Abstract

WAN, FENG. Commitment-Based Business Process Modelling and Enactment. (Under the direction of Professor Munindar P. Singh.)

Business process management faces challenges in dealing with business abnormalities and ever-changing business requirements. Traditional business process management approaches evolved from software engineering and workflow management where activities, messages and control logic are given prominence. The resulting models specify low-level details of execution and coordination. However, difficulties arise when modelling long-lived business transactions involving information updates and execution exceptions. To handle such situations, current approaches implement excessive activities without suitable abstractions, thereby arbitrarily fragmenting the business requirements.

We propose a commitment-based approach for business process modelling that formulates business processes as multiagent systems. Organizational structure and its effect on interactions are described using commitments and causality. Agents act as process executors and maintain the commitments made to each other. Updates and exceptions yield commitment operations under which processes are updated and reexecuted.

Our approach brings commitment semantics into business modelling and enables agent collaboration for business process enactment. We derive commitment protocols from agent conversations and generate agent execution models. We also formalize our approach using the π-calculus and prove its correctness. To demonstrate the practical use of our approach, we formalize multiparty agreements with commitments and present algorithms on how to detect agreement conflicts and build satisfiable commitment sets.

COMMITMENT-BASED BUSINESS PROCESS MODELLING AND ENACTMENT

BY

FENG WAN

A DISSERTATION SUBMITTED TO THE GRADUATE FACULTY OF

NORTH CAROLINA STATE UNIVERSITY

IN PARTIAL FULFILLMENT OF THE

REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

RALEIGH

JANUARY 2005

APPROVED BY:

DR. JAMES C. LESTER

DR. ROBERT ST. AMANT

DR. MUNINDAR P. SINGH

CHAIR OF ADVISORY COMMITTEE

DR. PETER R. WURMAN

Biography

I was born on April 16, 1972 in Beijing, China. Both my parents are professors at Peking University and I naturally became a student of every school attached to the Peking University including kindergarten, elementary, middle, high, undergraduate and graduate. I am very proud that in my high school year, because of my academic excellence, I was chosen into an experimental class which finished the high school education in two years instead of three. After obtaining my Master degree of Computer Science in July, 1997, I started pursuing my PhD degree in North Carolina State University and worked as a research assistant under Dr. Munindar P. Singh.

Due to personal reasons, I joined SimpliCTI Software Solutions, Inc. in January, 2000. However, I continued my graduate studies part time. I was a CTI (Computer Telephony Integration) consultant for SimpliCTI providing professional services for various customers. In five years, I have travelled to most of the states in USA and have worked for quite a few Fortune 500 companies. I have learned many cutting edge technologies and gained useful industrial experience which has helped motivate my PhD work. Due to my excellent work for SimpliCTI, I was promoted to Principal Architect in 2004.

Since year 2002, I started publishing more papers and gradually built up the foundation of my dissertation. I passed my PhD defense in November, 2004 and expect to graduate in May 2005.

Acknowledgments

I want to first thank my parents for their spiritual encouragement and support. Being university professors, they influenced me to always pursue higher academic achievement. I would also like to thank my wife, Hong, for her making the other half of my life meaningful.

My great thanks go to Dr. Munindar P. Singh, my advisor, mentor and good friend. I have learned so much from him on how to read and write academically, how to study and master my field of study, and how to improve and refine myself. I feel so lucky as being one of his students and am grateful for whatever he has done to me.

I also thank my committee members, Dr. Lester, Dr. St. Amant, and Dr. Wurman, for their valuable comments on my dissertation and enlightenment to me in many ways. To my school colleagues, particularly, Sudhir Rustogi, Jie Xing, Ashok Mallya, Amit Chopra, and Nirmit Desai, for their constructive discussions and comments which helped my work more solid.

And last, I want to thank SimpliCTI, the company I have been working for these years, for their financial support for me to pursue the Ph.D. Besides, the industrial experience that SimpliCTI provides me has helped motivate my research in many ways.

Table of Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

One of the main goals of today's eBusiness applications is to help companies produce value by improving customer and business partner satisfaction. To accomplish this goal, besides requirements developed by managers, a sophisticated and reasonable business process design plays an important role. A good design need not only specify domain-level business processes, but must also deal with the abnormalities arising around those processes. These abnormalities can be described as below.

Exception handling. Dealing with exceptions is one of the most common functions of process management. For example, in a travel system, a customer may cancel his trip, an airline may cancel or delay its flight, a hotel may over-sell its rooms. Although, there may be a low chance that an exception would happen, a good design must cover every one of them to avoid system failure. Researchers and practitioners are seeking efficient representations of exceptions to reduce the burden of exception handling.

Process revision. This often happens in long-lived transactions where some of the participants may re-execute their processes before the entire transaction ends. For example,

in a travel booking system, upon receiving a trip booking request from a customer, a travel agent sends individual requests to airlines, hotels, and car rental companies. However, before those agents send confirmations back to the travel agent, the customer may want to change his flight time or hotel location, which would trigger another request to be sent to his travel agent. This is allowed in travel booking systems, although different designs may treat it differently. While some systems require the travel agent wait for the results to be sent back from the airline and hotel agents before sending the customer's change request, others require that the travel agent cancel the corresponding initial request and send the change request as a fresh request afterwards. Most existing approaches treat each case as independent processes, ignoring the relationship among them, which makes the designs very inefficient and not expandable.

Alternative execution paths. Different execution paths may offer different value to a company. For example, for a process to sell software, a vendor may ask the customers to pay first before shipping CDs or authorizing downloads. However, customers may be reluctant to pay before making sure that the goods are acceptable. The customer's uncertainty can potentially affect sales. An alternative sales path could be letting the customer try out the software for a period of time before buying it. This path may produce both good and bad results: if the software is good, then many customers will buy; otherwise, few will buy. Another example could be that to repair a defective product, a company may require a customer to send back the defective product first before shipping a replacement; an alternative path could be shipping a replacement prior to receiving the defective one. This will speed up the availability of the product to the customer, which in turn can increase the customer's satisfaction. In other

words, companies will select a strategy to bring about the best value. A good process design should allow different execution paths under the same process to realize different values.

1.1 Challenges to Today's eBusiness Applications

The potential for abnormalities in business process execution poses major challenges for today's eBusiness design methodology. These challenges are described below.

Maintain long-lived transactions. Long-lived transactions are usually composite activities that can potentially last days or weeks. During such period, exceptions and process updates may occur within subactivities. The so-called ACID properties (atomicity, consistency, integrity and durability), which are essential for maintaining database integrity in conventional settings, of individual transactions, may not hold [Kaye, 2003]. Many existing approaches have developed specific ways to deal with exceptions such as executing compensation tasks, re-executing the failed tasks, or rolling back to the last checkpoint [BPEL, 2003]. However, these methods are highly task-specific, which makes their models quite complex. What we need is to declaratively specify task dependencies so that the exception handling processes can be automatically generated by participants instead of being statically specified in the design flow.

In addition, during process execution, each participant would receive reliable results from a predecessor or obtain satisfactory responses from the party from which it requested services. This requires all the participants to make promises on what they are delivering to others and to keep these promises until the entire transaction ends.

From this point of view, the participants' promises should become manipulable objects and be controlled by participants themselves to ensure a successful execution of the long-lived transactions.

Allow autonomy among business partners. eBusiness partners are autonomous entities that engage in business to promote their own interests, such as reducing operational costs or deriving maximum value out of a transaction. To maximize autonomy, participants may change their intermediate goals at runtime, resulting in different execution paths. This presupposes that the models should allow autonomy to the business partners so that they can flexibly choose different paths or even initiate new requests in the middle of the execution. However, existing approaches tend to impose static flows on the process execution and treat participants merely as task executors instead of as self-motivated agents. What kinds of specifications will enable autonomous execution is an interesting topic for research.

Another challenge of allowing autonomy is to enable interactions among the participants. These interactions include negotiations between producers and consumers to handle process updates and exceptions. Today's business processes exchange messages by following predefined protocols, which on the one hand ensure a proper execution sequence, but on the other hand restrict the autonomy of each participant so that they are not able to freely talk to each other to negotiate a best action on the fly. We should allow conversations among business participants during the enactment of their business logic. The conversations could potentially handle updates and exceptions locally and would not compromise the efficiency and the integrity of the entire execution.

Accommodate ever changing business requirements. Nowadays companies keep refining their business requirements to increase business process efficiency and robustness as well as to bring more value to their customers. The requirement changes often lead to design changes. Industrial practitioners have devised a few approaches to minimize the potential impact, for example, building adaptable workflow systems [Buhler and Vidal, 2005]. However, few approaches are capable of reducing the effort dealing with the exceptions introduced by the new requirements since the new exception handling steps usually complicate the existing design. What we need is a modular model in which new interactions can be easily added with exception handling automatically incorporated.

Facilitate building business agreements and detect conflicts. A business agreement is a set of domain-level rules that specify the dependencies and constraints among business participants. In a multiagent environment, these rules not only reflect the common regulations that all the participants must follow, but also the local policies specific to each individual party. Since most participants are self-interested, they may express their own rules as prerequisites for doing business with others. These rules could be imposed before or during business engagements, so we need a flexible specification language to manipulate them on the fly. We also need a way to determine the satisfiability of the rule set so that agreement conflicts, which prevent business activities from progressing, can be prevented. Most existing approaches tend to specify business agreements using workflows or FSM (Finite State Machine) in which rules are directly converted to the execution dependencies. However, keeping rules as manipulable entities is the key to maximizing execution flexibility.

1.2 Existing Approaches

Various modelling approaches have been used to specify eBusiness systems. These range from statecharts [Harel et al., 1987], which specify transitions within a standalone system, to workflows which coordinate multiple business activities [WfMC], to Web services standards which build the interfaces among communicating parties across the Web [W3C] [WSDL, 2002], and finally to the business process modelling languages which orchestrate participants to successfully do business with each other [BPEL, 2003] [Cabrera et al., 2003] [Cabrera et al., 2002]. However, these approaches yield complicated specifications when facing dynamically changing business requirements. In particular, the effort of dealing with autonomy and system abnormalities grows rapidly as the requirements change and grow. Below we describe the limitations of some existing models.

FSM-Based Models. An FSM-based model specifies how a system transitions from one state to the next, and what actions to perform upon receiving any internal or external events. This approach is useful for building a standalone sequential system, but has difficulty in specifying the interactions among multiple parties because of the exponential complexity of a global FSM in which all the participants execute.

In addition, an FSM-based model must cover every event and action to prevent its execution from breaking down. The majority of such events and actions are exceptions and their corresponding handlers. This inevitably imposes a heavy burden on designing a valid system whose requirements are changing.

The limitations of FSM-based models render them not appropriate to specify a complex distributed system where multiple parties interact autonomously.

Figure 1.1 shows an FSM for the flight ticket purchasing process in which tickets can be either available or not available, and after the tickets are confirmed, they can be

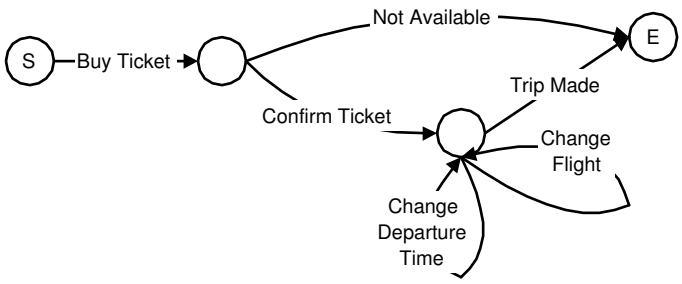

Figure 1.1: FSM Example

used, changed or cancelled.

Sequence Diagram. Sequence diagrams are often used to specify individual business scenarios where events and actions happen in a sequential order. You can think of each scenario as one transition path in an FSM. Different transition paths would be represented by different sequential diagrams. This approach is good at reflecting potential causality within a business process, since it denotes the order in which events happen. However, it is not effective for specifying concurrent processes and branches for exception handling and transaction updates. Figure 1.2 shows a sequence diagram for the same example as above.

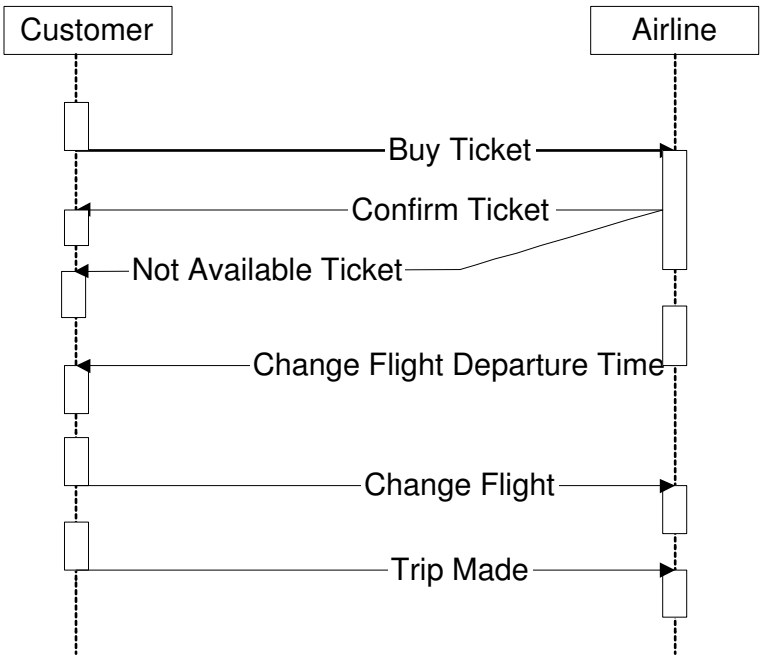

Figure 1.2: Sequence Diagram Example

Workflows. Workflow systems focus on task execution and control coordination. Unlike FSM-based systems, workflows do not model the states of a system, but validate the result of a task execution and how to chain a task's output to another next task's input. A workflow system is more useful to specify a pipeline system where a global task is processed piece by piece among consecutive task executors. Figure 1.3 shows how the previous example is designed using workflows.

Although there has been research on adaptive workflow systems [Buhler and Vidal, 2005], the focus on executing tasks makes workflow systems weak at specifying the behaviors of the task executors. To illustrate this, on the one hand, we can verify if a participant behaves properly only by his success in executing a task. However if he fails and if there may be several causes to the failures, it is difficult for a workflow system to itemize the causes and design a recovery execution path for each one.

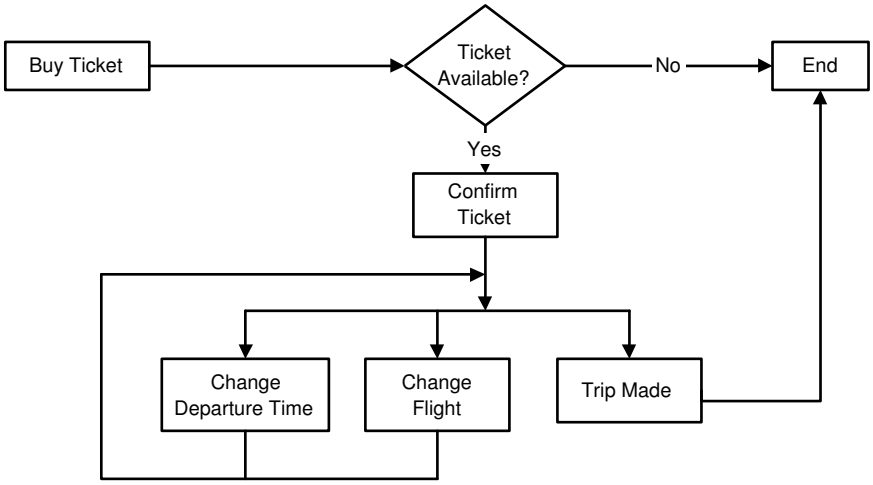

Figure 1.3: Workflow Example

On the other hand, in an eBusiness system, the participants are not the same type of workers within a given organization; they are different types of business entities and have their own interests for doing business. We cannot expect all the participants to follow a predefined execution path without giving them autonomy to choose when, where, and how to execute these tasks. In the extreme, the participants can even spawn new requests to trigger additional task executions. Therefore, workflow systems are not flexible and expressive enough to specify today's eBusiness applications.

Business Processes. Business processes automate transactions occurring in intrabusiness, business to business (B2B) and business to customer (B2C) environments. Business process management has evolved from workflow systems but with more powerful process semantics and reusable transactional protocols. Process build on the top of web services, which enables them to apply smoothly cross heterogeneous systems on the Web. Several standards have been proposed and are being introduced into practice, especially BPEL4WS (Business Process Execution Language for Web Services

[BPEL, 2003]), BPML (Business Process Modelling Language [BPML]), and WSCI (Web Service Choreography Interface [WSCI, 2002]).

Coordination and Transactions. WS-Coordination specifies the transactional protocols used in business process models. It generally involves a coordinator to handle protocol registration and enactment. Business participants must first register with a coordinator before interacting with other business partners. Built on the top of WS-Coordination, WS-Transaction specifies various type of transactional protocols. These protocols are divided into two categories, atomic transaction and business activity, which will be described in detail in Chapter 2.

1.3 Proposed Approach

Specifying detailed transactional protocols is good at ensuring a reliable process executions, however, it does not allow protocol flexibility and participants' autonomy. In other words, the process execution paths are statically specified at design phase, and participants cannot change the courses of interaction due to business abnormalities arisen at runtime. The problem here is that the business process specification still stay at lower level process coordination and message passing, and there is no higher level abstraction on how these processes and messages relate to the domain-level business requirements and obligational relations among participants.

By obligational relations, we mean each business participant, no matter if they are service requesters, providers or process executors, when they request or deliver services, they form underlying obligations to each other to produce and maintain valid outputs until the given transaction ends. For example, if a customer cancels a confirmed trip, he may need to pay his travel agent a penalty charge; or if a confirmed flight is cancelled, an airline

needs to find another available flight or refund the ticket. All these process are just specific action sequences when participants fulfill their obligations. The transactional protocols can handle them, but in a rigid manner within a narrow scope. In our view, capturing obligations among participants in business process design can yield a rich variety of interactions which not only generate flexible executions, but also maximizing participants' autonomy which may produce more value out of business engagements.

Based on the limitations of today's modelling approaches to meet the above challenges, we require a concept that supports process updates, on-the-fly conversations, and obligation fulfillments. We introduce *agents* and *commitments* from multi-agent theory into business process modelling.

1.3.1 Agents and Multiagent Systems

Agents have been widely studied as intelligent software entities that can perceive, reason and act proactively. Agents are designed to be autonomous and heterogeneous. By autonomy, we mean an agent is allowed to make its own decisions on what execution path it takes and what messages it sends. By heterogeneity, we mean an agent is not restricted by the environment in which it resides and the way it behaves in order to communicate with others. These two features distinguish agents from other software entities and help model a flexibly collaborative system.

When agents interact with each other to achieve their local or global goals, they form a multiagent system. The field of multiagent systems combines agents and distributed systems in which agents talk in agent communication languages and interact with each other via various protocols, such as ContractNet, Fish Market, and auction in general. A multiagent system abstracts a distributed system into a higher level so that the autonomy of various parties is preserved and the protocols can be made flexible and adaptive.

Commitments (also known as social commitments) are widely recognized as a key representation for the interactions in a multiagent system. A commitment is an abstraction of an obligation made by a debtor to a creditor. Agent interactions can be naturally understood in terms of the commitments the agents make, fulfill, cancel, or modify. Once a commitment is created, it ties its debtor and creditor agents together until it is fulfilled. The lifecycle of a commitment captures an interaction history between two agents. Also by analyzing the relations among commitments, we can capture the behaviors of a group of agents. Commitments enable us to model and analyze the behavior of autonomous agents and help establish agreements among these agents.

By introducing commitments into business process modelling, we endow process participants with autonomy and heterogeneity as would exist in a pure multiagent system. Our approach shifts the focus of system design from coordination and messaging protocols to commitment causality relations and lets the participants decide how to manipulate these commitments. In such a way, the participants can proactively accomplish their local and global tasks and flexibly handle task revisions and exceptions.

Figure 1.4 shows how we describe the previous example process using commitments. We use two commitments, one made from the customer to the airline, and the other made from the airline to the customer. All the actions or events seen in the previous approaches are generated by commitment operations and its lifecycle state machine. The dependencies among these actions and events are specified by a Commitment Causality Diagram (CCD), which is described in later chapters.

1.3.2 Benefits of Commitments

A commitment-based system has the following benefits over a traditional system.

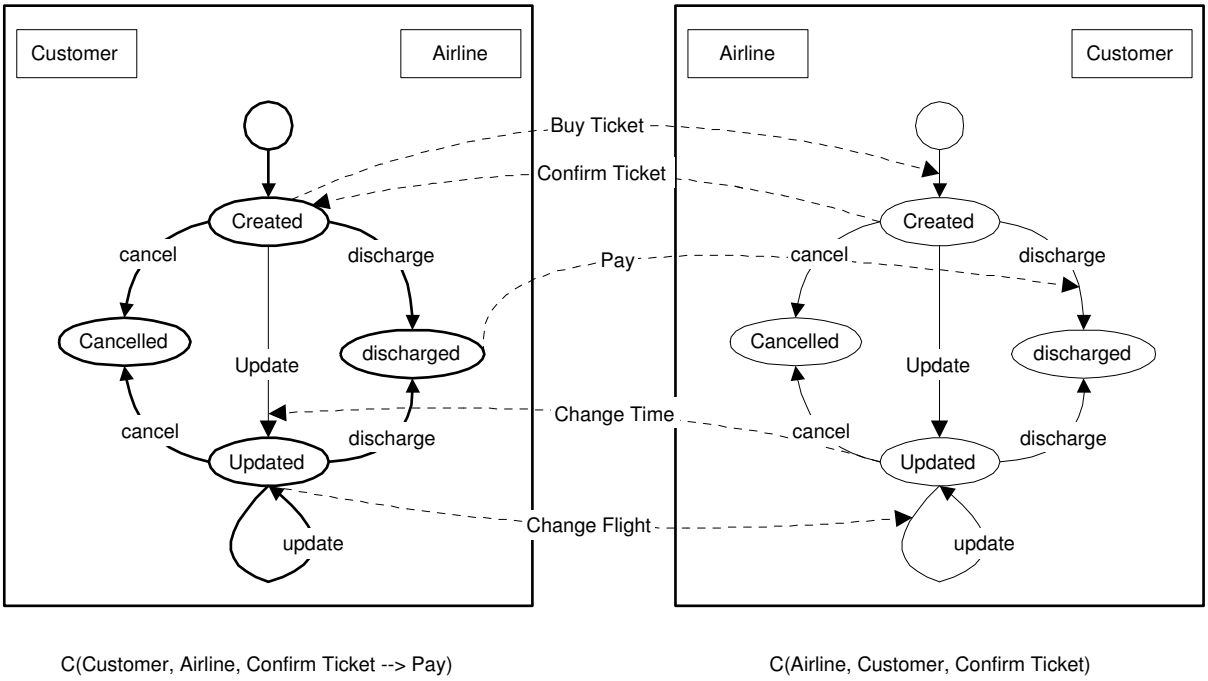

Figure 1.4: Commitment Example

Ensure validity and persistence across long-lived transactions. Parties in traditional systems send messages to each other, which cause changes on the data. The impact of a message is immediate and explicit. If the result needs to be updated, a separate message must be sent. Also, if the result is invalid or not satisfied by some of the parties, it is considered an exception. To bring a system back from an exception to a consistent state, either the sending parties need to send compensation messages or the receiving parties need to execute a rollback operation. All these additional messages or operations are separated from the original messages and need a complex protocol to govern their occurrence. In this paradigm, each party heavily relies on the protocol to exchange messages that are related to the same transaction. The disadvantage of this approach is obvious, because the more business logic there is, the more complex the protocol becomes.

Commitments can help since they can simplifies delivering consistency, satisfaction, and trust. If an agent sends a message to another agent to change some data, the former tells the latter that he will make sure the data will be eventually changed and the change will be correct and consistent. If it is not, the former will repeatedly deliver an update until the latter gets satisfied, or the former may change the commitment but still achieve the same goal agreed upon by the latter. In this paradigm, the above messages can be grouped into the same commitment. Each message represents a different stage of the commitment. The first message could be the creation of the commitment and each update or exception handling message represents an update of the commitment. If the commitment eventually satisfies the creditors, then the commitment will be discharged. The commitment can also be cancelled or released because the first agent wants to revoke the message he sends to the second agent.

In this manner, commitments essentially put threads of related messages and pieces of protocols together, which make the system interaction more expressive in terms of the business logic that a designer is implementing.

Preserve Autonomy and Heterogeneity. By communicating using commitments, agents tell each other what, but not how. Agents expose their commitments but not their behaviors. Agent-oriented programming differ from the object-oriented paradigm, which can also hide implementation details but has hard-coded interfaces. Agents can freely express themselves by commitments that can potentially change protocol execution and flexibly react to any unexpected events. By doing this, we preserve agent autonomy and heterogeneity. In other words, any party could potentially talk to another without a complex and rigid protocol being imposed.

Manipulate requirements and goals at high level. When designing or implementing a

multiagent system, we mainly deal with the local and global goals. A commitment can be considered as an operational element that contains the goals to achieve, the preconditions and postconditions to satisfy these goals, as well as the reference to related parties. By manipulating commitments, we could potentially eliminate the need to specify a message-oriented protocol. In other words, the message causality can be extracted into commitments, making the actual protocol execution requirement-driven and controlled by the lifecycle of the commitments involved. The runtime message sequence could vary depending on how the commitments evolve. In this way, the hurdle of handling exceptions and revisions can be overcome and the autonomy of agents can be maximized to achieve flexible implementations and interactions.

Capture relations in conversations. Commitments not only capture message causality but also expose the relationships among conversations. As described in Chapter 2, a conversation is a group of messages occurring between two agents. One conversation may involve several commitments between the two agents and the executions of the commitment operations represent the significant events that occur in this conversation. Deriving commitments inside a conversation can help identify the major goals that the two agents intend to achieve. Also by studying the context-related conversations we can detect the relationships among commitments and sketch out a business requirement flow. This helps us design a system in two directions. First, based on a business requirement, come up with a commitment causality diagram and further map it out to a message sequence diagram and agent models. Second, starting from a use case or an example conversation table, derive conversations, commitments and commitment causality diagram, then fine-tune the interaction model.

1.3.3 Applying Commitments

Based on the foregoing, the main idea of this research is to analyze agent interactions in terms of causally linked commitments, and to develop a methodology (with significant algorithmic components) using which commitment-based agents can be inferred. The advantage of doing so is to produce richly structured multiagent systems whose members interact flexibly.

Our main research focuses on how to incorporate commitments semantics into existing business process models so that business process specifications become requirement driven. The protocols become less restrictive and allow conversations among producer agents and consumer agents. The types of business applications that our approach applies to are:

- Long-lived and persistent business activities.
- Input can be updated and results can be reproduced.
- Business constraints are dynamically formed.
- Participants have allowed autonomy.
- Negotiation for optimal results is required.
- Compliance validation is required.

The philosophy behind our approach is summarized as follows:

- During design phase: **commitments + causality = business agreements**
- At run time: **commitment lifecycle + connectors = execution sequences**
  where the commitment lifecycle results in both domain-level processes and exception handling; and connectors are the runtime coordination logic, which follow the causality rules.

We propose a new modelling diagram called Commitment Causality Diagram (CCD). The CCD mainly specifies domain-level business requirements and only shows causal relations and the transitions where real domain value arises. At the ends of transitions are

the commitment operations, which manipulate commitments and execute underlying commitment protocols. We study how to obtain the CCD in two ways, from low-level agent conversations and from business requirement specifications. Also from the base technology, we convert one of the existing business process models into our CCD and prove that the conversion not only preserves the original coordination requirements but also enables conversations among agents.

Our model is complementary to existing industrial approaches and gives designers another view of business process. We add commitment objects and conversation structures to the existing business process specification so that each business process has its associated commitments along with the conversations that manipulate these commitments. All participants involved in a business process must agree on maintaining these commitments but are allowed to flexibly use any of the corresponding conversations. For example, a cancellation of a flight from an airline may result in two possible conversations. One, the travel agent may seek a new flight or a new airline without inquiring his customer. Two, the travel agent may consult with the customer if the latter wants to cancel the trip or seek an alternative flight or airline. Based on the travel agent's flexibility and policies, either conversation could be used but the final goal should be the same, that is the travel agent keeps his commitment to the customer to buy a ticket.

We derive agent models from a CCD. An agent model provides a programming framework for building an agent infrastructure. The internal states of an agent are commitment-based. The states transition only when there are operations occurring on the commitments. Along the transitions are conversations via which an agent communicate with others. These conversations can be implemented flexibly to achieve an effective business process execution. The new agent framework enables agents to make decisions at the level of commitment instead of at the level of messages or events. It also hides the implementation details

to achieve agent autonomy. Thus, an agent designer will put more focus on the business requirements and agreements instead of on coordination protocols.

1.4 Contributions

The main questions addressed by this research are the following.

- What is the basis for grouping multiple processes under the same commitment?
- How are long lived transactions represented under commitment semantics?
- How can we prove that the proposed approach is correct?

The following summarizes our main contributions:

1. **Engineering Methodology**
   (a) Representing agent interactions using CCD.
   (b) Designing CCDs based on business agreements.
   (c) Generating agent execution models from CCDs.
   (d) Proving the correctness of CCD.
2. **Formalization**
   (a) Expressing the proposed model using the π-calculus.
   (b) Deriving system behaviors.
   (c) Extending business logic.
   (d) Verifying model consistency.
3. **Applying the Approach**

(a) Incorporating the model into BPEL-like specifications.

(b) Formulating multiparty agreements using commitments.

(c) Detecting and resolving agreement deadlocks.

1.5 Outline of this Dissertation

We first present the technical framework associated with our approach in Chapter 2, which includes the definitions of important concepts including Commitment, Causality, Conversation, Connectors and Dooley Graphs. Chapter 3 presents engineering methodologies to generate commitments, CCDs, and agent models from agent conversations and Dooley Graphs. This includes describing the corresponding algorithms and their complexity. Chapter 4 formalizes our approach by mapping the CCDs into the π-calculus and illustrating how the formalization helps derive some useful properties and verifies the models. To prove the applicability of our approach, Chapter 5 shows how we use commitments to facilitate building a multiparty agreement and resolving agreement conflicts. Chapter 6 presents a broad set of literature, a summary of our contributions, and some important future directions.

Chapter 2

Technical Framework

This chapter describes the technical background behind our proposed methodology and theory. It includes the concepts of commitment, speech acts, agent conversations and Dooley Graphs. We illustrate these concepts by demonstrating a running example, which also forms the study domain of our methodology and the theories presented in the later chapters.

2.1 Commitments

Commitments are a key element of the semantics of agent communications [Singh, 2000a]. Recent work operationalizes commitments. Fornara and Colombetti develop an operational semantics that models the lifecycle of a commitment [Fornara and Colombetti, 2002]. Economou *et al.* show how deontic states and commitments can be detected from agents' finite state machines and how communications rely upon the protocols as executed by each agent [Economou et al., 2001]. Yolum and Singh show how protocols among agents can be encoded as commitment machines and automatically executed [Yolum and Singh, 2002].

2.1.1 Definitions

A commitment is an obligation from a debtor to a creditor about a particular condition. For debtor x, creditor y, and condition p, the relevant commitment is notated $\mathsf{C}(x, y, p)$.

- *Unconditional commitment.* A commitment whose condition is a simple proposition. If the condition specifies an action happening at once, then we called it an immediate commitment.
- *Conditional commitment.* A commitment of the form $\mathsf{C}(x, y, e \rightarrow p)$, where e is a condition (possibly interpreted as an event) and p is a condition to be brought about (possibly interpreted as an action). Here p is activated when e becomes true.

Any unconditional commitment that has a temporal proposition specifying an action happening in the future or a conditional commitment, are called non-immediate commitments.

2.1.2 Commitment Operations and Lifecycle

Commitments support the following operations and combine to capture mutual and multi-party scenarios.

- Creation, Crt(C(x,y,p)). Agent X commits to Y regarding P first time. This is an initial state of a commitment. It puts both agents into a significant local state since from then on, agent X should eventually fulfill what he commits and agent Y needs to monitor and verify whether the former has accomplished what he promises or not.

  If the commitment is immediate, then agent X fulfill the commitment at the

time he creates it and both parties will be released from this obligatory relation. However, if P is a non-immediate commitment, both the parties will keep track of the evolution of the commitment within a reasonable time range or before the end of the conversation, at which moment whether the commitment is fulfilled or violated will be judged.

- Update, Upd(C(x,y,p)). Agent X updates a commitment made to Y. This operation only applies to a non-immediate commitment. It occurs when something arising from X's internal or external world causes X intentionally to change the conditions or actions in P. If the changes are agreed upon by both parties and do not violate X's original promise to Y in nature, the commitment will continue to be held; otherwise Y will reject the operation which may result in the cancellation of the original commitment.

  This operation is an enhancement to previous research [Xing et al., 2001], in which to make such changes, a new commitment has to be created and the old one has to be cancelled, which breaks otherwise linked transactions into independent ones and makes it difficult to keep track of the ongoing interaction between two agents.

- Discharge, Dcg(C(x,y,p)). Agent X has fulfilled the commitment he makes to Y. In the other words, he satisfies the condition P. This operation marks the end of the commitment and releases the obligatory relation between the two agents. For an immediate commitment, its discharge is implied in its creation operation. Some non-immediate commitments can be fulfilled within a limited time range, like actions involved in B2B transactions, but other commitments may specify certain invariant properties that the debtor has to maintain forever. Such commitments will not be discharged buy may be cancelled or released.

- Cancel, Cnl(C(x,y,p)). Agent X cancels the commitment he makes to Y because he is not able to or he is not allowed to fulfill the commitment. This may happen when agent X's internal system breaks down or someone else does not fulfill its commitments to X. The cancellation usually incurs penalties on X or a corresponding compensation action from X, such as creating a new commitment to Y. This operation reflects the exceptions raised during agent interactions. To identify the chain of effects it makes on the other agents or the whole system is one of the interesting topics in this research.
- Release, Rls(C(x,y,p)). The commitment is released without being discharged. This could happen when the fulfillment of the commitment is not relevant to the ongoing business transactions any more. Both creditors and debtors do not depend on this commitment for to perform future actions. It is different from discharge and cancel since it never fulfill or violate the underlying agreement.
- Delegate, Dlg(C(x,y,z,p)). The creditor of the commitment has been changed from y to z. For example, a customer may transfer his flight ticket to his friend so that the travel agent needs to ensure his friend gets the ticket.
- Assign, Asn(C,(x,y,z,p)). The debtor of the commitment has been changed from x to z. For example, an airline may cancel a flight due to engine problem, then they could transfer their passengers to their partner airlines.

Of all the commitment operations, we utilize operation Create, Update, Discharge and Cancel, to depict the lifecycle of a commitment. These four operations can represent the common business process interoperation scenarios.

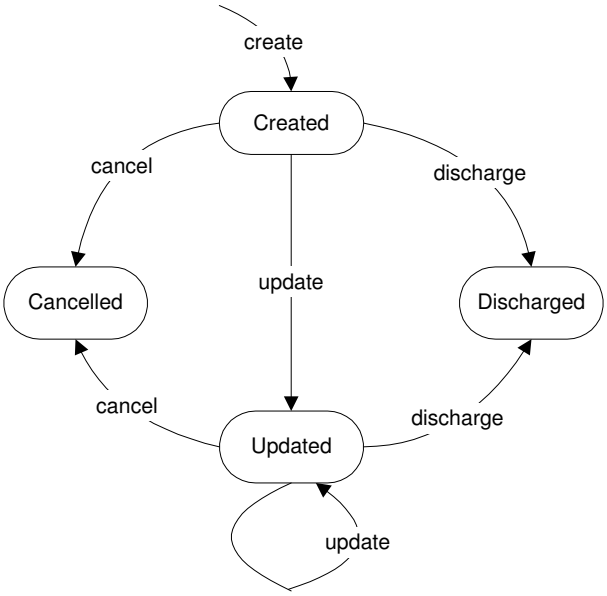

Figure 2.1: Commitment lifecycle

2.1.3 Commitment Patterns

Our previous research [Xing et al., 2001] has studied the combination patterns of the above commitment operations. It generalizes the following scenarios in which agents manipulate commitments to achieve the maximum autonomy and heterogeneity. Agent behavior models are also generated to accommodate these patterns.

- Entertain Request: A consumer role will accept requests from another role and perform its task about the request.
- Notify the Consumer: This comes into effect when a producer role has just completed its tasks for the first time. It informs another role of the computed data values, and becomes committed to the specified predicates.
- Entertain the Update: This comes into effect when a producer role will accept requests to correct some data values that another role may supply. It can abort the update, accept the correct request, or accept the reject request from its

consumer.

- Renotify Consumer: This comes into effect when a producer role has just completed its tasks for the second or a later iteration, and some of its existing commitments are violated by the recently completed task. The violation would typically occur because the predicates to which the agent had committed have been falsified by the results just obtained. The producer role will send the new results to its consumer.
- Dissatisfy the Consumer: This comes into effect when a consumer role is dissatisfied with the results. It sends a reject request to its producer role.
- Abort from the Producer: A producer role may cancel its commitment by sending an abort message to the other role.

2.2 Agent Interactions and Dooley Graphs

Dooley graphs segment agent interactions into courses of conversations [Parunak, 1996]. Each independent conversation reflects a fragment of agents' models and shows how agents behave in different stages of their interactions. Singh showed how to generate coordination requirements based on the agent skeletons created from a Dooley graph [Singh, 2000b].

2.2.1 Speech Acts

The starting point for design with Dooley graphs is to analyze an interaction by classifying the messages exchanged (communicative acts) in it and tagging the key relationships among the acts [Smith and Cohen, 1995]. Parunak proposed the acts Request, Refuse, Commit, Question, Inform, ACT, to which we add Cancel. ACT refers to an action that is external to the commitments, e.g., as needed to discharge a particular commitment. Cancel

simply enables a requester or a requestee to cancel his request or commitment, respectively. We also treat Request as a conditional commitment (defined in Section 3.1) made by a requester.

Parunak defined four relationships among pairs of utterances, namely, respond, reply, resolve, and complete. We add a fifth relationship, termed update. The following is the complete set of possible relationships between message u_i and u_j (here S_i and R_i are the sender and receiver of u_i):

- u_i responds to u_j iff S_i receives u_j and u_j causes S_i to send u_i. This simply denotes a causal relation and indicates that messages must follow the given order.
- u_i replies to u_j iff u_i responds to u_j and $R_i = S_j$. This tells us that the receiver of the Reply message is also the sender of the message to which it responds. It does not express more meaning but helps identify the characters in the conversations.
- u_i resolves u_j iff u_i replies to u_j and u_i is an Inform, Refuse, Commit, or an ACT. This means that something significant has occurred to the conversation initiator because it informs, commits, refuses, or acts. It is implied that u_i follows the "rules of engagement" defined in u_j.
- u_i completes u_j iff u_j is a Commit or a Request and u_i either cancels or fulfills the commitment made in the Commit or Request message.
- u_i updates u_j iff u_j is a Commit or a Request and u_i updates the commitment made in the Commit or Request message. One "Updates" may update another "Updates" message.

Although unconditional commitments can be directly obtained from Commit utterances,

the above five relations help identify conditional commitments (Requests) and causal relations among them.

2.2.2 Example Conversation Table

As our running example, consider a travel planning scenario. A customer (or passenger P) calls his travel agent (T) to book a trip. He makes a commitment that if T books the trip for him, he will pay for the trip as well as any processing costs. Upon receiving the order, T sends requests to airline (A), hotel (H), and car rental (R) agents to reserve air tickets, hotels, and cars, respectively. T also makes commitments that if A, H, R accept his requests and if P purchases the trip, then he will pay them. If A finds an available flight, he will make a commitment to reserve it. So will H and R, if they have available spots. Eventually, if all goes well, T will make a commitment to P to confirm the trip. However, P may cancel all or part of the trip. For instance, P may cancel the car rental if he will get a ride with a colleague. In this case, P updates his request; that is, he updates his original commitment. This update may cause T to update and cancel some of his commitments.

The example shows that any of the participants involved in a business process can initiate updates, which can cause a chained effect among other participants. Table 2.1 is the conversation table of this example. Note that both utterance 11 and 12 involve cancelling the car rental. However, utterance 11 intends to update the trip itinerary and still keep the conversation going, but utterance 12 intends to terminate the entire conversation. By introducing the Cancel act, we can precisely captures the boundaries of conversations in the scenarios where revisions or exceptions occur.

#	S	R	Act Type	Utterance	Respond to	Reply to	Resolve	Complete	Update
1	P	T	REQUEST	Book trip					
2	T	A_1	REQUEST	Buy ticket	1				
3	T	H_1	REQUEST	Reserve hotel	1				
4	A_1	T	REFUSE	Not Available	2	2	2		
5	T	A_2	REQUEST	Buy Ticket	4				
6	A_2	T	COMMIT	Confirm Ticket	5	5	5		
7	T	R	REQUEST	Rent car	1				
8	H_1	T	COMMIT	Confirm Hotel	3	3	3		
9	R	T	COMMIT	Confirm Car	7	7	7		
10	T	P	COMMIT	Send Itinerary	6, 8, 9	1	1		
11	P	T	REQUEST	Remove Car					1
12	T	R	CANCEL	Cancel Car	11			7	
13	T	P	COMMIT	Update Itinerary	11	11	11		10
14	H_1	T	CANCEL	Cancel Hotel				8	
15	T	P	QUESTION	Alternate Hotel?	14				
16	P	T	INFORM	Yes	15	15	15		
17	T	H_2	REQUEST	Reserve Hotel	16				
18	H_2	T	COMMIT	Confirm Hotel	17	17	17		
19	T	P	COMMIT	Update Itinerary	18	16			13
20	P	T	ACT	Pay for the trip	19	19		1	
21	T	A_2	ACT	Pay for the ticket	6, 20	6		5	

Table 2.1: Possible annotated interactions in the trip planning example

2.2.3 Dooley Graph Elements

The following defines the elements used in Dooley Graphs.

- *Characters*. These are the vertices in a Dooley graph. They represent the individual entities at different stages of the modelled interaction. In our example, the list of characters is $\{\mathtt{P}_1\text{–}\mathtt{P}_3, \mathtt{T}_1\text{–}\mathtt{T}_8, \mathtt{A}_1\text{–}\mathtt{A}_2, \mathtt{H}_1\text{–}\mathtt{H}_2, \mathtt{R}\}$
- *Conversation*. A sequence of utterances between two characters derived from a Dooley Graph. A character can only participate in one conversation. In Figure 2.2, the set of conversations is $\{\chi_1=\{1,10,20\}, \chi_2=\{2,4\}, \chi_3=\{5,6,21\}, \chi_4=\{3,8,14\}, \chi_5=\{7,9,12\}, \chi_6=\{11,13\}, \chi_7=\{15,16,19\}, \chi_8=\{17,18\}\}$.
- *Conversation initiator*. The sender of the first utterance in a conversation; e.g., the conversation initiator of χ_2 is $\mathtt{T}_2$.

- *Causality*. If the send or receive of an u_i precedes u_j then u_i is a potential cause of u_j. For a given conversation, the causes are annotated via the various relations among utterances. The internal conditions that enable the causes are not explicitly specified to preserve the autonomy and heterogeneity of the participants.
- *Context-related conversations*. Let utterances u_i and u_j occur in conversations χ_i and χ_j, respectively. If u_i updates u_j then χ_i is context-related to χ_j. For example, χ_1 and χ_6 are context-related conversations.
- *Role*. The abstraction of capabilities used by characters who deal with same type of transactions or are involved in context related conversations. In Figure 2.2, the characters inside a dotted circle belong to the same role.
- *Agent*. A concrete party who can play one or more roles. Our example has agents $\{P, T, A_1, A_2, H_1, H_2, R\}$. Notice that agent T plays 4 roles and has 8 characters, and A_1 and A_2 play the same role, which has characters, $\mathtt{A}_1$ and $\mathtt{A}_2$.

2.2.4 Example Dooley Graph

We can now capture the necessary details in our trip planning example. Table 2.1 shows an interaction with the key communicative acts and relationships identified. Notice the H_1 and H_2 are two hotels; H_2 is contacted when H_1 cancels. Similarly, A_1 and A_2 are two airlines. Figure 2.2 illustrates the corresponding Dooley graph based on the algorithm given by Singh [Singh, 2000b]. Parunak modularizes conversations to reuse agent modules [Parunak, 1996]. Singh focuses on deriving individual agent models from the relationship of the conversations [Singh, 2000b]. Huhns *et al.* incorporate exception handling into the characters involved in each agent model [Huhns et al., 2002].

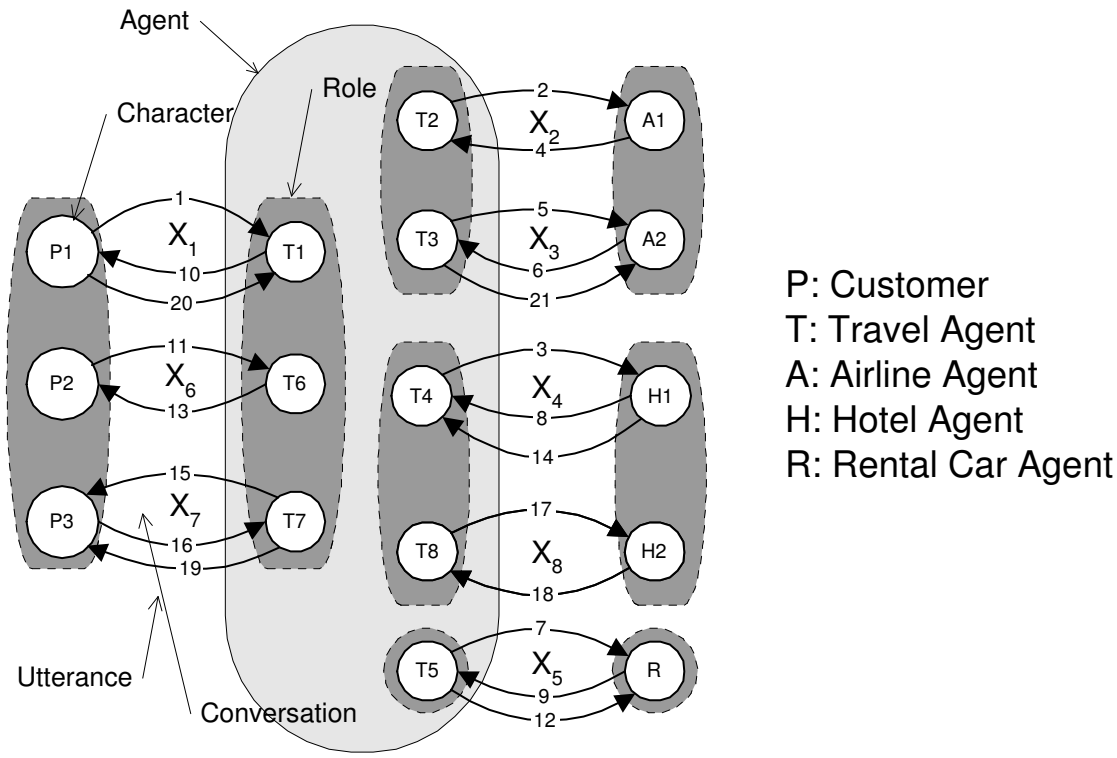

Figure 2.2: Dooley graph for trip planning

However, the above approaches concentrate on low-level interactions, such as the orderings of events. They do not study how high-level abstractions, such as commitments, can be induced. This dissertation demonstrates how commitments and relations among them can be derived from Dooley graphs and what the advantages are of looking at the models at commitment level.

2.3 Workflow Management System

The following are the basic elements involved in a typical workflow management system.

Task. A process or a transaction that accepts inputs, executes, and produces outputs.

A task represents one work step in the workflow. It can be a simple action or a

compound, which is comprised of a set of small subtasks.

Actor. A software module, a Web service, or an agent that executes a task. An actor is not only responsible for processing the task but also in charge of communication, such as receiving input from or sending output to other executors. In the context of our research, we uses agents as our actors. This naturally captures the autonomy, heterogeneity, and proactiveness of each component of a business process.

Capability. A set of functions or skills that are required to execute particular tasks. A capability specifies what input it can take and what output it can produce. It checks the preconditions and postcondition of tasks to ensure data consistency and integrity. The agents must have the matched capabilities before they are assigned to those tasks.

Role. The abstraction of the agents who have common capabilities. One role may own more than one capability and one agent may adopt more than one role which enable agents to execute several different tasks or a compound task.

Data Container. An abstract data storage from which an agent receives input and to which the agent sends output. The contents in data containers can be sent directly with the messages such as XML contents or can reside at an independent location such as in a database or maintained by third-party servers. In the latter case, correlation IDs are used by agents to reference those data.

Connector. A flow control element that controls the sequence of task executions based on business requirements. It evaluates the control events from incoming tasks and triggers the execution of outgoing tasks. Based on the flow logic, we can categorize the connectors into two groups: incoming connector and outgoing connector. An incoming connector has multiple incoming edges and

one outgoing edge which connects to a receiving agent. An outgoing connector has one incoming edge which connects to a sending agent and multiple outgoing edges. In this categorization, no connector connect multiple input tasks to multiple output tasks since we restrict every message to have only one source and one destination. The following is the list of connectors.

- AndJoin (Incoming): All its incoming edges must be triggered before its outgoing edge can be triggered.
- OrJoin (Incoming): Its outgoing edge can be triggered if one of its incoming edges is triggered.
- Fork (Outgoing): If its incoming edge is triggered then all of its outgoing edges must be triggered.

2.4 Business Process Execution Language

BPEL4WS is a service composition specification jointly developed by several industry leader companies such as IBM, Microsoft and BEA. It composes services described by WSDL into a compound business process with a flow control logic. Transaction protocols are also incorporated by adding the WS-Coordination and WS-Transaction frameworks. Because BPEL is increasingly recognized as an industry standard for specifying service compositions, it becomes an important counterpart to our approach. The following is a list of language elements used by BPEL4WS. These elements have similar concepts as workflow but with more expressiveness such as structured activities, which provides more programmable controls.

- Process. Mapped to tasks in workflow.
- Partner. Mapped to roles in workflow.

- PartnerLink. Mapped to capabilities in workflow. Only the roles specified in partner links can invoke the corresponding process.
- Variable. Mapped to Data containers in workflow.
- CorrelationSet. To correlate messages into sessions.
- Basic and Structured Activities. Used to implement process and flow control logic.
- faultHandlers, compensationhandler and eventHandlers. Activities triggered to handle fault, compensations and events.

2.5 Coordination and Transactions

Atomic Transaction. Atomic transactions achieve the "all or none" property in which an overall activity fails if one of its participating activities fails; or the overall activity succeeds only when all of its participating activities succeed.

1. **CompletionWithAck Protocol**. A participant sends its completion result to the coordinator and also receives a notification from the coordinator regarding the overall result.
2. **Completion Protocol.** A participant sends its completion result to the coordinator but without a notification from the latter.
3. **PhaseZero Protocol.** Participants wait for a notification from the coordinator before completing their processes.
4. **2PC Protocol.** An election on whether commit or abort is made before each participant commits or aborts its task.

5. **OutcomeNotification Protocol.** Each participant sends an outcome to the coordinator after finishing its execution.

Business Activity. Business activity protocols handle long-lived transactions and specifically deal with the business exceptions occurring within these transactions. Scopes are used to abstract different levels of process executions and exception handling.

1. **BusinessAgreement Protocol**. A nested scope participant registers for this protocol with its parent scope coordinator. A nested scope must know when it has completed all work for a given business activity.
2. **BusinessAgreementWithComplete Protocol**. A nested scope participant registers for this protocol with its parent scope coordinator.. A nested scope relies on its parent to tell it when it has received all requests to perform work within the business activity.

Chapter 3

Building Process Models

This chapter applies the framework of Chapter 2 to build a commitment-centric business process model. As illustrated in Chapter 1, commitments are specified in the design phase and manipulated during runtime. Our focus is on how to model long-lived transactions as persistent commitment objects by abstracting the low-level messages and activities into commitment operations. Our modelling technique is based on a deep analysis of agent interactions (based on speech acts) from which we can derive commitments, commitment causality, and agent execution models.

3.1 Deriving Commitments

Our communicative acts map to the following commitment operations: *create*, *update*, *discharge*, and *cancel*.

- *Request (without Update)*. Create $\mathsf{C}(x, y, e \rightarrow p)$. A conversation initiator usually requests a particular information or service. The antecedent e corresponds to a condition potentially satisfied by the requestee, e.g., a commitment to do

something. The consequent p corresponds to what the requester will do if the requestee satisfies e. Therefore, this action forms a conditional commitment.

- *Request and Update*. Update $\mathsf{C}(x, y, e \rightarrow p)$. This is a commitment update sent by a request creator. Either e or p could be changed, but the modified commitment still follows the original organizational rules.
- *Commit and Resolve*. Create $\mathsf{C}(x, y, p)$. If a Commit resolves an utterance, then it creates a commitment for the first time following the applicable rules. This may caused by a request sent from a conversation initiator and is also decided by the willingness of the commitment creator.
- *Commit and Update*. Update $\mathsf{C}(x, y, p)$. If a Commit updates an utterance, then it updates its original commitment. It is equivalent to cancelling the previous commitment and creating a new one while still following the original rules.
- *Act and Complete*. Discharge $\mathsf{C}(x, y, e \rightarrow p)$ or $\mathsf{C}(x, y, p)$. If an act (i.e., a non-communicative act) completes an utterance, then the action performer fulfills and discharges the commitment made in that utterance. Since a commitment need not be fulfilled in one message, several (Act and Complete) utterances may exist to complete the same utterance.
- *Act and Resolve*. Create and discharge $\mathsf{C}(x, y, p)$. If a non-communicative act resolves an utterance, then the action performer creates and fulfills a commitment at same time.
- *Cancel and Complete*. Cancel $\mathsf{C}(x, y, e \rightarrow p)$ or $\mathsf{C}(x, y, p)$, if a Cancel completes an utterance, then it cancels a conditional commitment (Request) or an unconditional commitment (Commit) made in that utterance.

From the mappings, we can prove that commitments are created within the boundaries of conversations because the commitments are created by utterances and, as a consequence,

both debtor and creditor are characters in a same conversation. Algorithm 1 below generates a complete list of commitments.

```
CommitList = {};
for each u_i do
    switch u_i do
        case Request w/o Update:
            Add C(S_i, R_i, e_i → p_i) to CommitList, where e_i = true and p_i =
            ToBeDecided.
        case (Commit and Resolve) or (Act and Resolve) w/o Update:
            Add C(S_i, R_i, p_i) to CommitList, where p_i is an action or a condition
            that S_i commits to R_i to perform or keep true.;
        case Act and Complete:
            u_j ← the utterance that u_i completes;
            if u_j is a Request then
                let C(S_j, R_j, e_j → p_j) be the corresponding commitment of u_j;
                p_j ← the Act;
                for each u_k that u_i responds to do
                    switch u_k do
                        case Commit + Resolve w/o Update:
                            e_j ← e_j ∧ C(S_k, R_k, p_k);
                        case Act + Resolve w/o Update:
                            e_j ← e_j ∧ theAct;
                        case Update:
                            u_l ← the original Resolve utterance that u_k updates fol-
                            lowing the update chain;
                            e_j ← e_j ∧ C(S_l, R_l, p_l);
                        case Act + Complete:
                            e_j ← e_j ∧ theAct;
```

Algorithm 1: Generate a list of commitments

The algorithm creates a conditional commitment for a Request message. For example, it can create $\mathsf{C}(P, T, e_1 \rightarrow p_1)$ for utterance u_1. It creates an unconditional commitment

for a Resolve message, e.g., C(A_2, T, Confirm) for utterance u_6. To decide the antecedent and consequent in a conditional commitment, the algorithm looks for an Act and Complete message u_i that completes a Request utterance u_j. If there is one, then the Act decides the consequent. For any utterance u_k that u_i responds to, the corresponding commitments or acts of u_k constructs the antecedent.

By executing the algorithm on Table 2.1, we obtain the following list of commitments. Here TBD indicates *to be decided*. All conditions of the form true $\rightarrow q$ are simplified to q.

- C_1 = C(P, T, C(T, P, SendItinerary)→Pay(P, T))
- C_2 = C(T, A_1,TBD)
- C_3 = C(T, A_2, C(A_2, T, Confirm) $\wedge$ Pay(P, T) →Pay(T, A_2))
- C_4 = C(T, H_1,TBD)
- C_5 = C(T, R, TBD)
- C_6 = C(A_2, T, Confirm)
- C_7 = C(H_1, T, Confirm)
- C_8 = C(R, T, Confirm)
- C_9 = C(T, P, SendItinerary)
- C_{10} = C(T, H_2, TBD)
- C_{11} = C(H_2, T, Confirm)

Some conditional commitments have undecided antecedents and consequents, because our initial table does not provide enough information to derive the dependencies. For example, for commitment C_2, because A_1 declines the request, we cannot determine what A_1 will commit to T if he accepts the request. For commitments C_4, C_5, and C_{10}, neither the antecedent nor the consequent of T can be determined, because there is no *Act and Complete* utterance sent by T in response to the commitments made by other parties. However, the travel domain may have a business rule or policy that cancellation charges may be

owed, e.g., if a passenger is a no show. Any such domain policies can be added during the final design.

Many commitment instances are derived from the conversation table. Some commitments update the others and some replace the others, but they essentially relate to the same debtors and creditors and deal with the same transactions. We treat all these related commitments as instances of the same *commitment class*. They are merely the results of operations performed on the same class. Next, we give the definition of a commitment class. Two commitments $\mathsf{C}(x_1, y_1, p_1)$ and $\mathsf{C}(x_2, y_2, p_2)$ belong to same class if and only if they satisfy the following requirements:

1. Role(x_1) = Role(x_2) and Role(y_1) = Role(y_2)

2. p_1 and p_2 deal with the same transaction, goods, or information (possibly $p_1 = p_2$).

Based on this definition, we can put commitment C_2 and C_3 into same class, because the only difference between them is that different instances of the same role A (A_1 and A_2) are involved. Likewise, C_4 and C_{10} are classified together. For each class, we use the commitment with the lowest subscript as the representative of the class but with bold font. The following lists each commitment class and its commitment instances in the example.

- $\mathbf{C}_1 = \{C_1\}$
- $\mathbf{C}_2 = \{C_2, C_3\}$
- $\mathbf{C}_4 = \{C_4, C_{10}\}$
- $\mathbf{C}_5 = \{C_5\}$
- $\mathbf{C}_6 = \{C_6\}$
- $\mathbf{C}_7 = \{C_7, C_{11}\}$
- $\mathbf{C}_8 = \{C_8\}$
- $\mathbf{C}_9 = \{C_9\}$

For a commitment class, any characters involved in its instance commitments are replaced by their corresponding roles. For example, in commitment class $\mathbf{C}_2$, character A_1 and A_2 are both replaced with A, and in class $\mathbf{C}_4$, H_1 and H_2 are replaced by H.

If a commitment class is a conditional commitment, then the antecedent of the class is the conjunction of the antecedents of the constituting commitments, and the consequent is the list of consequents of those commitments. For example, $\mathbf{C}_2$ = **C(T, A, (C(A, T, Confirm)** $\wedge$ **Pay(P, T))** $\rightarrow$ **Pay(T, A))** and $\mathbf{C}_4$ = **C(T, H, TBD)**. This is appropriate because the commitments in a class involve the same transactions.

3.2 Deriving Causal Relations

Causal relations among commitments are crucial to understanding the chain of commitment operations, because they drive an interaction along significant states where the real transactions of domain value occur. A *commitment causality diagram (CCD)* is a graph showing potential causality between each pair of commitment operations. A CCD highlights the important stages within the information flows and hides details of the interaction protocols that can vary depending on the actual implementation. From a designer's standpoint, a CCD reflects the high-level business logic that specifies what agreements should be achieved. From a CCD, we can infer the agent conversations needed to achieve and modify the commitments underlying these agreements. Once the commitments and their relations are identified, designers can always choose an optimal conversation that flows between any related pair of commitment operations.

Figure 3.1 shows a CCD derived for our running example. Each node consists of five elements, namely, the commitment class identifier and its associated four operations: *create* (Crt), *update* (Upd), *discharge* (Dcg), and *cancel* (Cnl). If an operation of a commitment

is causally related to another operation, then there is a directed edge from the causing operation to the caused operation. For example, if the creation of commitment $\mathbf{C}_1$ causes the creation of commitment $\mathbf{C}_2$, then there is an edge from Crt($\mathbf{C}_1$) to Crt($\mathbf{C}_2$).

If the source operation of an edge is an update or cancel, then we use a dotted line which represents an abnormal activity occurring in the process execution. In this example, an update of $\mathbf{C}_1$, which represents the customer's request to update his trip by cancelling the car rental, leads to Cnl($\mathbf{C}_5$), Cnl($\mathbf{C}_8$), and Upd($\mathbf{C}_9$), which represent the travel agent's cancellation of the car, the car rental company's cancellation of the car, and the travel agent's update of the itinerary, respectively. By using different line types, we can conveniently differentiate the abnormal activities from normal activities thus providing useful layered designs.

```
1 Construct the Responds graph for the entire conversations;
2 for each u_i that involves a commitment operation do
3     for each path in the u_i's Responds graph do
4         if ∃u_j that involves a commitment operations then
5             u_j ← the first utterance that involves commitment operation in the path;
6             Add a directed edge from u_i's operation to u_j's operation;
7             Mark the edge with the set of conversations that each utterance (except
              u_j) in the path belongs to;
8 for each commitment operation do
      Use dotted line to link an incoming edge to an outgoing edge if the conversations
      on the two edges are same or are causally related (e.g., Crt(C_7));
```

Algorithm 2: Generate a commitment causality diagram

Algorithm 2 generates a commitment causality diagram. This algorithm scans through each commitment operation and finds its immediate causally related operations. It examines each Responds path started from the original operation, since one operation can cause the occurrence of multiple other operations. The very first operation in each path must

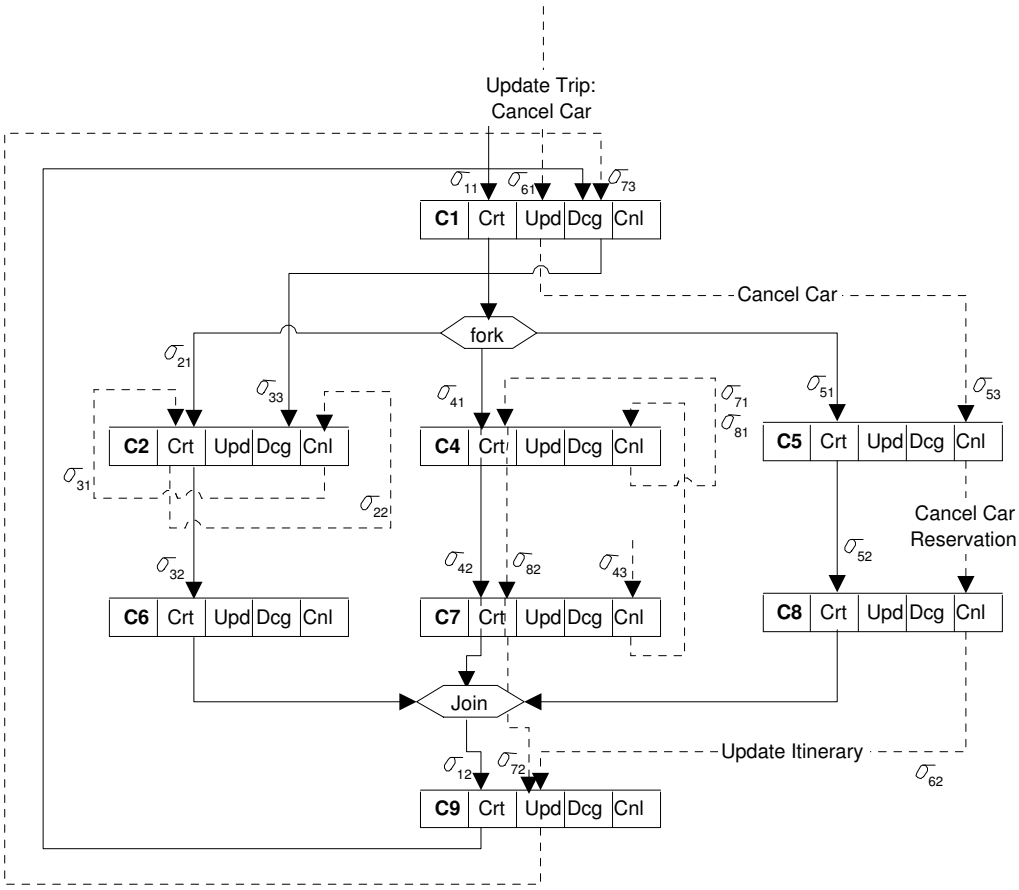

Figure 3.1: Commitment causality diagram for trip planning.

be the only operation directly caused along that path. For example, Figure 3.2 shows the Responds path of utterance u_1, wherein u_2, u_3, and u_7 are directly caused by u_1, but u_5 and u_6 are not. Notice that since multiple paths are considered, the occurrence of the original operation does not entail the occurrence of the first operation of any path. A subtlety in this algorithm is that different causal paths can pass through the same commitment operation (see line 8 of Algorithm 2). This enables the given role to be bound to different agents at run time. Figure 3.1 illustrates this point. Recall that $\mathbf{C}_4$ deals with the interaction of T with H (derived from H_1 and H_2). The cancellation of the instance of $\mathbf{C}_7$ causes a creation

of a new instance of $\mathbf{C}_4$ for H_2, which automatically voids the instance of $\mathbf{C}_4$ for H_1. A dotted edge is added from Cnl($\mathbf{C}_4$) to Cnl($\mathbf{C}_7$) to indicate this. The new instance of $\mathbf{C}_4$ then triggers a new instance of $\mathbf{C}_7$. The above scenario is reflected in the edges labeled χ_8 and the dotted lines within the Crt($\mathbf{C}_4$) and Crt($\mathbf{C}_7$) blocks.

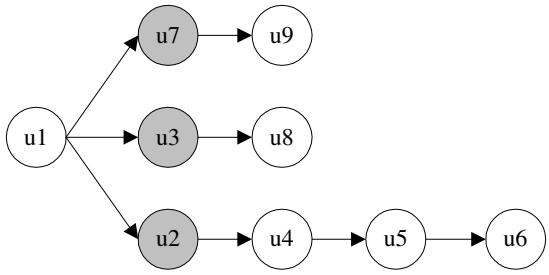

Figure 3.2: Responds tree of utterance u_1

A CCD illustrates how the commitments evolve during the interaction among the roles. For example, in Figure 3.1, P's initial request, which corresponds to the creation of $\mathbf{C}_1$, triggers the creation of $\mathbf{C}_2$, $\mathbf{C}_4$, and $\mathbf{C}_5$. This is a simple request, which follows a travel domain rule that causes T to create three subsequent requests without further negotiation.

The creation of the first instance of $\mathbf{C}_2$ leads to the creation of another instance of $\mathbf{C}_2$ because T and A_1 failed to negotiate in conversation χ_2. In this case, the first instance of $\mathbf{C}_2$ is implicitly cancelled (we add a dotted edge from Crt($\mathbf{C}_2$) to Cnl($\mathbf{C}_2$)). The second instance of $\mathbf{C}_2$ leads to the creation of $\mathbf{C}_3$, because A_2 in conversation χ_3 accepts the request and sends out a confirmation. This case indicates that the creation of $\mathbf{C}_2$ may lead to the creation of $\mathbf{C}_3$, but if it fails, it will be created again by T without any additional negotiation or consultation. In a real system, this may happen when the first preferred airline is sold out and T tries another acceptable one, if there is any. Because there must be a limit on the number of tries, when there are no more airlines to consider, the cancellation of $\mathbf{C}_2$ may lead to the cancellation or update of $\mathbf{C}_1$. For example, P may want to cancel the whole trip

or change the schedule of the trip.

Let us look at another case in which P updates the trip order by cancelling the car rental. This corresponds to the update of $\mathbf{C}_1$, which leads to the cancellation of $\mathbf{C}_5$ and update of $\mathbf{C}_9$. There is no edge between Cnl($\mathbf{C}_5$) and Cnl($\mathbf{C}_8$). However, to model a travel domain rule that the cancellation of car rental request will cancel any car rental confirmation, we add a dotted edge between those two cancellation operations.

The last case we study is the cancellation of $\mathbf{C}_7$. This leads to the creation of another instance of $\mathbf{C}_4$ by going through the conversation χ_7 in which case T asks P for an alternative hotel. This case tells us that between each pair of commitment operations, there may be more conversations and roles involved to decide whether the conditions behind the potential causal relation are satisfied at run time.

Here is a summary of how each type of path in CCD represents the actual business process scenario.

- The creation paths reflect the business logic and specify the interdependencies among business processes.
- The update paths enable revisions of business processes. Each update has to be made within update domain. The update domain evaluates each update and triggers different causal links.
- The discharge paths terminate each causally related commitments. If the discharge links are not present in a CCD, we may consider it an incomplete business process model.
- The cancel paths handle failures and exceptions. The causal links and commitment operations along the cancel paths reflects how exceptions are treated at the domain-level.

By analyzing the CCD we can get a view on how commitments are created, updated,

and discharged in a normal business flow and how they are cancelled because of exceptions or revoked requests. This view is not complete since the original conversation table provides only partial case scenarios. We can either provide more complete agent interactions or try to complete and refine the diagram by hand. The first option is obviously not acceptable, since you have to provide many low-level messages and can be easily confused in trying to capture causal relationships among all the messages. The second option is more appealing since we would already have captured significant transactions and events at the commitment level—the only things remaining would be additional causal edges between some pairs of commitment operations.

3.2.1 Commitment Update Domain

Commitment update facilitates transaction reentrance and exception handling. What can be updated within a commitment is an interesting topic. Here we introduce a concept called "Commitment Update domain" which specifies the invariants that the activation condition, discharge condition and predicate of a commitment do not violate. It also defines the scope within which the agreement can be adjusted and adds flexibility to agent interactions.

For example, an agreement between a customer and a travel agent may state that, the travel agent must look for a flight ticket on a given date (9/10) and within a given price range (< 700 USD). Initially, the travel agent confirms the first choice, 8am, on 9/10 for 650 USD. He holds the ticket for the customer before the latter pays him. However, for some reason the flight gets cancelled, but the travel agent finds another one, at 2PM, on 9/10 for 680 USD, which still satisfies the original agreement. The customer is notified of this change and still pays for the ticket.

We call the original agreement forms the "Update Domain" of the travel agent's commitment to the customer. If a commitment update operation violates the domain invariant,

then the customer may cancel his commitment to the travel agent without any obligation.

In a business process specification, we could derive these kinds of update domains from domain-level policies. The research on this aspect will focus on the formalization of domain invariants and detection of update violations. This topic is beyond the scope of this dissertation.

3.2.2 Accommodating Commitment Patterns

In previous research, we proposed a small number of commitment patterns so as to streamline the interactions among agents [Xing et al., 2001]. The main patterns are Entertain Request, Notify the Consumer, Entertain Update, Renotify the Consumer, Satisfy the Consumer (Retry), and Resign (Section 2.1.3 summarize these patterns). These patterns are the behaviors of a single agent. However, by applying them to a CCD, we can state that any operation performed on one commitment needs to be propagated to the operation of another commitment to which the first one is causally related. For example, if the creation of one commitment causes the creation of another one, then updating or cancelling the first one should affect the second one too.

In the CCD of Figure 3.1, there is no out-edge from the cancellation of $\mathbf{C}_1$, which corresponds to the request cancellation from P. We can add three edges connecting this node to the cancellations of $\mathbf{C}_2$, $\mathbf{C}_3$, and $\mathbf{C}_4$, respectively. If this cancellation happens after the discharge of $\mathbf{C}_1$ (in which case P has paid for the flight ticket), then a new conversation may be involved between Cnl($\mathbf{C}_1$) and Cnl($\mathbf{C}_2$) that will resolve a refund or surcharge.

For any existing edges, we can optimize or reduce the conversations along the edges. For example, the cancellation of hotel (Cnl($\mathbf{C}_7$)) creates a new $\mathbf{C}_4$ through conversation χ_8. Can we eliminate this conversation? For instance, the agent may automatically find another hotel within two miles of the old one, so that it can avoid potential delay resulting from an

email round trip or a phone call.

Please note that any additional conversations may spawn new commitments. We don't allow commitment operations on the causality edge between two other commitment operations. Therefore, we need to repeatedly regenerate the CCD whenever new conversations are added.

3.3 Generating Agent Models

The last step of our approach is generating agent models. The agent models are represented in commitment-centric statecharts and are derived from the commitment causal relation diagram. Each statechart is composed of two parts, dealing with creation and revision, respectively. The creation phase represents the negotiation of initial agreements and the formation of commitment chains, while the revision phase represents the update and cancellation of the agreements and any subsequent effects on the commitment chains. Algorithm 3 generates a statechart for an agent. For reasons of space, the algorithm is not as detailed as the others, but highlights the major steps. Figure 3.3 is the example statechart for T.

3.4 Computational Complexity Results

In this section we analyze the complexities of these algorithms. Throughout the proofs, we use the following notation:

$$
\begin{aligned}
k &= \text{the number of agents} \\
n &= \text{the number of utterances} \\
m &= \text{the number of conversations} \\
p &= \text{the number of commitments}
\end{aligned}
$$

```
1  for each operation α on each commitment created by agent A do
2      Create a new state s_1;
3      Create a new transition to s_1 and mark it as "Send α";
4      for each incoming edge to α do
5          β ← The source operation of the edge;
6          Create a new state s_2;
7          Create a new transition to s_2 and mark it as "Receive β" or any communi-
           cation involved on the edge;
8  Construct the "creation" part of the statechart by linking and merging the states and
   transitions that involve only the creations of commitments;
9  Construct the "revision" part of the statechart by Linking and merging the states
   and transitions that involve update, cancel, discharge or the second creations of
   commitments;
10 Connect these two statecharts together by the state where the last commitment is
   created;
11 Put any concurrent states and transitions into AND states and merge their common
   transitions;
12 Use AND connector to converge transitions leading to a same state if these transi-
   tions are the AND preconditions of the state;
```

Algorithm 3: Generate statechart for an agent

$$p_A = \text{the number of commitments created by agent A}$$

$$l = \text{the maximum length of the } \textit{Responds to} \text{ paths in which only the begin and end utterance involve commitment operations}$$

Since one utterance can respond to an arbitrary number of utterances that are sent by the same agent, which makes an unrealistic upper bound of the time cost (O(n^2)) for traversing *Responds to* graphs, we want to limit such number to one so that the time cost can be bounded by the number of agent (k). In other words, if one agent sends an utterance to another agent, the latter will respond immediately and will not wait for more utterances sent from the same agent. It is true in most interaction scenarios where only one utterance would be often used to trigger the consequent actions. Based on such motivation, we make

an assumption that, if u_i responds to both u_j and u_k, then $S_j \neq S_k$. This assumption ensures that the number of utterances that one utterance can respond to is not greater than the number of agents.

3.4.1 Generate a List of Commitments

Theorem 1. *The time cost of Algorithm 1 is $O(p + km)$ and the space cost is $O(p)$.*

Proof.

1. In each "Case Request" or "Case Commit and Resolve or Act and Resolve," we create one commitment, so both the total time and space cost should be $O(p)$, where p is the number of commitments.

2. In each "Case Act and Complete," since u_i only completes a Request utterance u_j whose number should be less than m (the number of conversations), and also the number of u_k is less than k (the number of agents), the total time cost is $O(km)$. Since there is no memory allocated here, the space cost should be 0.

3. Summing the results in the above two lines, the total time cost is $O(p + km)$ and the total space cost is $O(p)$.

□

3.4.2 Generate a Commitment Causality Diagram

Theorem 2. *The time cost of Algorithm 2 is $O(kln)$ and the space cost is $O(kn)$.*

Proof.

1. To construct the *Responds to* graph, we scan each utterance and find the utterances that it responds to. Since the maximum *Responds to* messages should be less than the number of agents, the time cost is $O(kn)$. We also allocate one unit of space for each *Responds to* link, so the space cost is also $O(kn)$.

2. In the loop between line 3 and line 7, we scan through each utterance that involves commitment operations and find the closest utterances u_j in each of its *Responds to* path that involves commitment operations. Since the length of the paths between u_i and u_j should be less than l, the total time cost is $O(kln)$. There is no new space allocated here so the cost is 0.

3. Summing the results in the above steps, the total time cost is $O(kln)$ and the total space cost is $O(kn)$.

□

3.4.3 Generate Statechart for an Agent

Theorem 3. *The time cost of Algorithm 3 is $O(kp_A)$ and the space cost is $O(kp_A)$.*

Proof. 1. For line 2 and line 3, both the time and space cost is $O(1)$

2. For lines 4 through 7, since the maximum number of incoming edges for each operation should be less than k, both the time and space cost is $O(k)$

3. The outermost loop repeats at most $4p_A$ times.

4. Based on the above, both the time and space cost is $O(kp_A)$

□

3.5 Completeness and Soundness

To verify if a model is technically correct, we must prove its completeness and soundness. The following defines the completeness and soundness of a CCD in terms of the conversations in which a multiagent system is engaged.

Definition 1. ***Completeness:*** *A CCD is complete if and only any conversation that exists in the agent conversation table can be generated by the CCD.*

Definition 2. ***Soundness:*** *A CCD is sound if and only any possible edge in the CCD belongs to or carries a conversation.*

Based on Singh's algorithm [Singh, 2000b], we can derive that a conversation must be initiated by either a Request or a Question, and must include one Resolve (Accept, Reject or Inform) message. Therefore the idea of proof is to examine each edges in CCD and associate them to existing conversations (completeness) or possible conversations (soundness).

Theorem 4. *A CCD generated by Algorithm 2 is both complete and sound.*

Proof. **Completeness**

1. For any conversation χ, let u be its initiation message and v be its Resolve message.

2. There are five possible combinations of u and v. (NIL means no commitment operation involved, and $\hookrightarrow$ and $\dashrightarrow$ mean a trigger and a sequence of triggers, respectively)

   (a) [u = Request w/o update, v = Commit] $\Rightarrow$ [$\hookrightarrow$ Crt(C_1) $\dashrightarrow$ Crt(C_2)]

   (b) [u = Request w/ update, v = Commit] $\Rightarrow$ [$\hookrightarrow$ Upd(C_1) $\dashrightarrow$ Upd(C_2)].
   Based on Algorithm 2, the edges coming into either the above pair of operations generates conversation χ;

   (c) [u = Request w/o update, v = Refuse] $\Rightarrow$ [$\hookrightarrow$ Crt(C_1) $\rightarrow$ Cnl(C_1)].
   There is always a default edge from Crt(C) to Cnl(C) to carry a conversation that ends with a refusal to a non-update request;

   (d) [u = Request w/ update, v = Refuse] $\Rightarrow$ [$\hookrightarrow$ Upd(C_1) $\rightarrow$ NIL].
   The CCD generates a no-target edge which carries a conversation χ that causes no commitment changes;

(e) $[u$ = Question, v = Inform$] \Rightarrow [\hookrightarrow$ NIL $\rightarrow$ NIL$]$.

This is a non-transactional conversation and the CCD places it on a edge whose source and target may or may not be a commitment operation node which depends on whether conversation χ is triggered or causes any transactional operations or not.

3. In the above cases, the CCD can generate the conversation χ. Therefore, the CCD is complete.

Soundness

1. Based on the semantics of the commitment operations, there are only 11 possible edges (see below) out of 16 combinations of two commitment operations. The remaining 5 edges are excluded because of the following rules:

   - An Update operation cannot directly lead to a Creation, however, it may indirectly achieve this through a Cancel operation.
   - A Discharge operation can only leads to another Discharge.
   - A Cancel operation cannot lead to a Discharge.

   In addition, for reentrant connectors, the edges are considered to be different combinations of individual sources and targets. We also add the 12th case in which the source or the target of the edge does not have commitment operations.

   Let $\mathsf{C}_1(x, y, p_1)$ and $\mathsf{C}_2(y, z, p_2)$ be the commitments of the source and target operations, respectively.

2. Creation $\rightarrow$ Creation. If $x \neq z$, then the edge carries a "Request w/o Update" message ($Crt(\mathsf{C}_2)$) as the initiation of a conversation (1a); if $x = z$, then the edge carries a "Commit w/o Update" message ($Crt(\mathsf{C}_2)$) which resolves the request message

($Crt(\mathsf{C}_1)$) in the same conversation (1b).

3. Creation → Update. If $x \neq z$, then the edge carries a "Request w/ Update" message ($Crt(\mathsf{C}_2)$) as the initiation of a conversation (2a); if $x = z$, then the edge carries a "Commit w/ Update" message ($Crt(\mathsf{C}_2)$), which updates the "Request w/o Update" message ($Crt(\mathsf{C}_1)$) in the same conversation (2b).

4. Creation → Discharge. If $x \neq z$, then the edge carries an "ACT w/ Complete" message ($Crt(\mathsf{C}_2)$ + $Dcg(\mathsf{C}_2)$), which performs an action immediately; if $x = z$, then the edge carries an "ACT w/ Complete" message which completes a "Request w/o Update" message ($Crt(\mathsf{C}_2)$), where C_2 resembles C_1 and C_1 resembles C_2 in case 1b.

5. Creation → Cancel. This case only applies when C_1 and C_2 represent a same commitment and the edge carries a Refusal to the Request ($Crt(\mathsf{C}_1)$) so that the commitment is created and cancelled afterwards.

6. Update → Update. If $x \neq z$, then the edge carries a "Request w/ Update" message ($Crt(\mathsf{C}_2)$) as the initiation of a conversation (5a); if $x = z$, then the edge carries a "Commit w/ Update" message ($Crt(\mathsf{C}_2)$) which updates the "Request w/ Update" message ($Crt(\mathsf{C}_1)$) in the same conversation (5b).

7. Update → Discharge. If $x \neq z$, then the edge carries an "ACT w/ Complete" message ($Crt(\mathsf{C}_2)$ + $Dcg(\mathsf{C}_2)$), which performs an action immediately; if $x = z$, then the edge carries an "ACT w/ Complete" message which completes a "Request w/ Update" message ($Crt(\mathsf{C}_2)$), where C_2 resembles C_1 and C_1 resembles C_2 in case 1b or 5b.

8. Update → Cancel. If $x \neq z$, then the edge carries a "Cancel w/ Complete" message ($Cnl(\mathsf{C}_2)$) to cancel the "Request" message in 1a, 2a, or 5a; if $x = z$, then the edge

carries a "Cancel w/ Complete" message ($Cnl(\mathsf{C}_2)$) which cancels the "Commit" message ($Crt(\mathsf{C}_1)$) in 1b, 2b, or 5b.

9. Discharge → Discharge. If $x \neq z$, then the edge carries a "Cancel w/ Complete" message ($Cnl(\mathsf{C}_2)$) to cancel the "Request" message in 1a, 2a, or 5a; if $x = z$, then the edge carries a "Cancel w/ Complete" message ($Cnl(\mathsf{C}_2)$), which cancels the "Commit" message ($Crt(\mathsf{C}_1)$) in 1b, 2b, or 5b.

10. Cancel → Creation. Similar to case 4, this case only applies when C_1 and C_2 represent the same commitment and the edge carries a new "Request" or a new "Commit" message ($Crt(\mathsf{C}_1)$), which denotes the re-creation of a commitment after its cancellation.

11. Cancel → Update. This case only applies when $x \neq z$ since otherwise the two commitments belong to a same conversation in which the commitments must both exist or neither of them should exist. Therefore, in the event of $x \neq z$, the edge carries a "Request w/ Update" message ($Crt(\mathsf{C}_2)$) as the initiation of a conversation

12. Cancel → Cancel. If $x \neq z$, then the edge carries a "Cancel w/ Complete" message ($Cnl(\mathsf{C}_2)$) to cancel the "Request" message in 1a, 2a, or 5a; if $x = z$, then the edge carries a "Cancel w/ Complete" message ($Cnl(\mathsf{C}_2)$), which cancels the "Commit" message ($Crt(\mathsf{C}_1)$) in 1b, 2b or 5b.

13. The source or the target of the edge does not have commitment operation. The edge carries one or more "Query and Inform" conversations which does not involve transactional actions.

From the itemization above, we conclude that every edge belongs to or carries a conversation and this conversation has a Request (w/ or w/o Update) message or a Query message

and one Resolve (Commit w/ or w/o Update, and Inform) message. Therefore, the CCD is sound.

□

3.6 Convert CCD to XML Specifications

The CCD XML adds richer syntax to a CCD itself so that the subtlety of business processes can be captured and the conversion between other business process models and CCD is formalized more precisely. Figure 3.4 is the UML model for the commitment-based business process.

The XML specification of the commitment-based model is described below.

- **Commitment**

```
<Commitment name="...">
    <Debtor name="..."/>
    <Creditor name="..."/>
    <Proposition UpdateInvariant=.../>
    <ActivationCondition UpdateInvariant=.../>
    <DischargeCondition UpdateInvariant=.../>
    <ViolationCondition UpdateInvariant=.../>
    <DataDomain/>
</Commitment>
```

- **Conversation**

```
<Conversation name="...">
    <ConsumerCharacter name="..."/>
```

```
    <ProviderCharacter name="..."/>
    <ConversationSegments>
        <Segment name="..."/>
        ...
        <Segment name="..."/>
    </ConversationSegments>
</Conversation>
```

- **ConversationSegment**

```
<ConversationSegment name="...">
    <Conversation name="..."/>
    <Utterance name="...">
        <Utterance1/>
        ...
        <UtteranceN/>
    </Utterance>
</ConversationSegment>
```

- **ConversationLink**

```
<ConversationLink name="...">
    <SourceCommOp Commitment="..." Operation="..."/>
    <TargetCommOp Commitment="..." Operation="..."/>
    <ActivationCondition/>
    <ConversationSegments>
        <Segment name="..."/>
        ...
```

```
        <Segment name="..."/>
    </ConversationSegments>
</ConversationLink>
```

- **CommitmentOperation**

```
<CommitmentOperation name="..." type="...">
    <Commitment name="..."/>
    <CausedConvLinks>
        <Link name="..."/>
        ...
        <Link name="..."/>
    </CausedConvLinks>
    <CausingConvLinks>
        <Link name="..."/>
        ...
        <Link name="..."/>
    </CausingConvLinks>
</CommitmentOperation>
```

3.7 Incorporating Commitment Semantics into BPEL

This section describes how to incorporate commitment semantics into the popular BPEL specification to provide more flexible service-oriented interactions. In our commitment-based view of service composition, agents request and provide services by making commitments. The creation of the commitments corresponds to sending service requests and

delivering services. This is mapped to the traditional service model. However, unlike traditional services that end their transaction when a result is delivered, in our approach, it is just the beginning of the lifecycle of commitments. Agents can update or cancel these commitments as often as required by the given business logic. These updates and cancellations automatically modify the underlying service requests and responses.

To incorporate commitments, we need to modify service definitions to support commitment operations, by doing which the services become updatable and cancellable without further methods being added. Figure 3.4 shows the new service model expressed in the Unified Modelling Language (UML) [UML].

Service requesters and providers communicate with each other by sending *Messages*. Each *Message* includes the *Service* being requested or provided, the *InputItem* or *OutputItem* to be passed or delivered, the *Commitment* involved, and the *CommitmentOp* that manipulates the *Commitment*. Each *Commitment* has two associated *Roles*, debtor and creditor. To make a valid *Commitment*, agents must agree to perform the commitment's roles [Wan et al., 1999]. We capture *Role* as an independent entity to explicitly document the parties involved in a service episode with its associated commitments. Doing so enables us to let the agents change roles. For example, an agent may delegate some of its commitments to another agent who was not initially involved in the service episode.

A *Commitment* can be of two types, *RequesterCommitment* and *ProviderCommitment*. A *RequesterCommitment* usually conveys a requester agent's willingness to perform some actions for his provider if the latter will satisfy the former's service request. It is associated with both the *InputItem* and *OutputItem* in that it passes the *InputItem* to the service provider and also uses the properties of the *OutputItem* to specify commitment conditions. For example, a traveler may commit to paying an airline if the latter finds him a flight. A *ProviderCommitment* is created by a service provider who commits to the satisfaction

to his requester. For example, an airline would commit to rebooking the traveler if the flight schedule changes (and letting him know of the change). It is associated only with the *OutputItem* since it only applies to what the provider produces for the requester. For this reason, the two kinds of commitments do not have the same structure. The following shows the message templates of different commitment operations performed on the two types of commitments.

Creation of RequesterCommitment. The creation of a commitment for requesting a service. It should contain all initial input data required by the service. The commitment involved specifies the liabilities on the requester upon delivery of the service or the cancellation of the service request.

```
<env:Header>
    <m:ServiceName>
    ...
    </m:ServiceName>
</env:Header>
<env:Body>
    <CommitmentOp type="Creation"/>
    <Input1> ... </Input1>
    ...
    <InputN> ... </InputN>
    <RequesterCommitment>
        <CommId> ... </CommId>
        <Condition> ... </Condition>
        <Action> ... </Action>
```

```
        </RequesterCommitment>
    </env:Body>
```

Update of RequesterCommitment The update of a commitment for the service requested. It passes part of the initial input data that need to be updated.

```
<env:Header>
    <m:ServiceName>
    ...
    </m:ServiceName>
</env:Header>
<env:Body>
    <CommitmentOp type="Update"/>
    <UpdateInput1> ... </UpdateInput1>
    ...
    <UpdateInputM> ... </UpdateInputM>
    <RequesterCommitment>
        <CommId> ... </CommId>
        <Condition> ... </Condition>
        <Action> ... </Action>
    </RequesterCommitment>
</env:Body>
```

Cancellation of RequesterCommitment The cancellation of a commitment for the service requested. It voids the original service request and also takes any liabilities if required.

```
<env:Header>
    <m:ServiceName>
    ...
    </m:ServiceName>
</env:Header>
<env:Body>
    <CommitmentOp type="Cancel"/>
    <RequesterCommitment>
        <CommId> ... </CommId>
        <Condition> ... </Condition>
        <Action> ... </Action>
    </RequesterCommitment>
</env:Body>
```

Discharge of RequesterCommitment The discharge of a commitment for a requested service. It releases any commitments associated with the original service request and ends the history for this transaction.

```
<env:Header>
    <m:ServiceName>
    ...
    </m:ServiceName>
</env:Header>
<env:Body>
    <CommitmentOp type="Discharge">
    <RequesterCommitment>
```

```
        <CommId> ... </CommId>
        <Condition> ... </Condition>
        <Action> ... </Action>
    </RequesterCommitment>
</env:Body>
```

Creation of ProviderCommitment The creation of a commitment for service delivery. It should contain all the initial output data required by the service response The commitment involved specifies the conditions that need to be satisfied upon service delivery.

```
<env:Header>
    <m:ServiceResponseName>
    ...
    </m:ServiceResponseName>
</env:Header>
<env:Body>
    <CommitmentOp type="Creation">
    <Output1> ... </Output1>
    ...
    <OutputN> ... </OutputN>
    <ProviderCommitment>
        <CommId> ... </CommId>
        <Condition> ... </Condition>
        <Action> ... </Action>
    </ProviderCommitment>
```

```
</env:Body>
```

Update of ProviderCommitment The update of a commitment for a delivered service. It contains part of the output data that needs to be updated because some changes occur within the service provider or in response to the chain of updates triggered from other services.

```
<env:Header>
    <m:ServiceResponseName>
    ...
    </m:ServiceResponseName>
</env:Header>
<env:Body>
    <CommitmentOp type="Update">
    <UpdateOutput1> ... </UpdateOutput1>
    ...
    <UpdateOutputM> ... </UpdateOutputM>
    <ProviderCommitment>
        <CommId> ... </CommId>
        <Condition> ... </Condition>
        <Action> ... </Action>
    </ProviderCommitment>
</env:Body>
```

Cancellation of ProviderCommitment The cancellation of a commitment for a delivered service. This happens when the service results are no longer available. The service has to be cancelled

```
<env:Header>
    <m:ServiceResponseName>
    ...
    </m:ServiceResponseName>
</env:Header>
<env:Body>
    <CommitmentOp type="Cancel">
    <ProviderCommitment>
        <CommId> ... </CommId>
        <Condition> ... </Condition>
        <Action> ... </Action>
    </ProviderCommitment>
</env:Body>
```

Discharge of ProviderCommitment The discharge of a commitment for a delivered service. It releases any commitments associated with the service delivery and ends the history for this transaction.

```
<env:Header>
    <m:ServiceResponseName>
    ...
    </m:ServiceResponseName>
</env:Header>
<env:Body>
    <CommitmentOp type="Discharge">
    <ProviderCommitment>
```

```
            <CommId> ... </CommId>
            <Condition> ... </Condition>
            <Action> ... </Action>
        </ProviderCommitment>
    </env:Body>
```

3.8 Case Study

This section examines an online shopping process to show how we model the process using CCD and what business activities can be captured by the CCD. A Customer P purchases a computer from Newegg.com using a VISA credit card (B). Newegg.com has two departments, sales (N_1) and shipping (N_2). The sales department receives an order from the customer and then talks to the VISA card company for card verification. If the card is successfully verified, the sales department notifies the shipping department to start packing and shipping. The commitments involved in this case are as follows:

- $\mathsf{C}_1 = \mathsf{C}(P, N_1, \text{Order})$
- $\mathsf{C}_2 = \mathsf{C}(N_1, P, \text{Order} \wedge \text{Credit Approved} \rightarrow$
  $(\text{Ship within 24 hours}) \wedge (\text{Not to charge before shipping})$
- $\mathsf{C}_3 = \mathsf{C}(N_1, B, \text{Order} \rightarrow \text{Credit Verification})$
- $\mathsf{C}_4 = \mathsf{C}(B, N_1, \text{Credit Verification} \rightarrow \text{Credit Approved} \mid \text{Credit Rejected})$
- $\mathsf{C}_5 = \mathsf{C}(N_1, N_2, \text{Credit Approved} \rightarrow \text{Processing})$
- $\mathsf{C}_6 = \mathsf{C}(N_2, N_1, \text{Processing} \rightarrow \text{Ship within 24 hours})$

The CCD for this process is shown in Figure 3.5. The following shows a list of commitment operations that reflect significant domain level transactions.

- Create(C_1) = {Place an order}

- Update(C_1) = {Change order quantity, Change billing, ...}
- Discharge(C_1) = {Credit card charged}
- Cancel(C_1) = {Cancel order because the customer found cheaper deal, ...}
- Create(C_2) = {Confirm order}
- Update(C_2) = {Ship within 5 days because of back order, ...}
- Discharge(C_2) = {Confirm shipping}
- Cancel(C_2) = {Cancel Order because of item discontinued, ...}
- Create(C_4) = {Start verification process}
- Discharge(C_4) = {Credit approved | Credit rejected}
- Create(C_6) = {Start processing}
- Update(C_6) = {Process within 5 days because of back order, ...}
- Discharge(C_6) = {Send to shipping carrier}

From this case, we can see that the model consists of only 6 commitments but specifies as many as 14 activities. The dependencies among these activities follow commitment causality by default. The commitment creditors are always the receivers of the corresponding commitment operations. For example, the "Confirm shipping" means the customer receives an shipping confirmation from the seller. Compared to traditional workflows or business process specifications, we canefficiently put as many as activities into the same diagram by grouping them using the given commitment semantics.

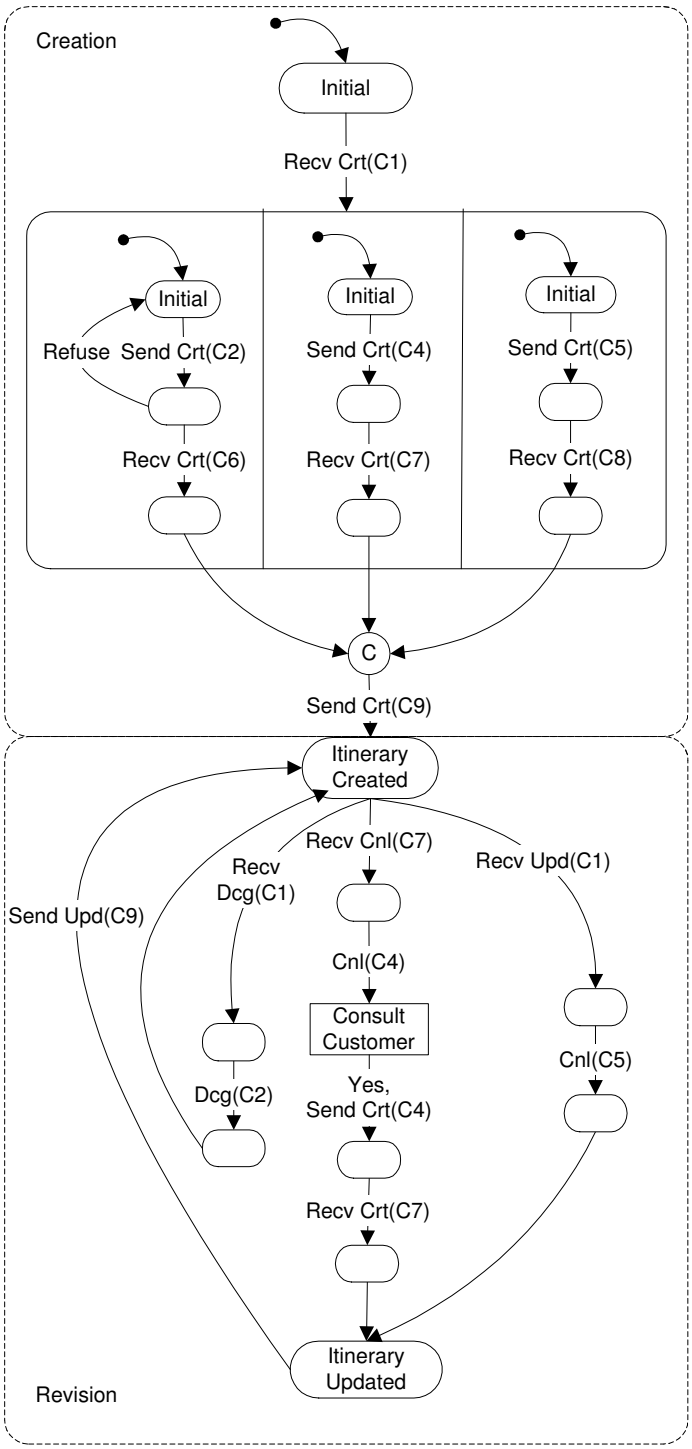

Figure 3.3: Travel agent behavior model statechart

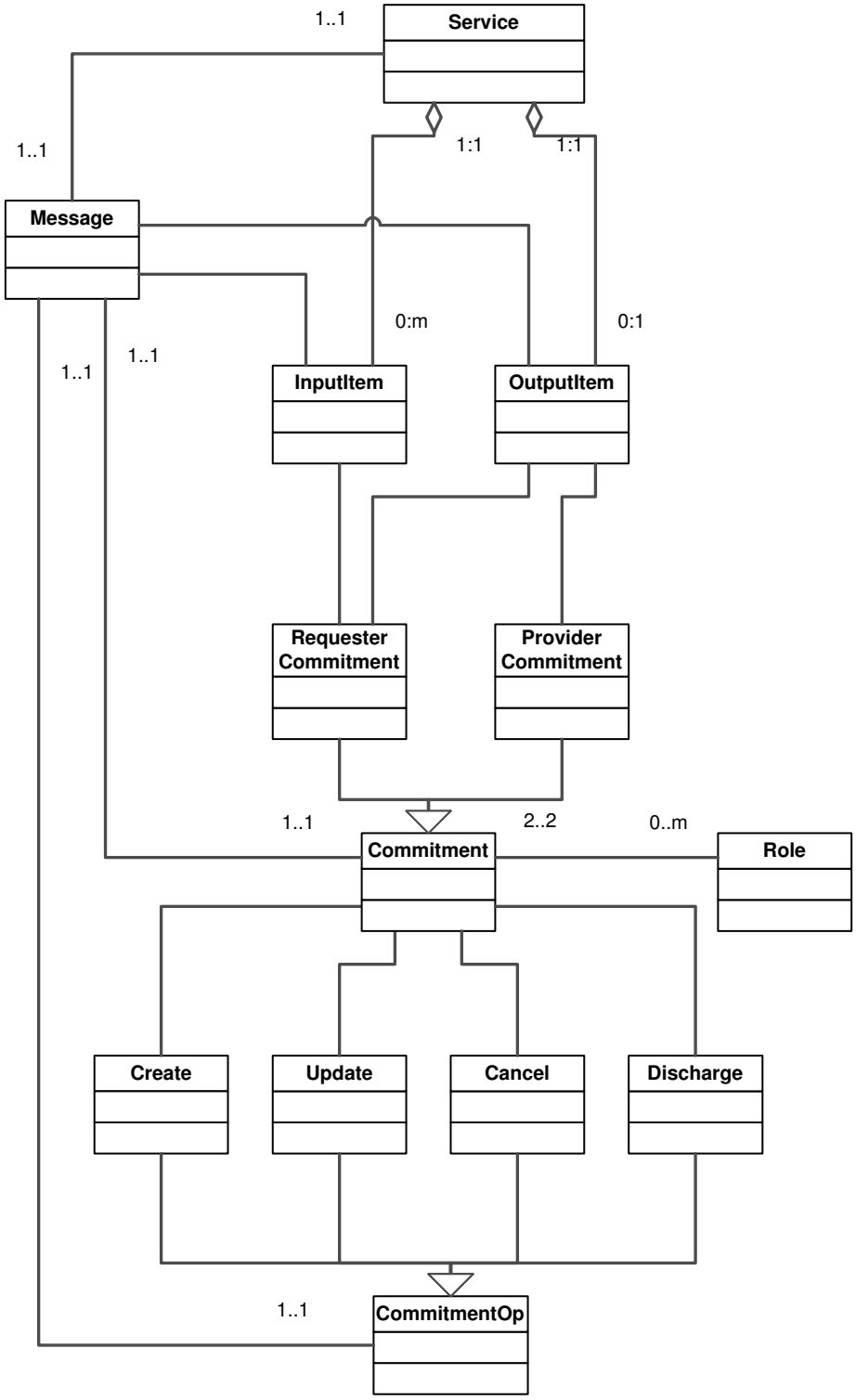

Figure 3.4: Commitment-based business process UML

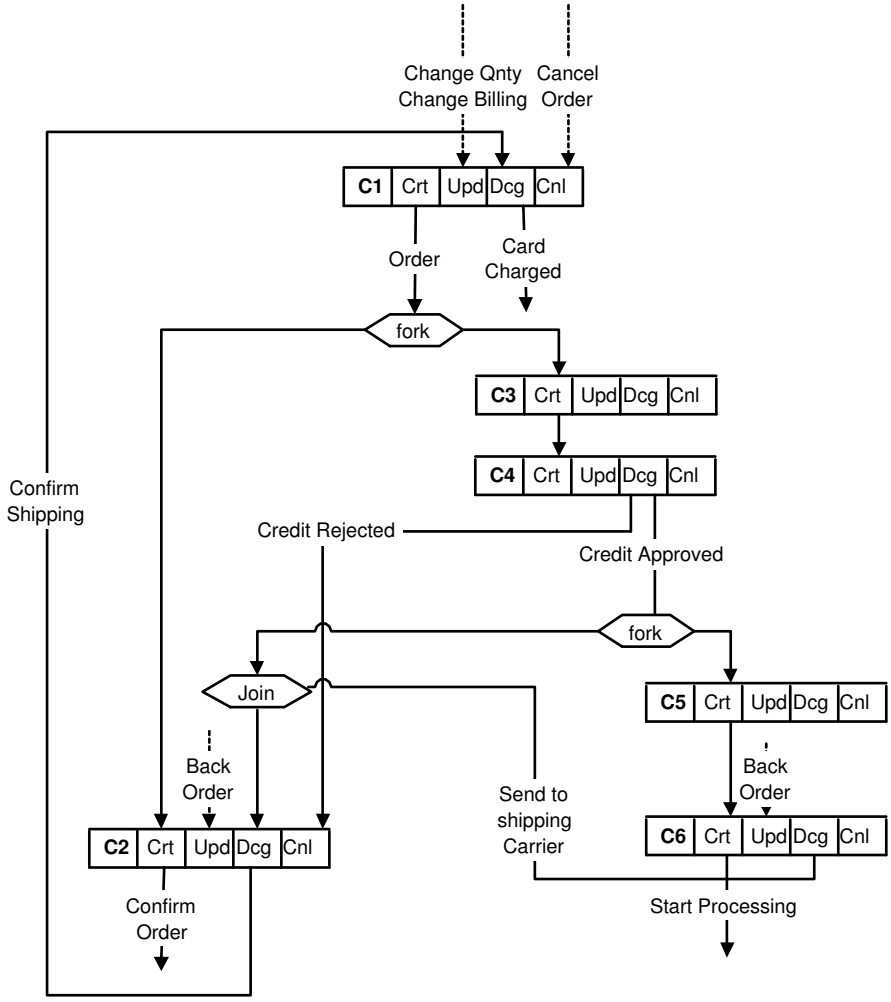

Figure 3.5: Case study

Chapter 4

Formalization in the π-Calculus

This chapter formalizes our commitment-based models using the π-calculus. The formulation enable us to automatically derive system behaviors, add business logics and verify model consistency based on the derivation rules of the π-calculus. It opens a direction on machine-assisted design and verification for our proposed approach.

4.1 π-Calculus Overview

The π-calculus is a process algebra capable of expressing mobile and concurrent systems [Sangiorgi and Walker, 2001]. The key concepts of the π-calculus are name binding and passing among concurrent processes. It provides four kinds of basic actions.

$$\pi \ ::= \ \overline{x}\langle\widetilde{y}\rangle \mid x(\widetilde{z}) \mid \tau \mid [\widetilde{z} = \widetilde{y}]\pi$$

Here $\overline{x}\langle\widetilde{y}\rangle$ sends tuple $\langle\widetilde{y}\rangle$ via name (or channel) x; $x(\widetilde{z})$ receives tuple $\widetilde{z}$ via name x; τ is an internal action not observable externally; and $[\widetilde{z} = \widetilde{y}]\pi$ means perform capability π if $\widetilde{z}$ and $\widetilde{y}$ are same. These actions form the basis of a complete process syntax defined as

below:

$$P \ ::= \ M \mid P \mid P' \ \mid \ \nu z P \mid !P$$
$$M \ ::= \ \mathbf{0} \mid \pi.P \mid M + M'$$

- **0** is a null process which does nothing.
- $\pi.P$ is a process with π as the first action; process P cannot be executed before π is performed.
- $M + M'$ is a choice between M and M'. Only one of them will proceed and the other will be nullified.
- $P|P'$ denotes two concurrent processes P and P'. They can execute independently but can also interact with each other through shared names.
- $\nu z P$ restricts the use of name z in P so that it cannot be shared outside P.
- $!P$ is a repetition of P where $!P \ = \ P \mid !P$.

A common reduction of π-calculus, called PR-INTER, is shown below: $(\overline{x}\langle\widetilde{y}\rangle.P_1 + M_1) \mid (x(\widetilde{z}).P_2 + M_2) \rightarrow P_1|P_2\{\widetilde{y}/\widetilde{z}\}$. In this reduction, both action $\overline{x}\langle\widetilde{y}\rangle$ and $x(\widetilde{z})$ are performed and $\widetilde{z}$ is bound to $\widetilde{y}$.

For simplicity, we define the following notation to receive a constant $Const$ through channel a (as mentioned above, τ stands for an internal basic action):

$$a(Const) \stackrel{def}{=} a(y).[y = Const]\tau$$

The constants in our models are any utterance u_i and any commitment C_j.

4.2 Formulating the Model

Now we show how each element of Dooley graphs can be represented in the π-calculus.

4.2.1 Characters

We use β to represent the process executed by characters and use the character name as its subscript. Since a conversation is a sequence of utterances, the character is a single process executing a sequence of actions. The syntax of β is $\beta ::= \pi.\beta \mid \mathbf{0}$, where π has the following three forms:

- $\overline{a}u$. Send utterance u through a channel named by the character's name a.
- $b(u)$. Receive utterance u through a channel named by the communicating character's name b.
- $\overline{Op_i}\mathsf{C}_i$. Send commitment C_i through channel Op_i where $Op_i \epsilon \{Crt_i, Upd_i, Dcg_i, Cnl_i\}$. The operation channels for each commitment are unique. This action notifies connectors (described in Section 4.2.6) that the commitment operation Op_i has performed on commitment C_i. $\overline{Op_i}\mathsf{C}_i$ is generated after a receiving action $b(u)$ in which utterance u maps to $\overline{Op_i}\mathsf{C}_i$. The mapping between utterances and commitment operations was studied previously in [Wan and Singh, 2003].

The character processes for the above example are

$$
\begin{aligned}
\beta_{P_1} &= \overline{P_1}u_1.T_1(u_{10}).\overline{Crt_8}\mathsf{C}_8.\mathbf{0} \\
\beta_{P_2} &= \overline{P_2}u_{11}.T_6(u_{13}).\overline{Upd_8}\mathsf{C}_8.\mathbf{0} \\
\beta_{P_3} &= T_7(u_{15}).\overline{P_3}u_{16}.T_7(u_{19}).\overline{Upd_8}\mathsf{C}_8.\overline{P_3}u_{20}.\mathbf{0} \\
\beta_{T_1} &= P_1(u_1).\overline{Crt_1}\mathsf{C}_1.\overline{T_1}u_{10}.\mathbf{0} \\
\beta_{T_6} &= P_2(u_{11}).\overline{Upd_1}\mathsf{C}_1.\overline{T_6}u_{13}.\mathbf{0} \\
\beta_{T_7} &= \overline{T_7}u_{15}.P_3(u_{16}).\overline{T_7}u_{19}.P_3(u_{20}).\overline{Dcg_1}\mathsf{C}_1.\mathbf{0} \\
\beta_{T_2} &= \overline{T_2}u_2.A_1(u_4).\mathbf{0} \\
\beta_{T_3} &= \overline{T_3}u_5.A_2(u_6).\overline{Crt_5}\mathsf{C}_5.\overline{T_3}u_{21}.\mathbf{0}
\end{aligned}
$$

$$
\begin{aligned}
\beta_{T_4} &= \overline{T_4}u_3.H_1(u_8).\overline{Crt_6}\mathsf{C}_6.H_1(u_{14}).\overline{Cnl_6}\mathsf{C}_6.\mathbf{0} \\
\beta_{T_8} &= \overline{T_8}u_{17}.H_2(u_{18}).\overline{Crt_6}\mathsf{C}_6.\mathbf{0} \\
\beta_{T_5} &= \overline{T_5}u_7.R(u_9).\overline{Crt_7}\mathsf{C}_7.\overline{T_5}u_{12}.\mathbf{0} \\
\beta_{A_1} &= T_2(u_2).\overline{Crt_2}\mathsf{C}_2.\overline{A_1}u_4.\mathbf{0} \\
\beta_{A_2} &= T_3(u_5).\overline{Crt_2}\mathsf{C}_2.\overline{A_2}u_6.T_3(u_{21}).\overline{Dcg_2}\mathsf{C}_2.\mathbf{0} \\
\beta_{H_1} &= T_4(u_3).\overline{Crt_3}\mathsf{C}_3.\overline{H_1}u_8.\overline{H_1}u_{14}.\mathbf{0} \\
\beta_{H_2} &= T_8(u_{17}).\overline{Crt_3}\mathsf{C}_3.\overline{H_2}u_{18}.\mathbf{0} \\
\beta_R &= T_5(u_7).\overline{Crt_4}\mathsf{C}_4.\overline{R}u_9.T_5(u_{12}).\overline{Cnl_4}\mathsf{C}_4.\mathbf{0}
\end{aligned}
$$

4.2.2 Roles

We use ρ to represent the process executed by a role. It is composed by the repetition of the choices of its character processes since at any time for a given role, only one character process can be engaged and the role can execute its character processes repeatedly. The subscripts of roles are the combination of two interacting characters. For example, one of the travel agent roles, $\langle T_1, T_6, T_7\rangle$, which interacts with role $\langle P_1, P_2, P_3\rangle$, is denoted as TP, and the corresponding customer role is denoted as PT. The role processes for the above example are

$$
\begin{aligned}
\rho_{\text{PT}} &= !(\beta_{P_1} + \beta_{P_2} + \beta_{P_3}) \\
\rho_{\text{TP}} &= !(\beta_{T_1} + \beta_{T_6} + \beta_{T_7}) \\
\rho_{\text{TA}} &= !(\beta_{T_2} + \beta_{T_3}) \\
\rho_{\text{TH}} &= !(\beta_{T_4} + \beta_{T_8})
\end{aligned}
$$

$$\begin{aligned}
\rho_{\text{TR}} &= !\beta_{T_5} \\
\rho_{\text{AT}} &= !(\beta_{A_1} + \beta_{A_2}) \\
\rho_{\text{HT}} &= !(\beta_{H_1} + \beta_{H_2}) \\
\rho_{\text{RT}} &= !\beta_R
\end{aligned}$$

4.2.3 Agents

We use α to represent the process executed by an agent that is composed by the actions performed by the roles it plays. The list of agent processes for the above example are

$$\begin{aligned}
\alpha_P &= \rho_{\text{PT}} \\
\alpha_T &= \rho_{\text{TP}} \mid \rho_{\text{TA}} \mid \rho_{\text{TH}} \mid \rho_{\text{TR}} \\
\alpha_A &= \rho_{\text{AT}} \\
\alpha_H &= \rho_{\text{HT}} \\
\alpha_R &= \rho_R
\end{aligned}$$

4.2.4 Conversations

A conversation is a sequence of utterances between two characters. We use ξ to represent the processes involved in a conversation. The conversation processes are the combination of the two interacting character processes. Consequently, the list of conversation processes for the above example are

$$\begin{aligned}
\xi_{\chi_1} &= \beta_{P1} \mid \beta_{T1}, & \xi_{\chi_2} &= \beta_{T2} \mid \beta_{A1} \\
\xi_{\chi_3} &= \beta_{T3} \mid \beta_{A2}, & \xi_{\chi_4} &= \beta_{T4} \mid \beta_{H1} \\
\xi_{\chi_5} &= \beta_{T5} \mid \beta_R, & \xi_{\chi_6} &= \beta_{P2} \mid \beta_{T6}
\end{aligned}$$

$$\xi_{\chi 7} = \beta_{P3} \mid \beta_{T7}, \qquad \xi_{\chi 8} = \beta_{T8} \mid \beta_{H2}$$

4.2.5 Conversation Segments

A conversation segment is a partition of a conversation that has one of the following three forms: $P.\overline{Op}\mathsf{C} \mid P'$, $P \mid P'.\overline{Op}\mathsf{C}$ or $P \mid P'$, where process P and P' are sequences of actions that do not contain any commitment operations and all the utterances involved in P or P' only respond to utterances within P and P'. Conversation segments either result in a transactional activity (a commitment operation), such as reserving a hotel or confirming a ticket; or a nontransactional activity, such as a query for information or negotiation about price. The dependencies among conversation segments are decided by connectors, as described in later sections.

We use σ to represent conversation segments and use both the conversation subscript and the segment number as the subscript of σ. We prefix each conversation segment with a receiving action TR_i to trigger its execution. The addition of the trigger action makes a conversation ξ differ from its segments. However, since the triggers are consumed when put in conjunction with the connectors, we can consider conversations and segments as equivalent when reasoning with connectors. The conversation segments for our example are

$$\begin{aligned}
\sigma_{11} &= \mathrm{TR}_{11}.(\overline{P_1}u_1.\mathbf{0} \mid P_1(u_1).\overline{Crt_1}\mathsf{C}_1.\mathbf{0}) \\
\sigma_{12} &= \mathrm{TR}_{12}.(T_1(u_{10}).\overline{Crt_8}\mathsf{C}_8.\mathbf{0} \mid \overline{T_1}u_{10}.\mathbf{0}) \\
\sigma_{21} &= \mathrm{TR}_{21}.(\overline{T_2}u_2.\mathbf{0} \mid T_2(u_2).\overline{Crt_2}\mathsf{C}_2.\mathbf{0}) \\
\sigma_{22} &= \mathrm{TR}_{22}.(A_1(u_4).\overline{Cnl_2}\mathsf{C}_2.\mathbf{0} \mid \overline{A_1}u_4.\mathbf{0}) \\
\sigma_{31} &= \mathrm{TR}_{31}.(\overline{T_3}u_5.\mathbf{0} \mid T_3(u_5).\overline{Crt_2}\mathsf{C}_2.\mathbf{0})
\end{aligned}$$

$$
\begin{aligned}
\sigma_{32} &= \mathrm{TR}_{32}.(A_2(u_6).\overline{Crt_5}\mathsf{C}_5.\mathbf{0} \mid \overline{A_2}u_6.\mathbf{0}) \\
\sigma_{33} &= \mathrm{TR}_{33}.(\overline{T_3}u_{21}.\mathbf{0} \mid T_3(u_{21}).\overline{Dcg_2}\mathsf{C}_2.\mathbf{0}) \\
\sigma_{41} &= \mathrm{TR}_{41}.(\overline{T_4}u_3.\mathbf{0} \mid T_4(u_3).\overline{Crt_3}\mathsf{C}_3.\mathbf{0}) \\
\sigma_{42} &= \mathrm{TR}_{42}.(H_1(u_8).\overline{Crt_6}\mathsf{C}_6.\mathbf{0} \mid \overline{H_1}u_8.\mathbf{0}) \\
\sigma_{43} &= \mathrm{TR}_{43}.(H_1(u_{14}).\overline{Cnl_6}\mathsf{C}_6.\mathbf{0} \mid \overline{H_1}u_{14}.\mathbf{0}) \\
\sigma_{51} &= \mathrm{TR}_{51}.(\overline{T_5}u_7.\mathbf{0} \mid T_5(u_7).\overline{Crt_4}\mathsf{C}_4.\mathbf{0}) \\
\sigma_{52} &= \mathrm{TR}_{52}.(R(u_9).\overline{Crt_7}\mathsf{C}_7.\mathbf{0} \mid \overline{R}u_9.\mathbf{0}) \\
\sigma_{53} &= \mathrm{TR}_{53}.(\overline{T_5}u_{12}.\mathbf{0} \mid T_5(u_{12}).\overline{Cnl_4}\mathsf{C}_4.\mathbf{0}) \\
\sigma_{61} &= \mathrm{TR}_{61}.(\overline{P_2}u_{11}.\mathbf{0} \mid P_2(u_{11}).\overline{Upd_1}\mathsf{C}_1.\mathbf{0}) \\
\sigma_{62} &= \mathrm{TR}_{62}.(T_6(u_{13}).\overline{Upd_8}\mathsf{C}_8.\mathbf{0} \mid \overline{T_6}u_{13}.\mathbf{0}) \\
\sigma_{71} &= \mathrm{TR}_{71}.(T_7(u_{15}).\overline{P_3}u_{16}.\mathbf{0} \\
&\qquad \mid \overline{T_7}u_{15}.P_3(u_{16}).\overline{\mathrm{TR}_{81}}.\mathbf{0}) \\
\sigma_{72} &= \mathrm{TR}_{72}.(T_7(u_{19}).\overline{Upd_8}\mathsf{C}_8.\mathbf{0} \mid \overline{T_7}u_{19}.\mathbf{0}) \\
\sigma_{73} &= \mathrm{TR}_{73}.(\overline{P_3}u_{20}.\mathbf{0} \mid P_3(u_{20}).\overline{Dcg_1}\mathsf{C}_1.\mathbf{0}) \\
\sigma_{81} &= \mathrm{TR}_{81}.(\overline{T_8}u_{17}.\mathbf{0} \mid T_8(u_{17}).\overline{Crt_3}\mathsf{C}_3.\mathbf{0}) \\
\sigma_{82} &= \mathrm{TR}_{82}.(H_2(u_{18}).\overline{Crt_6}\mathsf{C}_6.\mathbf{0} \mid \overline{H_2}u_{18}.\mathbf{0})
\end{aligned}
$$

4.2.6 Reentrant Connectors

Traditional task connectors, e.g., as in traditional process modelling languages [BPEL, 2003], specify the data or control flows among the business tasks that they connect. But, as already motivated, in many business applications, the processes are long-lived and the components are autonomous service providers. These service providers can raise exceptions or generate updates based on their local reasoning. For example, a customer may

update his trip itinerary or the hotel may cancel a reservation because of a problem at the intended facility. Existing models would treat such updates disjointly from the original execution, creating separate processes to handle them without explicitly relating them to the original transactions. Thus traditional approaches unnecessarily complicate the modelling and enactment of processes.

We introduce *reentrant connectors*, which not only control the initial process coordination requirement, but also deal with any updates sent or exceptions raised from the initial processes. The essence of our connectors is to allow processes to repeatedly deliver results through the same control logic until an entire transaction ends. To accomplish this, we use commitments as described above and let our connectors to coordinate on commitment operations to ensure that commitment causal relations are not violated. Reentrant connectors differs from the traditional ones in the following aspects.

- Both inputs and outputs of reentrant connectors are the commitment operations that convey commitment causal relations instead of process dependencies.
- The decision-making process in our connectors consists not only of Boolean logic, but also triggers for conversations. By engaging in conversations, agents are able to negotiate and collaborate to find optimal business values based on their local decisions. Thus we preserve agent autonomy.
- The operations performed on a commitment are fed to the same connectors where the original commitment is involved. In this way, activities related to the same processes are glued to the same connectors.

Figure 4.1 is the graphical illustration of a generic reentrant connector. Each connector contains two parts, one for initial creation and the other for revision. The initial creation process captures the business requirement level dependencies, which map to the service

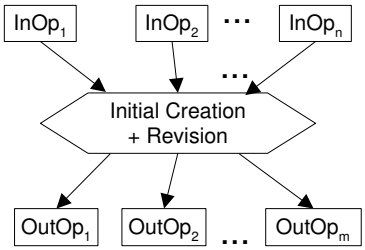

Figure 4.1: Reentrant connectors

request and response chain. The revision process handles updates, cancellations, and terminations occurring along the above established chains. In our model, since we use commitment to capture the stages of agent interactions, the revision processes can be naturally attached to the existing commitments, thus making multiagent system modelling clear and extensible.

Here we define three commonly used connector in the π-calculus.

- Join Connector (Γ_{join}). The join connector has multiple input commitment creation links and but only one output commitment creation link. Initially the join connector requires that all its input links verify creation of the corresponding commitments; the output would be a commitment creation as well. After the initial join, commitment Update, Discharge, and Cancel may come in through any input link. For Update, the connector may output a commitment update or discharge. For Discharge, the connector will output a commitment discharge only when all the input links have Discharge operations present. For cancellation, depending on business rules, the connector may output commitment cancellation, update, or nothing.

  As an example, the join connector executed by agent T on commitment C_5, C_6, and C_7 is expressed as follows,

$\Gamma_{join}(\langle \mathsf{C}_5, \mathsf{C}_6, \mathsf{C}_7 \rangle \rightarrow \mathsf{C}_8) =$

$$\begin{aligned}
&\nu d\ (\ Crt_5(C_5).\overline{S}d.\mathbf{0} \mid Crt_6(C_6).\overline{S}d.\mathbf{0} \mid Crt_7(C_7).\overline{S}d.\mathbf{0} \\
&\mid S(x).S(x).S(x).\overline{\mathrm{TR}_{12}} \\
&.!(\ Cnl_6(\mathsf{C}_6).\overline{Cnl_3}\mathsf{C}_3.\mathbf{0} + Crt_6(\mathsf{C}_6).\overline{\mathrm{TR}_{72}}.\mathbf{0} \\
&+ Upd_8(\mathsf{C}_8).\overline{\mathrm{TR}_{73}}.\mathbf{0} +\ Cnl_7(\mathsf{C}_7).\overline{\mathrm{TR}_{62}}.\mathbf{0}) \\
&)
\end{aligned}$$

The above connector only reflects the control logic of our running example. Initially, the connector waits for all the three commitment creations (by consuming three $S(x)$) and then executes conversation segment σ_{12} which creates commitment C_8. After the initial creation, it enters an infinite loop in which it may receive the cancellation of C_6 or C_7, or another creation of C_6. It then executes conversation segment σ_{71}, σ_{72}, or σ_{62} as appropriate, which lead to either the creation of C_3 or C_6 or an update of C_8.

- Fork Connector (Γ_{fork}). The fork connector has only one commitment creation input link but multiple output commitment creation links. Initially upon receiving a commitment creation on its input link, the fork connector triggers all commitment creations on its output links. After the initial execution, commitment Update, Discharge, and Cancel may come in through the input link. For update, depending on the stated business rules, it may affect part or all of its output commitments with either update or cancellation operations. For discharge, the connector should send commitment Discharges to all its output links. For cancellation, the connector should send cancellation to all its output links.

  As an example, the fork connector executed by agent T on commitment C_1 is

expressed as follows

$\Gamma_{fork}(\mathsf{C}_1 \rightarrow \langle \mathsf{C}_2, \mathsf{C}_3, \mathsf{C}_4 \rangle) =$

$$\nu d(Crt_1(C_1).\overline{S}d.\overline{S}d.\overline{S}d.!(Upd_1(C_1).\overline{\mathrm{TR}_{53}})$$
$$\mid S(x).\overline{\mathrm{TR}_{21}}.\mathbf{0} \mid S(x).\overline{\mathrm{TR}_{41}}.\mathbf{0} \mid S(x).\overline{\mathrm{TR}_{51}}.\mathbf{0})$$

Upon receiving a creation of C_1, the fork connector triggers all three conversation segments σ_{21}, σ_{41}, and σ_{51}, which in turn create commitments C_2, C_3, and C_4, respectively. Later, there is an update of C_1, which is justified to trigger conversation σ_{53} only. The latter cancels commitment C_4.

- Linear Connector Γ_l Linear connectors are the same as fork connectors but with only one output commitment creation link. As an example, the linear connector $\Gamma_l(\mathsf{C}_3 \rightarrow \mathsf{C}_6)$ is expressed as follows

$$\begin{aligned}\Gamma_l(\mathsf{C}_3 \rightarrow \mathsf{C}_6) &= Crt_3(\mathsf{C}_3).\overline{\mathrm{TR}_{42}}.!(Cnl_3(\mathsf{C}_3).\overline{\mathrm{TR}_{71}}.\mathbf{0} \\ &\quad + Crt_3(\mathsf{C}_3).\overline{\mathrm{TR}_{82}}.\mathbf{0})\end{aligned}$$

4.3 Reasoning on the Formulations

Using the formulations of all the elements given above, we now define the process for an entire system Υ, which is the combination of agent processes and connector processes: $\Upsilon ::= \alpha_1 | \ldots | \alpha_n | \Gamma_1 | \ldots | \Gamma_m$. From this definition, we can derive some useful properties and also verify system correctness.

4.3.1 Behavior Derivation

Given a commitment operation $Op(\mathsf{C})$ and a system Υ, we can derive consequent conversations and commitment operations. This gives a designer a means to verify whether

the multiagent system works as designed. As an example, to derive the consequent commitment operations of $Cnl_6(\mathsf{C}_6)$ (the hotel cancels the room reservation), we can start the reduction process from the revision part of $\Gamma_{join}(\mathsf{C}_5, \mathsf{C}_6, \mathsf{C}_7)$ since this connector covers the control logic of C_6. To save space, we only show the processes that are used in each derivation step and use "$\ldots$" to denote the remaining processes. We also put the action performed in each step on derivation arrows.

$Cnl_6(\mathsf{C}_6).\overline{Cnl_3}\mathsf{C}_3.\mathbf{0} \mid \ldots$

$$
\begin{aligned}
&\xrightarrow{\overline{Cnl_6}\mathsf{C}_6} && \overline{Cnl_3}\mathsf{C}_3.\mathbf{0} \mid Cnl_3(\mathsf{C}_3).\overline{\mathrm{TR}_{71}}.\mathbf{0} \mid \sigma_{71} \mid \ldots \\
&\xrightarrow{\overline{Cnl_3}\mathsf{C}_3} && \overline{\mathrm{TR}_{71}}.\mathbf{0} \mid \mathrm{TR}_{71}.(T_7(u_{15}).\overline{P_3}u_{16} \\
& && \qquad \mid \overline{T_7}u_{15}.P_3(u_{16}).\overline{\mathrm{TR}_{81}}) \mid \sigma_{81} \mid \ldots \\
&\xrightarrow{\overline{\mathrm{TR}_{71}}} && \overline{\mathrm{TR}_{81}} \mid \mathrm{TR}_{81}.(\overline{T_8}u_{17} \mid T_8(u_{17}).\overline{Crt_3}\mathsf{C}_3) \mid \ldots \\
&\xrightarrow{\overline{\mathrm{TR}_{81}}} && \overline{Crt_3}\mathsf{C}_3 \mid Crt_3(\mathsf{C}_3).\overline{\mathrm{TR}_{82}}.\mathbf{0} \mid \sigma_{82} \mid \ldots \\
&\xrightarrow{\overline{Crt_3}\mathsf{C}_3} && \overline{\mathrm{TR}_{82}}.\mathbf{0} \mid \mathrm{TR}_{82}.(H_2(u_{18}).\overline{Crt_6}\mathsf{C}_6 \mid \overline{H_2}u_{18}) \mid \ldots \\
&\xrightarrow{\overline{\mathrm{TR}_{82}}} && \overline{Crt_6}\mathsf{C}_6 \mid Crt_6(\mathsf{C}_6).\overline{\mathrm{TR}_{72}}.\mathbf{0} \mid \sigma_{72} \mid \ldots \\
&\xrightarrow{\overline{Crt_6}\mathsf{C}_6} && \overline{\mathrm{TR}_{72}}.\mathbf{0} \mid \mathrm{TR}_{72}.(T_7(u_{19}).\overline{Upd_8}\mathsf{C}_8 \mid \overline{T_7}u_{19}) \mid \ldots \\
&\xrightarrow{\overline{\mathrm{TR}_{72}}} && \overline{Upd_8}\mathsf{C}_8 \mid \ldots \\
&\xrightarrow{\overline{Upd_8}\mathsf{C}_8} && \ldots
\end{aligned}
$$

Now we know the chained effects of $\overline{Cnl_6}\mathsf{C}_6$ is $\overline{Cnl_3}\mathsf{C}_3 \xrightarrow{\overline{\mathrm{TR}_{71}}\overline{\mathrm{TR}_{81}}} \overline{Crt_3}\mathsf{C}_3 \xrightarrow{\overline{\mathrm{TR}_{82}}} \overline{Crt_6}\mathsf{C}_6 \xrightarrow{\overline{\mathrm{TR}_{72}}} \overline{Upd_8}\mathsf{C}_8$. Further, since σ_{71} and σ_{72} are the segments of conversation ξ_{χ_7}, and σ_{81} and σ_{82} are the segments of ξ_{χ_8}, and we have $\xi_{\chi_7} = \beta_{P3} \mid \beta_{T2}$ and $\xi_{\chi_8} = \beta_{T8} \mid \beta_{H2}$, we also know this transition involve four characters $\mathtt{P}_3$, $\mathtt{T}_2$, $\mathtt{T}_8$ and $\mathtt{H}_2$ and three agents P, T, and H.

4.3.2 Adding Business Logic

We can accommodate further business requirement variations by simply expanding the connector decision logics. For example, the customer may request a trip package (a bundled flight, hotel, and car) to get discount. If he cancels the car (reflected by the path $Upd_1(\mathsf{C}_1) \rightarrow Cnl_4(\mathsf{C}_4) \rightarrow Cnl_7(\mathsf{C}_7)$), the prices for his airline and hotel may be increased. We can modify the connector $\Gamma_{join}(\langle \mathsf{C}_5, \mathsf{C}_6, \mathsf{C}_7 \rangle \rightarrow \mathsf{C}_8)$ to implement this logic. In the consequent actions of $Cnl_7(\mathsf{C}_7)$, we introduce an additional action choice $\overline{Upd_2}\mathsf{C}_2.\overline{Upd_3}\mathsf{C}_3.\mathbf{0}$. Now the join connector becomes

$$\begin{aligned}
&\nu d\ (\ Crt_5(C_5).\overline{S}d.\mathbf{0} \mid Crt_6(C_6).\overline{S}d.\mathbf{0} \mid Crt_7(C_7).\overline{S}d.\mathbf{0} \\
&\qquad \mid S(x).S(x).S(x).\overline{\mathrm{TR}_{12}} \\
&.!(\ Cnl_6(\mathsf{C}_6).\overline{Cnl_3}\mathsf{C}_3.\mathbf{0} + Crt_6(\mathsf{C}_6).\overline{\mathrm{TR}_{72}}.\mathbf{0} + \ Upd_8(\mathsf{C}_8).\overline{\mathrm{TR}_{73}}.\mathbf{0} \\
&\qquad + Cnl_7(\mathsf{C}_7).(\overline{\mathrm{TR}_{62}}.\mathbf{0} + \overline{Upd_2}\mathsf{C}_2.\overline{Upd_3}\mathsf{C}_3.\mathbf{0})))
\end{aligned}$$

Note that the decision logic on which actions to take is not explicitly shown in the connector. In our model, we emphasize the potential commitment causality and conversation consequences, and leave local decisions to the agents. Doing so maximizes agent autonomy and heterogeneity.

4.3.3 Verifying Model Consistency

Definition 3. *A system Υ is consistent if and only if any runtime commitment instance is discharged within finite time.*

The definition says that for a successful execution of a business process, all the participants must eventually fulfill their commitments. This property is useful in a highly distributed and heterogeneous environment where a system can only be validated by verifying

the agents' commitments made to each other.

Theorem 5. *Assume a commitment will be discharged if its triggering commitment is discharged. Then if all commitment creation paths form a rooted-DAG (Directed Acyclic Graph) and there is an edge from one of the commitment creations to the discharge of the root commitment, then the system Υ is consistent.*

Proof. :

1 Let C_r be the tree root commitment and C_a be the commitment which discharges C_r.

2 Since all creation paths form a tree, there exists a behavior derivation that produces $\overline{Crt_r}\mathsf{C}_r \Rightarrow ... \Rightarrow \overline{Crt_a}\mathsf{C}_a$

3 We also have $\overline{Crt_a}\mathsf{C}_a \Rightarrow \overline{Dcg_r}\mathsf{C}_r$

4 Based on 2) and 3), we know C_r is discharged within finite time

5 Based on the assumption that, all the discharge paths also form a rooted-DAG. Therefore for any commitment C_i, there exists a behavior derivation which produces $\overline{Dcg_r}\mathsf{C}_r \Rightarrow ... \Rightarrow \overline{Dcg_i}\mathsf{C}_i$

6 Based on 4) and 5), all the commitments that are created because of $\overline{Crt_r}\mathsf{C}_r$ are also discharged within finite time.

□

This resembles a service request and response chain that is triggered by a single requester. When the root requester is satisfied by a commitment where it is the creditor, it discharges its commitment, which in turn discharges the entire chain of commitments.

Chapter 5

Applying the Approach: Building Multiparty Agreements

In a multiagent system, agents can autonomously decide on whether to perform or not perform a particular action. When agents coordinate with each other to achieve a global task, they need to first create a multiparty agreement, which would satisfy global goals as well as the agents' individual constraints. Multiparty agreements in agent communities are more subtle than in other software systems where agreements are represented by fixed protocols and each party has to follow the same execution sequences. Agents are able to perceive, reason and act so that they can more freely express themselves and flexibly interact with each other.

Researchers have studied multiparty agreements from several perspectives, such as developing FIPA standards [FIPA, 2003] for heterogeneous agents to communicate with and understand each other; implementing domain-specific protocols such as the fish market and auction protocols; approaches toward building software agents such as designing AUML [Odell et al., 2000] and statecharts [Harel et al., 1987]; and developing business processes

using BPEL [BPEL, 2003]. Existing approaches define interaction frameworks that limit agents' choices.

An example scenario is when a buyer wants to buy some goods from a seller. The buyer may require the seller to ship the goods before he would pay. The seller may require the buyer to pay before he would ship. Various approaches exist to resolve such situations in the real world. For instance, the buyer can make an advance deposit; or the buyer can give the seller his credit card but the seller will ship the goods before he charges the card; or the buyer and seller can use an escrow service to ensure successful execution of all the instructions. In the computer world, the different approaches lead to different protocols that each software agent must follow. This leads to two questions of interest. How can the agents derive different protocols to achieve agreement? What semantics must be incorporated into the agreements?

There is a rich literature on how agents form teams and negotiate on global execution plans, e.g., [Tambe, 1997]. Here we present another idea which starts from the representation of multiparty agreements based on commitments. As explained above commitments represent the obligations made between pairs of agents and are used to model interactions in a multiagent system. Commitments help agents express promises and monitor each other's compliance without regard to internal details. This chapter uses commitments as the basic elements to form multiparty agreements. Next, it detects potential deadlocking constraint dependencies and resolves them by executing selected protocols that are proposed here.

5.1 Representing Multiparty Agreements using Commitments

Let's first discuss the basic forms of commitments of relevance to agreements. A multiparty agreement is expressed by a set of the following four basic commitments. These commitments are differentiated by the preconditions that need to be satisfied for the debtor to make a commitment to its creditor.

F_1 : Unconditional Commitment $\mathsf{C}(x, y, q)$.

Agent x commits to y to perform q unconditionally.

For example, $\mathsf{C}(seller, buyer, ShipGoods)$.

F_2 : Action-Triggered Commitment $\mathsf{C}(x, y, p \rightarrow q)$.

Agent x commits to y to perform q only if p happens.

For example, $\mathsf{C}(seller, buyer, InStock \rightarrow ShipGoods)$.

F_3 : Unconditional Commitment-Triggered Commitment $\mathsf{C}(x, y, \mathsf{C}(z, x, p) \rightarrow q)$.

Agent x commits to y to perform q only if agent z commits to x to perform p.

For example, $\mathsf{C}(seller, buyer, \mathsf{C}(buyer, seller, Pay) \rightarrow ShipGoods)$. Here z = y = $buyer$.

F_4 : Conditional Commitment-Triggered Commitment $\mathsf{C}(x, y, \mathsf{C}(z, x, p \rightarrow r) \rightarrow q)$.

Agent x commits to y to perform q only if agent z commits to x to perform r when p happens.

For example, $\mathsf{C}(seller, buyer, \mathsf{C}(buyer, seller, ShipGoods \rightarrow Pay) \rightarrow ShipGoods)$.

Here z = y = $buyer$.

Based on the commitment definition described in Chapter 2, we can see that form F_1 is an unconditional commitment and form F_2, F_3 and F_4 are conditional commitments.

Classifying conditional commitments into the three forms enable us to study the subtleties of the causal dependencies. For example, the fulfillment of F_2 relies on a concrete action to be performed, but the fulfillment of F_3 relies only on a promise that is built upon the trust between the two agents. F_4 also relies on promises but is weaker than F_3 since the promise is conditional and may not be fulfilled if the condition does not hold.

Definition 4. *A multiparty agreement A is given by a set of commitments $\{\mathsf{C}_1, \mathsf{C}_2, \cdots, \mathsf{C}_n\}$ where $\mathsf{C}_i \epsilon \{F_1, F_2, F_3, F_4\}$*

The following two commitments C_1 and C_2 forms our sample agreement $A = \{\mathsf{C}_1, \mathsf{C}_2\}$, in which a buyer commits to a seller that if the seller commits to ship the goods then the buyer will pay for it, and the seller commits to the buyer to ship the goods.

$$\begin{aligned} \mathsf{C}_1 &= \mathsf{C}(buyer, seller, \mathsf{C}(seller, buyer, ShipGoods) \rightarrow Pay) \\ \mathsf{C}_2 &= \mathsf{C}(seller, buyer, ShipGoods) \end{aligned}$$

5.1.1 Derivation Rules

Here we give a set of rules to reduce an agreement (or a commitment set) to a set of conditions that all the agents would eventually bring about. The purpose of the inferences is to show how the interactions progress given a commitment set. This will also give us a way to detect potentially deadlocking agreements. For simplicity, we do not consider the *cancel* and *update* operations, which usually digress from normal executions and do not help in detecting deadlocks introduced by the original commitment set.

In the following rules, the notation $p \xrightarrow{t} q$ states that if p happens, then q must happen within a finite time t, otherwise it is a violation.

$E_1 \quad : \quad Discharge(\mathsf{C}(x, y, q)) \Rightarrow q$

$$E_2 \;:\; Discharge(\mathsf{C}(x, y, p \rightarrow q)) \Rightarrow q$$

$$E_3 \;:\; Create(\mathsf{C}(x, y, p)) \Rightarrow Discharge(\mathsf{C}(x, y, p))$$

$$E_4 \;:\; p \wedge Create(\mathsf{C}(x, y, p \rightarrow q)) \Rightarrow Discharge(\mathsf{C}(x, y, p \rightarrow q))$$

$$E_5 \;:\; Create(\mathsf{C}(z, x, p)) \wedge Create(\mathsf{C}(x, y, \mathsf{C}(z, x, p) \rightarrow q))$$
$$\Rightarrow Discharge(\mathsf{C}(x, y, \mathsf{C}(z, x, p) \rightarrow q)) \xrightarrow{t} Discharge(\mathsf{C}(z, x, p))$$

$$E_6 \;:\; Create(\mathsf{C}(z, x, p \rightarrow r)) \wedge Create(\mathsf{C}(x, y, \mathsf{C}(z, x, p \rightarrow r) \rightarrow q))$$
$$\Rightarrow Discharge(\mathsf{C}(x, y, \mathsf{C}(z, x, p \rightarrow r) \rightarrow q)) \wedge (p \xrightarrow{t} Discharge(\mathsf{C}(z, x, p \rightarrow r)))$$

E_1 and E_2 show that the results of the *discharge* of both unconditional and conditional commitments are the conditions or actions they bring about; E_3 states that an unconditional commitment will be eventually discharged after it is created. E_4 states that an action-triggered commitment will be discharged after the action occurs; E_5 states that an unconditional commitment-triggered commitment will be discharged after the unconditional commitment is created, and this discharge eventually triggers the discharge of the unconditional commitment; E_6 states that a conditional commitment-triggered commitment will be discharged after the conditional commitment is created, and the conditional commitment must be discharged if its condition is eventually satisfied.

5.1.2 An Example Derivation

Based on the above rules, we have the following derivation for our example agreement A:

$$\begin{aligned} A \;\longrightarrow\;& Create(\mathsf{C}(seller, buyer, ShipGoods)) \wedge \\ & Create(\mathsf{C}(buyer, seller, \mathsf{C}(seller, buyer, ShipGoods) \rightarrow Pay)) \\ \xrightarrow{E_5}\;& Discharge(\mathsf{C}(buyer, seller, \mathsf{C}(seller, buyer, ShipGoods) \rightarrow Pay)) \xrightarrow{t} \end{aligned}$$

$$
\begin{aligned}
& Discharge(\mathsf{C}(seller, buyer, ShipGoods)) \\
\xrightarrow{E_2} \; & Pay \xrightarrow{t} Discharge(\mathsf{C}(seller, buyer, ShipGoods)) \\
\xrightarrow{Pay} \; & Discharge(\mathsf{C}(seller, buyer, ShipGoods)) \\
\xrightarrow{E_1} \; & ShipGoods
\end{aligned}
$$

By collecting all the actions or conditions on the derivation arrows, which is Pay in this example, and the results of the derivation, which is ShipGoods, we see that both actions in the two commitments have been performed. Thus we say this agreement is satisfiable.

5.2 Building CCD from Commitment Sets

In the previous chapter, we introduced an algorithm to generate a CCD from a given agent conversation based on speech act semantics. Here we introduce Algorithm 4, which builds a CCD from a given commitment set. In this algorithm, the causality is derived only for creation and discharge operations and we manually add the update and cancel operations later. This is different from the previous algorithm, where the causality among updates and cancellations can also be induced from the conversations. The CCD generated by Algorithm 4 directly represents the original business agreements and can help verify the satisfiability of a business process design. The symbol $*$ refers to a wildcard that matches any agent or condition. Figure 5.1 shows the CCD derived for our sample agreement A.

5.3 Building Satisfiable Agreements

In a multiagent system, each individual agent can have its local constraints. The agents' commitments not only specify their protocols, but also factor in their local constraints (which in essence limit what the agents can promise others). However, since the agents

```
1 Draw a node for each commitment with its identifier and two operations, Crt and
  Dcg;
2 for each commitment C_i do
      switch C_i do
3         case C(x, y, p) or C(x, y, * → p)
              For any commitment C_j where C_j = C(y, *, p → *) or
              C(y, *, C(x, y, p) → *), add an edge from Dcg(C_i) to Dcg(C_j);
4         case C(x, y, p → *) or C(x, y, C(*, x, p) → *)
              For any commitment C_j where C_j = C(*, x, p) or C(*, x, * → p), add
              an edge from Crt(C_i) to Crt(C_j);
5         case C(x, y, C(*, x, * → r) → q)
              For any commitment C_j where C_j = C(*, x, * → r), add an edge from
              Crt(C_i) to Crt(C_j);
```

Algorithm 4: Building CCD from Commitment Sets

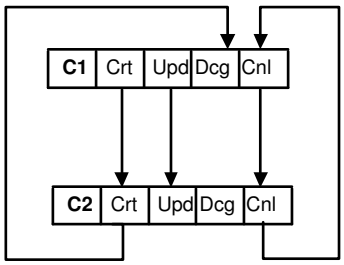

Figure 5.1: An Example of Commitment Causality Diagram

are autonomous, the constraints of different agents may form cyclic dependencies. Let us return to the example described above. A buyer wants the seller to ship the goods first before he makes the payment, but the seller may want the buyer pays first before he ships the goods. The two commitments can be expressed as below.

$$\mathsf{C}_1 = \mathsf{C}(buyer, seller, ShipGoods \rightarrow Pay)$$

$$\mathsf{C}_2 = \mathsf{C}(seller, buyer, Pay \rightarrow ShipGoods)$$

Apparently, neither party will proceed because of the deadlocking dependencies. Our

goal is to detect these cyclic constraint dependencies and propose several protocols to resolve them and produce a satisfiable commitment set. Here we first define what is a satisfiable multiparty agreement.

Definition 5. *A multiparty agreement is satisfiable if and only if for any* $\mathsf{C}(x_i, y_i, q_i)$*,* q_i *will eventually become true or is performed; or for any* $\mathsf{C}(x_i, y_i, p_i \rightarrow q_i)$*,* q_i *will eventually become true or is performed if* p_i *becomes true.*

We have an intuition that, if the CCD derived from a commitment set has cycles involving either *create* or *discharge* nodes, then the preconditions of the commitments on those cycles form deadlocking dependencies. This means that no condition would be brought about, so that the commitment set is not satisfiable. Therefore, we have the following theorem.

Theorem 6. *If the CCD derived from a multiparty agreement shows cycles on its* create *or* discharge *nodes, then the corresponding multiparty agreement is not satisfiable.*

Proof. 1. First we define $p(t)$ as true or false at a time t

2. If there is a cycle on a set of *create* (or *discharge*) nodes, then based on case 4 (or case 3 for discharge node) in Algorithm 4, we have a commitment set $\{\mathsf{C}_1, \mathsf{C}_2, \ldots, \mathsf{C}_n\}$ where

$$\begin{aligned} \mathsf{C}_1 &= \mathsf{C}(x_1, x_2, p_1 \rightarrow p_2) \\ \mathsf{C}_2 &= \mathsf{C}(x_2, x_3, p_2 \rightarrow p_3) \\ \ldots \\ \mathsf{C}_n &= \mathsf{C}(x_n, x_1, p_n \rightarrow p_1) \end{aligned}$$

3. Assume C_1 will eventually be discharged. Then we can find a time t_m, such that $p_2(t_m)$;

4. Based on derivation rule E_4, we can find a time t_1, $t_1 < t_m \wedge p_1(t_1)$, and $\forall t$, $t \leq t_1 \wedge \neg p_2(t)$.

5. For the same reason, we can find a time t_2, $t_2 < t_1 \wedge p_n(t_2)$ and $\forall t$, $t \leq t_2 \wedge \neg p_n(t)$.

6. Repeating step (5), we can find a time t_n, $t_n < t_{n-1} \wedge p_2(t_n)$ and $\forall t$, $t <= t_n \wedge \neg p_3(t)$.

7. From steps (4) through (6), we have $t_n < t_1 \wedge p_2(t_n)$. This result conflicts with "$\forall t$, $t \leq t_1 \wedge \neg p_2(t)$" in (4). Therefore, C_1 can not be discharged and the same conclusion applies to C_2, ..., C_n.

8. Since no commitment on any cycle can be discharged, the multiparty agreement is not satisfiable.

□

Now we would like to examine the satisfiability of the above example. By executing Algorithm 4 on the above commitment set, we obtain the CCD shown in Figure 5.2.

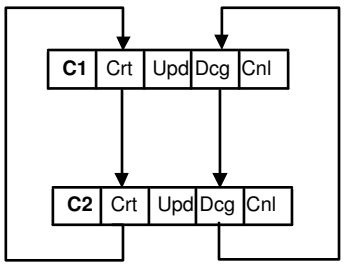

Figure 5.2: A deadlocking agreement

From the CCD, we can see that both the *create* and the *discharge* paths are cyclic. Therefore, this commitment set is not satisfiable.

Deadlocking constraints imposed on a group of agents do not mean that these agents cannot engage activities at all. Autonomous agents can negotiate to serve their interests.

To break these deadlocking dependencies, all the agents may choose to commit what they promise to do for the others regardless of what constraints they impose on the others, or some of the agents may concede to satisfy the others first before their own constraints are satisfied. All these approaches lead to a variety of protocols for forming satisfiable multiparty agreements.

For the sake of simplicity, this dissertation only introduce the protocols that resolve the constraint conflicts created from the action-triggered commitments (F_2). For generality, we choose a three commitments scenario shown below:

$$\mathsf{C}(x, y, p \rightarrow q), \mathsf{C}(y, z, q \rightarrow r), \mathsf{C}(z, x, r \rightarrow p)$$

By applying Algorithm 4, we can tell that the *create* and *discharge* path are cyclic and, therefore, a deadlock exists. The following protocols describe different approaches to resolve this inconsistency.

2PC Protocol. This protocol is similar to the one widely used in database systems where task executors either all commit or all abort their transactions to achieve task atomicity and preserve system consistency [Gray and Reuter, 1993]. For our research, we apply this protocol to agents who have deadlocking constraints, which prevents them from discharging any commitment to each other. The goal of the protocol is to make sure that all the involved agents commit first before their preconditions are met. Our 2PC protocol only resolves the cyclic *discharge* path since, in this scenario, the discharge of one commitment will satisfy the precondition of another commitment and we can let all the discharges unconditionally happen so that all commitments will be fulfilled. The following shows the steps of the 2PC protocol.

By executing the 2PC protocol, q, r, and p will be unconditionally performed by x, y, and z, respectively. Once these conditions become true, they also satisfy each

1 A coordinator tells all the agents that are involved in a cyclic *discharge* path that a 2PC protocol is started;
2 Each agent sends yes or no to indicate whether it wants to unconditionally discharge its commitments or not;
3 If all the answers are yes, then the coordinator sends yes to each agent. Then each agent replaces its conditional commitment with a corresponding unconditional commitment by removing the preconditions;
4 If at least one answer is no, then the coordinator sends no to each agent. That is, no solution is found.

Algorithm 5: 2PC protocol

precondition in the above commitments. In terms of this aspect, the 2PC protocol essentially converts all the conditional commitments to their corresponding unconditional commitments (provided all agents are willing). Therefore, the above three commitments become

$$\mathsf{C}(x, y, q), \mathsf{C}(y, z, r), \mathsf{C}(z, x, p)$$

An assumption of the 2PC protocol is that, for any commitment $\mathsf{C}(x, y, p \rightarrow q)$, p is not required to happen before q, but must happen eventually. However, some commitments may require that p happens before q can happen. In this case, not all the agents are willing to commit unconditionally. Then we need to seek other protocols to resolve the conflicts.

Unconditional Yield. If one agent is willing to convert its conditional commitment to an unconditional commitment, we say that this agent yields unconditionally. In the above example, agent x may promise y to perform q without being satisfied by p first. This usually happens when the debtor of p, which is z in this example, has developed enough credit with x, which makes the latter believe that p will be eventually performed by z, even after x's unilateral concession. Here we construct a protocol

to convey x's intention and propagate it to other agents to make corresponding commitment changes.

1 A coordinator notifies all the agents that are involved in a cyclic *discharge* path and an Unconditional Yield protocol is started;
2 Each agent sends yes or no to indicate whether it wants to unconditionally discharge its commitments or not;
3 If at least one agent answers yes, then the coordinator picks the first agent (say agent x) who answers yes and notifies all the agents;
4 The agent x converts its action-triggered commitment (F_2) to an unconditional one (F_1) and all other agents will keep their commitments (F_2) unchanged.

Algorithm 6: Unconditional yield protocol

By executing the protocol on the above example, the three commitments are changed to

$$\mathsf{C}(x, y, q),\ \mathsf{C}(y, z, q \rightarrow r),\ \mathsf{C}(z, x, r \rightarrow p)$$

Agent x will commit q unconditionally to y based on its belief that agent z will eventually commit p to it if r happens.

Conditional Yield. Conditional yield is similar to unconditional yield, but differs in that the agent willing to make an unconditional commitment does not have enough trust in any other agents. It must conditionally rely upon other agents' promises to it before it can perform its action. In such a case, the agent will replace its action-triggered commitment with a conditional commitment-triggered commitment. The protocol is described as Algorithm 7.

By executing the protocol on the above example, the three commitments are changed to

$$\mathsf{C}(x, y, \mathsf{C}(z, x, r \rightarrow p) \rightarrow q),\ \mathsf{C}(y, z, q \rightarrow r),\ \mathsf{C}(z, x, r \rightarrow p)$$

1 A coordinator notifies all the agents that are involved in a cyclic *discharge* path and a Conditional Yield protocol is started;
2 Each agent sends yes or no to indicate whether it want to conditionally discharge its commitments or not;
3 If at least one agent answers yes, then the coordinator picks the first agent (say, agent x) who answers yes and notifies the results to all the agents;
4 The agent x converts its action-triggered commitment (F_2) to a conditional commitment (F_4) and all other agents will keep their commitments (F_2) unchanged.

Algorithm 7: Conditional yield protocol

Looking back the two conflicting commitments shown in section 5.3, the Conditional Yield protocol will produce the following outcome.

$$\begin{aligned} \mathsf{C}_1 &= \mathsf{C}(buyer, seller, \mathsf{C}(seller, buyer, Pay \rightarrow ShipGoods) \rightarrow Pay) \\ \mathsf{C}_2 &= \mathsf{C}(seller, buyer, Pay \rightarrow ShipGoods) \end{aligned}$$

This result shows that the buyer yields to the seller to pay first by taking the promise from the seller saying if the buyer pays then the seller ships goods.

To improve efficiency, in the future work, we will study decentralized protocols which allow agents to relax their constraints without a centralized coordinator and the voting processes. In such decentralized settings, we will need to consider protocol safety to prevent agents who do not have global views of constraints from discharging their commitments prematurely, which may lead to inconsistencies.

Chapter 6

Discussion

Our contribution is to bring commitment semantics into business process modelling and to enable agent collaborationss during process enactment while preserving their autonomy and heterogeneity. The new model not only specifies the services that the agents provide, but also captures how the agents interact. Commitments can be treated as common states during agent interaction, which represent the agreements that are achieved in the interaction at any given point. Any update, fulfillment, cancellation, or modification of the commitments affects how the services are delivered. By capturing the commitments in supply chains, we can control the lifecycle of service delivery and are able to handle most business processes.

Our base technology starts from agent conversations tables. Given an agent interaction marked up with the various columns of a conversation table, as in Table 2.1, our approach first extracts commitments and commitment relations from it. It then generates the agent models by fine-tuning the relation diagrams. Our approach enables designers to model a commitment-centric agent system based on an understanding of how the agents interact with each other by making, fulfilling, and modifying commitments. The main idea of this

approach is in designing the commitment lifecycle in terms of the causal relations among the commitments. Each commitment must be taken care of by its creator and any operation performed on a commitment must be properly propagated to other commitments to ensure that inconsistencies do not arise.

Our approach fits into a methodology that involves a combination of heuristic reasoning by a human designer (e.g., creating and marking up example interactions; enriching a CCD by hand if desired; coding domain-specific rules) and algorithmic reasoning (e.g., deriving a CCD and generating agent models). In this way, it goes beyond pure methodological approaches, which concentrate on the heuristic element.

Here is the summary of the key ideas in this research

1. Model business process as networks of commitments. A commitment is defined as a directed obligation from one party to another. Commitments can
   (a) Naturally represent rules and causality, and can be manipulated on the fly.
   (b) Help specify, hold, and verify promises that are key to multiparty collaboration.
2. Abstract and encapsulate causally related processes using commitments.
3. Specify revisions and exceptions in a controlled manner.
4. Capture business agreements and protocols via commitment causality, which reduce the gap between the requirements and designs.
5. Benefits of using commitments
   (a) Flexible process executions, since the commitment lifecycle support multiple execution paths.

(b) Adaptive process modelling, since the commitments can accommodate changing business requirements.

6.1 Literature

When we bring autonomous agent into business process modelling, we are mostly dealing with agent-oriented software engineering (AOSE) [Jennings, 2000]. AOSE has been a popular topic for building electronic commerce systems and global supply chains. Recent approaches have considered the purely methodological aspects of AOSE. Our program of research seeks to develop rigorous methodologies for AOSE and for multiagent systems design with the view of supporting AOSE. Our contribution here is on the technical aspects of how to create commitment-based representations.

Parunak originated the idea of using Dooley graph to decompose agent conversation and differentiate courses of stages, so agent interaction can be modularized and reused [Parunak, 1996]. This is a good starting point to scoping characters and conversations, and helpful for designing an agent behavior model. We extend this approach by adding more communicative acts (cancel request) and an additional relation among utterance (*update*), so agent conversations become more meaningful in terms of the relations with each other and business process requirement can be more accurately captured.

Singh previously found that the Dooley graph is capable of deriving skeletons for heterogeneous agents [Singh, 2000b]. However, the main results dealt with the low-level agent interaction protocols. Our study found that the commitment dependencies are more useful than commitments themselves since the lifecycle of each commitment relies on the relations with other commitments and actions. They could be nested deeply within each other commitments. Any break in the commitment dependency chain could affect all related

commitments. Therefore, the major advantage of our approach over Singh's is to generate agent models from commitment relations, but not from the message sequences alone.

Huhns *et al.* use Dooley graphs to deal with exception handling in supply chains [Huhns et al., 2002]. After deriving an agent conversation and an agent model, they add branches to the agent skeletons to capture possible exceptions. However, the effects of agent internal exceptions on other agents' behavior are not clearly captured. Our approach generalizes exception handling into two commitment operations, cancel and update. Our commitment causal relation diagram shows what are the chains of effects of these operations, so the designer may have a generic way to deal with different types of exceptions that arise in the system.

Fornara and Colombetti [Fornara and Colombetti, 2002] present an operational semantics of commitments relating to communicative acts. They map each communicative act to a commitment operation and depict a commitment state diagram during a course of conversation. This helps discover the protocols of making and fulfill commitments within the given conversations and differentiating states in the negotiation process. However, like other commitment-based approaches, this one still deals with commitments between two agents. The relations among commitments are not explored.

Verdicchio and Colombetti recently presented ideas of generalizing commitments relations in supply chain systems [Verdicchio and Colombetti, 2002]. However, they are limited to money, process, and information flows. How each commitment operation influences the chain of commitments is not clearly specified. Such relationships are accommodated and exploited by our approach.

Economou *et al.*'s approach is similar to ours, but for a different purpose [Economou et al., 2001]. Their premise is that the protocol is fixed. If certain deontic states are entered, then the corresponding commitments have to be fulfilled. Since our approach captures the

commitment causal relations and is intended to help for protocol design, the deontic states can always be changed. However, as the design is completed, the protocol is fixed. Thus it becomes possible to accommodate some of the topics that Economou *et al.* study, e.g., inferring commitments from protocols.

Agent UML (AUML) provides various kinds of diagrams for a designer to specify multiagent system behavior and internal agent models [Odell et al., 2000]. AUML is weak in its handling of multiagent concepts and abstractions. Our commitment causality diagram can be a complementary, alternative way to specify system requirements. Adding CCDs to AUML would enhance the expressiveness of AUML to a commitment level.

The π-calculus has found serious application in software engineering [Achermann, 2002] and business process modelling [Mehta et al., 2001] [Gwen Salan, 2004]. These works use the π-calculus to model computations, but conceptually the computations they consider do not have any special features geared to open environments. In particular, they lack anything analogous to manipulable commitments or reentrance. We use π-calculus to provide a theoretical foundation to our commitment-based approach. This formalization better uses the power of the π-calculus to capture the essence of higher-level abstractions. Using the π-calculus helps us not only derive useful properties on the relations among characters, roles, agents, commitments, and connectors, but also verify the correctness of a model. Petri nets are another prominent modelling approach for concurrent systems [van der Aalst and van Hee, 2002]. We find the algebraic nature of the π-calculus to be more natural for representing commitments than the graph representation of Petri nets.

The key strength of Web services lies in enabling method invocation and service requests, using established standards such as HTTP and XML, so that heterogeneous systems can communicate with each other and participate in distributed business processes [Buhler and Vidal, 2004] [Jin, 2004] [Petrone, 2004]. One of the major research challenges for the

expansion of Web services is the ability to enable additional expressive power in models while maintaining simple service descriptions and hiding the implementation complexity of service providers. This is because a complicated service description adds complexity to service composition and workflow management, and would yield a rigid system vulnerable to exceptions. When we introduce commitments and reentrant connectors, we allow processes to be correlated under the same commitments and coordinated by the original connectors so as to reduce the complexity of the composition specifications. This makes the entire process coherent even as it is modelled in a manner that can be readily enacted by autonomous, heterogeneous agents.

Business process and web service choreography standards have evolved for several years, e.g., see [Curbera et al., 2002] [zur Muehlen et al., 2004]. Some recent work on Web service composition can be found in [Fu et al.] [Desai and Singh] [Melloul and Fox]. In such models, each agent is designed as service provider [McIlraith et al., 2001]. Agents can receive and send service requests to each other by following the protocol specified for each service. They never deliver services to others without being requested. A group of service providers can form a supply chain system if the services they provide are involved in a global business process. Service-based designs are a popular approach in today's ebusiness systems. However, in terms of the adaptability and flexibility, practitioners are still struggling for a better interaction model to make a supply chain more robust, because current approaches lack the abstractions to handle different protocols and the data interfaces needed to accommodate different business situations. In previous research [Wan et al., 1999, Xing et al., 2001], we proposed incorporating a set of commitment patterns into an agent-enabled workflow system to enhance system interoperability and modularity. Here we introduce commitments into what may be thought of as a workflow system enabled by Web services. Commitments enable us not only to record the service delivery history, but

also to specify what properties or conditions must be maintained. If any of these conditions is violated, we can always trace down the cause and restore the system to a consistent state again. Therefore, embedding commitments into service descriptions is also key to enabling reliable and persistent Web services.

A recent development in teamwork theory [Pynadath and Tambe, 2003] presents the idea of forming teams among heterogeneous agents who have no coordination capabilities in a manner that ensures robust execution of the teamwork. The authors use a proxy called Teamcore to wrap those standalone agents with teamwork capabilities so that any type of agents can join a team and collaborate with others to achieve a global goal. This research differs from ours in that it searches for passive agents with special capabilities and convert them to team members, whereas ours is based on active agents who seek services from or do business with other agents. It is these active agents who decide how to perform a team task instead of a team organizer who picks agents and imposes specific behaviors on them.

The research of constraint satisfaction problems (CSP) in distributed systems, e.g., [Liu et al., 2002], has shown much promise. The models behind CSP are computational since the entire multiagent environment and individual agents are represented by variables, formulas, and constraints, which are made ready to compute based on mathematical rules. However, these models in most cases require homogeneous agents, who sense and act in exactly the same manner. Thus they are not suitable for a real business-to-business world where parties are heterogeneous and loosely coupled. Our approach trades off complexity in the agent models with flexibility of the agents' behaviors. The outcome of our problem-solving is a set of consistent commitments.

Both the Gaia [Zambonelli et al., 2003] and the Tropos [Bresciani et al.] [Castroa et al.] methodologys incorporate the notion of agents into software engineering. Gaia abstracts organizational roles and rules needed to build a flexibly enacted multiagent system, while

Tropos captures both early and late requirements of software development and reduces the gap between what a system must do and why, thus providing the extra flexibility needed to cope with application complexity. Although both the methodologies intend to build an autonomous and proactive system, they more focus on system architectural design within static organizational settings. By contrast, our approach is geared toward forming dynamically enacted business processes in which commitment networks are the only primary constraints imposed on each interacting party. Therefore our approach is more suitable to model business activities whose environment is dynamically changing.

6.2 Directions

There are quite a few interesting topics that can be explored along our proposed direction. First, integrate the commitment-based calculus into business process specification so as to apply the approach in practical development. This would allow machine-assisted verification and testing of the business process specifications. Second, devise an approach to automatically detect commitment conflicts not only before but also during the execution of business process. This would enable the participants' behaviors to be judged on the fly while their autonomy is maximized. Third, incorporate deeper aspects of commitments including a treatment of context group and additional commitment operations (e.g., *assign*, *delegate*) to help seamlessly design long-lived transactions. This would allow us to capture more subtleties among agent interactions and help us in designing applications that more closely represent the real world. Fourth, develop a decentralized protocol to resolve commitment conflicts. This not only helps improve the protocol efficiency, but also removes the single point of failure resulting from a centralized coordinator.

Bibliography

Franz Achermann. *Forms, Agents and Channels Defining Composition Abstraction with Style*. PhD thesis, University of Berne, 2002.

BPEL. Business process execution language for web services, version 1.1, May 2003. www-106.ibm.com/developerworks/webservices/library/ws-bpel.

BPML. Business process modelling language, November 2002. www.bpmi.org.

Paolo Bresciani, Anna Perini, Paolo Giorgini, Fausto Giunchiglia, and John Mylopoulos. Tropos: An agent-oriented software development methodology. *Autonomous Agents and Multi-Agent Sytems*, 8.

Paul Buhler and Jos M. Vidal. Enacting bpel4ws specified workflows with multiagent systems. In *In Proceedings of the Workshop on Web Services and Agent-Based Engineering*, 2004.

Paul Buhler and Jos M. Vidal. Towards adaptive workflow enactment using multiagent systems. *Information Technology and Management*, 6(1), 2005.

Felipe Cabrera, George Copeland, Bill Cox, Tom Freund, Johannes Klein, Tony Storey, and Satish Thatte. Web services transaction (WS-Transaction), August 2002. http://www-106.ibm.com/ developerworks/ webservices/ library/ ws-transpec/.

Luis Felipe Cabrera, George Copeland, William Cox, Max Feingold, Tom Freund, Jim Johnson, Chris Kaler, Johannes Klein, David Langworthy, Anthony Nadalin, David Orchard, Ian Robinson, John Shewchuk, and Tony Storey. Web services coordination (WS-Coordination), September 2003. ftp://www6.software.ibm.com/ software/ developer/ library/ ws-coordination.pdf.

Jaelson Castroa, Manuel Kolp, and John Mylopoulos. Towards requirements-driven information systems engineering: the tropos project. *Information Systems*, 27.

Francisco Curbera, Matthew Duftler, Rania Khalaf, William Nagy, Nirmal Mukhi, and Sanjiva Weerawarana. Unraveling the Web services Web. *IEEE Internet Computing*, 6: 86–93, 2002.

Nirmit Desai and Munindar P. Singh. Protocol-based business process modeling and enactment. *Proceedings of the IEEE International Conference on Web Services*.

Gregg Economou, Maksim Tsvetovat, Katia Sycara, and Massimo Paolucci. Implicit commitments through protocol-level semantics. In *Proceedings of the 2nd Workshop on Norms and Institutions in MAS*, 2001.

FIPA. FIPA interaction protocol specifications, 2003. FIPA: The Foundation for Intelligent Physical Agents, http://www.fipa.org/repository/ips.html.

Nicoletta Fornara and Marco Colombetti. Operational specification of a commitment-based agent communication language. In *Proceedings of the 1st International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 535–542. ACM Press, July 2002.

Xiang Fu, Tevfik Bultan, and Jianwen Su. Realizability of conversation protocolswith message contents. *Proceedings of the IEEE International Conference on Web Services*.

Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, San Mateo, 1993.

Marco Schaerf Gwen Salan, Lucas Bordeaux. Describing and reasoning on web services using process algebra. *Proceedings of the IEEE International Conference on Web Services*, pages 42–49, 2004.

David Harel, Amir Pnueli, Jeanette P. Schmidt, and Rivi Sherman. On the formal semantics of statecharts. In *IEEE Symposium on Logic in Computer Science*, pages 54–64. IEEE Computer Society Press, 1987.

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

Nicholas R. Jennings. On agent-based software engineering. *Artificial Intelligence*, 177 (2):277–296, 2000.

Tao Jin. Utilizing web services in an agent based transaction model (abt). In *In Proceedings of the Workshop on Web Services and Agent-Based Engineering*, 2004.

Doug Kaye. *Loosely CoupledThe Missing Pieces of Web Services*. RDS Press, 2003.

Jiming Liu, Han Jing, and Y.Y. Tang. Multi-agent oriented constraint satisfaction. *Artificial Intelligence*, 136:101–144, 2002.

Sheila A. McIlraith, Tran Cao Son, and Honglei Zeng. Semantic Web services. *IEEE Intelligent Systems*, 16(2):46–53, March 2001.

Bimal Mehta, Marc Levy, Greg Meredith, Tony Andrews, and Brian Beckman. BizTalk server 2000 business process orchestration. *IEEE Data Engineering Bulletin*, 24(1): 35–39, March 2001.

Laurence Melloul and Armando Fox. Reusable functional composition patterns for web services. *Proceedings of the IEEE International Conference on Web Services*.

James Odell, H. Van Dyke Parunak, and Bernhard Bauer. Extending UML for agents. In *Proceedings of the Agent Oriented Information Systems Workshop at the 17th National Conference on Artificial Intelligence (AAAI)*, 2000.

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

Giovanna Petrone. Managing flexible interaction with web services. In *In Proceedings of the Workshop on Web Services and Agent-Based Engineering*, 2004.

David V. Pynadath and Milind Tambe. An automated teamwork infrastructure for heterogeneous software agents and humans. *Journal of Autonomous Agents and Multi-Agent Systems*, 7:71–100, 2003.

Davide Sangiorgi and David Walker. *The π-Calculus: a Theory of Mobile Processes.* Cambridge Universtity Press, 2001.

Munindar P. Singh. A social semantics for agent communication languages. In *Proceedings of the 1999 IJCAI Workshop on Agent Communication Languages*, volume 1916 of *Lecture Notes in Artificial Intelligence*, pages 31–40, Berlin, 2000a. Springer-Verlag.

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

I.A. Smith and P.R. Cohen. Toward a semantics for a speech act based agent communications language. In *Proc. of CIKM'95 Workshop on Intelligent Information Agents.*, 1995.

Milind Tambe. Agent architectures for flexible, practical teamwork. In *Proceedings of the National Conference on Artificial Intelligence*, pages 22–28, 1997.

UML. The unified modelling language. http://www.uml.org/.

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

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

W3C. Web services. http://www.w3c.org/.

Feng Wan, Sudhir K. Rustogi, Jie Xing, and Munindar P. Singh. Multiagent workflow management. In *Proceedings of the IJCAI-99 Workshop on Intelligent Workflow and Process Management: The New Frontier for AI in Business*, 1999.

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

WfMC. The workflow management coalition. http://www.wfmc.org/.

WSCI. Web service choreography interface 1.0, July 2002. wwws.sun.com/ software/ xml/ developers/ wsci/ wsci-spec-10.pdf.

WSDL. Web Services Description Language, 2002. http://www.w3.org/TR/wsdl.

Jie Xing, Feng Wan, Sudhir K. Rustogi, and Munindar P. Singh. A commitment-based approach for business process interoperation. *IEICE Transactions on Information and Systems*, E84-D(10):1324–1332, October 2001. Special issue on *Autonomous Decentralized Systems and Systems Assurance*.

Pınar Yolum and Munindar P. Singh. Commitment machines. In *Proceedings of the 8th International Workshop on Agent Theories, Architectures, and Languages (ATAL-01)*, pages 235–247. Springer-Verlag, 2002.

Franco Zambonelli, Nicholas R. Jennings, and Michael Wooldridge. Developing multiagent systems: The gaia methodology. *ACM Transactions on Software Engineering and Methodology*, 12(3):317–370, July 2003.

Michael zur Muehlen, Jeffrey V. Nickerson, and Keith D. Swenson. Developing web services choreography standards – the case of rest vs. soap. *Decision Support Systems*, 36, 2004.