ABSTRACT

RAJAGOPALAN, KOUSHIK N. Tools for Business Protocols. (Under the direction of Professor Munindar P. Singh).

Because of their centrality to modern applications of computing, the modeling and enactment of service engagements among autonomous participants is a crucial research direction. We adopt an interaction-oriented approach to service engagements in which interactions among two or more participants are abstractly defined as business protocols. In particular, a business protocol identifies two or more roles and describes the messages that the roles exchange along with the meanings of the messages and any constraints on the exchange of such messages. In this way, a business protocol offers a high-level description of an interaction underlying a service engagement, which each participant (playing one or more roles in the protocol) can realize with some flexibility based on its local policies.

Although several technical challenges have been addressed with respect to business protocols, currently no good tools exist that provide an effective means of developing and enacting business protocols. In this thesis, we seek to fill this gap by developing a suitable tool set. We begin from a conceptual model of a business protocol and use it as a basis for a specification language. We develop three tools based on the above conceptual model. First, for protocol specification, we develop a form-based tool with which to create protocols. Second, to enable implementing the modules for participants adopting various roles, we develop a role skeleton generator that yields the local views of a protocol with respect to each role. This tool incorporates within it a check for enactability and confirms that the role skeletons it generates can interoperate successfully. Third, we develop a state diagram generator that maps each protocol to a state diagram showing how the common or social state of the interaction would progress as the protocol is enacted. This state diagram can be used for an alternative implementation and further analysis.

We evaluate our tools by exercising them on use cases that are well-known in the literature.
Tools for Business Protocols

by

Koushik N. Rajagopalan

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 in Computer Science Raleigh, North Carolina 2010

APPROVED BY:

__________________________  ____________________________
Dr. Robert St. Amant        Dr. Kemafor Anyanwu

__________________________
Dr. Munindar P. Singh
Chair of Advisory Committee
DEDICATION

To my parents; and my friends
BIOGRAPHY

Koushik Narayanan Rajagopalan hails from Tuticorin, Tamil Nadu, India. He holds a Bachelor of Engineering degree from Thiagarajar College of Engineering, affiliated to Anna University. After working for a few years at Tata Consultancy Services Limited, he joined the MS program at North Carolina State University in August 2008.
ACKNOWLEDGMENTS

I would like to thank my adviser, Dr Munindar P. Singh, for guiding and helping me through this thesis. He has taught me the basics of research, and a lot about the area. I also thank my committee members Dr Kemafor Anyanwu and Dr Robert St. Amant for their support.

I thank Chung-Wei Hang, Prashant Kediyal and Kartik Tadanki for the most thoughtful comments and ideas.

I would like to thank my friends, without whom my time in Raleigh would not have been as fun-filled as it was. I will be taking with me a lot of pleasant memories of these two years. I would also like to thank my parents for believing in me and supporting me in all ways.
# TABLE OF CONTENTS

LIST OF TABLES ................................................................. vii

LIST OF FIGURES ............................................................... viii

Chapter 1 Introduction ......................................................... 1
  1.1 The Problem .................................................................. 1
  1.2 An Example Scenario: Purchase Protocol ................................. 5
  1.3 Approach in Brief .......................................................... 7
  1.4 Summary of Contribution .................................................... 8

Chapter 2 Background .............................................................. 9
  2.1 Conceptual Model ............................................................ 9
    2.1.1 Roles .................................................................... 10
    2.1.2 Messages ................................................................ 10
    2.1.3 Propositions ........................................................... 10
    2.1.4 Commitments ........................................................... 11
    2.1.5 Commitment Operation ................................................. 11
    2.1.6 Rules .................................................................... 12
    2.1.7 Expression ............................................................... 14
  2.2 The State of an Enactment ..................................................... 14
  2.3 Proves, Aware, Subscribed to ................................................. 15
  2.4 The Proves Checker ........................................................... 16
  2.5 The Awareness Checker ..................................................... 18

Chapter 3 Tools ...................................................................... 20
  3.1 Architecture .................................................................... 20
  3.2 Enactability Checker .......................................................... 20
    3.2.1 Localization of Choice ............................................... 22
    3.2.2 Message Verifiability ................................................... 22
    3.2.3 Commitment Condition Verifiability ............................... 22
    3.2.4 Commitment Operation Verifiability ............................... 22
    3.2.5 Commitment Discharge Verifiability .............................. 22
  3.3 Role Skeleton Generator ...................................................... 23
  3.4 State Diagram Generator .................................................... 23
  3.5 Protocol Editor ................................................................. 25

Chapter 4 Validation .............................................................. 26
  4.1 Purchase Scenario ............................................................ 26
  4.2 Wire Transfer Scenario ....................................................... 33
  4.3 Nonenactable protocols ...................................................... 39
LIST OF TABLES

Table 4.1 Messages in the purchase protocol ........................................ 27

Table 4.2 Messages in the money transfer protocol ............................... 34
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Nonlocal choice.</td>
<td>2</td>
</tr>
<tr>
<td>1.2</td>
<td>A local view is different from the global view.</td>
<td>3</td>
</tr>
<tr>
<td>1.3</td>
<td>A simple payment interaction.</td>
<td>4</td>
</tr>
<tr>
<td>1.4</td>
<td>State diagram of the simple purchase protocol.</td>
<td>5</td>
</tr>
<tr>
<td>2.1</td>
<td>A conceptual model for protocols.</td>
<td>11</td>
</tr>
<tr>
<td>2.2</td>
<td>Pay-by-check protocol.</td>
<td>13</td>
</tr>
<tr>
<td>3.1</td>
<td>Architecture.</td>
<td>21</td>
</tr>
<tr>
<td>4.1</td>
<td>Role skeleton of bank.</td>
<td>30</td>
</tr>
<tr>
<td>4.2</td>
<td>Role skeleton of buyer.</td>
<td>31</td>
</tr>
<tr>
<td>4.3</td>
<td>Role skeleton of shipper.</td>
<td>32</td>
</tr>
<tr>
<td>4.4</td>
<td>Role skeleton of seller.</td>
<td>32</td>
</tr>
<tr>
<td>4.5</td>
<td>State diagram of the purchase protocol.</td>
<td>33</td>
</tr>
<tr>
<td>4.6</td>
<td>Role skeleton of customer.</td>
<td>36</td>
</tr>
<tr>
<td>4.7</td>
<td>Role skeleton of origin bank.</td>
<td>38</td>
</tr>
<tr>
<td>4.8</td>
<td>Role skeleton of destination bank.</td>
<td>40</td>
</tr>
<tr>
<td>4.9</td>
<td>State diagram of the wire transfer protocol.</td>
<td>41</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

With the prevalence of Internet-based services, cross-organizational service engagements have become crucial in commercial and scientific software systems. The participants in such service engagements are autonomous, heterogeneous and dynamic [1]. Therefore, understanding interactions between the participants is essential for modelling service engagements. Business protocols describe such interactions in high-level terms. They help the autonomous participants communicate despite their heterogeneity and dynamism. Further, business protocols are centered on the notion of commitments based on which participants may track each other’s compliance to their agreements.

1.1 The Problem

Incorporating a (business) protocol within a working system requires the following steps: (1) Defining the participants and the rules of a protocol. (2) Checking all of the possible states that a protocol may lead to. (3) Checking for errors and extracting a local view of the interaction that can be incorporated within the design and implementation of an agent. The process of defining a protocol requires a formal way of defining the various attributes of the interaction being specified: the participants, the facts about the world, the agreements between the participants, and the rules of interaction. For this purpose, an easy and user friendly editor that aids a designer would be valuable. Currently, protocol designers do not have such a tool.

A protocol may yield an incorrect enactment. For example, consider a business
protocol in which the roles, say, CreditCardCompany, Seller, and Bank participate in a business collaboration. Let \textit{cardPaymentMsg} be a message sent by CreditCardCompany to Seller and \textit{checkPaymentMsg} be a message sent by Bank to Seller. Now, let us assume that there is a rule that states that for some condition, \( \psi \), \textit{cardPaymentMsg} and \textit{checkPaymentMsg} are mutually exclusive. This can lead to problematic execution scenarios such as the one shown in Figure 1.1. In this scenario, CreditCardCompany sends \textit{cardPaymentMsg} to Seller. Bank assumes that the CreditCardCompany will not send \textit{cardPaymentMsg} and sends \textit{checkPaymentMsg}. This creates an incorrect conversation even though each of the roles behaved correctly. The problem just described is called \textit{nonlocal choice}. The other category of problems that may affect a protocol is called \textit{blindness}. Desai and Singh [2] discuss the types of problems that lead to incorrect enactment of a protocol. A protocol has to be verified to make sure that it avoids such incorrect enactments.

A designer has a high-level view of the enactment and specifies a protocol in a global frame of reference. However, each role possesses a limited view of the real world. It can only observe the messages that it sends or receives. This difference in perspectives poses a challenge in adopting a protocol for enactment. Let \( P \) be the protocol and \( P(r) \) represent the rules specified from a global perspective. To use a protocol for service engagement, a role has to derive a set of rules from \( P(r) \) based upon the possible observations made by the role. So, the challenge here is translating the rules into a form that can be understood and evaluated by a role.

For example, consider a protocol that describes the interaction shown in Figure 1.2.
In this scenario, there are four roles: Payer, Shipper, Payee, and Bank. The payer sends sendCheckInMailMsg to the shipper. The shipper on the receipt of sendCheckInMailMsg, sends deliverMailMsg to payee. The payee, on receipt of deliverMailMsg, sends encashMsg to bank. The protocol defines two rules:

If sendCheckInMailMsg has not occurred, then deliverMailMsg should not occur \( (1.1) \)

If sendCheckInMailMsg has not occurred, then encashMsg should not occur \( (1.2) \)

Let us consider Rule 1.2. The sender of encashMsg has no way of knowing if sendCheckInMailMsg has occurred or not. We can observe sendCheckInMailMsg from a global perspective. But observations of the payee does not include sendCheckInMailMsg. The payee should have this rule translated into a form that it can evaluate based on what it observes.

If deliverMailMsg has not been received, then encashMsg should not be sent \( (1.3) \)

Rule 1.3 is more relevant to the payee than Rule 1.2. An agent needs rules that are relevant to its observations.

A protocol designer cannot premeditate all the possible enactments that a protocol can allow. There may be certain enactments that a protocol may allow but they may lead to an inconsistency in the business collaboration that the protocol represents. In other cases, the protocol may not allow a valid enactment of the protocol. Ideally such scenarios should
be avoided. We need to generate all possible enactments of a given protocol so that the
designer can identify a need for any modification in the specification.

Let us consider the scenario of a simple purchase protocol. The goal of the protocol
is to enable a buyer to make a purchase. There are two roles: Buyer and Seller. The seller
makes an offer to the buyer by sending \textit{offerMsg}. The occurrence of \textit{offerMsg} creates
OrderCommitment. The buyer can send \textit{payMsg} to make a payment. The buyer can send
\textit{rejectMsg} to cancel the offer. The message, \textit{rejectMsg}, cancels OrderCommitment. We find
that it allows the following enactments:

1. \textit{offerMsg, payMsg, rejectMsg}.
The payer receives a reject payment while the bank actually pays the payee.

2. \textit{offerMsg, rejectMsg, payMsg}.
The bank sends a confirmation message even before payment is done.

In the list above, we show a conversation by listing the messages in the order in
which they were exchanged.

Figure 1.4 shows the possible enactments of the protocol and the way the com-
mitment undergoes a change at each step. A protocol designer might want to understand
all the possible transitions that a commitment can be subject to during the enactment
of a protocol. Visualization of all allowed enactments helps a designer reason about the
commitments.
In our work, we develop tools that address these challenges.

1.2 An Example Scenario: Purchase Protocol

Let us consider electronic commerce. In this scenario, there are buyers and sellers who participate in the transaction. Let us consider that the buyer needs a product from the seller. The buyer requests a quote (by sending a message \textit{rfq}) to different sellers. Sellers may choose to respond with a quote (by sending a message \textit{offerDetails}) or not respond at all. The buyer chooses the quote that best serves its goals and accepts that quote (\textit{acceptQuote}). The buyer may reject other quotes (\textit{rejectQuote}). The buyer, after accepting, would then use a payment method like credit card or PayPal to pay for the product (\textit{payment}). Once the \textit{payment} is received, the seller ships the product to the buyer (\textit{ship}). For this the seller may use a shipping service provided by, say, DHL or Fedex.

It can be observed that the interactions occur between multiple parties that may represent different organizations. The organizations act independently of each other, even while they cooperate to produce more value for each other. Traditional approaches tend to specify the interactions from a perspective of a single participant. A business protocol specifies the interactions from a global perspective. Organizations make their own decisions but conform to the rules of the protocol.

![State diagram of the simple purchase protocol.](image-url)
A protocol that defines the interaction between the participants in the scenario above would define the following roles: buyer, seller, bank and shipper.

For modelling an interaction with a bank, let us use payment by credit card, where the payer requests the bank to pay from the payer’s account and the bank can choose to accept or reject based on its policies. When the bank accepts the payment request, it sends the payment (pay) to the payee and notifies the payer (confirmPaymentReq). We define all the messages described above: rfq, offerDetails, acceptOffer, rejectOffer, requestPayment, rejectPaymentReq, confirmPaymentReq, pay and ship. We also define some rules, these include:

If payment has not occurred, then ship should not occur \( (1.4) \)

If quoteDetails has occurred, then either payment should not occur or rejectQuote should not occur \( (1.5) \)

From a designer’s perspective, Rules 1.4 and 1.5 are correct. However, there are problems with the specification from a role’s perspective. Consider Rule 1.4 that states that ship cannot occur before payment. The shipper (sender of ship message) has no way of knowing if payment has occurred or not. This is a case of blindness.

Consider the Rule 1.5 that states that if quoteDetails message occurs, then either payment should not occur or rejectQuote should not occur. Though this rule makes sense from the perspective of the seller, it can lead to incorrect enactment. The payment message is sent by the bank. The bank has no way of controlling the occurrence of rejectQuote. This is an example of nonlocal choice.

Since the shipper does not observe payment, the shipper cannot use Rule 1.5 for its enactment. The shipper agent, to participate in the interaction, it has to derive new rules based on what the shipper can observe.

Also, a need for additional rules cannot be identified unless we get to know all possible enactments of a protocol. As a protocol designer, we cannot premeditate all allowed enactments. In this scenario, we might have to include a rule, say, if quoteDetails has not occurred, then payment should not occur. But in a complex protocol, reasoning about a need for modification to the rules becomes difficult.
1.3 Approach in Brief

The participants, the facts about the world, agreements between the participants and the rules of interactions all form the part of the domain model of a protocol. In our approach, the domain model is the basis for the tools that we develop. We adopt the OWL-P model proposed in [3]. The process of specifying the participants and the constraints on the interaction is called protocol specification. The document produced out of the process is protocol specification document. To address the challenge of protocol specification, we design and implement an editor. The editor produces a protocol specification document. For this, we made appropriate specializations to the OWL-P model.

Properties such as nonlocal choice and blindness are the causes of incorrect enactment of a protocol. To ensure that a given protocol accounts for these challenges, we implement the enactability checker. This component checks for the five properties listed by Desai and Singh [2] that ensure that a protocol satisfying these properties avoids incorrect enactment.

The next challenge that we address is the generation of the role skeletons from a protocol specification. A role skeleton is a role’s perspective of a protocol. Desai and Singh [2] discuss the problem of extracting a role skeleton from a protocol. They describe an algorithm through which, given a protocol and a role, a role skeleton is generated. We follow the algorithm described in [2], with a few modifications. We integrate the enactability checker with this tool for validation of the given protocol. The tool can be used by the protocol developer to extract a role skeleton. The input to this tool is a protocol and a role; and the output is a set of rules extracted from the protocol but translated to the role’s perspective.

Finally, we address the challenge of visualization of all the enactments that a given protocol allows. It is done by generating all enactments and then pruning the ones that are disallowed. The problem reduces to generation of a tree in which the nodes are the states of a protocol and the edges are the messages that are exchanged. The root of the tree is the state where no message is exchanged. The possible transitions from a given node are all the messages that can be exchanged. Given a protocol, the tool for visualization generates a state diagram of all allowed enactments.
1.4 Summary of Contribution

We address the challenges by designing and building the tools that solve the problems described in the previous sections.

A tool for protocol specification should be user friendly and should be capable of specifying a complex protocols. Our Protocol Editor accounts for these challenges while providing a user interface through which we can specify a protocol.

A tool to generate a role skeleton should accept a protocol and a role for which the role skeleton should be generated. Such a tool should be capable of being integrated with a larger tool. We build a tool that can produce a role skeleton using the approach described in [2]. We also provide suitable interfaces that help in integrating our tools with other tools. We also introduce an error checking module with this role skeleton generator that checks for the enactability of a protocol.

The visualization of protocol enactment is done through a tool called State Diagram Generator. This tool makes use of an important component, the Rule Evaluator, that acts as an inference engine. We design and implement an algorithm for rule evaluation. The State Diagram Generator has a suitable interface through which we can integrate it with other tools.

We discuss the background and concepts in Chapter 2 and the design and implementation of these tools are discussed in Chapter 3. Chapter 4 shows two example scenarios and how our tools can be used in such scenarios. Chapter 5 discusses related works and points out areas of future work.
Chapter 2

Background

This chapter introduces a few concepts related to business protocols. It starts with the model of a protocol. It then proceeds to introduce the concept of a state of enactment. Finally, it introduces properties like proves, awareness, and subscription that we will use in our tools.

2.1 Conceptual Model

The conceptual model is the core artefact based on which all the tools have been developed. We adopt the OWL-P model proposed by Desai et al. [3]. Agents are software entities that are situated in an environment, persistent, and autonomous in the sense that they are in total control of their behaviour. A protocol is a description of the interaction between two or more participants to achieve certain business goals. The agents, in order to collaborate with other agents, adopt a role from a protocol. Roles form an abstraction of the business partners in an interaction. The facts about the environment are represented as propositions. Every proposition has zero or more parameters called slots. For example, a payment of $100 may be denoted as payment($100) where payment is a proposition with a slot, amount, that takes a value of $100. The agents interact with each other through messages. A message is represented in a rule system as a proposition. The messages have slots too. A message derives its slots from the proposition that represents the message in the rule system. For example, let there be a message $m$ and let a proposition $p$ represent $m$ in the rule system. In this case, if $p$ has slots, say, slot1, slot2, and slot3, then $m$ also
A commitment is a data structure that represents an element of a contractual agreement made by the participants. It has a creditor and a debtor, which are roles, and an antecedent and a consequent, which are expressions. We introduce a data structure called expression to represent a Boolean expression of propositions which can be translated to a form that is compatible with any rule system. It is useful in specifying the antecedent and consequent of a commitment. A protocol dictates rules that govern the interactions between the participants. Figure 2.1 shows the domain diagram that we use. The following sections describe these concepts to a finer detail.

2.1.1 Roles

Roles are an abstract boundaries of the participants of a business protocol interaction. Agents adopt the roles and enact a protocol. The agents should interact in a manner that is allowed by the protocol.

2.1.2 Messages

A message is an unit of interaction. It has a sender and a receiver. It also has zero or more slots that hold objects to be exchanged. Slots hold data values. For each message, there exists a proposition that represents the occurrence of the message in the rule system. A message can also bring about a change in the state of a commitment by enabling commitment operations. When a message is sent or received, its associated proposition gets enabled.

2.1.3 Propositions

Propositions are facts about the world. They assume a Boolean value. They may also represent, semantically, the occurrence of a message. They have one or more parameters called slots [3]. A proposition is said to be enabled if the value of the proposition is true. It is disabled, otherwise. All propositions are disabled at the start of enactment. As the role makes observations and inferences, propositions are enabled. We assume that a proposition that has been enabled cannot be disabled.
2.1.4 Commitments

Commitments formalize the contractual obligations that may arise between two participants [4]. Commitment is a 4-tuple represented as \((\alpha, \beta, \omega_1, \omega_2)\). It represents the fact that an agent, \(\alpha\), owes a consequent, \(\omega_2\), to another agent, \(\beta\), provided an antecedent, \(\omega_1\), holds. Commitments can also be unconditional. In such cases, the antecedent, \(\omega_1\), is represented by \(true\). Since at design-time, the agents are non-existent and since we can place a constraint an agent’s behaviour by limiting the the behaviour of the role, we choose roles to represent the creditor and debtors of a commitment in design time. The antecedent and consequent of a commitment are of type Expression.

2.1.5 Commitment Operation

Commitment operations are actions that manipulate the state of a commitment. A commitment operation has four parts: (1) Action (2) Commitment (3) From role (4) To role. The commitment, say c, indicates that the state of c has to be modified. The from role and the to role indicate the roles in which the state of the c has to be changed. The action can take one of the following values: CREATE, RELEASE, ASSIGN, DELEGATE, and
A commitment operation has a set of roles that are subscribed to the commitment operation \[2\]. The details are discussed in Section 2.3. They affect only those roles that are subscribed to the commitment action.

2.1.6 Rules

Rules of a protocol describe the constraints on the interactions between the roles. For example, let us consider the example described in Section 1.2. The seller sends the offer details through a message \textit{offerDetailsMsg} that is represented in rule system as \textit{offerDetails}. The buyer, if it accepts the offer, sends the acceptance to the seller through the message \textit{acceptOfferMsg} represented in the rule system as \textit{acceptOffer}. The buyer, if it rejects the offer, sends the reject message \textit{rejectOfferMsg} represented in the rule system as \textit{rejectOffer}.

As a designer, we need to place a simple restriction on the buyer agent that it has to choose between \textit{acceptOfferMsg} and \textit{rejectOfferMsg} and cannot send both the messages. This restriction can be represented as a rule \textit{offerDetails} → ¬\textit{acceptOffer} ∨ ¬\textit{rejectOffer}.

This rule represents the constraint that if the proposition, offerDetails is enabled, then either one or both of the propositions acceptOffer or rejectOffer should be false. Each of these propositions represent a message. Let \( p \) be a proposition and \( m \) be a message that is represented by \( p \). When \( p \) is disabled, \( m \) is also disabled. Disabling a proposition through a rule makes sure that the message that can enable the proposition does not occur. A rule that states that a message \( m_1 \) should occur before another message \( m_2 \), is specified as \( ¬p_1 \rightarrow ¬p_2 \) where \( p_1 \) and \( p_2 \) are propositions that represent \( m_1 \) and \( m_2 \) respectively.

A rule has a body, a head and a list of clauses. It is represented in the format body → head. Any set of propositions is said to satisfy a rule if whenever the body is true, head is also true. The head and body are of type Boolean expression.

Example

Let us consider a simple pay by check scenario. There are three participants: Payer, Payee, and Bank. The intent of the interaction is that an agent that plays the role of payer makes a payment to an agent playing the role of payee. The payee makes the payment through a check. The payee, on the receipt of the check, encashes the check by interacting with the bank. Figure 2.2 shows the interaction between these participants. We
Figure 2.2: Pay-by-check protocol.

specify pay-by-check protocol with following artefacts:

- Roles: Payer, Payee, and Bank.

- Propositions: sendCheckProp(checkNumber, amount), encashProp(checkNumber), accountCreditedProp(amount), checkRejectProp(checkNumber), informCheckAcceptedProp(checkNumber), informCheckRejectProp(checkNumber), and payment(amount).

- Messages: sendCheckMsg, encashMsg, accountCreditedMsg, checkRejectedMsg, informCheckAcceptedMsg, and informCheckRejectMsg

- Commitments: PaymentCommitment (payer, payee, encashMsg, payment)

- Rules:

  1. When sendCheckMsg is sent, create PaymentCommitment.
     
     This rule is specified as sendCheckProp → CREATE(PaymentCommitment).

  2. encashMsg cannot occur before sendCheckMsg.
     
     This rule is specified as ¬sendCheckProp → ¬encashProp.
3. \textit{accountCreditedMsg} cannot occur before \textit{encashMsg}.
   This rule is specified as \( \neg \text{encashProp} \rightarrow \neg \text{accountCreditedProp} \).

4. \textit{checkRejectedMsg} cannot occur before \textit{encashMsg}.
   The rule is specified as \( \neg \text{encashProp} \rightarrow \neg \text{checkRejectedMsg} \).

5. \textit{informCheckAcceptedMsg} cannot occur before \textit{accountCreditedMsg}.
   This rule is specified as \( \neg \text{accountCreditedProp} \rightarrow \neg \text{informCheckAcceptedProp} \).

6. \textit{checkRejectMsg} cannot occur before \textit{checkRejectMsg}.
   This rule is specified as \( \neg \text{checkRejectProp} \rightarrow \neg \text{informCheckRejectProp} \).

7. \textit{accountCreditedMsg} and \textit{checkRejectedMsg} are mutually exclusive.
   This rule is specified as \( \text{encashProp} \rightarrow \neg \text{accountCreditedProp} \lor \neg \text{checkRejectedProp} \).

8. An occurrence of \textit{accountCreditedMsg} implies that payment has been done.
   This rule is specified as \( \text{accountCreditedProp} \rightarrow \text{payment} \).

\textbf{2.1.7 Expression}

An expression has an operator and a list of operands and represents a Boolean expression that can be evaluated to produce a Boolean result. The operands of an expression can be an expression or a proposition. The operators can be the Boolean operators AND, NOT, and OR.

\textbf{2.2 The State of an Enactment}

The state of an enactment of a protocol is a list of states, one for each agent playing a role in the protocol. The state of enactment of an agent playing a given role is a conjunction of each enabled proposition and the negation of each disabled proposition. For example, let payment and goodsDelivered be two propositions. If payment is made but goods have not been delivered,

- Enabled propositions: payment, ...
- Disabled propositions: goodsDelivered, ...
- State: \((\text{payment} \land \ldots) \land (\neg \text{goodsDelivered} \land \neg \ldots)\).
A state is considered to be inconsistent if there exists a proposition that is both enabled as well as disabled. A state is considered to be consistent, if it is not inconsistent.

2.3 Proves, Aware, Subscribed to

The following concepts form the core in the algorithm for enactability checking and role skeleton generation.

Proves

Given a protocol that specifies two propositions, prop1 and prop2, prop1 proves the occurrence of prop2 if

- there exists a rule of the form \(-\text{prop2} \rightarrow \neg\text{prop1}\).
- there is a proposition, prop3, and there exists a rule \(-\text{prop3} \rightarrow \neg\text{prop1}\) and prop3 proves prop2.

For example, consider pay-by-check protocol. Here, encashProp proves sendCheckProp.

Aware

Given a protocol that specifies a role, \(\alpha\) and a proposition prop. \(\alpha\) is said to be directly aware of prop if

1. prop represents a message that \(\alpha\) receives.
2. prop represents a message that \(\alpha\) sends.

Given a protocol that specifies a role, \(\alpha\) and a proposition prop. \(\alpha\) is said to be indirectly aware of prop if \(\alpha\) is not directly aware of prop and

1. prop is proved by another proposition that represents a message that \(\alpha\) receives.
2. prop can be inferred through a rule that is of the form \(\psi \rightarrow \text{prop}\) and \(\alpha\) is aware of \(\psi\).

A role \(\alpha\) is aware of a proposition prop if \(\alpha\) is directly aware of prop or \(\alpha\) is indirectly aware of prop.

For example, let us consider pay-by-check protocol. An agent playing the role of bank is directly aware of encashProp and indirectly aware of sendCheckProp.
Subscription

A role, \( \alpha \), is said to be subscribed to a commitment operation, \( op \), if one of the following conditions is satisfied:

1. if \( op \) is one of the following: \textit{create}, \textit{release}, and \textit{cancel}; and \( \alpha \) is the creditor or the debtor of the commitment.

2. if \( op \) is \textit{delegate} and \( \alpha \) is the delegatee or the debtor of the commitment.

3. if \( op \) is \textit{assign} and \( \alpha \) is the assignee or the creditor of the commitment.

For example, let us consider pay-by-check protocol. The rules of the protocol specifies \textit{create} (PaymentCommitment). The roles payer and payee are subscribed to \textit{create} (PaymentCommitment).

The algorithm that we use to check enactability and generate role skeleton depend on these properties extensively. We develop components that can check these properties.

2.4 The Proves Checker

A common inference that needs to be made based on the rules of a protocol is that of \textit{proves}. Given two propositions, say \( p \) and \( q \), we should be able to check if \( p \) proves \( q \).

The Proves Checker is a component that, given a protocol, extracts the rules of the protocol and identifies, for each proposition \( p \), all the propositions that are proved by \( p \). A challenge here is identifying the propositions that are indirectly proven by \( p \). For example, given three propositions, \( p \), \( q \), and \( r \); if \( p \) proves \( q \) and \( q \) proves \( r \) and there is no rule that tells that \( p \) proves \( r \), our component should be able to identify \( q \) and \( r \) in the list of propositions proven by \( p \). To accomplish this task, we view each proposition as a node of a graph and \textit{proves} as edge of the graph. If \( p \) and \( q \) are two propositions, then they are viewed as two nodes, say \( P \) and \( Q \). If \( p \) proves \( q \) then we view it as a directed edge from \( P \) to \( Q \). This allows us to use graph algorithms like depth first traversal which can be useful to a) find all the nodes in a graph that can be reached from a given node b) detect loops (or cycles) in the graph which implies that the specification is not sound.
Approach

The algorithm that we implemented is as follows:

Step 1: Initialize a map, `provesMap`, that holds a proposition as its key and a list of propositions as its value.

Step 2: Iterate through the rules. If a rule of the form $\neg prop2 \rightarrow \neg prop1$ is found, then prop1 proves prop2. Add prop2 to the list of propositions proved by prop1 in `provesMap`.

Step 3: Once all the rules are processed, do a depth first traversal through the `provesMap`. Add the children of every child node to a given node. This step allows us to complete the list of all propositions that are proved by a given proposition. For example, if prop1 proves prop2 and prop2 proves prop3, then after we perform the depth first traversal, we add prop3 to the list of propositions proved by prop1.

For example, in pay-by-check protocol, among other rules, we have these rules:

- $\neg sendCheckProp \rightarrow \neg encashProp$.
- $\neg encashProp \rightarrow \neg accountCreditedProp$.
- $\neg accountCreditedProp \rightarrow \neg informCheckAcceptedProp$.

When we apply our algorithm on pay-by-check protocol, after Step 2, we have `provesMap` with `sendCheckProp` in the list of propositions proved by `encashProp`, `encashProp` in the list of propositions proved by `accountCreditedProp`, and `accountCreditedProp` in the list of propositions proved by `informCheckAcceptedProp`.

We choose propositions in an arbitrary order and perform a depth first traversal. For example, we might visit `encashProp` and find that `sendCheckProp` does not prove any other propositions. So, we do not make changes to `sendCheckProp`’s list. We might then visit `informCheckAcceptedProp`. It proves `accountCreditedProp` which has not been visited yet. So, we visit `accountCreditedProp` and find that it proves `encashProp`. Since we have visited `encashProp`, we add all the propositions proved by `encashProp` to the list of all propositions proved by `accountCreditedProp`. So, the new list for `accountCreditedProp` has `encashProp` and `sendCheckProp`. We then return to `informCheckAcceptedProp` and add `encashProp` and `sendCheckProp` to `informCheckAcceptedProp`’s list.
2.5 The Awareness Checker

A common functionality in our tool set is to check, given a role $\alpha$ and a proposition $p$, if $\alpha$ is aware of $p$. Also we need to identify if the role is directly aware of or indirectly aware of the proposition.

The goal of the awareness checker is that, given a protocol, extract all the roles and for a role, get all the propositions that the role is aware of.

Approach

Step 1: Initialize a map, $awareMap$, with role as the key and value as a list of propositions that it is aware of. Initially, the list should be empty.

Step 2: For the receiver of each message, get the list of propositions, listOfAwareProp, that the receiver is aware-of (use $awareMap$). To listOfAwareProp, add the proposition that represents the message.

Step 3: For each role, get the list of propositions, listOfAwareProp, that the role is aware-of. For each proposition, prop, in listOfAwareProp, add to the list all the propositions that prop proves.

Step 2: For the sender of each message, get the list of propositions, listOfAwareProp, that the sender is aware-of. To listOfAwareProp, add the proposition that represents the message.

Step 4: For each rule of the form $prop1 \rightarrow prop2$, add $prop2$ to each list that contains $prop1$

Let us consider pay-by-check protocol. After Step 1, we have $awareMap$ that contains the following:

- Payer: (informCheckAcceptedProp, informCheckRejectProp).
- Payee: (accountCreditedProp, checkRejectedProp).
- Bank: (encashMsg).

After Step 2, we have:
• Payer: (informCheckAcceptedProp, informCheckRejectProp, accountCreditedProp, checkRejectedProp, encashProp, sendCheckProp).

• Payee: (accountCreditedProp, checkRejectedProp, encashProp, sendCheckProp).

• Bank: (encashMsg, sendCheckProp).

After Step 3, we have:

• Payer: (informCheckAcceptedProp, informCheckRejectProp, accountCreditedProp, checkRejectedProp, encashProp, sendCheckProp).

• Payee: (accountCreditedProp, checkRejectedProp, encashProp, sendCheckProp, informCheckAcceptedProp, informCheckRejectProp).

• Bank: (encashMsg, sendCheckProp, accountCreditedProp, checkRejectedProp).

After Step 4, we have:

• Payer: (informCheckAcceptedProp, informCheckRejectProp, accountCreditedProp, checkRejectedProp, encashProp, sendCheckProp, payment).

• Payee: (accountCreditedProp, checkRejectedProp, encashProp, sendCheckProp, informCheckAcceptedProp, informCheckRejectProp, payment).

• Bank: (encashMsg, sendCheckProp, accountCreditedProp, checkRejectedProp, payment).

In most cases, our tool has to deal with expressions (or combinations of propositions). A role is aware of an expression, if it is aware of every proposition in the expression.
Chapter 3

Tools

This chapter describes the architecture of the tools. It then goes on to explain the approach that we use to implement each of the tools.

3.1 Architecture

We develop four tools: Protocol Editor, Enactability Checker, Role Skeleton Generator, and State Diagram Generator. The Protocol Editor creates a protocol specification document. The Enactability Checker (which gets integrated with Role Skeleton Generator) checks for the enactability properties of a protocol. Role Skeleton Generator generates a role skeleton of a role from a given protocol. The State Diagram Generator helps in visualizing enactments and helps us reason about them.

Figure 3.1 explains the architecture and the way these components interact to produce a desired output.

3.2 Enactability Checker

The Enactability checker is a tool that checks if a protocol can guarantee correct enactments. Desai and Singh [2] discuss the causes of incorrect enactment of protocols. They propose five properties that, when satisfied, ensures that the protocol allows only correct enactments. The properties include:

1. Localization of Choice: When two messages are mutually exclusive, then they should
have the same sender. Formally, if there is a rule of the form $\psi \rightarrow \neg p \lor \neg q$, where $p$ and $q$ are propositions that represent messages $m_1$ and $m_2$ respectively and $\psi$ is the body of the rule, then $m_1$ and $m_2$ should have the same sender.

2. Message Verifiability: When a message is blocked by a condition, the sender of the message should be aware of the blocking condition. Formally, if there is a rule of the form $\psi \rightarrow \neg p$, where $p$ is a proposition that represents a message, $m$, and $\psi$ represent an expression is the body of the rule, then the sender of $m$ should be aware of $\psi$.

3. Commitment Condition Verifiability: The creditor and the debtor of a commitment should be aware of its antecedent and consequent.

4. Commitment Operation Verifiability: If a role is subscribed to a commitment operation, $op$, then it should be aware of the $op$.

5. Commitment Discharge Verifiability: There should be at least one enactment where a commitment, once created, is discharged.
Enactability Checker checks for these properties. The following sections explain the implementation of the modules that check for these properties is implemented.

3.2.1 Localization of Choice

Checking for localization of choice is done by iterating through all the rules. If, there exists a rule that states that propositions prop1 and prop2 are mutually exclusive, then get the message msg1 and msg2 that are represented by prop1 and prop2 respectively. Check if msg1 and msg2 have the same sender.

3.2.2 Message Verifiability

Checking for message verifiability is done by iterating through all the rules. If, there exists a rule of the form $\omega \rightarrow prop$, where prop is a proposition that represents a message msg and $\omega$ is an expression representing a condition, then check if the sender of msg is aware of $\omega$.

3.2.3 Commitment Condition Verifiability

For any commitment operation, op, that is a RELEASE of a commitment, use awareness checker to check if the roles subscribed to the op are aware of the antecedent and the consequent.

3.2.4 Commitment Operation Verifiability

For each commitment operation that occurs in the rules, get the list of roles that are subscribed to the commitment operation. For each subscribed role, $\alpha$, use awareness checker to check if $\alpha$ is aware of the commitment operation.

3.2.5 Commitment Discharge Verifiability

For each commitment, create a state with propositions CREATE of commitment, antecedent, consequent. Use Rule Evaluator to make more inferences and get the new state. Check if the given state is consistent.
Once all these conditions are checked, the algorithm to generate role skeleton is called. The algorithm returns a set of new rules that constitute the role skeleton of a target role.

### 3.3 Role Skeleton Generator

This component extracts a role skeleton. The algorithm implemented is as follows.

Given a protocol and a role $\alpha$:

**Step 1:** For each rule, take the head of the rule.

**Step 2:** If the head is of the format $\neg p \lor \neg q$ where $p$ and $q$ are propositions that represent the messages sent by the role $\alpha$; then add the rule to the output list of rules.

**Step 3:** If the head is of the format $\neg p$ where $p$ is a proposition that represents a message received by $\alpha$; then add it to the output list of rules.

**Step 4:** If the head of the format $p$ where $p$ is a proposition that $\alpha$ is aware of, then add it to the output list of rules.

**Step 5:** For each rule in the output list of rules, if it is of the form $\neg p$ where $p$ represents a message, $m$, that $\alpha$ sends, then extract the body of the rule. For each proposition in the body of the rule that $\alpha$ is indirectly aware of, say $r$, replace $r$ with a proposition that proves $r$ such that $\alpha$ is directly aware of $r$. Change the head to the form $\neg !m$ where $!m$ represents the send of $m$.

**Step 6:** For each rule in the output list of rules, if it is of the form $\neg p$ where $p$ represents a message, $m$, that $\alpha$ receives, then extract the body of the rule. For each proposition in the body of the rule that $\alpha$ is not aware of, remove it from the body. Change the head to the form $\neg ?m$ where $?m$ represents the receive of $m$.

### 3.4 State Diagram Generator

The State Diagram Generator generates all possible states of a protocol. Since the state of enactment is the set of states of all roles, we need to maintain the states of all roles
in a given state. In other words, for each role, all the propositions that are enabled during the current enactment have to be kept track of.

The first step is the extraction of role skeleton from protocol specification. The role skeleton extraction in this step follows a slightly different algorithm than what is described above. Given a protocol and a role, $\alpha$;

Step 1: Iterate through all the rules of the protocol. Create a list of rules, outputList, to which all the role skeleton’s rules are added.

Step 2: If the rule is of the form $\psi \rightarrow \neg p$;
   If $p$ represents a message that the role sends, then add the rule to the outputList
   If $p$ represents a message that the role receives, then add $p \rightarrow \neg \psi$ to outputList.

Step 3: For any other rule, if $\alpha$ is aware of the head of the rule, then add it to the outputList.

A set of agents enacting a protocol makes a transition from one state to another by sending or receiving a message. We consider only the quiescent states that occur after all messages are received by their respective receivers. The generation of a state diagram is done by starting from the initial state when no message has been exchanged. At any given state, the possible transitions are the set of all message that has not been used by its ancestors. When a state is consistent, it can be added to the tree and an attempt to visit its child nodes can be made. When a state is inconsistent, it can be ignored.

Approach

Step 1: The State Diagram generation is construction of a tree. The tree has a root node when no message has been exchanged. The set of all messages form the possible transitions.

Step 2: Take the root node and do a depth first traversal.

Performing DFS: The following steps illustrate the algorithm that we use to generate the state diagram.

Step 1: For a given node, $startNode$, iterate through the list of possible messages. For each message, $action$, do the following steps.
Step 2: Create a node, childNode by applying the action on startNode. This involves copying all the propositions from startNode to childNode; adding the proposition, prop that represents action; calling Rule Evaluator to make more inferences about the state.

Step 3: If childNode is consistent, then add it to the tree. Change the states of commitments based on the new commitment operations that may be observed. Perform DFS with childNode. If childNode is inconsistent, skip the node and go to Step 2.

Step 4: If no childNode is produced or if no childNode was added to the tree, then add the startNode to the list of leaf nodes.

The final output of the state diagram is a list of leaf nodes. The users can iterate through the list of leaf nodes and perform operations on the tree to make more meaningful inferences.

3.5 Protocol Editor

The Protocol Editor is the tool that aids in protocol specification. It is an Eclipse Plugin and produces a protocol specification document. The User Interface is sufficient to handle all the elements of a protocol specification.
Chapter 4

Validation

This chapter employs two common scenarios to show how our tools help the designers of a protocol and the developers of a participating system. The first scenario is a simple on-line purchase protocol as described in Section 1.2. The second scenario is a simple wire transfer protocol for money.

In brief, each agent plays a role in enacting a protocol, the protocol defines the rules of interaction and allows the agents to frame their policies. A protocol designer uses the protocol editor to create a protocol specification document. This protocol specification document is then used as the input to the role skeleton generator and a role skeleton corresponding to a role is produced. The protocol specification document is also used as an input to generate a state diagram.

4.1 Purchase Scenario

We build on the example scenario described in the Section 1.2. There are four roles: buyer, seller, shipper and bank. The agent playing buyer starts the conversation by sending request for quote message, \( rfqMsg \) represented in the rule system by the proposition \( rfqProp \) which has a slot, Item. The seller quotes Item price \( offerDetailsMsg \), represented in the rule system by offerDetailsProp. The buyer may respond with an accept message \( acceptOfferMsg \) if the buyer decides to accept the offer made by the seller. The \( acceptOfferMsg \) is represented in the rule engine by acceptOfferProp. The buyer also has an option to reject the offer through \( rejectOfferMsg \), represented in the rule system by rejectOffer-
Table 4.1: Messages in the purchase protocol

<table>
<thead>
<tr>
<th>Message</th>
<th>Slots</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>rfqMsg</td>
<td>Item, quantity</td>
<td>Request for quote message from buyer to seller</td>
</tr>
<tr>
<td>offerDetailsMsg</td>
<td>Item, quantity, price</td>
<td>Details about the Items and price from the seller to the buyer</td>
</tr>
<tr>
<td>rejectOfferMsg</td>
<td>Item, price</td>
<td>Buyer informs the seller that it has rejected the offer</td>
</tr>
<tr>
<td>requestPaymentMsg</td>
<td>amount, card-Number, verification</td>
<td>Buyer requests the bank to send a payment</td>
</tr>
<tr>
<td>rejectPayReqMsg</td>
<td>cardNumber</td>
<td>Bank informs the buyer that it has rejected the request for payment</td>
</tr>
<tr>
<td>confirmPaymentMsg</td>
<td>amount, card-Number</td>
<td>Bank informs the buyer that it has paid the amount</td>
</tr>
<tr>
<td>cardPaymentMsg</td>
<td>amount</td>
<td>Bank credits the seller’s account with the amount</td>
</tr>
<tr>
<td>shipRequestMsg</td>
<td>Item</td>
<td>Seller request the shipper to ship the Item</td>
</tr>
<tr>
<td>shipmentMsg</td>
<td>Item</td>
<td>Shipper ships the Item to the buyer</td>
</tr>
<tr>
<td>acknowledgeMsg</td>
<td>Item</td>
<td>Buyer releases the commitment</td>
</tr>
</tbody>
</table>

Prop. If the buyer sends an accept message, it has to pay the seller through the bank. It requests for payment by sending requestPaymentMsg, with creditCardNumber and amount. The message requestPaymentMsg is represented as requestPaymentProp. The bank can choose to accept the request or reject the request. If it chooses to accept, it sends out the payment to the seller through paymentByCardMsg, represented as paymentByCardProp. It then sends a confirmation message to the buyer through paymentConfirmedMsg, represented as paymentConfirmedProp. The seller, on receiving paymentByCardMsg, delegates the commitment to the shipper through shipRequestMsg, represented as shipRequestProp. The shipper ships the Item by sending the shipMsg, represented as shipProp. The buyer then acknowledges the receipt of the shipMsg by sending acknowledgeMsg, represented as acknowledgeProp. The messages, their corresponding propositions, slots, and description are tabulated in Table 4.2.

The core of the protocol is the commitment, OrderCommitment, that has the
buyer as the initial creditor, the seller as the debtor, payment as the antecedent and ship as the consequent. The 4-tuple (seller, buyer, payment, ship) represent the commitment as created initially. The protocol transition can be explained as a change in the state of the commitment. For example, when a payment is made, the seller delegates the commitment to the shipper. The shipper then discharges the commitment by sending the *ship* message.

To ensure consistent enactment, we define the following rules in the protocol. These rules specify the commitment operations that need to take place, thus they change the state of a commitment.

1. The offerDetailsProp should create OrderCommitment.
2. The rejectOfferProp should cancel OrderCommitment.
3. The shipRequestProp should delegate OrderCommitment to the shipper.
4. The acknowledgeProp should release OrderCommitment.

Next, additional rules arise out of data flow requirements:

1. The seller cannot make an offer unless a buyer requests for an offer to be made on an Item.
2. The buyer cannot reject an order before an order is placed. Also, it cannot make a payment before the price is known.
3. The seller cannot send a good unless he knows what the buyer wants.
4. A shipper cannot ship an Item before a request for shipment is made by seller by requesting a pickup.
5. The bank cannot pay or reject a payment request unless a payment request is made.
6. The bank cannot send a confirmation of payment without a payment being made.
7. The buyer cannot send an acknowledgeMsg before shipMsg is received.

Finally, a few more rules are needed to specify mutual exclusion.

1. The confirmation and reject payment messages are mutually exclusive.
2. Accepting and rejecting an offer are mutually exclusive.
Results

In the scenario above, the protocol is specified using the Protocol Editor. Since the protocol is specified from a global perspective we use the Role Skeleton Generator to generate role skeleton, which is a set of rules from the role’s perspective.

The role skeleton generated for bank is shown in Figure 4.1. The rules of the bank’s role skeleton are as follows:

1. If requestPaymentMsg is not received, then cardPaymentMsg cannot be sent.
2. If requestPaymentMsg is not received, then rejectPayReqMsg cannot be sent.
3. If requestPaymentMsg is received, then either rejectPayReqMsg should not be sent or cardPaymentMsg should not be sent.
4. If cardPaymentMsg is not sent, then confirmPaymentMsg should not be sent.

The role skeleton generated for the shipper is shown in Figure 4.3. The rules of the shipper are as follows:

1. If shipRequestMsg is not received, then shipmentMsg is not received.
2. If shipmentMsg is not sent and acknowledgeMsg is not available, then acknowledgeMsg should not be received.
3. If shipRequestMsg received, then delegate OrderCommitment.
4. If acknowledgeMsg sent, then release orderCommitment.

The state diagram generator generates the possible states of a protocol. The states generated by the state diagram generator are shown in Figure 4.5. It generates all the allowed enactments of a protocol. Since, in a tree, we can trace the path from the root node to the leaf node through the transitions, we give all leaf nodes of the state diagram as a list of transitions from the root node. These are:

1. rfqMsg, offerDetailsMsg, rejectOfferMsg
2. rfqMsg, offerDetailsMsg, requestPaymentMsg, rejectPayReqMsg
3. rfqMsg, offerDetailsMsg, requestPaymentMsg, cardPaymentMsg, shipRequestMsg, shipmentMsg, acknowledgeMsg, confirmPaymentMsg

4. rfqMsg, offerDetailsMsg, requestPaymentMsg, cardPaymentMsg, shipRequestMsg, shipmentMsg, confirmPaymentMsg, acknowledgeMsg

5. rfqMsg, offerDetailsMsg, requestPaymentMsg, cardPaymentMsg, confirmPaymentMsg, shipRequestMsg, shipmentMsg, acknowledgeMsg

6. rfqMsg, offerDetailsMsg, requestPaymentMsg, cardPaymentMsg, confirmPaymentMsg, shipRequestMsg, shipmentMsg, acknowledgeMsg,

In Figure 4.5, for brevity, we use ‘*, message1, message2’. Here, message1 and message2 are messages that act as a transitions. We use ‘*’ to denote a list of all transitions that happened before message1. The order of messages is important. When considering a state, even a slight change in order may lead to a different state than an intended state. But, it has to be noted that, semantically, two states can be same. This can be inferred by comparing the propositions and the commitments of the two states. For example, in the list of states given above, items 5 and 6 represent two different enactments that are semantically the same.

<table>
<thead>
<tr>
<th>Bank</th>
<th>Buyer</th>
<th>Seller</th>
<th>Shipper</th>
</tr>
</thead>
<tbody>
<tr>
<td>requestPaymentMsg</td>
<td>cardPaymentMsg</td>
<td>confirmPaymentMsg</td>
<td>Bank's role skeleton:</td>
</tr>
<tr>
<td>1) If requestPaymentMsg is not received, then cardPaymentMsg should not be sent</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2) If cardPaymentMsg is not sent, then confirmPaymentMsg should not be sent</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 4.1: Role skeleton of bank.
Buyer’s role skeleton:

1) If offerDetailsMsg has not been received, then requestPaymentMsg cannot be sent.
2) If offerDetailsMsg has not been received, then rejectOfferMsg cannot be sent.
3) If rfqMsg has not been sent and offerDetailsMsg is not available, then offerDetailsMsg cannot be received.
4) If requestPaymentMsg has not been sent and rejectPayReqMsg is not available, then rejectPayReqMsg cannot be received.
5) If requestPaymentMsg has not been sent and confirmPaymentMsg is not available, then confirmPaymentMsg cannot be received.
6) If offerDetailsMsg is received, then either requestPaymentMsg should not be sent or rejectOfferMsg should not be sent.
7) If shipmentMsg is not received, then acknowledgeMsg should not be sent.
8) When acknowledgeMsg is sent, then release orderCommitment.

Figure 4.2: Role skeleton of buyer.
Shipper’s role skeleton:
1) If shipRequestMsg is not received, then shipmentMsg should not received.
2) If shipmentMsg is not sent and acknowledgeMsg is not available, then acknowledgeMsg should not be received.
3) If shipRequestMsg received, then delegate OrderCommitment.

Seller’s role skeleton:
1) If rfqMsg not received, then offerDetailsMsg should not be sent.
2) If offerDetailsMsg has not been sent and rejectOfferMsg is not available, then rejectOfferMsg should not be received.
3) If cardPaymentMsg has not been received, then shipRequestMsg should not be sent.
4) If offerDetailsMsg is sent, then create OrderCommitment.
5) When shipRequestMsg is sent, then release OrderCommitment.
6) If rejectOfferMsg is received, then cancel OrderCommitment.
4.2 Wire Transfer Scenario

Our second evaluation scenario is the wire transfer protocol. Here, there are three roles: customer, originBank and destinationBank.

The customer requests for a wire transfer to the originBank through a message wireReqMsg represented in the rule system as wireReqProp. The originBank would then verify with the destinationBank about the validity of the request by sending wireValidityReqMsg, represented by wireValidityReqProp. The destinationBank would then respond with wireReqValidMsg, represented by wireReqValidProp or wireReqInvalidMsg, represented by wireReqInvalidProp. If wireReqValidMsg is sent by the destinationBank, the originBank responds to the customer with quoteForWireMsg, represented by quoteForWireProp. If wireReqInvalidMsg is sent by the destinationBank, the originBank responds to the customer with wireReqRejectedMsg. The acceptQuoteMsg is sent by the customer to the originBank that creates a commitment with originBank as the creditor, customer as the debtor, wireProp as the antecedent and payToBankProp as the consequent. The originBank then sends wireMsg to the destinationBank, where wireMsg is represented by wireProp. The destinationBank responds with wireSuccessMsg if the wire transfer was successful. In this
Table 4.2: Messages in the money transfer protocol

<table>
<thead>
<tr>
<th>Message</th>
<th>Slots</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>wireReqMsg</td>
<td>amount</td>
<td>customer requests for a wire transfer</td>
</tr>
<tr>
<td>wireValidityReqMsg</td>
<td>amount</td>
<td>The originBank requests the destination-Bank if the transfer can take place</td>
</tr>
<tr>
<td>wireReqValidMsg</td>
<td>amount</td>
<td>The destinationBank responds that the wire request can be fulfilled</td>
</tr>
<tr>
<td>quoteForWireMsg</td>
<td>amount, charges</td>
<td>The originBank responds to the customer that the quote</td>
</tr>
<tr>
<td>wireReqInvalidMsg</td>
<td>amount</td>
<td>The destinationBank responds that the wire request cannot be fulfilled</td>
</tr>
<tr>
<td>acceptQuoteMsg</td>
<td>amount, charges</td>
<td>The customer accepts the quote</td>
</tr>
<tr>
<td>wireReqRejectedMsg</td>
<td>amount</td>
<td>The originBank responds to the customer that the quote</td>
</tr>
<tr>
<td>wireMsg</td>
<td>amount</td>
<td>The originBank wires the money to the destinationBank</td>
</tr>
<tr>
<td>wireSuccessMsg</td>
<td>amount</td>
<td>The destinationBank informs the origin-Bank that the wiring of money was successful</td>
</tr>
<tr>
<td>wireFailureMsg</td>
<td>amount</td>
<td>The destinationBank informs the origin-Bank that the wiring of money was not successful</td>
</tr>
<tr>
<td>moneyWiredMsg</td>
<td>amount</td>
<td>The originBank informs the customer that the wiring of money was successful</td>
</tr>
<tr>
<td>moneyNotWiredMsg</td>
<td>amount</td>
<td>The originBank informs the customer that the wiring of money was not successful</td>
</tr>
</tbody>
</table>

In case, the originBank can send `moneyWiredMsg`, represented by `moneyWiredProp`, to the customer informing that the wire transfer was a success. If the wire transfer was a failure, the destinationBank responds with `wireFailureMsg`, represented as `wireFailureProp`. The originBank can send `moneyNotWiredMsg`, represented by `moneyNotWiredProp`, to the customer informing that the wire transfer was a failure.

For the wire transfer protocol we define a QuoteCommitment, which has a customer as the debtor, originBank as the creditor, wireProp as the antecedent and payToBank as the consequent.

Next, we define the following rules for the commitment operations in the protocol:
1. acceptQuoteMsg creates QuoteCommitment.

2. If acceptQuoteMsg and wireSuccessMsg have occurred, then payToBank is enabled. (This rule helps in commitment discharge.)

   Additionally, the following rules are necessitated by the data flow:

1. wireValidityReqMsg cannot occur before the occurrence of wireReqMsg.
2. wireReqValidMsg and wireReqInvalidMsg cannot occur without wireValidityReqMsg.
3. wireReqValidMsg and wireReqInvalidMsg are mutually exclusive, given wireValidityReqMsg.
4. quoteForWireMsg cannot occur before the occurrence of wireReqValidMsg.
5. wireReqRejectedMsg cannot occur before the occurrence of wireReqInvalidMsg.
6. acceptQuoteMsg can occur only if wireReqValidMsg has occurred.
7. wireMsg can occur only if acceptQuoteMsg has occurred.
8. moneyWiredMsg can occur only if wireSuccessMsg has occurred.
9. wireFailureMsg can occur only if wireMsg has occurred.
10. moneyWiredMsg can occur only if wireSuccessMsg has occurred.
11. moneyNotWiredMsg can occur only if wireFailureMsg has occurred.

Finally, for debiting money from the customer’s account, we need the following condition:

1. If moneyWiredMsg, then payToBank.

Results

As before, we use the protocol editor to specify the protocol. We then use the Role Skeleton Generator to generate the role skeletons described below.

The role skeleton for the customer is shown in Figure 4.6. The rules in the role skeleton are as follows:
1. If quoteForWireMsg is not received, then acceptQuoteMsg should not be sent.
2. If acceptQuoteMsg is sent, then create QuoteCommitment.
3. If moneyWiredMsg is received, then release QuoteCommitment.
4. If moneyNotWiredMsg is received, then release QuoteCommitment.

Figure 4.6: Role skeleton of customer.
The role skeleton for the originBank is shown in Figure 4.7. The rules in the role skeleton are as follows:

1. If wireReqMsg is not received, then wireValidityReqMsg should not be sent.
2. If wireValidityReqMsg is not sent and wireReqValidMsg is not available, then wireReqValidMsg should not be sent.
3. If wireValidityReqMsg is not sent and wireReqInvalidMsg is not available, then wireReqInvalidMsg should not be sent.
4. If wireReqValidMsg is not received, then quoteForWireMsg should not be sent.
5. If wireReqInvalidMsg is not received, then wireReqRejectedMsg should not be sent.
6. If quoteForWireMsg is not sent and acceptQuoteMsg is not available, then acceptQuoteMsg should not be sent.
7. If acceptQuoteMsg is not received, then wireMsg should not be sent.
8. If wireMsg is not sent and wireSuccessMsg is not available, then wireSuccessMsg should not be received.
9. If wireMsg is not sent and wireFailureMsg is not available, then moneyNotWiredMsg should not be received.
10. If wireSuccessMsg is not received, then moneyWiredMsg cannot be sent.
11. If wireFailureMsg is not received, then moneyNotWiredMsg should not sent.
12. If acceptQuoteMsg and wireSuccessMsg are received, then payToBank proposition is enabled.

The role skeleton for the destinationBank is shown in Figure 4.8. The rules in the role skeleton are as follows:

1. If wireValidityReqMsg is not received, then wireReqValidMsg should not be sent.
2. If wireValidityReqMsg is not received, then wireReqInvalidMsg should not be sent.
originBank’s role skeleton:
1) If wireReqMsg is not received, then wireValidityReqMsg should not be sent.
2) If wireValidityReqMsg is not sent and wireReqValidMsg is not available, then wireReqValidMsg should not be sent.
3) If wireValidityReqMsg is not sent and wireReqInvalidMsg is not available, then wireReqInvalidMsg should not be sent.
4) If wireReqValidMsg is not received, then quoteForWireMsg should not be sent.
5) If wireReqInvalidMsg is not received, then wireReqRejectedMsg should not be sent.
6) If quoteForWireMsg is not sent and acceptQuoteMsg is not available, then acceptQuoteMsg should not be sent.
7) If acceptQuoteMsg is not received, then wireMsg should not be sent.
8) If wireMsg is not sent and wireSuccessMsg is not available, then wireSuccessMsg should not be received.
9) If wireMsg is not sent and wireFailureMsg is not available, then moneyNotWiredMsg should not be received.
10) If wireSuccessMsg is not received, then moneyWiredMsg cannot be sent.
11) If wireFailureMsg is not received, then moneyNotWiredMsg should not sent.
12) If acceptQuoteMsg is received and wireSuccessMsg are received, then payToBank proposition is enabled.

Figure 4.7: Role skeleton of origin bank.
3. If wireValidityReqMsg is received, then either wireReqValidMsg is not sent or wireReqInvalidReq is not sent.

4. If wireMsg is not received, then wireSuccessMsg should not be sent.

5. If wireMsg is not received, then wireFailureMsg should not be sent.

6. If wireMsg is received, then either wireSuccessMsg should not be sent or wireFailureMsg should not be sent.

7. If wireSuccessMsg is sent, then payToBank proposition is enabled.

### 4.3 Nonenactable protocols

Let us consider the example described in Section 1.1 for nonlocal choice. There are two roles $\alpha$ and $\beta$. $\alpha$ sends a message $m_1$ and $\beta$ sends a message $m_2$. A rule specifies that $m_1$ and $m_2$ are mutually exclusive. We had discussed that this protocol may lead to incorrect enactment. When our role skeleton is used, it invokes a component called enactability checker. The enactability checker throws an exception: ProtocolNotEnactableException.

Let us consider another example for message verifiability. Let us consider a protocol where there are three roles $\alpha$, $\beta$, and $\gamma$. Let $\alpha$ send a message $m_1$ to $\beta$. Let $\gamma$ send a message $m_2$ to $\beta$. Let us introduce a rule that says, if $m_1$ has not occurred, then $m_2$ should not occur. This specification also throws ProtocolNotEnactableException in our enactability checker.
destinationBank's role skeleton:

1) If wireValidityReqMsg is not received, then wireReqValidMsg should not be sent.
2) If wireValidityReqMsg is not received, then wireReqInvalidMsg should not be sent.
3) If wireValidityReqMsg is received, then either wireReqValidMsg is not sent or wireReqInvalidReq is not sent.
4) If wireMsg is not received, then wireSuccessMsg should not be sent.
5) If wireMsg is not received, then wireFailureMsg should not be sent.
6) If wireMsg is received, then either wireSuccessMsg should not be sent or wireFailureMsg should not be sent.
7) If wireSuccessMsg is sent, then payToBank proposition is enabled.

Figure 4.8: Role skeleton of destination bank.
Figure 4.9: State diagram of the wire transfer protocol.
Chapter 5

Conclusions

A business protocol describes interaction among autonomous participants in terms of their commitments to one another. Commitments support defining and tracking the compliance of the participants. Business protocols can be combined to produce many business interactions. We identify a set of tools that support modelling and using business protocols.

The primary concept for the tool design and implementation is a conceptual model for protocols. We propose a conceptual model that is based on the OWL-P framework [3].

Our tools are based on technical properties such as proves and awareness that reflect relationships between messages. The development of the components that help us check these properties or make inferences based on these properties forms a crucial step. We introduce new algorithms (for Rule Evaluators, Enactability Checker) or refine existing algorithms (for Role Skeleton Extractor).

This thesis describes the approach that we adopted to implement these tools.

5.1 Related Works

Multiagent systems for business process management, which preserve the autonomy of heterogeneous systems, have been around for quiet sometime. Singh [1991] introduced the notion of social commitments (from one agent to another) and distinguished them from psychological commitments (from an agent to itself). The approach of this thesis and protocols generally follow social commitments. Jennings [5] proposes the notion of commitments as the core concept of multiagent system though his notion of commitments is
psychological and not suited to open systems. Singh [4] formalizes a set of legal and deontic concepts using commitments. Desai et al. [3] propose the OWL-P framework that defines the domain model of a protocol. They define the structure of messages, propositions, commitments, commitment operations and rules. They leave a few details, like the format of rules, to the designers in the assumption that the rules would be in accordance to the choice of rule engine. We, instead, adopt an object-oriented approach and, as a consequence, make the protocol specification process independent of the choice of the platform. Rovatsos [6] proposes the idea of identifiers of a commitment. Each commitment has a unique identifier. We adopt a similar approach in our model.

Oryx is an open source web-based editor business process editor [7]. It helps us design a business process flow independent of the choice of business process language. Oryx uses JSON for descriptions of objects, SVG to store information about graphical components and JavaScript to implement special callback functions. The flow is represented in RDF [8], specifically, eRDF triples in the server and can be translated to any language. Our tool performs a similar functionality. It helps us define a protocol independent of the choice of language on which the agent is implemented. We use Eclipse’s in-built controls to design a user interface. During the process of protocol specification, we make modifications to the protocol object. It is then serialized in RDF/XML [8]. When our editor opens a new protocol specification document, it reads the RDF triples and creates a protocol object and displays various attributes of the protocol object. We use Jena’s APIs to serialize the protocol object as well as to read the RDF/XML document to produce a protocol object [9]. Our editor is not server based. Rather, it is a plugin and can be installed in the user’s workstation.

Desai and Singh [2] discuss the enactment of a protocol and propose five properties that ensure correct enactment of a protocol. They also propose an algorithm to extract role skeletons from a protocol specification. The representation of the state is based on the concepts from LISP language.

We design an inference engine that operates on the domain objects that we had defined. Forgy [10] proposes an algorithm (known as the Rete algorithm) that is efficient to the order of number of clauses of a rule. The rule engine JESS [11], [12] implements this algorithm. But, our scenario differs from the assumptions made in the algorithm. The Rete algorithm assumes that the knowledge base almost always remains fixed and undergoes a
small percentage of change every time a new tuple (proposition) is asserted. The algorithm remembers the previous results and makes inferences based only on the modified proposition. Our tools need a rule engine that can make inferences on a set of propositions. They do not have previous executions and the rules need to make inferences on the whole of the knowledge base at a time. Another mismatch is that our tools are implemented on the assumption that a proposition, once enabled, cannot be disabled. But the Rete’s algorithm does not have such assumptions and is designed for generic cases.

5.2 Future Work

Chopra and Singh [13] propose a formalization of commitments to ensure alignment despite asynchronous communication. We currently do not support such an approach. We also do not support protocol design based on the Commitment-Based Service-Oriented Architecture [14]. These topics can be important areas of study in the future.
Bibliography


