ABSTRACT

XING, JIE. **Commitment-Based Interoperation for E-Commerce**. (Under the direction of Dr. Munindar P. Singh.)

Successful e-commerce presupposes techniques by which autonomous trading entities can interoperate. Although much progress has been made on data exchange and payment protocols, interoperation in the face of autonomy is still inadequately understood. Current techniques, designed for closed environments, support only the simplest interactions.

This dissertation concentrates on two themes. First, we develop a generic agent interaction model that supports agent coordination. We propose metacommitment patterns, which accommodate revisions and exceptions, to model agent interaction. We formalize metacommitment patterns declaratively in temporal logic. We apply statecharts to specify behavior models of agents who follow our commitment patterns. The statecharts provide an operational semantics, which can be used as a rigorous basis for agent coordination. We propose agent behavior models and prove that it operationally supports our temporal logic semantics. In this manner, we provide the basis for formally designing coordinated multiagent systems. Second, we apply agent behavior models for interoperation in e-commerce. This approach consists of (1) behavioral models to specify autonomous, heterogeneous agents representing different trading entities (businesses, consumers, brokers), (2) a metamodel that provides a language (based on XML) for specifying a variety of service agreements and accommodating exceptions and revisions, and (3) an execution architecture that supports persistent and dynamic (re)execution. Our implementation uses existing Java tool kits for parsing XML and building communicating agents. The

main contributions of this dissertation are in developing some theoretical aspects of agent interaction with an emphasis on e-commerce.

In addition, the proposed approach can also provide a rigorous basis for future standards for interoperation in e-commerce.

COMMITMENT-BASED INTEROPERATION FOR E-COMMERCE

BY

JIE XING

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

OPERATIONS RESEARCH

RALEIGH

JULY 2001

APPROVED BY:

CHAIR OF ADVISORY COMMITTEE

BIOGRAPHY

Jie Xing was born on December 14, 1966 in Yantai city, People's Republic of China. He got his Bachelor of Science degree in Applied Mathematics from Shanghai JiaoTong University, Shanghai, People's Republic of China in 1989. He received his Master of Engineering degree in Computer Aided Design from Beijing Institute of Technology, Beijing, People's Republic of China. After his graduation, he worked as a system engineer in China Academy of Railway Science, Beijing, People's Republic of China. He focused on multi-platform and multi-protocol computer system research. He joined in Hong Kong VANDA Group in 1994 and worked as network specialist and project manager. His responsibilities are network design, network implementation, and network troubleshooting. He was a Ph.D. student in the program of Operations Research of North Carolina State University from August 1997 to May 2001. In his Ph.D. study, Jie Xing focuses on multia-gent systems and workflow systems with special interests in agent coordination, agent interaction, process interoperation, and applying agents for e-commerce.

ACKNOWLEDGEMENTS

Many thanks to my adviser, Dr. Munindar P. Singh for his advice and support in my study and life; special thanks to Dr. William J. Stewart for his encouragement and support; thanks to my Ph.D. committee members, Dr. Mladen A. Vouk and Dr. Peter R. Wurman for their time and valuable comments; many thanks to Feng Wan, Sudhir K. Rustogi and Zhenggang Cheng for extensive discussion and collaboration on our prototype systems; thanks to Bin Yu and Pinar Yolum for some valuable discussion; many thanks to my family for their support, encouragement and patience.

Table of Contents

List of Figures **vii**

1 Introduction **1**
1.1 Motivation . . . 3
1.2 E-commerce Reference Architecture . . . 7
1.3 Database Transactions and Groupware . . . 8
1.4 Agents . . . 10
1.4.1 Communication . . . 12
1.4.2 Interaction . . . 14
1.4.3 Coordination . . . 14
1.5 Our Approach . . . 15
1.6 Contributions . . . 17
1.7 Organization . . . 18

2 Commitments **19**
2.1 Social Commitments . . . 20
2.2 Two-Phase Commit Protocol . . . 22
2.3 Commitment in CSCW . . . 24
2.4 Commitment and its Operations . . . 26
2.4.1 Commitment Creation . . . 28
2.4.2 Commitment Resolution . . . 30
2.4.3 Commitment Observation . . . 30
2.4.4 Revisions . . . 31

3 Metamodel **32**
3.1 Metamodel for Process Interoperation . . . 33
3.2 Agent Knowledge Representation Language . . . 35
3.3 Metacommitment Patterns . . . 37
3.4 Example Models . . . 38
3.5 Specification Language . . . 43

4 Formalization of Metacommitment Patterns **44**
4.1 Computation Tree Logic . . . 44
4.1.1 CTL Syntax . . . 44
4.1.2 CTL Semantics . . . 45
4.2 Conversation-Oriented Metacommitment Patterns . . . 48
4.2.1 Communication Primitives . . . 49
4.2.2 Specification Language . . . 50
4.2.3 Formalization . . . 51
4.3 Task-Oriented Metacommitment Patterns . . . 52
4.3.1 Task-Related Events and Operations . . . 53
4.3.2 Specification Language . . . 53
4.3.3 Formalization . . . 53

5 Operational Semantics **56**
5.1 Statecharts . . . 56
5.1.1 Hierarchy in Statecharts . . . 57
5.1.2 Concurrency in Statecharts . . . 57
5.1.3 Communication in Statecharts . . . 58
5.2 Agent Behavior Models . . . 58
5.2.1 Basic Agent Behavior Model . . . 60
5.2.2 Reactive Agent Behavior Model . . . 61
5.2.3 Monitor Agent Behavior Model . . . 63
5.2.4 Agent Request and Informed Behavior Models . . . 64
5.3 Push and Pull Models . . . 66
5.4 Tasks . . . 69
5.5 Control Connectors . . . 70

6 Soundness of Operational Semantics **73**
6.1 Definition of a Statechart . . . 73
6.2 Pattern Composition . . . 76
6.2.1 Annotation and Translation . . . 77
6.2.2 Merging and Hierarchy . . . 78
6.3 CTL Structure Derivation . . . 81
6.4 CTL Model Checker . . . 83
6.5 Soundness . . . 87
6.5.1 Soundness for Monitor Agent Behavior Model . . . 87
6.5.2 Soundness for Reactive Agent Behavior Model . . . 89
6.5.3 Soundness for Basic Agent Behavior Model . . . 91
6.6 Intra-agent and Inter-agent Pattern Operational Sequences . . . 93
6.7 Exception Handling . . . 96

7 Architecture, Implementation, and Application **100**
7.1 Architecture and Implementation . . . 100
7.1.1 Specification . . . 101
7.1.2 Commitment Data Interface . . . 101
7.1.3 Execution . . . 102
7.2 Applications . . . 102
7.2.1 Travel Planning . . . 102
7.2.2 Contract Net Protocol . . . 105
7.2.3 Distributed Database Interoperability . . . 106

8 Related Work **109**
8.1 Interoperability Standards . . . 109
8.1.1 Existing Workflow Interoperability . . . 109
8.1.2 Interoperation with Existing Systems . . . 112
8.1.3 Existing Languages and Protocols . . . 113
8.1.4 Discussion . . . 116
8.2 Software Agents . . . 117
8.2.1 Commitments . . . 118
8.2.2 Negotiations . . . 119
8.2.3 Process Coordination and Agent Cooperation. . . . 121
8.2.4 Process Management . . . 123
8.2.5 Discussion . . . 124
8.3 Transactions . . . 125
8.3.1 Advanced Transaction Model Workflow . . . 125
8.3.2 Exception Handling . . . 127
8.3.3 Atomic Transaction Extension . . . 127
8.3.4 Transactions as Actions . . . 128
8.3.5 Discussion . . . 129

9 Conclusions and Future Research **130**
9.1 Conclusions . . . 130
9.2 Future Research . . . 132

List of Figures

1.1 Order fulfillment process 2
1.2 Contract net protocol 5
1.3 An e-commerce scenario that does not follow a simple nesting of requests and responses. Here an exception at the customer causes revisions to be sent between vendors. 6
1.4 Layered reference architecture for e-commerce 7

2.1 Finite state automata illustrating the 2-phase commit protocol (cited from [60]) . . 22
2.2 Phases of the customer-performer model 25
2.3 Establishing commitments for a reactive system 29
2.4 Establishing commitments for some other situations 30
2.5 Revision situations 31

3.1 Commitment-based metamodel for interoperation 33
3.2 Applying metacommitment patterns to the contract net protocol 39
3.3 Interoperating to plan a trip 40

4.1 A CTL structure 46
4.2 The corresponding tree for start point s_0 46
4.3 (a) $\mathsf{AG}p$: p is invariant, (b) $\mathsf{AF}p$: p is inevitable, (c) $\mathsf{EF}p$: p potentially holds. (Here, $\bullet = p, \circ = \neg p$) 48

5.1 A statechart example 57
5.2 Basic agent behavior model 60
5.3 Reactive agent behavior model 62
5.4 Monitor agent behavior model 63
5.5 Agent request behavior model 65
5.6 Agent infomed behavior model 66
5.7 Push model, (a) producer, (b) consumer 67
5.8 Pull model, (a) producer, (b) consumer 68
5.9 Combinations of pull and push models for a consumer 69
5.10 Statecharts of typical task executions, (a) transactional, (b) 2PC transactional, (c) non-transactional 69

5.11 Statechart representation of typical connectors, (a) linear, (b) and-join, (c) and-fork 71
5.12 Statechart representation of typical connectors, (d) condition, (e) or-join 72

6.1 Statecharts generated from metacommitment patterns 78
6.2 Merge two statecharts 80
6.3 CTL structure derived from the statechart S_{ca} 84
6.4 The CTL structure derived from the statechart S_{ma} 87
6.5 The CTL structure derived from the statechart S_{ra} 90
6.6 Inter-agent interaction cases 96

7.1 System architecture 100
7.2 A snippet of our XML file for Example 3 104
7.3 Architecture for distributed database interoperation 107
7.4 Example for an agent approach of distributed database interoperability 108

8.1 Logical resource model (cited from WfMC) 111

Chapter 1

Introduction

The study of autonomous, decentralized systems gains importance with the expansion of business-to-business (B2B) electronic commerce. We would like autonomous businesses to be able to operate together logically without being subject to central control. This means we should be able to specify and obtain the correct *interoperation* of their processes.

XML-based standards for data exchange, payment protocols, and e-commerce transactions are being proposed. These provide an important basis for achieving flexible multiparty supply chains and virtual businesses. However, as we show below, current approaches do not fully accommodate the subtleties of interoperation in e-commerce. The key challenge is not in merely producing an XML-based notation, but in ensuring its expressiveness for real-life situations and in providing a formal operational semantics to it.

Current software technology, based on distributed computing and databases and dealing exclusively with low-level abstractions, is inadequate for the proper treatment of interoperation. This is because the associated abstractions—remote methods and distributed transactions—address only the simpler challenges in integrating heterogeneous systems. Remote methods offer no conceptual abstractions for composing activities, whereas transactions require a tighter coupling of activities than is appropriate. Even the extended transaction models require the specification of compensatory methods or transactions, which are in practice usually impossible to guarantee.

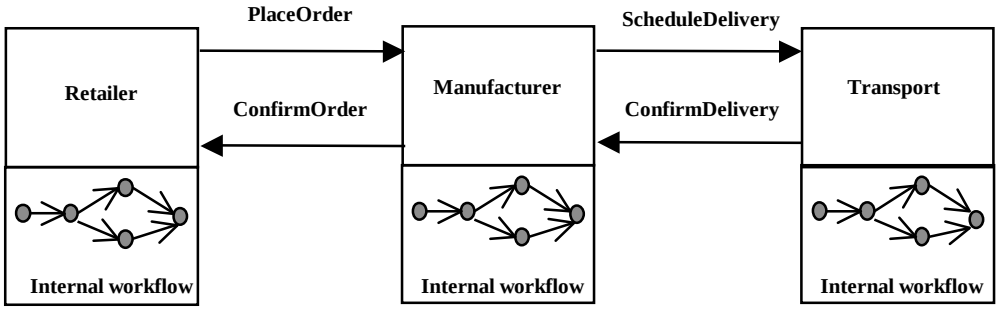


Figure 1.1: Order fulfillment process

Example 1 Figure 1.1 shows an order fulfillment process example at a high level. This commerce process involves three organizations, a retailer, a manufacturer and a transport company. The retailer starts an order fulfillment process by placing orders with the manufacturer. The retailer makes an arrangement to produce goods with the manufacturer. The manufacturer schedules to deliver the goods to the customer. Each participant uses a workflow engine to manage its business process. The workflow engine interoperations achieve the commerce process automation. ▮

Major technical challenges are not properly addressed by existing approaches:

- *Autonomy.* In e-commerce, the interacting parties must retain their autonomy, limited only by their contracts. They can make decision based on their own beliefs. However, in order to form and participate in business processes, they must be able to compromise on their autonomy so that they can cooperate with others. For example, the retailer doesn't want to disclose details of its process definition models to the manufacturer. It wouldn't like to grant access to its internal information about its ongoing process. Similarly, the manufacturer doesn't want to disclose its information on its process definition models and its organization database to the retailer. Each makes its decision independently.
- *Heterogeneity.* In e-commerce, the interacting software may be of any internal design and construction, may be upgraded at different rates, and may run on different platforms. We call this heterogeneity *construction autonomy*. For example, the retailer, the manufacturer and the transport company can use different workflow engines on

different operating systems. This may imply a sea of non-automated processes.

- *Exceptions.* Because e-commerce presupposes heterogeneity, interoperation frequently runs into exceptions. These exceptions are not only low level such as about communication or software, but also high level such as "the goods the consumer wants to buy is not produced by the deadline in manufacturer organization." In particular, the exceptions across organizational boundaries can have significant effects. For example, if the goods is delivered to a wrong address, the whole business process could be canceled.
- *Revisions.* Because e-commerce interactions can be long-lived, potentially lasting forever, they must release their results prematurely. This means that results ought to be able to be revised. For example, a customer orders some computer products from a retailer. The customer may want to change the order configurations after purchasing, but before delivery. The retailer has to check with its manufacturer. If these products are still being assembled, the retailer can inform the manufacturer to modify the configurations.

1.1 Motivation

Business-to-Business electronic commerce places certain demands on interoperation of different autonomous trading partners, especially for dynamically creating and running virtual businesses. There are many business models used in electronic commerce like e-procurement, electronic market places, virtual communities, value chain service provider, value chain integration, collaboration platforms, and information brokerages. Computationally the models can be represented as a set of tasks or steps. It is natural that workflow techniques be used to address the above problems. Current workflow management systems are widely used within enterprises. These workflow management systems are implemented in different languages, on different platforms, and especially, use in different models. It is a challenge to interoperate these different workflow systems across business boundaries to achieve process automation. Much progress has been made on data exchange

(e.g. electronic data interchange (EDI)) standards and online payments. Interoperation of business processes is only now beginning to emerge. E-commerce business processes are complicated and involve multiple organizations. Business relationships between the involved organizations are established in a dynamic competitive environment. The competition also puts increasing pressure on companies to connect their business processes to the processes of other organizations to provide services to customers.

A workflow is the implementation and automation of a particular business process. A workflow management system is the software that automates the coordination and control of tasks during business process execution [14]. Workflow management systems have been used in many different applications such as banking loan processing, telecommunication management, product procurement, software development and proposal evaluation. This has led to different approaches for specifying workflow systems. A wide range of languages, such as script-based, net-based, rule-based and logic-based, are suggested that differ in their expressiveness, formal foundations, and ease of use. The Workflow Management Coalition (WfMC), an industry consortium, aims at a unified terminology and a standardization interface between key components of workflow management systems. WfMC interface 4 is such an interface to provide workflow engine-to-engine interoperation [14]. Object Management Group's (OMG) jointFlow [51], SWAP [6] and Wf-XML [15] address the similar concerns. These interfaces help interoperation across workflow vendor's boundaries, but not organizational boundaries.

Below are two examples. The first illustrates a traditional contract net protocol. The second introduces a business process for travel planning. The two examples describe the regular interaction scenarios as well as some revision and exception scenarios. These scenarios can be expressed as sequence diagrams in UML. A sequence diagram shows the interaction between agents over time.

Example 2 The Contract Net Protocol (CNP) is among the most well known protocols in Distributed Artificial Intelligence (DAI) [17]. CNP involves one manager agent and several contractor agents. The manager calls for bids for a number of widgets with a defined price and deadline. Some

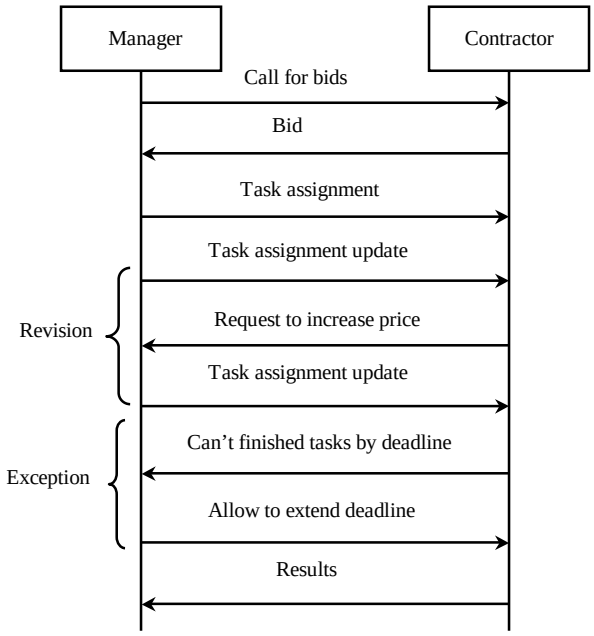


Figure 1.2: Contract net protocol

of the contractors propose bids. The manager chooses a bid and assigns a task to the selected contractor. When the contractor finishes its assigned task, the manager evaluates the results. Figure 1.2 shows the information exchange between the manager and the winning contractor. Traditional CNP specifications cannot handle the following revisions and exceptions.

- The manager may change its task assignment requesting additional widgets with the same price and deadline after it has awarded the task to the contractor. The contractor has to perform the revised task again and send the revised results to the manager.
- The contractor may send requests to increase the price if the manager changes the task assignment.
- The contractor may be unable to finish its task by the specified deadline. The manager may then cancel the task assignment or grant an extension. ▮

Example 3 A customer comes up with a need to travel to a certain city (with multiple hotels and airports) on a certain date. She contacts her travel agent who in turn requests an airline and a hotel

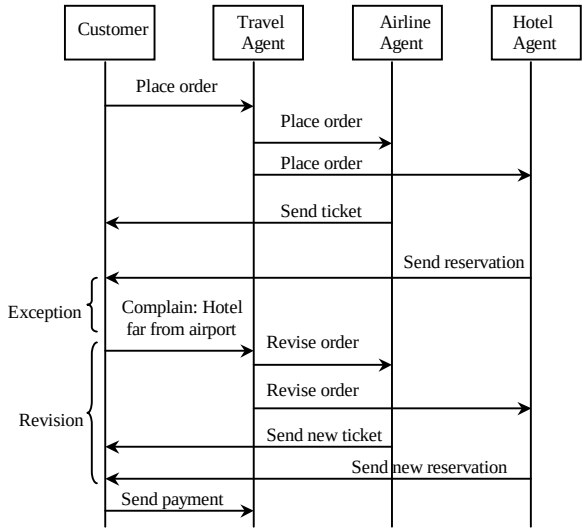


Figure 1.3: An e-commerce scenario that does not follow a simple nesting of requests and responses. Here an exception at the customer causes revisions to be sent between vendors.

clerk to make appropriate reservations. The clerks are to send confirmations to the traveler. The customer may have some additional requirements that are not initially brought out. For example, the hotel may not be close to the airport chosen by the airline clerk and, say, for a flight that arrives late in the evening. The hotel location is an important constraint of the customer. If the customer's requirements are not met, she complains to her travel agent. He would then make a revised request to the clerks, let's say by trying for an earlier flight. ▮

Although small, Example 3 is not trivial. It combines the B2B and B2C aspects of e-commerce. Four autonomous parties are involved, and communication among them does not follow a simple nesting of requests and responses. Even a task that executes successfully may need to be revisited as shown in Figure 1.3. Current situations work well for the simpler situations up to the first travel plan. However, because they offer no conceptual or operational support for exceptions, programmers are left to fend for themselves. Typically, they are forced to employ ad hoc techniques that lead to reduced productivity and ineffective solutions.

1.2 E-commerce Reference Architecture

We propose a layered reference architecture for e-commerce as shown in Figure 1.4. This architecture consists of e-commerce workflow applications, business process, business transactions, common business components and related middleware and tools. Each upper layer is constructed based on the lower layer. The layered reference architecture gives us a clear picture to specify and analyze e-commerce applications. It also shows the components of process interoperation.

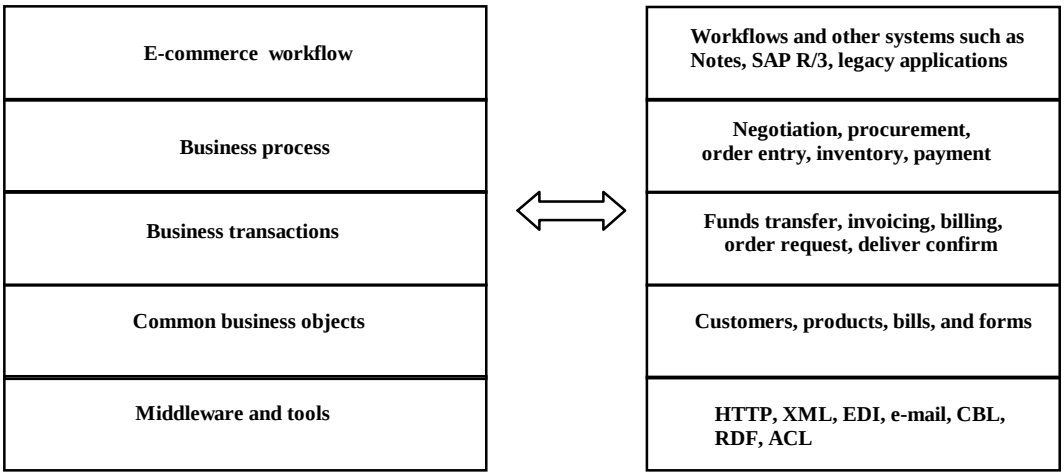


Figure 1.4: Layered reference architecture for e-commerce

1. The layer of e-commerce workflow applications provides a means for developing inter-organization applications. The workflow systems are used to connect these applications to process-oriented applications. It automates business processes across organization boundaries.

2. The business process layer specifies business execution sequences and business logic. It consists of generic business services such as negotiation, procurement, order entry, inventory, payment and special services on special business domains.

3. The layer of business transactions provides transaction supports such as order request, funding transferring, billing, delivering confirmation, and invoicing. It also specifies contractual

obligations and sanctions.

4. The common business objects layer uses CBL and RDF to represent common business objects such as customers, products, bills, and all kinds of forms.

5. The middle-ware and tools layer provides runtime environment for business process instances and tools are used to describe business objects.

1.3 Database Transactions and Groupware

Most current approaches for process interoperation offer ways of structuring activities through means reminiscent of flowcharts. These approaches neglect the organizational relationships among the interoperating components. Some approaches consider the relationship among the components, but fail to provide rigorous semantics for the relationships. Three requirements for process interoperation are:

1. Data integrity: correctness of data despite concurrent access and failures.

2. Control and data flow: how triggering or control information and data flows through the systems.

3. Organization structure: how the various components relate to each other in achieving coherent behavior, e.g., whether a control signal is expected and would not be ignored depends on the organizational structure of the components.

Traditional transactions are traditional computations that satisfy a number of useful proprieties, in particular the so-called ACID properties [25]:

- Atomicity: all or none of a transaction happens.
- Consistency: a transaction preserves the consistency of the database.
- Isolation: intermediate results of a transaction are not visual externally.
- Durability: when a transaction concludes successfully, its effects are permanent.

If the individual transactions are programmed correctly, the database management system guarantees consistency for any arbitrary concurrent mix of transactions. Atomicity is essential to ensure that the integrity of distributed data is preserved. Consequently, the actions or sub-transactions that constitute a transaction must either (a) all happen, thereby transforming the database from a consistent state to a new consistent state, or (b) each fails to happen, thereby leaving the database in its original (consistent) state.

To realize the above transactional properties requires a mutual commit protocol, e.g., two-phase commit protocol, to be executed. However, that might be impossible, for example, hotel and airline are independent enterprises and their databases may not even have visual pre-commitment states, essential to execute a mutual commitment protocol. Even if the pre-commit states are visual, it is seldom acceptable to lock resources while communicating with a remote site. Thus traditional transactions are unacceptable in heterogeneous environments.

Traditional nested transactions provide integrity, but restrict the other aspects. Extended transaction models (ETMs) also focus on integrity, but allow freer control and data flow as the cost of relaxing the integrity requirements. A number of extended transaction models have been proposed, e.g., [8]. The extended transaction models allow partial results to be released, and then attempt to restore consistency through actions to compensate for the effects of erroneously completed actions. Database workflow approaches ignore the integrity aspects, but deliver the control and data flow requirements by specific applications. Some workflow scheduling approaches exist that provide functionality to capture control flow among aspects. For example, they ignore social commitments altogether.

Workflows in groupware also provide application-specific control and data flow without regard to integrity. Some groupware approaches, which study organizational structure, do not consider quite a rich form of commitments. For example, information control nets are primarily geared toward control and data flow aspects [45].

The notion of commitments finds applicability in some groupware tools. For example, [42] shows how the flow of work in an organization is expressed through commitments in the action-Workflow tool. The tool comes with a fixed set of specifications from which the developer can choose. Although the participants can decide whether a given task is successfully performed, there is no notion of failure recovery, of commitments being canceled, or other operations on commitments. Still, we believe this is an interesting system that shows how much can be achieved through the careful use of commitments. We will discuss it in Section 2.3.

1.4 Agents

Software agents are currently the subject of much research in many interrelated fields. While much of the agent community has concentrated on building exemplar agent systems, defining theories of agent behavior and inter-agent communications, there has been less emphasis on defining the techniques required to build practical interoperating agents.

Traditionally, communication protocols are specified by defining the orders in which communicative acts may take place. This holds for protocol formalisms such as finite state machines, push-down automata, formal grammars, Petri nets, and temporal logic. Thus the practitioners are forced into protocols with no content at all. In general, our research has focused on developing the methodology and techniques for building practical interoperational agents. We consider not only the agent interaction sequence, but also the interaction information. The information is the commitment among agents. The main problem is to structure activities in a manner that can respect the autonomy of the information resources. The database approaches are restrictive. The agent approaches are flexible, but there is a need for tools and formal approaches for designing them. In particular, there is a need for a notation of commitment that flexibly reflects the organizational structure of how agents interact. Data integrity can be maintained by agent interaction. A set of control connectors are associated with the agents for specifying control flow. Organization structure can be specified in the interaction. Thus we think that our agent approach can satisfy the requirements for

interoperation. The following rules are used to instruct us to build that agent.

- *Formal.* Formal semantics yields (1) clarity of specifications to guide implementers and (2) assurance of software. Clarity and assurance are significant for agent interoperation, because agent interoperation is meant to be realized in different agents implemented by different vendors.
- *Declarative.* The semantics should describe *what* rather than *how*. A declarative semantics can be applied to a variety of settings, not just those that satisfy some low-level operational criteria.
- *Operational.* This is necessary to provide syntax to represent the agent behavior.
- *Verifiable.* We should be able to determine whether an agent acts according to the given semantics. Otherwise, the semantics are not of much use, especially in an open setting.
- *Meaningful.* The semantics should be intuitive and relate the content of the communications to some substantive feature of the context in which communications occur. Meaningless tokens provide no guidance about how to design and improve protocols.

Interaction is a super set of all different kinds of communication and interoperation.

In order to foster interoperability and reusability of agents, FIPA suggests the abstract agent architecture. We considered agent interoperation as the following three parts, which are agent management and transport, messaging transport and communication protocols. These parts focus on different issues to deal with agent interoperation.

- *Agent management*: it establishes the logical reference model for the creation, registration, location, communication, migration and retirement of agents. It provides white pages, such as agent location, naming and control access services; yellow pages, such as service location and registration services; agent message transport services.
- *Agent message transport*: it deals with the delivery and representation of messages across different network transport protocols. It consists of message transport protocols

(MTPs), such as IIOP, HTTP and WAP, message envelope representations that are suitable for each MTP, such as an XML encoding for HTTP and a bit-efficient encoding for WAP, FIPA ACL representations, such as a string encoding, an XML encoding and a bit-efficient encoding.

- *Agent communication*: communication between agents in FIPA is based on a model of semantically grounded communication. The basis of communication between FIPA agents is through the use of communicative acts (also called performatives) that are based on speech act theory. It consists of a set of FIPA interaction protocols, which describe entire conversations between agents for the purpose of achieving some interaction or effect, such as, auctioning, issuing a call for proposals, negotiating brokering services and the registration and deregistrations of subscriptions.

We will focus on some issues in details.

1.4.1 Communication

Communication enables the agents in a multiagent system to exchange information on the basis of which they coordinate their actions and cooperate with each other. This raises the important questions of what communication protocols and mechanisms enhance cooperation between communicating agents. One popular approach used to enable agents to intercommunicate is through a shared data repository in which information can be posted and retrieved. Agents don't know each other. That means that one agent doesn't know other agents' identities. For example, blackboard systems [12] are an often used model of shared memory in AI. This approach is suitable to the communication that doesn't cross organization boundaries. There is another kind communication. Each agent needs to have an identifier. Agents can directly exchange messages, or they can organize themselves into a group and communicate through special facilitator agents, or they can broadcast their messages.

Directed Message Exchange. Directed communication involves establishing direct physical links with other agents using a protocol such as TCP/IP. There is a centralized server (such as conversation manager in COOL [3] and AgentNameServer in JATLite [31]). The centralized server is like a directory server where one agent can join the server by *register* and leave the server by *unregister*. FIPA 1997 specification suggests such an agent directory that contains information about all agents in a particular environment provides the facilities for an agent to identify and access other agents.

Group with Facilitator. When the number of agents in a system becomes very large, the cost and processing involved in directed message exchange are very large for the centralized agent server. A popular alternative approach that eliminates these difficulties to directed message exchange is to organize the set of agents into different groups. Each group has a facilitator. Agents don't communicate with each other directly. Instead, they communicate through a special facilitator agent. These facilitators ack like routers to transform application-level messages and route them to appropriate places. They are kept to inform about their individuals' needs and abilities.

Broadcast Communication. In the situation, where a message has to be sent to all the agents in the environment, or a sender agent doesn't know who the recipient will be. Broadcast communication is necessary. On the one hand, if there are a large number of agents in the system, then the network bandwidth used to transmit the message is significant when the message length is substantial. On the other hand, broadcast prevents networks from overloading by doing away with the need of making multiple copies of the same message and transmitting them to other agents. For example, in the CNP approach to interoperation, the manager agent in need for service distributes requests for proposals (broadcast message) to other agents. In a specification-sharing approach, agents broadcast their specifications and capabilities to other agents.

Each agent needs to have an address to identify itself. Agents need to register their services. Agents can be divided into different categories by grouping the agents with similar services into a

single category. The three communication mechanisms can be used for different application scenarios.

1.4.2 Interaction

Interaction means a type of collective action where one agent takes an action or makes decision that has been influenced by the presence or knowledge of another agent [7]. The inherently heterogeneous and distributed nature of a multiagent system makes the implementation of the interaction among agents a difficult process. Thus, it is important to design an expressive common language for communication where agents can communicate with their peers by exchanging messages and interact together through explicit linguistic actions. In the multiagent community *Speech Act Theory* [55] is one of the most common methods used to construct and formalize agent communication language. Speech act theory originates in the observation that utterances by agents are not simply propositions that are true or false, but attempts on the part of the agent to convey some knowledge. Speech act theory uses the concept of performatives to allow an agent to convey its knowledge to another agent. A speech act language that is commonly used in the multiagent community is KQML [21]. It helps high-level cooperation and interoperation among agents. The FIPA 1997 specification [22] defines an interaction protocol as an explicitly shared multiagent plan, which contains communicative acts (like speech acts).

Another question here is how to represent the exchanged information. XML is such a means of representation to the problem. XML can encode information and services with a meaningful structure that computers can readily understand. XML allows agents to access different information resources.

1.4.3 Coordination

The third issue that we must deal with in a multiagent system is to assure overall system coordination and coherence. Jennings states that without coordination the benefits of decentralized

problem-solving vanish and the community may quickly degenerate into a collection of chaotic and incoherent individuals [44]. In order to bring about a coherent solution, the objective of a coordination process should be to ensure that all the necessary portions of the overall problem are included in the activities of at least a agent, that agents interact in a manner that permits their activities to be developed and integrated into an overall solution, the team members act in a purposeful and consistent manner, and that all of these are achievable with the available computational resources.

The three main arguments proposed by Jennings as to why actions of multiagents need to be coordinated are as follows.

- There are dependencies between agents' actions, which can occur when the individual goals of the agents are related.
- There is a need to meet global constraints. These constraints exist when the solution being developed by the system must satisfy certain conditions if it is to be deemed successful.
- No individual agent has sufficient competence, resources, or information to solve the entire problem. Many problems cannot be solved by a agent who works in an isolated environment because the agent does not possess necessary expertise, resource, or information.

Speech act theory's power has been exploited to formalize the agent plans within conversations, or groups of utterance. A conversation is an agent's plan to achieve some goal based on interactions with other agents. In order to specify agent coordination, agent conversations can be visualized in term of automata models. An agent's action produce different states and the current state of a plan(conversation) influences how the agent will act/react in the next moment.

1.5 Our Approach

We focus on interaction and coordination for agent interoperation. In brief, we develop the theoretical underpinnings of engineering flexible interactions among agents. We apply agents to process

interoperation in a way that naturally addresses the above challenges. We rely on two crucial properties of agents. First, the agents must interact at a high level by forming and managing *commitments* to one another. These commitments are about the information they exchange, about changes to that information, and about each other's needs. For example, an agent would not only send results to others, but may commit to notify its consumers if it modifies those results or to satisfy its consumers with respect to some requests. Second, the agents must be *persistent.* This is essential so the agents may form, manage, and act according to their commitments. For example, an agent may retry a task until it obtains results acceptable to it and to its peers. The agent would receive updates and send updated results to its consumers. If the agents didn't outlast their tasks, such actions would be impossible.

The agents reason about the formation and manipulation(including revocation) of social commitments. The manipulation of commitments is constrained through specified *metacommitments.* Our metamodel includes a small set of carefully engineered *metacommitment patterns* through which a given interaction can be structured. The metacommitment patterns are given a formal operational semantics based on statecharts [28] and translated into a set of rules that can be executed by different agents.

The interoperation is specified generically in terms of roles played by different components. Agents volunteer to play different roles—thus the agents' autonomy is respected. However, agents may play a role only if (1) they have the capabilities to perform the tasks required of that role and (2) are willing to adopt the metacommitments specified for that role. The assignment of agents to roles happens dynamically. An agent may play more than one role. The roles can be nested when the agents form a team that plays a role.

Conventional software engineering lacks the abstractions necessary to model multiagent systems. We consider the problem of creating specifications for agent behavior and interaction to achieve the necessary coordination to support various kinds of communicative or "conversational" interaction. Following [10, 58], we propose to model the interaction among agents in terms of agents' commitments to one another. We show how temporal logic [19] and statecharts [28] can be

applied in modeling and operationalizing commitments.

We model interactions as a set of composable metacommitment patterns. We formalize the patterns in temporal logic. Each pattern invokes commitment operations. These patterns cover different requirements for the interaction. We develop agent behavior models in statecharts. An executable behavior model generated by a statechart satisfies these commitment patterns. Whereas current approaches include only simple request-response interactions, our approach incorporates more flexible interactions that need accommodate exceptions and revisions. Statecharts provide an operational semantics. Importantly, we show how the operational semantics relates to the declarative semantics for agent interaction.

We develop a simple metamodel that supports interoperating business processes. Like traditional metamodels, our metamodel captures the structuring of activities. However, unlike traditional metamodels, our metamodel includes concepts dealing with persistence and commitments to support exceptions and revisions. The metamodel is represented by UML.

Object-oriented UML provides some methodology to represent collaboration among objects such as sequence diagrams, interaction diagrams, Petri nets, and state diagrams. UML is gaining wide acceptance for representation of engineering artifacts in object-oriented software. Agents will be the next step beyond objects. Researchers are exploring extensions to UML to represent agent interaction. Here we borrow these means to specify agent coordination. However current approaches only represent the interaction sequence. Instead our approach not only considers the interaction sequence, but also uses commitment patterns to store the interaction knowledge to ensure the consistency of their interactions.

1.6 Contributions

First, we develop agent behavior models that support agent coordination. We propose commitment patterns, which accommodate revisions and exceptions, to model agent interaction. We formalize commitment patterns declaratively in temporal logic. We apply statecharts to specify behavior

models of agents who follow our commitment patterns. The statecharts provide an operational semantics, which can be used as a rigorous basis for agent coordination. We propose agent behavior models and prove that they operationally support our temporal logic semantics. In this manner, we provide the basis for formally designing coordinated multiagent systems. Second, we apply agent behavior models for interoperation in e-commerce. This approach consists of (1) behavioral models to specify autonomous, heterogeneous agents representing different trading entities (businesses, consumers, brokers), (2) a metamodel that provides a language (based on XML) for specifying a variety of service agreements and accommodating exceptions and revisions, and (3) an execution architecture that supports persistent and dynamic (re)execution. Our implementation uses an existing Java tool kits for parsing XML and building communicating agents. The main contributions of this dissertation are in

- Developing some theoretical aspects of agent interaction.
- Developing some approaches for process interoperation.
- Applying the theory to e-commerce scenarios.

In addition, it can also provide a rigorous basis for future interoperation standards in e-commerce.

1.7 Organization

The dissertation is organized as follows. Chapter 2 introduces the social commitments and communicative primitives, which are related to the commitment. Chapter 3 introduces a metamodel for agent interaction and process interoperation and describes metacommitment patterns. Chapter 4 formalizes the metacommitment patterns. Chapter 5 introduces operational semantics and several agent behavior models. Chapter 6 gives the soundness of operational semantics. Chapter 7 introduces architecture, implementation, and application. Chapter 8 introduces the related work. Section 9 gives our conclusions and future directions.

Chapter 2

Commitments

In the contract net protocol example 2, a contractor promises to produce some widgets for a manager by a specified date. That is a commitment that the contractor promises the manager to perform the task. For a commitment to be meaningful, one agent making the commitment must intend to perform the task as committed. That is, when the contractor agrees to complete the task on a date, he must really intend to do so. To responsibly do this, he must think through the task, estimate the work involved, and then set the completion date.

The necessary elements of a commitment between two parties are then:

- One party wants a task performed.
- Another party promises to perform the task.
- There exist some constraints for the task.

A commitment often occurs when one party requests something to be done and another party promises to get it done. Neither one wants an unrealistic task. In our example, the manager wants an earlier date, which may be agreed to by the contractor. He is interested in getting his task completed as soon as possible. The contractor would also like to complete the task as soon as possible. However he is not sure how big the task really is and whether there is any other urgent task to do. He would like to have a later schedule. With these opposing objectives, it is clear that commitments should be changed to accommodate the interaction. The contractor wants the longest date he can reasonably get, and the manager wants the shortest.

In Section 2.1, we briefly introduce social commitments. In Section 2.2 and 2.3, we show two commitment applied examples. In Section 2.4, we give the social commitment definition, its operations, and some interaction scenarios.

2.1 Social Commitments

There are three kinds of commitments: internal, social, and collective. Internal commitment corresponds to the commitment defined by Cohen and Levesque [16]. It refers to a relation between an agent and an action. An agent decides to do something by executing an action based on its persistent goals. The agent stops executing the action only when the agent has achieved the goal, or it is impossible to achieve, or it is no longer motivated. The collective commitment is just an internal commitment of a collective agent or group. In other words, a set of agents is internally committed to a certain intention and there is mutual knowledge about that. We follow [10, 58] to discuss the social commitment. Castelfranchi [10] thinks that a social commitment is seen as the glue of group, of collective activity, it links the agent with the joint goal and the common solution, it links the members with each other. A social commitment is not an individual commitment shared by many agents. The social is already collective to characterize a goal or intention shared by many agents and unachievable independent of each other.

A social commitment is a form of goal adoption. For example, x is committed to y to do action a. x and y share a goal: x does a. x wants to do a and x adopts the goal of y. If the x is socially committed to y, then y can:

- Control if x does what he promises.
- Require that x does it.
- Complain/protest with x if he doesn't do a.
- Make good his losses (pledges, compensations).

x can:

- Be indebted to y.

- Acknowledge to be in debt to y and y's rights.
- If x is socially committed to y, he has a duty, an obligation, and he ought to do what he is committed to. Thus when x is committed, a is more than an intention of x, it is a special kind of goal, more cogent.

The definition of a social commitment is the following: x and y mutually know that x intends to do a and that this is y's goal, and with a, y has specific rights on x.

Singh[58] thinks of commitments as being toward conditions to be achieved rather than actions. Using the conditions facilitates nesting commitments and constructing high-order commitments easily. An agent's commitments typically constrain him to act in accordance with them. A commitment is discharged when the desired condition is obtained. This condition might be a plain physical requirement, or a requirement that some other agent recognizes that some physical requirement is satisfied. This is more general because performing an action is the same as satisfying a condition.

Although agents normally act in accordance with their commitments, they sometimes cannot or would not do so. Thus, some commitments must be canceled. Indeed, in most settings and applications where the agent and multiagent systems are useful, inflexibility is undesirable and the agents must retain some autonomy beyond their commitments. However, commitments should not be canceled arbitrarily, because that would subvert their very purpose.

Although social commitments can be established implicitly, here we emphasize the commitment established through an explicit communication act. Singh [58] introduced the notion that communication acts are related to a commitment between speaker and hearer. Directives presuppose a commitment by hearer to do as told; in order to succeed, they lead to a specific commitment by hearer. Assertives commit the speaker to the statement expressed. Permissives make the speaker committed to allowing the relevant assertion to hold or to specifically release a prior commitment of which the speaker is the creditor or context group.

2.2 Two-Phase Commit Protocol

The goal of commit protocol in database area is to have all sites agree either to commit or to abort a transaction. A commit is an unconditional guarantee that the transaction will be completed. In other words, the effects of its actions on the database will be permanent. An abort is unconditional to back out of the transaction, and none of the effects of its actions will persist.

Social commitments can be considered as general commitments to enforce interaction between organizations. Database commitments can be considered as a special case of a social commitment. It hasn't the notions of commitment cancellation and nesting commitments. Two-phase commitment protocol is an application of a database commitment. Because two-phase commit protocol is widely used in the distributed database management system, we investigate it as a reference to specify the interaction between organizations.

In a successful two-phase commit [25, 60], the protocol assumes that one of the cooperating process acts a coordinator. Other processes are referred to as cohorts. This protocol assumes that a stable storage is available at each site and the write-ahead log protocol is active. At the beginning of the transaction, the coordinator sends a start transaction message to every cohort.

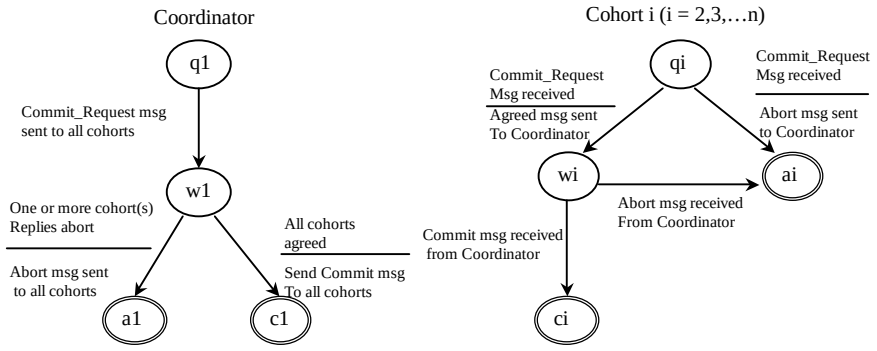


Figure 2.1: Finite state automata illustrating the 2-phase commit protocol (cited from [60])

Phase I

At the coordinator

- The coordinator sends a COMMIT-REQUEST message to every cohort requesting the cohorts to commit.
- The coordinator waits for replies from all the cohorts.

At cohorts:

On receiving the COMMIT-REQUEST message, a cohort takes the following actions. If the transaction executing at the cohort is successful, it writes UNDO and REDO log on the stable storage and sends an AGREED message to the coordinator. Otherwise it sends an ABORT message to the coordinator.

Phase II

At the coordinator

- If all the cohorts reply AGREED and the coordinator also agrees, then the coordinator writes a COMMIT record into log. Then it sends a COMMIT message to all the cohorts. Otherwise, the coordinator sends an ABORT message to all the cohorts. The coordinator then waits for acknowledgements from each cohort.
- If an acknowledgement is not received from any cohort within a timeout, the coordinator resends the commit/abort message to that cohort.
- If all the acknowledgements are received, the coordinator writes a COMPLETE record to log (to indicate the completion of the transaction).

At cohorts:

- On receiving a COMMIT message, a cohort releases all the resources and locks held by it for executing the transaction, and sends an acknowledgement.
- On receiving an ABORT message, a cohort undoes the transaction using the UNDO log record, releases all the resources and locks held by it for performing the transaction, then sends an acknowledgement.

The phase I is also called the prepare phase of commit protocol, and phase II is called the commit phase. If any of the cohorts votes ABORT at the prepare phase, the commit fails, and rollback is

called (abort). If all votes COMMIT, the commit step writes a record in the log saying that the transaction is committed. Each cohort will be invoked in phase 2. When the record appears in durable memory, the transaction is logically committed. If the system fails prior to that instant, the commit step will have failed. If the system fails after that instant, the commit step will be carried forward by the restart logic.

The two-phase commit protocol can be represented by the finite state automata (see Figure 2.1). The protocol that enforces global atomicity is referred to as a commit protocol. In distributed database systems, the transactions must be processed at every site, or none of sites maintains the integrity of the database. The commit protocols also fall into the fault-tolerant design techniques in that they help the system behave in a certain way in the presence of failures.

Here the commit protocol is taking a well-proved algorithm for controlling inter-dependencies among the cohorts. Although widely used in X/OPEN and OTS from OMG, and several commercial database systems, 2PC is considered quite inefficient because it introduces a substantial delay in transaction processing. It is a blocking protocol. This couldn't be accepted with the long-lived tasks found in an e-commerce application.

2.3 Commitment in CSCW

Commitment in CSCW is another commitment application. It is a customer-oriented approach. Customer satisfaction, response time and customer request are the main focus of attention. It abstracts the interaction scenarios with pair-wise commitments. Thus it is thought to apply the commitments between the organizations to model the business processes. The approach is similar as ours. Thus we introduce it as a reference as ours.

In current CSCW systems, the customer-performer model [41] is based on a conversation-oriented approach. A business process is interpreted as a sequence of customer relationships. During the business process the roles of customer and performer may change. However for each relationship, the customer and the performer must be well defined. The *action workflow* system uses the

customer-performer model as a basic element for modeling workflows; it is referred to as a workflow loop.

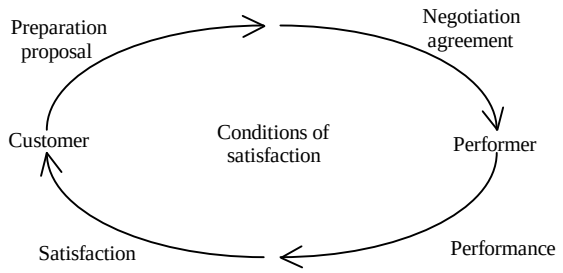


Figure 2.2: Phases of the customer-performer model

A workflow loop incorporates all interactions between customer and performer. It consists of four generic phases (see Figure 2.2). In each of the four-phases, speech acts are applied to specify activity alternatives of all involved parties.

- Preparation/Proposal: an initial contact is established. The customer requests the completion of a service. The customer advertises a certain service.
- Negotiation/Agreement: the customer and the performer negotiate about what is expected of the performer. The result of this phase is a mutual agreement between the two parties on the condition of satisfaction.
- Performance: the performer executes the activities associated with the requested service and subsequently informs the customer about the results. This phase can trigger additional customer-performer relationships, usually involving other parties.
- Satisfaction: the customer evaluates the service results according to the agreed conditions of satisfaction and informs the performer about the evaluation. If the service is not accepted as delivered, further actions may be required by the performer.

In the customer-performer model, there is a commitment established in the first two phases. The commitment is resolved in the last two phases. It also allows us to trigger another loop where another commitment is established. The model allows us to specify business process by applying

the commitment in high level. However there is no notation of failure recovery in the model. The commitment couldn't be canceled.

2.4 Commitment and its Operations

A social commitment C*(x, y, G, p)* relates a debtor role *x*, a creditor role *y*, and a condition *p*, in the scope of a context group *G*. The condition *p* may involve relevant predicates and commitments, allowing the commitments to be nested or conditional. The context group G is the organization within which the commitment exists. Making the context explicitly enables us to specify which roles exist in it and what kinds of commitments exist among them.

For example, in Figure 1.2 of the contract net protocol, a contractor denoted as ctr promises a bid to the manager denoted as mgr. CNP is denoted as the context group of the contract net protocol and $bidp$ is denoted as combination of predicates about the number, price and date of the widgets the manager wants to order. A commitment that the manager commits to order some widgets from the contractor is denoted as C(ctr, mgr, CNP, $bidp$).

Definition 1 A commitment can be viewed as an abstract data type. The following operations are relevant in the thesis. Here, x and y denote roles, and c denotes a commitment of the form C*(x, y, G, p)* if x commits y about p in the context group G.

- *Create(x, c)* establishes the commitment c. The *create* operation can only be performed by the debtor of the commitment.
- *Cancel(x, c)* cancels the commitment c. Generally, cancellation of a commitment is followed by the creation of another commitment to compensate for the former one.
- *Release(y, c)* or *Release(G, c)* releases the debtor from the commitment c. It can be performed either by the creditor or the context group, to mean that the debtor is no longer obliged to carry out its commitment. ▮

A large number of researchers construct agent systems by using a commitment as a mental

phenomenon. Here we emphasize that a commitment is a social commitment, which describes inter-agent relationships. We use the commitment as an agent's social properties to construct agent interaction in a multiagent system. We have given social commitment notions and related operations. Commitments in fault-tolerant design and distributed database have given us some light to how multiple parties interact to guarantee the property: either all commit, or none of them commit. The commitment in CSCW has shown that business process can be represented by some triggerable loops of commitments. For the purpose of manipulating our social commitment, we consider commitment creation, resolution, observation, and some revisions.

Each commitment has an identifier that is generated by the commitment debtor. A commitment can be represented by commitment identifier, debtor, creditor, context group and commitment conditions. Each agent stores its commitments. If the commitment that the agent participates is established, the commitment can be queried. If some resolution conditions are satisfied, the commitment has to be removed. A commitment can be considered as some information, which is stored in a file, or a database, etc. We can check whether a new commitment is consistent with the current commitments of an agent.

Commitments can be used to record the interaction information. Both creditor and debtor share the information. This avoids some fraud. For example, an agent promises to fulfill something for another agent by the agreed deadline. If it doesn't finish the fulfillment by the deadline, the agent has to make some compensation for another agent because they have the commitment records.

Each of commitment operation, *create, release, cancel*, can be performed by one message [63]. In order to make an operation by using one message, some assumptions need to be considered so that we can focus on the commitment itself, which is independent of the communications.

- Communication is reliable and effective. That means that the message cannot be lost and the message will reach the desired parties.
- The participants are persistent. That means that the participants are alive to reply or communicate with other parts.
- The participants are responsible for replying to any request if there is any request.

- However in an open, e-commerce environment, the assumptions are often relaxed. First one participant may be busy on its tasks and may not reply quickly. One participant may ignore some requests because it isn't willing to reply other participants. A business partner may reply late, or won't reply any more in some time.

In this paper, we will use the above assumptions for commitment operations. Some communication primitives can be used to perform these operations. The other primitives are related to the commitment operations directly. We use:

- *Inform*: a role tells another role something and the commitment (exchange information) between the two roles is established.
- *Request*: a role asks another role for some information.
- *Correct*: a role asks another role for some updated information, which is a special request.
- *Reject*: a role doesn't accept the information (a commitment), which is supplied by another role.
- *Withdraw*: a role retracts the information (a commitment), which the role supplied for another role.
- *Refuse*: a role doesn't accept the request to provide any information for another role.

2.4.1 Commitment Creation

In many scenarios, we can construct agents as reactive systems (see Figure 2.3). A creditor sends a *request* to a debtor. The debtor may make the following decisions: (a) the debtor may accept the request and he can establish a commitment by sending a *inform* message; (b) the debtor may *refuse* the request from the creditor. In case (a), a commitment has been established. In case (b), the debtor may not satisfy the creditor's current request instead he may refuse the request by sending a *refuse* message. If a creditor doesn't satisfy the information the debtor supplies, it can continue their interaction (shown in case (c)). When a creditor doesn't satisfy the information, it doesn't accept the

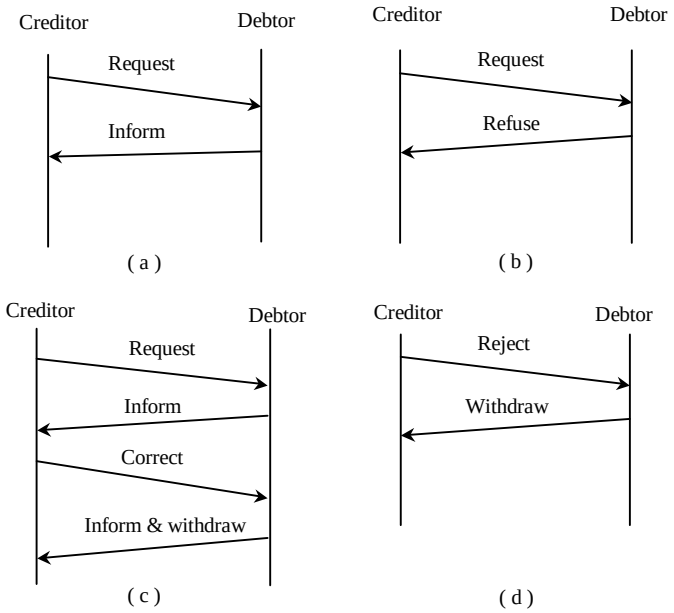


Figure 2.3: Establishing commitments for a reactive system

information by sending a *reject message* (shown in (d)). When a creditor sends a request or correct request to its debtor, it sets a timer. If the debtor doesn't reply the request by the time, a timeout is generated and the creditor sends a *reject* message to its debtor (shown in case (d)).

Figure 2.4 shows some other situations for commitment establishment. In case (a), the commitment is established when the debtor sends an *inform* message. In case (b), the debtor informs the creditor of the latest information and withdraws the old information if there is new information available and previous commitment is invalid. The creditor reasons whether it can accept the commitment at the time. If it would not accept the information, it would send a *reject* message (shown in (c)). After sending an *inform* information, the debtor may retreat the previous information by sending a *withdraw* message (shown in (d)).

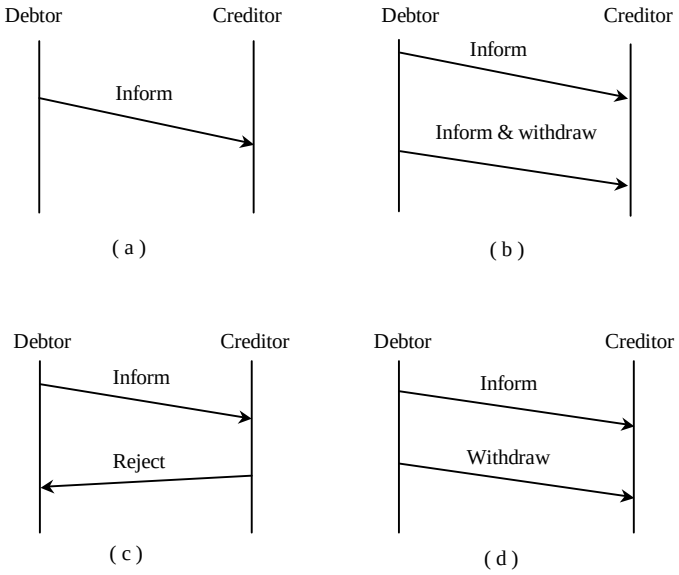


Figure 2.4: Establishing commitments for some other situations

2.4.2 Commitment Resolution

There are three ways to resolve a commitment: (a) normal discharge, (b) debtor cancels the commitment, or (c) creditor releases the commitment.

In case (a), the commitment will be discharged when the commitment conditions (for example, time condition) are satisfied.

In case (b), the debtor may cancel the commitment. It can send a *withdraw* message to a creditor and remove the commitment with the creditor.

In case (c), the creditor may release a commitment. It can send a *reject* message to the debtor.

2.4.3 Commitment Observation

We also provide two messages for an agent to query/answer questions about a commitment. The *query* is used for one agent to send other agents *query* messages to query the related commitments with other agents. The *answer* message is used to provide the related commitments to a query agent.

It may also allow some observers to query the commitments of other agents in their observation domain.

2.4.4 Revisions

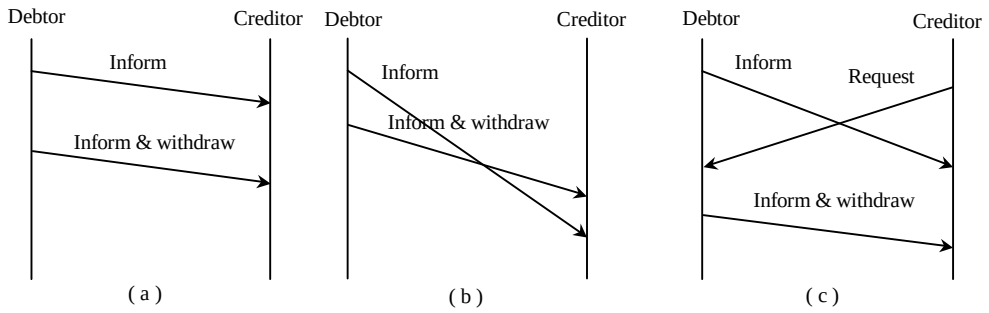


Figure 2.5: Revision situations

Although we may assume that communication is reliable and effective, there are still some revision situations, such as the messages go out of order, or an *inform* message isn't for responding the request. There are two *inform* messages from the same agent with the same commitment identifier (shown in (a)). In case (b) and (c), there are message order problems (shown in (b) and (c)).

Chapter 3

Metamodel

A *metamodel* is a language for specifying models. Our metamodels are a language for specifying agent interactions and specifically the desired process interoperation.

A commitment is the core of our metamodel. There are two kinds of commitment applications. One is conversation-oriented. Another is task-oriented. For the conversation-oriented commitment application between agents in e-commerce scenarios, we need to consider the commitment as what one party promises another party. It often results in an agreement. We say that a party promises another party to do something, while another party promises the party to do other thing. The commitment here is used to enforce their interactions. For the task-oriented commitment application, we consider the commitment as what one party agrees to perform for another party. The commitment is often propagated down the control flow. Thus we have to distinguish them, but we can use the same metacommitment patterns with different formalizations.

We introduce a metamodel for process interoperation in Section 3.1. We present agent knowledge representation language, which can be used to represent agent interaction information in Section 3.2. We propose agent metacommitment patterns in Section 3.3. We apply the metamodel to two examples in Section 3.4. We give the specification language for metacommitment patterns in Section 3.5.

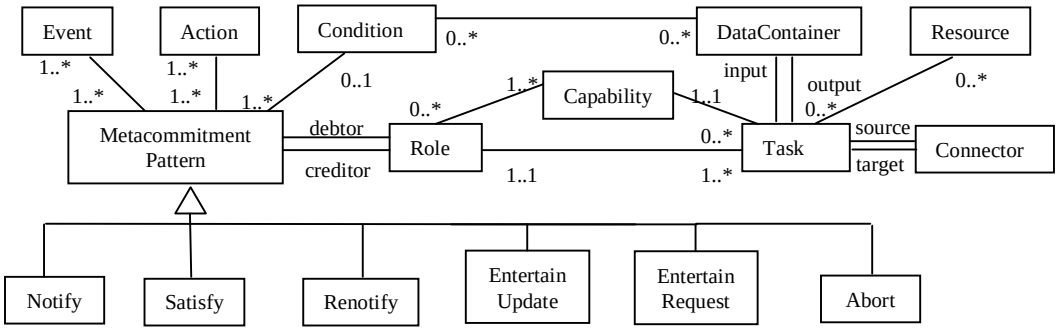


Figure 3.1: Commitment-based metamodel for interoperation

3.1 Metamodel for Process Interoperation

Traditional metamodels for process interoperation are based on flowcharts or activity diagrams, which are of limited help in handling complex situations. Coding the associated procedures or scripts is tedious work. While some of the complexity of the scripts arises from accessing legacy data, much of it is because of the ad hoc manner in which exceptions are handled.

We develop a simple metamodel that supports interoperating businesses represented by autonomous agents. Like traditional metamodels, our metamodel captures the structure of activities. However, unlike traditional metamodels, our metamodel includes concepts dealing with persistence and commitments to support exceptions and revisions. Figure 3.1 presents class diagram of our metamodel in UML [23]. We now discuss the main concepts of this metamodel.

- A *task* identifies a definite piece of work. Tasks may be atomic or compound. We model a task as taking inputs and producing results, both of which map to *resources* (such as databases). A task is executed only if it is triggered; upon completion it produces an event, which (through the connectors introduced below) may trigger other tasks.
- *Capabilities* implement tasks by defining the primary processing required of a task. They are handlers to procedural code that would do the bulk of the computation in an enterprise.

- *Connectors* capture the control flow among the tasks. They also capture the data flow among tasks by binding the outputs and inputs of successive tasks. On the surface, these connectors are traditional and include *linear*, *and-fork*, *condition*, *or-join*, and *and-join*. However, their operational semantics are nontraditional and support reentrance, which is essential for managing commitments properly, especially to enable revisions.
- *Data dependency* captures the data flow along tasks. Each task has its data container, which can be input, or output. One task output can be another task input. Thus we say that there exists data dependency between the two tasks. The data dependency could be along control connectors. It may be across two or more control connectors.
- A *role* provides the capabilities required of any task assigned to it. During execution, a specific agent with matching capabilities is bound to each role. Importantly, roles are used to specify the applicable commitments. Thus, roles capture the underlying organizational structure. Agents can volunteer to assume certain roles that would require them to perform certain tasks by executing their capabilities. Thus, an agent playing a role must implement all the capabilities that the role provides and must honor the metacommitments of that role.
- A *commitment* relates a debtor role, a creditor role, and a condition, in the scope of a context group [58]. This means that the debtor is obliged to the creditor to satisfying the given condition. The context group is the virtual organization that provides the scope of the interoperation. The condition of the commitment involves the relevant predicates from the domain. In our example, the airline can commit to the given traveler that she is confirmed on a particular flight.

  Importantly, commitments are flexible, and can be revoked or modified. For example, even a confirmed flight booking or an entire flight can be canceled. However, the revocation or modification of commitments is constrained through *metacommitments*, which are commitments whose condition itself involves commitments. In our example,

the airline can commit to the traveler that if the flight that she is on is canceled, the airline will book her on an alternative flight. In other words, if the airline has to cancel the commitment (of the booking), then the airline will create an alternative commitment. That is, the above is a metacommitment.

Our metamodel codifies the important kinds of metacommitments as *metacommitment patterns.* We have found a small set of patterns that cover many real-life examples as well as other research scenarios.

In most cases, the metacommitment patterns apply in conjunction with the connectors, which carry the information about which commitments are made. However, in some cases, patterns can apply across two or more connectors.

3.2 Agent Knowledge Representation Language

In order to represent agent interaction information, we need to have a language to represent it. Agent interaction information is a part of agent knowledge. In this section, we will discuss the agent knowledge representation language. We will use the language to formalize agent metacommitment patterns in Chapter 4. The study for choosing a language for representing agent knowledge is the area of knowledge representation. There are many aspects for choosing such a language for an agent. The knowledge representation language can be a modal language, a first order language, or any other language that fits the construction of a particular type of agents. A necessary constraint for the language is that it should have a clear inference mechanism for deriving facts from stored facts and associated update mechanisms for updating stored facts. We also use the language to represent our commitment conditions and reason from agent knowledge. We allow that the language can be plugged in because agent language is separated from our agent interaction.

Knowledge Interchange Format (KIF) is a computer-oriented language for the interchange of knowledge among disparate programs. It has declarative semantics (the meaning of expressions in

the representation can be understood without appeal to an interpreter for manipulating those expressions); it is logically comprehensive (it provides for the expression of arbitrary sentences in the first-order predicate calculus); it provides for the representation of knowledge about the representation of knowledge; it provides for the representation of non-monotonic reasoning rules; and it provides for the definition of objects, functions, and relations.

We choose the first order predicate languages as the agent knowledge representation language in this thesis. First order languages can naturally model many types of information, are well-known and have a lot of expressive power. There are many good introductions to first order logic [43, 20].

The knowledge of an agent is formulas from some first order language L. A first order language L is built from a signature $\sigma = \langle Pred, Func \rangle$ of predicate symbols $Pred$ and function symbols $Func$ with associated arities, and a countably infinite set of variables Var with typical elements x, y, z, ...,x_1, etc. Terms are built from $Func$ and Var as usual and we use $Term$ to denote the set of all terms.

Definition 2 *A language of agent knowledge* is the set of first order formulas L inductively defined by

- if $p \in Pred$ of arity n and t_1, ... , $t_n \in Term$, then $p(t_1, ..., t_n) \in L$,
- if t_1, $t_2 \in Term$, then $t_1 = t_2 \in L$
- $\phi \in L$, then $\neg\phi \in L$
- $\phi \in L$ and $\psi \in L$, then $\phi \wedge \psi \in L$
- $\phi \in L$ and $x \in Var$, then $\forall\, x(\phi) \in L$ ▮

Other logical constructions likes disjunction $\vee$, implication $\rightarrow$, and existential qualification $\exists$ are defined as the usual abbreviations. A number of special syntactic classes and concepts are associated with first order languages. First, the formula of the form $p(t)$ is called an atom; the set of atoms is denoted by At. The usual entailment relation for first order logic is denoted by $\models$. Informally $T \models \varphi$ if φ is implied by the set of sentences T.

3.3 Metacommitment Patterns

Design patterns are valuable to object-oriented software design. We consider an interaction context among agents. Patterns of interaction, each of which captures an important scenario, are defined in term of the participants' metacommitments. Metacommitment patterns make agent behavior coherent during interactions. The combination of these patterns can create different kinds of agent behavior models. We can apply these agent behavior models to our workflow application. Our patterns have two properties crucial for inclusion in a metamodel. They are:

- *Minimal*, in that each pattern imposes the fewest reasonable restrictions to maximize the agents' autonomy.
- *Composable*, so that patterns may be assigned in any combination by a modeler.

The patterns relate to both conversation-intensive, and task-oriented scenarios. An agent can reason about its actions based on its local knowledge. Each pattern involves two roles, a consumer and a producer. An agent plays different roles in different patterns. Predicates with some arguments (i.e., data values) are used to represent the agents' interaction information. In task-oriented scenarios, the tasks are explicitly represented.

1. *Entertain request.* This means that a consumer role will accept requests from another role and reason, or perform its task about the request. It can refuse the request or accept a *reject* request from the consumer role. It can start its execution by itself. For example, a contractor will entertain a call for bids and start reasoning about the call.

2. *Notify the consumer.* This comes into effect when a producer role has just completed its reasoning, or its tasks for the first time. It informs another role of the computed data values, and becomes committed to the specified predicates. For example, a contractor who proposes a bid will notify the manager and commit to the price.

3. *Entertain 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. For example, a manager may request additional widgets at the same price and deadline. The contractor reasons again with the updated data values.

4. *Renotify the consumer.* This comes into effect when a producer role has just completed its reasoning, or its tasks for the second or a later iteration, and some of its existing commitments are violated by the recently completed reasoning. 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. For example, the contractor informs the manager of a higher price to a changed task assignment from its consumer.

5. *Dissatisfy.* This comes into effect when a consumer role is dissatisfied with the results. It sends a reject request to its producer role. For example, the manager sends reject information to the contractor to reject the task assignment and release the associated commitment.

6. *Abort.* A producer role may cancel its commitment by sending an abort message to the other role. For example, the contractor cancels the task assignment if it is unable to finish the task by the deadline.

3.4 Example Models

Example 4 The following shows how our metamodel applies to the CNP of example 2. Here we use the UML to represent the interaction scenarios.

- *Roles* : there are two kinds of roles *Manager* and *Contractor*.
- *Context group* : this context group consists of *Manager* and a number of *Contractor*s and is denoted by CNP.
- *Predicates* : $price(p)$, $num(n)$, $date(d)$. For example, $price(p)$ may represent that the price p is less than some defined value.

We now apply our patterns to CNP. We mainly discuss the patterns that the contractors use for different scenarios.

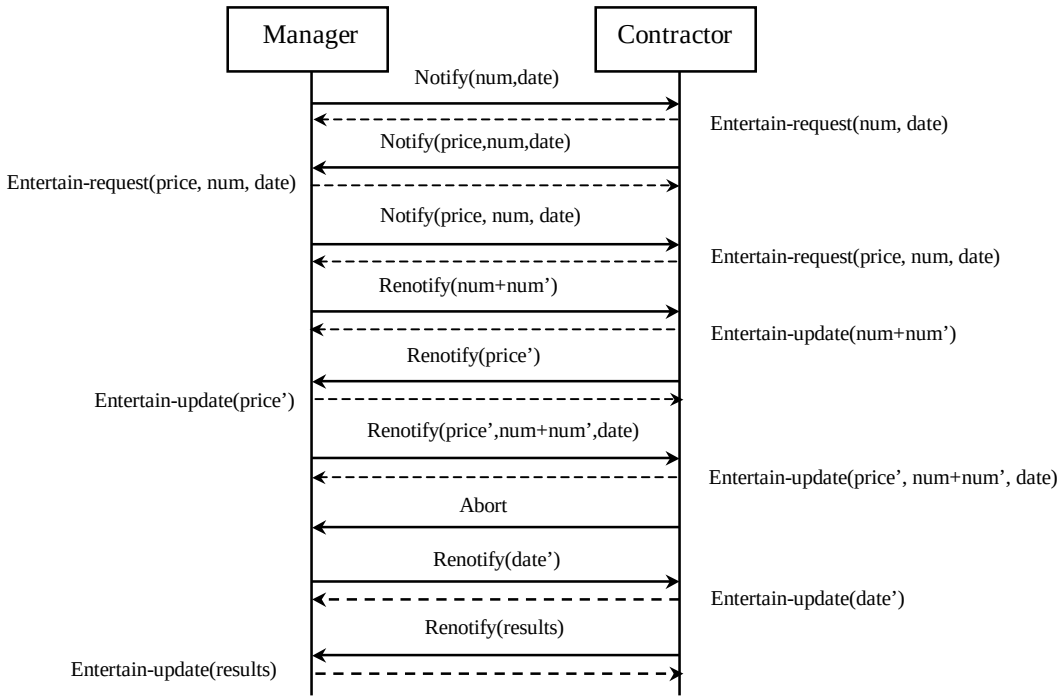


Figure 3.2: Applying metacommitment patterns to the contract net protocol

1. After the contractor receives a call for bids, *entertain-request* pattern forces it to reason about the request. The predicates $price(p)$, $num(n)$, $date(d)$ are created.

2. The contractors apply *notify* with predicates $price(p)$, $num(n)$, $date(d)$ for a bid and a commitment is created.

3. After the manager awards a task assignment to a contractor with $price(p)$, $num(n)$ and $date(d)$, the contractor applies another instance of *entertain-request* that forces it to perform the assigned task. The predicates $price(p)$, $num(n)$, $date(d)$ are created.

4. The contractor applies *notify* which forces it to send task results to the manager and commits to the results.

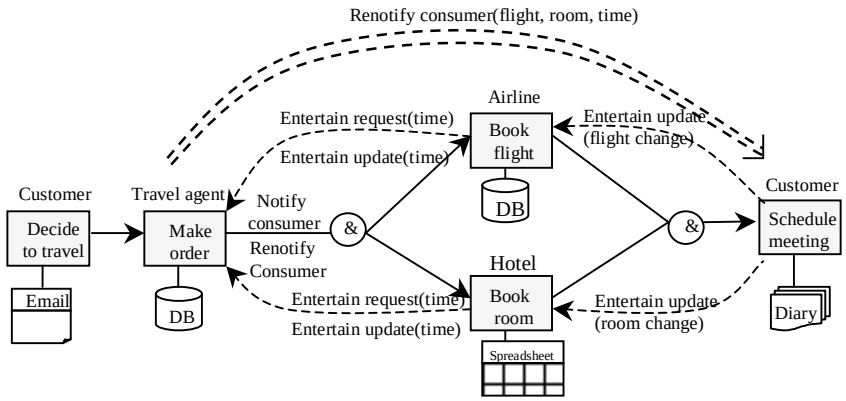


Figure 3.3: Interoperating to plan a trip

5. The manager wants to update a task assignment with an additional n' widgets. The contractor may be forced to reason again by *entertain-update* and inform a higher price p' by using *renotify* and a related commitment is updated.

6. If the contractor cannot meet the request from the manager, it can withdraw the task assignment and send a task cancellation to the manager, which it may be allowed to do by *abort* pattern.

7. The manager may reject the task assignment by using *dissatisfy*, or send an extended date d'. The contractor can reason again by using *entertain-update.*

The first four scenarios above cover the traditional approach for CNP. The last three scenarios handle natural revisions and exceptions that are not accommodated by traditional models. ▮

Example 5 Figure 3.3 illustrates our metamodel applied on Example 3. We use the following notations. Rectangles represent tasks to each of which a role is assigned. Each task has associated resources. Solid arrows represent the connectors. Associated with each connector is a metacommitment from its source role to its target role. Single-dashed lines represent metacommitments from

the target of a connector to a role. Double-dashed lines represent metacommitments that go across more than one connector.

- *Roles*: there are four roles, *customer*, *travel agent*, *airline agent*, and *hotel agent*.
- *Context group*: this group consists of four roles, *customer*, *travel agent*, *airline agent*, and *hotel agent*. We denoted the group as TA.
- *Task*: there are five tasks in the example.
  - *Decide to travel*: the customer performs the task for specifying a definite place, time or price.
  - *Make order*: travel agent performs the task by processing customer requirements and decomposing requirements for airline agent and hotel agent.
  - *Book flight*: airline agent finds available fights for the customer.
  - *Book room*: hotel agent finds available rooms for the customer.
  - *Schedule a meeting*: the customer checks the order of the flight and hotel and decides if he accepts the order.
- *Predicates*: $price(\vec{p})$, $flight(\vec{f})$, $hotel(\vec{h})$
- *Connectors*:
  - *Linear*: after task *decide to travel* is completed, *make order* starts to execute.
  - *And-fork*: source task is *make order*. Two tasks *book flight* and *book room* are target tasks. *make to order* forks two tasks *book flight* and *book room*.
  - *And-join*: source tasks are *book flight* and *book room*. They synchronize at the connector. Target task is *schedule the meeting*.

The control and data flows are as described in Figure 3.3. The applicable metacommitment patterns are that the roles will notify and renotify other roles, entertain requests and updates, dissatisfy, and abort. We now apply our metamodel to the travel example.

1. After the travel agent receives the request from the customer, this forces it to perform the task *make order* by *entertain-request* pattern.

2. After the travel agent finishes its task, it notifies the hotel and airline agents about the customer's order information respectively, such as the date, place, and price by using *notify* pattern.

3. When the hotel and airline agents receive the request from the travel agent, this forces them to perform their tasks *book flight* and *book room* using *entertain-request* pattern.

4. After the airline and hotel agents finish their tasks, they notify the flight, hotel and price information to the customer by using *notify* pattern.

5. If the customer is dissatisfied with the itinerary schedule and suggests a different date or destination city, he renotifies the travel agent about the revisions by using *renotify* pattern.

6. When the travel agent receives the revision requests, it entertains this update on account of *entertain update* pattern. If the effects of the revisions are significant and lead to the violation of a prior commitment, the travel agent may propagate the effect of revisions to hotel and/or airline agent(s).

7. When the hotel, or airline agent receives the revision information, it entertains the updated and reexecutes *book flight* or *book room*.

8. After the hotel, or airline agent finishes its tasks, it can send the updated information to the customer if any of the prior commitments don't hold any more.

9. Sometimes the customer's requirements cannot be satisfied by the airline agent after the airline agent tries several times. The airline agent sends the abort information to the travel agent by *abort* pattern. The travel agent tries other airline agents by *notify* pattern, or send an abort information to the customer by *abort* pattern. It depends on the customer's decision about whether it wants to abort the order, or renotify the travel agent of a new requirement. ∎

3.5 Specification Language

We now define the syntax of the specification language through the following grammar whose starting symbol is *Specification*. The braces $\{$ and $\}$ indicate that the enclosed items are repeated 0 or more times; the remaining symbols are terminals. $\ll$ and $\gg$ delimit comments.

L1. *Specification* $\longrightarrow$ $\{$*Role*$\}$, $\{$*PatternSpec*$\}$ $\ll$Sets of roles and patterns$\gg$

L2. *PatternSpec* $\longrightarrow$ *PatternFormula*($Role$, $Role$, $Group$, $Predicate$) | *PatternFormula*($Role$, $Role$, $Group$, $Predicate$, $Predicate$) $\ll$Formula of metacommitment patterns$\gg$

A specification consists of sets of roles and pattern specifications. Each pattern specification is denoted by a formula name along with roles involved in it, the context group these roles are in, and the predicate. A pattern formula can be expressed as a CTL [19] formula, which is introduced in next Chapter.

Chapter 4

Formalization of Metacommitment Patterns

We have presented the metacommitment patterns in Chapter 3 and given the specification language. In this Chapter, we would like to formalize them. We introduce the syntax and semantics of Computation Tree Logic in Section 4.1. We formalize the communication primitives and conversation-oriented metacommitment patterns in Section 4.2. We formalize the task-oriented metacommitment patterns in Section 4.3.

4.1 Computation Tree Logic

The formal declarative semantics for the metacommitment patterns is given in Computation Tree Logic (CTL) [19], a branching-time logic. The following development suffices for our purposes. We give a semantics to CTL in terms of a CTL structure, roughly a finite rooted directed graph whose paths correspond to different computations. One can imagine the graph being unraveled into an infinite tree. In the following, AP is a set of atomic propositions.

4.1.1 CTL Syntax

The following Backus-Naur Form (BNF) grammar with a distinguished start symbol L gives the syntax of $\mathcal{L}$. $\mathcal{L}$ is based on a set Φ of atomic propositions. Below, *slant* typeface indicates non-terminals; $\longrightarrow$ and $|$ are meta-symbols of BNF specification.

L3. $L \longrightarrow$ *Prop* $\ll$atomic propositions: members of Φ $\gg$

L4. $L \longrightarrow \neg L$ $\ll$negation$\gg$

L5. $L \longrightarrow L \wedge L$ $\ll$conjunction$\gg$

L6. $L \longrightarrow \mathsf{A}\ P$ $\ll$universal quantification over paths$\gg$

L7. $L \longrightarrow \mathsf{E}\ P$ $\ll$existential quantification over paths$\gg$

L8. $L \longrightarrow \mathsf{R}\ P$ $\ll$selecting a real path$\gg$

L9. $P \longrightarrow L\ \mathsf{U}\ L$ $\ll$until: operator over a single path$\gg$

4.1.2 CTL Semantics

The meanings of formulas generated from L are given relative to a model and a point in the model. The meanings of formulas generated from P are given relative to a path and a point on the path. The boolean operators are standard. Useful abbreviations include $\mathsf{false} \equiv (p \wedge \neg p)$, for any $p \in \Phi$, $\mathsf{true} \equiv \neg\mathsf{false}$, $p \vee q \equiv \neg(\neg p \wedge \neg q)$ and $p \rightarrow q \equiv \neg p \vee q$. The temporal operators A and E are quantifiers over paths. Informally, $p\mathsf{U}q$ means that on a given path from the given point, q will eventually hold and p will hold until q holds. $\mathsf{F}q$ means "eventually q" and abbreviates $\mathsf{true}\mathsf{U}q$. $\mathsf{G}q$ means "always q" and abbreviates $\neg\mathsf{F}\neg q$. Therefore, $\mathsf{E}\ p\ \mathsf{U}\ q$ means that on some future path from the given point, q will eventually hold and p will hold until q holds. $\mathsf{RF}q$ means "select a real path " and q will holds.

Definition 3 A CTL structure τ is a four-tuple (Q, R, P, s_0), where

- Q is a finite set of points.
- R is a binary relation on Q, which gives the possible transitions between points and must be total; that is, $\forall\ x \in Q\ \exists\ y \in Q.\ (x, y) \in R$. The relation also indicates branching time.

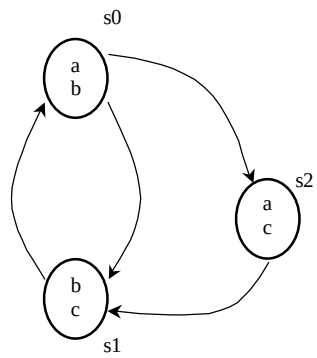


Figure 4.1: A CTL structure

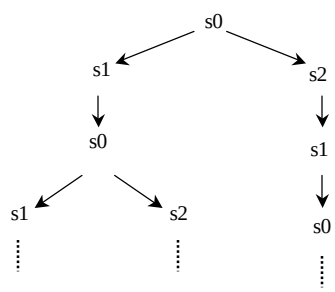


Figure 4.2: The corresponding tree for start point s_0

- P: $Q \rightarrow 2^{AP}$ assigns to each point the set of atomic propositions that are true at that point, where AP are atomic propositions.
- s_0 is the root of Q. $P(s_0) = \emptyset$ ∎

Figure 4.1 shows a CTL structure.

A path is an infinite sequence of points $(s_0, s_1, s_2, ...)$ such that $\forall\, i.\ (s_i, s_{i+1}) \in R$. For any structure $\tau = (Q, R, P, s_0)$, there is an infinite computation tree with root labeled s_0 such that $s \rightarrow t$ is an arc in the tree iff $(s, t) \in R$. Figure 4.2 shows the computation tree rooted at s_0 that is generated from the CTL structure of Figure 4.1.

We use standard notation to indicate truth in a structure: $\tau, s_0 \models f$ means that formula f holds at point s_0 in structure τ. When the structure τ is understood, we simply write $s_0 \models p$. The relation $\models$ is defined inductively as follows:

- $s_0 \models p$ iff $p \in P(s_0)$.
- $s_0 \models \neg f$ iff $\neg(s_0 \models f)$.
- $s_0 \models f_1 \wedge f_2$ iff $s_0 \models f_1$ and $s_0 \models f_2$.
- $s_0 \models \mathsf{AX} f_1$ iff for all points t such that $(s_0, t) \in R$, $t \models f_1$.
- $s_0 \models \mathsf{EX} f_1$ iff for some point t such that $(s_0, t) \in R$, $t \models f_1$.
- $s_0 \models \mathsf{A}[f_1 \cup f_2]$ iff for all paths $(s_0, s_1, ...)$ $(\exists i.\ (i \geq 0 \wedge s_i \models f_2 \wedge \forall j.\ (0 \leq j < i \rightarrow s_j \models f_1)))$.
- $s_0 \models \mathsf{E}[f_1 \cup f_2]$ iff for some path $(s_0, s_1, ...)$, $(\exists i.\ (i \leq 0 \wedge s_i \models f_2 \wedge \forall j.\ (0 \leq j < i \rightarrow s_j \models f_1)))$.

We also use the following abbreviations as temporal connectives in writing CTL formulas:

- $\mathsf{AF}(f) \equiv \mathsf{A}[\text{True} \cup f]$ intuitively means that f holds in the future along every path from s_0; that is, f is inevitable.
- $\mathsf{EF}(f) \equiv \mathsf{E}[\text{True} \cup f]$ means that there is some path from s_0 that leads to a point at which f holds; that is, f potentially holds.
- $\mathsf{EG}(f) \equiv \neg\mathsf{AF}(\neg f)$ means that there is some path from s_0 on which f holds at every point.
- $\mathsf{AG}(f) \equiv \neg\mathsf{EF}(\neg f)$ means that f holds at every point on every path from s_0; that is, f holds globally.

Figure 4.3 shows how some simple correctness properties would be represented using these connectives. By combining these connectives, we can use to specify some protocols. For example, a mutual exclusion protocol can be specified as follows.

There are two processes in the protocol. There are three points for each process.

- N_i: the Non-critical region

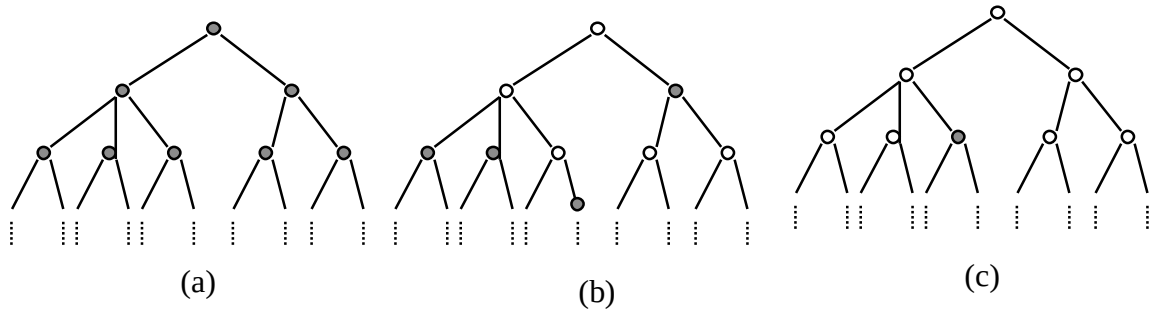


Figure 4.3: (a) $\mathsf{AG}p$: p is invariant, (b) $\mathsf{AF}p$: p is inevitable, (c) $\mathsf{EF}p$: p potentially holds. (Here, $\bullet = p, \circ = \neg p$)

- T_i: the Trying region
- C_i: the Critical region

For the protocol, three properties need to be satisfied.

- $\mathsf{AG}\neg(C_1 \wedge C_2)$ for mutual exclusion
- $\mathsf{AG}(T_i \rightarrow \mathsf{AF}C_i)$ where i = 1,2 for liveness
- $\mathsf{AG}(N_i \rightarrow \mathsf{EX}T_i)$ where i = 1,2 for non-blocking

Our formal language involves the usual boolean connectives plus some temporal connectives. Roughly $\mathsf{AG}f$ holds in s iff f is true at all future points on all paths through s. $\mathsf{AF}f$ holds at s iff f holds eventually on each path through s.

For checking if a CTL structure satisfies a formula f, we applied the model checker algorithm in Chapter 6.

4.2 Conversation-Oriented Metacommitment Patterns

Most studies of communication in multiagent systems are based on speech act theory. The main idea of speech act theory, namely, to treat communications as actions, remains attractive. An illocution is the core component of a communication and corresponds to what the communication might be designed to accomplish independent of both how the communication is physically carried out (the

locution) and the effect it has on a listener (the perlocution). It is customary to classify agent communication language (ACL) primitives or message types into a small number of categories based on the different types of illocutions.

We now define the domain-independent propositions that correspond to communicative actions of a generic agent. These propositions invoke some commitment operations.

4.2.1 Communication Primitives

We formalize our communication primitives as propositions. $Pred$, $Pred_1$, and $Pred_2$ are predicates with domain arguments. As before, x and y denote two roles.

- *Inform*$(x, y, Pred)$: x informs y about $Pred$. This is the simplest case for a conversation. Here x sends a message to y.
- *Request*$(x, y, Pred)$: x requests y to make $Pred$ true. Here x may expect that y will reason or perform the related task when y receives x's request.
- *Correct*$(x, y, Pred_2, Pred_1)$: x requests y to correct $Pred_1$ by $Pred_2$. This primitive allows x to send an updated request information to y. Here y may reason or perform its task again.
- *Reject*$(x, y, Pred)$: x doesn't accept y's information about $Pred$. This primitive allows x to cancel its request.
- *Withdraw*$(x, y, Pred)$: x retracts information about $Pred$ from y. This primitive allows x to cancel its information that has been sent to y. It is often used with *inform* primitive.
- *Refuse*$(x, y, Pred)$: x would not accept y's request. This primitive allows x not to accept a request from y to achieve $Pred$.

The above communication propositions *inform*, *reject*, and *withdraw* imply some commitment operations. $Pred$ is defined as the predicate with a vector of domain arguments $\vec{v}$.

- $\forall\, x, y, Pred, \vec{v}$: $\mathsf{AG}[\mathit{inform}(x, y, Pred(\vec{v})) \rightarrow \mathsf{AF}[\mathit{create}(x, \mathsf{C}(x, y, G, Pred(\vec{v})))]]$.

This means that x commits to inform y that $Pred(\vec{v})$ is true.

- $\forall\, x, y, Pred, \vec{v}$: $\mathsf{AG}[reject(x, y, Pred(\vec{v})) \rightarrow \mathsf{AF}[release(x, \mathsf{C}(y, x, G, Pred(\vec{v})))]]$.

  This means that y is released from its commitment to $Pred(\vec{v})$ when x requests to reject the $Pred(\vec{v})$. If there isn't $\mathsf{C}(y, x, G, Pred(\vec{v})))$, we assume that *release*$(x, \mathsf{C}(y, x, G, Pred(\vec{v})))$ is true.

- $\forall\, x, y, Pred, \vec{v}$: $\mathsf{AG}[withdraw(x, y, Pred(\vec{v})) \rightarrow \mathsf{AF}[cancel(x, \mathsf{C}(x, y, G, Pred(\vec{v})))]]$.

  This means that x can cancel the $Pred(\vec{v})$.

4.2.2 Specification Language

We define the following domain-independent event propositions about status changes of agent reasoning.

- *Start*: a role starts its reasoning by itself.
- *Computed_done*: a role finishes its reasoning for the first time.
- *Update*: a role starts its reasoning again when something is changed.
- *Recomputed_done*: a role finishes its reasoning for the second or a later time.
- *Abort*: a role aborts its reasoning.
- *Valid_commitment*: a role's commitments with another role are still valid after the role just finishes its reasoning for the second or a later time.

Based on the commitment operations and communication primitives, we now give our specification language for commitment patterns.

L10. *PatternFormula* $\longrightarrow$ $\mathsf{AG}[statusProp|commProp \rightarrow \mathsf{AF}commProp|PatternFormula]$

L11. *CommProp* $\longrightarrow$ *communicationOP*$(Role, Role, Pred)$|*communicationOP*$(Role, Role, Pred, Pred)$

L12. *CommunicationOP* $\longrightarrow$ *inform*|*request*|*correct*|*reject*|*withdraw*|*refuse* $\ll$domain independent propositions$\gg$

L13. *StatusProp* $\longrightarrow$ *start*|*update*|*computed_done*|*recomputed_done*|*abort*|*valid_commitment*
≪status propositions≫

4.2.3 Formalization

Let's first make an assumption: only if a role has the capabilities to reason, the role starts to entertain the request or update. $\triangleq$ is used to represent a definition symbol. The left of $\triangleq$ is definition name. The right of $\triangleq$ is the definition body.

We now apply the specifications to formalize our metacommitment patterns.

- When x finishes its reasoning for the first time, which means *computed_done* is true, x informs y about $Pred(\vec{v})$.
  notify$(x, y, G, Pred) \triangleq \forall\, \vec{v}.\ \mathsf{AG}[$*computed_done* $\rightarrow \mathsf{AF}$*inform*$(x, y, Pred(\vec{v}))]$
- When x finishes its reasoning for second, or a later times, *recomputed_done* becomes true. Then x checks if its current commitments(represented as *valid_commitment*) hold with new results. If *valid_commitment* is false, x informs y of the predicates associated with new values and informs y to cancel the commitment with old values.
  renotify$(x,y,G,Pred,Pred) \triangleq \forall\, \vec{v_1},\ \vec{v_2}.\ \mathsf{AG}[$*recomputed_done* $\wedge \neg$*valid_commitment* $\rightarrow \mathsf{AF}[$*inform*$(x,\ y,\ Pred(\vec{v_2})) \wedge$ *withdraw*$(x,\ y,\ Pred(\vec{v_1}))]]$
- When a *start* event triggers x to start its reasoning, x will use *notify* pattern to inform its consumer y of the results, or accept a *reject* request from y.
  entertain-request$(x, y, G, Pred) \triangleq \forall\, \vec{v}.\ \mathsf{AG}[$*start* $\rightarrow \mathsf{AF}[$*notify*$(x,\ y,\ G,\ Pred(\vec{v})) \vee$ *reject*$(y,\ x,\ Pred(\vec{v}))]]$
  When y sends x a request about $Pred(\vec{v})$, this will trigger x to reason about the request. Finally x uses *notify* pattern to inform the results, or x refuses the request from y, or accept a *reject* request from y.
  entertain-request$(x, y, G, Pred) \triangleq \forall\, \vec{v}.\ \mathsf{AG}[$*request*$(y,\ x,\ Pred((\vec{v}))) \rightarrow \mathsf{AF}[$*notify* $(x,\ y,\ G,\ Pred\ (\vec{v})) \vee$ *refuse*$(x,y,Pred\ (\vec{v})) \vee$ *reject*$(y,\ x,\ Pred(\vec{v}))]]$

- When an *update* event triggers x to start its reasoning again, x will use *renotify* pattern to inform its consumer y of the updated results, it can abort the *update* event, or it accept a reject request from y.

  entertain-update$(x, y, G, Pred, Pred) \underline{\triangle} \forall \vec{v_1}, \vec{v_2}.$ AG[*update* $\rightarrow$ AF[*renotify* $(x, y, Pred(\vec{v_2}), Pred(\vec{v_1})) \vee$ *abort*$(x, y, G, Pred(\vec{v_1})) \vee$ *reject*$(y, x, Pred(\vec{v_1}))$]]

  When y sends x a correct request, x reasons about that again. Finally x uses *renotify* pattern to inform y of the new results, x may be allowed to withdraw the previous commitment by using *abort* pattern, or it accept a reject request from y.

  entertain-update$(x, y, G, Pred, Pred) \underline{\triangle} \forall \vec{v_1}, \vec{v_2}.$ AG[*correct*$(y, x, Pred(\vec{v_2}), Pred(\vec{v_1})) \rightarrow$ AF[*renotify*$(x, y, Pred(\vec{v_2}), Pred(\vec{v_1})) \vee$ *abort*$(x, y, G, Pred(\vec{v_1})) \vee$ *reject*$(y, x, Pred(\vec{v_1}))$]]

- This pattern allows that x withdraws the commitment it has had with y.

  abort$(x, y, G, Pred) \underline{\triangle} \forall \vec{v}.$ AG[*abort* $\rightarrow$ AF*withdraw*$(x, y, Pred(\vec{v}))$]

- If x sends y a reject about $Pred(\vec{v_1})$, y can withdraw the commitment with x by using metacommitment pattern *abort.* Or, y may send a new predicate $Pred(\vec{v_2})$ to satisfy x again by using commitment pattern *renotify.*

  dissatisfy$(x, y, G, Pred, Pred) \underline{\triangle} \forall \vec{v_1}, \vec{v_2}.$ AG[*reject*$(x, y, Pred(\vec{v_1})) \rightarrow$ AF[*abort*$(y, x, G, Pred(\vec{v_1})) \vee$ *renotify*$(y, x, Pred(\vec{v_2}), Pred(\vec{v_1}))$]]

4.3 Task-Oriented Metacommitment Patterns

We define the following domain-independent task-oriented propositions that correspond to transitions in a generic agent that performs some given tasks. We generically represent task t in our metacommitment patterns because t is an important part of our workflow specification.

4.3.1 Task-Related Events and Operations

- *Computed_done*(x, t): a role x finishes its task t execution successfully for the first time.
- *Recomputed_done*(x, t): a role x finishes its task t execution successfully for a second or later times.
- *Abort*(x, t): a role x can abort its task t.
- *Do*(x, t): a role x performs its task t at the first try.
- *Redo*(x, t): a role x performs its task t at the second or later try.
- *Undo*(x, t): a role x compensates its task t.

4.3.2 Specification Language

Based on the task-oriented propositions and communication primitives, we now give our specification language for task-oriented commitment patterns.

L14. *PatternFormula* $\longrightarrow$ AG[*statusProp*|*commProp* $\rightarrow$ AF[*taskProp*| *commProp*| *PatternFormula*]]

L15. *commProp* $\longrightarrow$ *communicationOP*($Role$, $Role$, $Pred$)|*communicationOP*($Role$, $Role$, $Pred$, $Pred$)

L16. *communicationOP* $\longrightarrow$ *inform*|*request*|*correct*|*reject*|*withdraw*|*refuse* $\ll$domain independent propositions$\gg$

L17. *statusProp* $\longrightarrow$ *start*|*valid_commitment*|*update* $\ll$status propositions$\gg$

L18. *taskProp* $\longrightarrow$ *computed_done*|*recomputed_done*|*abort*|*do*|*redo*|*undo* $\ll$task propositions$\gg$

4.3.3 Formalization

We now apply the specifications to formalize our metacommitment patterns.

- When x finishes its tasks, which means *computed_done*(x, t) is true, x informs y about $Pred(\vec{v})$.

  notify$(x, y, G, Pred) \triangleq \forall \vec{v}, t.$ AG[*computed_done*$(x, t) \rightarrow$ AF*inform*$(x, y, Pred(\vec{v}))$]

- When x finishes its tasks for the second, or a later time, *recomputed_done*(x, t) becomes true. Then x checks if its current commitments(represented as *valid_commitment*) hold with new results. If *valid_commitment* is false, x informs y of the predicate associated with the new values and withdraws the previous commitment.

  renotify$(x, y, G, Pred, Pred) \triangleq \forall \vec{v_1}, \vec{v_2}, t.$ AG[*recomputed_done*$(x, t) \wedge \neg$*valid_commitment* $\rightarrow$ AF[*inform*$(x, y, Pred(\vec{v_2})) \wedge$ *withdraw*$(x, y, Pred(\vec{v_1}))$]]

- A *start* event triggers x to perform its task, or accept a *reject* request from y.

  entertain-request$(x, y, G, Pred) \triangleq \forall \vec{v}, t.$ AG[*start* $\rightarrow$ AF[*do*$(x, t) \vee$ *reject*$(y, x, Pred(\vec{v}))$]]

  When y sends x a request about $Pred(\vec{v})$, this triggers x to reason about the request, x refuses the request, or accepts a *reject* request from y.

  entertain-request$(x, y, G, Pred) \triangleq \forall \vec{v}, t.$ AG[*request*$(y, x, Pred(\vec{(v)})) \rightarrow$ AF[*do*$(x, t) \vee$ *refuse*$(x, y, Pred(\vec{v})) \vee$ *reject*$(y, x, Pred(\vec{v}))$]]

  When task t is completed, x may send out a notification, but that notification would be in the scope of another pattern.

- An *update* event triggers y to perform its task again, x redoes its task, x uses *abort* pattern to reject to entertain the changes, or it accepts a reject request from y.

  entertain-update$(x, y, G, Pred, Pred) \triangleq \forall \vec{v_1}, \vec{v_2}, t.$ AG[*update* $\rightarrow$ AF[$redo(x, t) \vee$ *abort*$(x, y, G, Pred(\vec{v_1})) \vee$ *reject*$(y, x, Pred(\vec{v_1})$]]

  When y sends x a correct request, x performs its task about that again, x may be allow to withdraw the previous commitment by using *abort* pattern, or it accept a reject request from y.

  entertain-update$(x, y, G, Pred, Pred) \triangleq \forall \vec{v_1}, \vec{v_2}, t.$ AG[*correct*$(y, x, Pred(\vec{v_2}), Pred(\vec{v_1})) \rightarrow$ AF[$redo(x, t) \vee$ *abort*$(x, y, G, Pred(\vec{v_1})) \vee$ *reject*$(y, x, Pred(\vec{v_1})$]]

Finally x redoes its task.

- When x aborts task t, x needs to withdraw the commitment it has had with y.

  $\textit{abort}(x, y, G, Pred) \triangleq \forall\ \vec{v}.\quad \mathsf{AG}[\textit{abort}(x, t) \rightarrow \mathsf{AF}[\textit{withdraw}(x,\ y,\ Pred(\vec{v})) \wedge \textit{undo}(x,\ t)]]$

- If x sends y a reject about $\vec{v_1}$, y can withdraw the commitment with x by using commitment pattern *abort.* Or y may send a new predicate $Pred(\vec{v_2})$ to satisfy x again by using commitment pattern *renotify.*

  $\textit{dissatisfy}(x, y, G, Pred, Pred) \triangleq \forall\ \vec{v_1},\ \vec{v_2},\ t.\quad \mathsf{AG}[\textit{reject}(x,\ y,\ Pred(\vec{v_1})) \rightarrow \mathsf{AF}[(\textit{abort}(y, x, G, Pred(\vec{v_1})) \vee \textit{renotify}(y,\ x,\ Pred(\vec{v_2}),\ Pred(\vec{v_1})))]]$

Chapter 5

Operational Semantics

To support our operational semantics for agent interaction, we require that the agents follow some predefined behavioral models, which are expressed as statecharts. Statecharts are well established in software engineering as a means to specify reactive systems, and concurrent computations [28, 52]. Typically, such descriptions involve complex sequences of events, actions, conditions, and information flow. A statechart is composed of *states* (OR-states, AND-states, and PRIM-states) and *transitions.* In Section 5.1, we introduce three major characteristics of a statechart. In Section 5.2, we present agent behavior models. In Section 5.3, we show pull and push models. In Section 5.4 and Section 5.5, we give the task and control connector representations of a statchart.

5.1 Statecharts

Statecharts [27] are a visual formalism to specify the behavior of complex reactive systems. A reactive system is characterized by being, event-driven, continuously having to react to external and internal stimuli. Statecharts extend conventional finite state machine to include the notions of hierarchy, concurrency (orthogonality), and communication. Statecharts also have the ability to be viewed at varied levels of details through either abstraction or refinement. In a nutshell, one can say

Statecharts = state-diagrams + depth + orthogonality + broadcast-communication

Figure 5.1 is a simple example of a statechart. We will use the example to introduce concepts

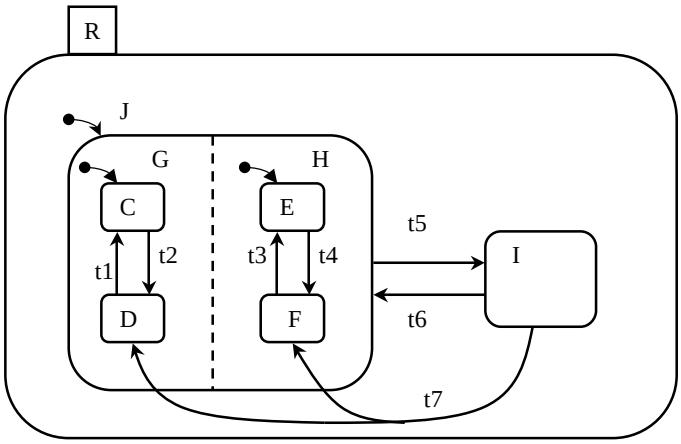


Figure 5.1: A statechart example

about statecharts.

5.1.1 Hierarchy in Statecharts

Hierarchy in statecharts is achieved by grouping states into super-states. This is an XOR (exclusive-or) decomposition. State R in our statchart example includes two states, J and I. State J has two sub-states, G and H. If the system is in state R, then it can only be either in sub-state J or sub-state I, but not both (exclusive-or relation). Here, the arrow in the statechart indicates the transition direction; the label over the arrow indicates the transition event (including the conditions) and action. Suppose that the system is in state I. When transition t_6 is satisfied, the system goes directly to sub-state G and H of state J. The unlabeled dotted arrow to sub-state G of J indicates that C is the default state of state G. Therefore, the system goes to C when transition t_6 is triggered. When transition t_5 is triggered, it goes to state I as a whole. When transition t_7 is triggered, the system directly goes to the sub-state D of G and sub-state E of H.

5.1.2 Concurrency in Statecharts

Concurrency in statecharts is shown through orthogonality, which is an AND decomposition and captured by the partition of the statechart.

The state J has two sub-states G and H, which are two orthogonal components (see the dashed boundary). When the system is in J, it must be in both G and H. G and H also have XOR decompositions. State G has two sub-states C and D, while state H has E and F. Therefore, if the system is in J, it must be in one of four possible pairs of sub-states, which are $\langle C,E\rangle$, $\langle C,F\rangle$, $\langle D,E\rangle$ and $\langle D,F\rangle$ (one from G, another from H). Now, if the system is in the state I, when transition t_6 is triggered, it enters state J and enters sub-states C of G and E of H (following the arrow labeled t_6, which starts from state I). If the system is in sub-state E of H and either C or D of G, then, when transition t_4 is triggered, it goes to F and stays in whatever sub-state of G it was in.

5.1.3 Communication in Statecharts

The communication mechanism in statecharts [27] is based on broadcast, whereby signals generated by a subsystem are immediately available to other concurrent subsystems through a broadcast style communication.

5.2 Agent Behavior Models

Agent interaction is basically a specialized type of agent behavior. The behavior of an agent may be viewed conceptually in three categories: what it does internally, what it acts externally, or how it reasons to external stimuli. We consider that the internal behavior happens, but we don't consider how it happens. We mainly concern the inter-agent behavior. We overview some of work that is related to agent behavior models.

Agent-0 [57] is a simple language to specify agent behavior. Agent-0 is an implementation of the Agent-Oriented Programming (AOP) principle that supports three fundamental mental states: *Belief*, *Capability* and *Commitment*. Beliefs are logical statements about the world that the agent believes to be true or false. Capabilities are actions the agent is able to perform. Commitments are guarantees that an agent will carry out an action at a certain time. Three basic message types are provided: *inform*, *request* and *unrequest*. Inform allows agents to communicate mental states to

other agents, whilst request allows an agent to ask another agent to perform an action, and unrequest cancels a request. Commitment rules consist of antecedent conditions that are matched against incoming messages and the agent's internal states (such as beliefs). If a rule is triggered, the agent will generate a commitment to do a private action, refrain from an action, or send a message (i.e., inform). These commitments are then executed at the time specified by the agent who sent the request. Agent-0 allows to use nesting message, for example, to ask a question, a *request*(*inform* ($?x$)) message is sent.

Zeus [46] incorporates the behavior of the entire agent into a state machine format: specifically, a directed graph. Zeus agents pursue a goal, and to achieve that goal the agent executes its behavior defined along a directed graph that best describes what they wish to do to achieve that goal. Zeus uses the state information contained within the graph to determine whether it should converse with another agent to achieve the goal. If the agent requires assistance from another agent, an interaction is used for the conversation.

COOL [4] is an agent coordination language that focuses on the specification of coordination between agents. It uses finite state machines to describe the flow of communication between agents based on a fixed number of speech acts, such as propose, accept, and reject. COOL focuses on a specific way to model coordination between agents and not on the architecture of an agent itself. It provides prescribed specific protocols for specific functionality, not for the generic interaction between agents.

Agent-0, Zeus, and COOL provide different approaches to specify agent behavior models. We mainly focus on agent interaction. We don't elaborate the agent internal reasons here. Thus we only represent all agent internal reasons in *computing* and *recomputing* states. In Chaper 2 we introduce the interaction scenarios. We use the debtor and creditor to represent roles because these roles are related to commitment notions. Here we consider agent behavior models. We use the producer and consumer notion. In the following sub-secitons, we will sythesize these interaction scenarios to generate different agent behavior models. Figure 5.2, Figure 5.3, and Figure 5.4 show three agent behavior models for producer agent. Figure 5.2 represents a basic agent behavior model denoted

by S_{ca} in a statechart. Figure 5.3 represents reactive agent behavior model denoted by S_{ra} in a statechart. Figure 5.4 represents monitor agent behavior model denoted by S_{ma} in a statechart. Figure 5.5 and Figure 5.6 show two agent behavior models for consumer agents. An agent can instantiate several threads for different interaction scenarios. Each thread follows our agent behavior models. For example, the contractor agent has a thread for proposing a bid and another thread for performing task assignment. Both threads use the same behavior model S_{ca}.

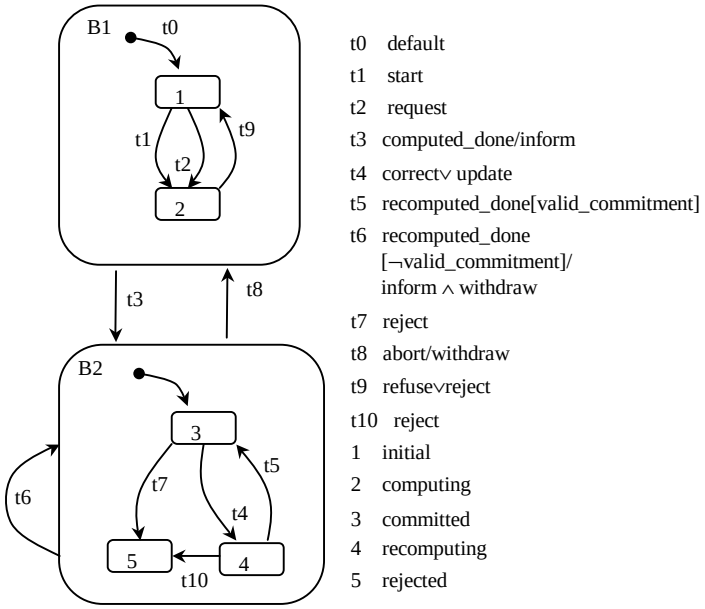


Figure 5.2: Basic agent behavior model

5.2.1 Basic Agent Behavior Model

This is a producer agent model. Its main functions are (1) to provide information to its consumer agent, and (2) commit its consumer agent to provide any updated information. In order to achieve these two functions, the producer agent should have the following properties.

- The agent can provide the results after it finishes its reasoning. The reasoning is based on the request from its consumer agent, or its internal decision.

- The agent can provide the updated results after it finishes its reasoning again. The reasoning is based on the correct request from its consumer agent, or its internal decision.
- The agent can refuse any request and abort the results it provides.
- The agent can handle the reject message of its consumer agent.

We will apply statecharts to specify the basic agent behavior model with these properties.

For the model S_{ca}, on receiving an initial trigger signal, an agent is in state 1. After the agent receives a start event, or a request event, it begins reasoning in state 2. The start event means that the agent can start its reasoning by itself. The request event means that the agent can start its reasoning by accepting requests from other agents. Upon completion of the reasoning, the agent sends the reasoning results to some selected agents and commits to the sent results. The agent is in state 3. In state 2, the agent may send a refuse message to refuse the request, or the agent can stop its reasoning when it receives a reject message.

A correct or an update event causes the agent to reexecute its reasoning in state 4. After finishing its reasoning, the agent obtains new results. If the results change substantially to invalidate the agent's commitments, the agent sends the updated results again (canceling the old commitments and creating new ones). If the agent receives a reject event in state 3, the agent transits to state 5 and reasons in state 5. The agent may receive a reject request in the state 4. In that case the agent transits to state 5 and reasons in this state. If an abort event occurs, it withdraws the commitments with its consumer agent. If a recomputed_done event occurs and current commitments are invalid, the agent sends new results to satisfy its consumer agent.

5.2.2 Reactive Agent Behavior Model

This is a producer agent model. Its main function is to response the request from its consumer agent. In order to achieve the function, the producer agent should have the following properties.

- The agent can provide the results after it finishes its reasoning. The reasoning is based on the request from its consumer agent, or its internal decision.

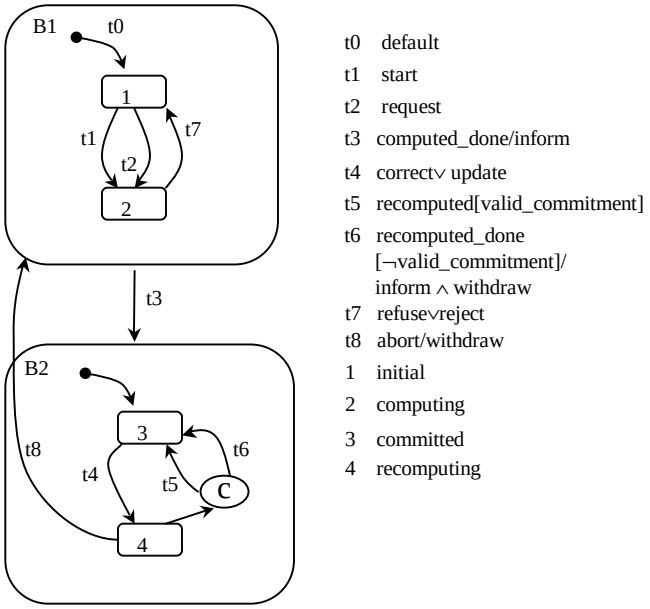


Figure 5.3: Reactive agent behavior model

- The agent can provide the updated results after it finishes its reasons again. The reasoning is based on the correct request from its consumer agent.
- The agent can refuse any request and abort the results it provides.

We will apply statecharts to specify the reactive agent behavior model with these properties.

For the model S_{ra}, on receiving a default trigger signal, an agent is in state 1. After the agent receives a start event, or a request event, it begins reasoning in state 2. The start event means that the agent can start its reasoning by itself. The request event means that the agent can start its reasoning by accepting requests from other agents. Upon completion of the reasoning, it sends the results to some selected agents and commits to the results sent. The agent is in state 3. In state 2, the agent can send a refuse message to refuse the request, or the agent can stop its reasoning when it receives a reject message.

A correct or update event causes the agent to reexecute its reasoning in state 4. If the results change substantially to invalidate the agent's commitments, the agent sends the updated results again (canceling the old commitments and creating new ones). Otherwise the agent keeps the current

commitment and transits to state 3. The agent will stay in state 4 to react to any request from other agents. An abort event causes the agent to send a message to withdraw the results to the consumer.

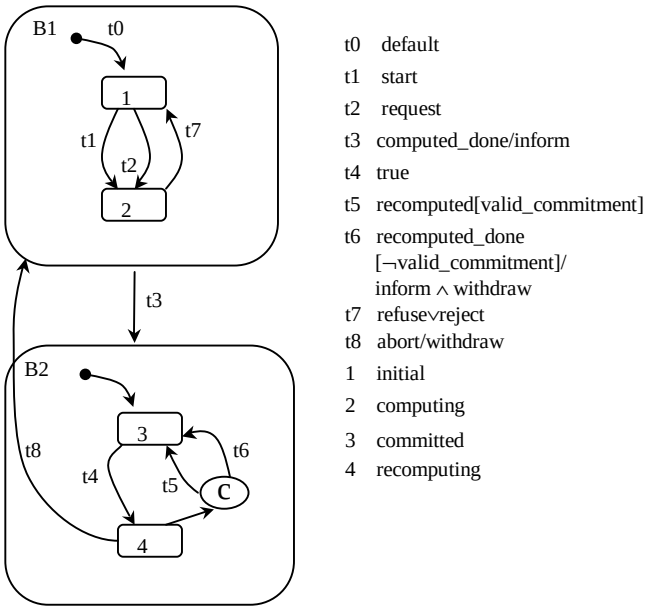


Figure 5.4: Monitor agent behavior model

5.2.3 Monitor Agent Behavior Model

An agent periodically supplies information that is then processed by the information consumer to update his own knowledge base. The periodical information is automatically delivered to the consumer agent without any explicit request message. The method seems particularly suitable for the conveyance of sensor data; if the sensor data changes, the new value is automatically forwarded to the agent that processes these data, e.g., a robotics agent planning a robot's movements or actions.

This is a producer agent model. Its main function is to monitor any change and provide the information for its consumer agent. In order to achieve the function, the producer agent should have the following properties.

- The agent can provide the results after it finishes its reasoning. The reasoning is based on the request from its consumer agent, or its internal decision.

- The agent can provide the updated results to its consumer agent.
- The agent can refuse any request and abort the results it provides.

We will apply statecharts to specify the monitor agent behavior model with these properties.

For the model S_{ma}, on receiving a default trigger signal, an agent is in state 1. After the agent receives a start event, or a request event, it begins reasoning in state 2. The start event means that the agent can start its reasoning by itself. The request event means that the agent can start its reasoning by accepting requests from other agents. Upon completion of the reasoning, it sends the results to some selected agents and commits to the results sent. The agent is in state 3. In state 2, the agent can send a refuse message to refuse the request, or the agent can stop its reasoning when it receives a reject message.

The agent likes an execution loop. It keeps reexecuting its reasoning in state 4. If the results change substantially to invalidate the agent's commitments, the agent sends the updated results again (canceling the old commitments and creating new ones). Otherwise the agent keeps the current commitment and transits to state 4. An abort event causes the agent to send a message to withdraw the results to the consumer.

5.2.4 Agent Request and Informed Behavior Models

The basic agent behavior model handles the request from other agents. However an agent should have rights to request some information when the agent needs it at any time.

An agent request behavior model is a model for a consumer agent. This request agent model should have the following properties.

- The agent can ask its producer to provide information.
- The agent can ask its producer to provide updated information.
- The agent can reject the information its producer supplies.

We will apply statecharts to specify the request agent behavior model with these properties.

In the agent request behavior model, there are five states, *initial*, *requested*, *committed* and

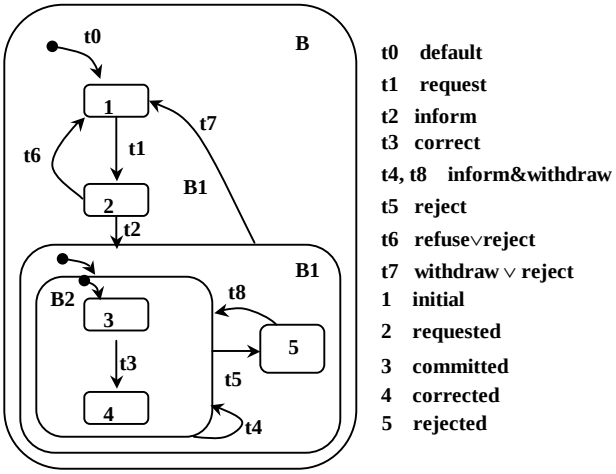


Figure 5.5: Agent request behavior model

corrected, rejected. When the agent wants to request something from its producer, it uses the request behavior model. It first sends a request message to its producer. It may receive the requested information, or a refuse message from its producer. A timer starts when a request is sent. If it cannot receive any information by the time, it will send a reject message to cancel the requested information.

When the agent is in the committed state, it can send a correct message if the agent reasons and needs to some updated information. The agent may be informed to accept some new information and to withdraw old information, or only to withdraw the old information. When the agent doesn't receive any information by the deadline, it sends a reject message to cancel the correct request. If it cannot get any reply about the reject information, it will send another reject information to close the conversation.

In some cases, a producer informs its consumer agent about the results. The consumer agent handles the information by using agent informed behavior model. Once all of information is received, it will use its basic agent behavior and works as the producer of other agents.

An agent informed behavior model is a model for a consumer agent. This informed agent model should have the following properties.

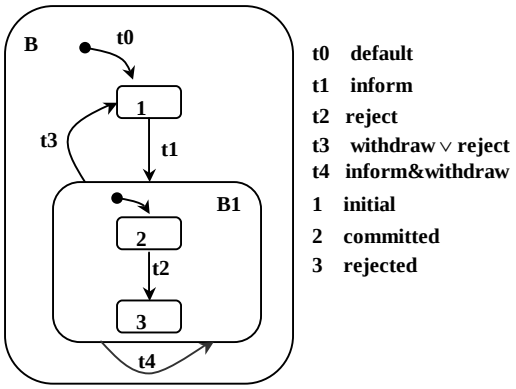


Figure 5.6: Agent infomed behavior model

- The agent can accept any information that a producer agent provides.
- The agent can accept any updated information that a producer agent provides.
- The agent can reject the information a producer supplies.
- The agent can accept a withdraw from a producer agent.

We will apply statecharts to specify the informed agent behavior model with these properties.

It has three states, *initial*, *committed*, and *rejected*. When it receives any information from its producer agent, it enters *committed* state. When some new information is received, it enters *committed* state. The model allows to send a message to reject the results, and handle a withdraw message from its producer.

5.3 Push and Pull Models

Push model. The producer commits to inform its consumer about the results it has obtained (see Figure 5.7).

- After a *start* event is issued by a producer, it reasons. The producer has two choices. After it finishes its reasoning successfully, it sends an *inform* message about the reasoning results to its consumer. Sometime it may receive a *reject* message from its consumer during its reasoning, it goes to its initial state and doesn't send any message

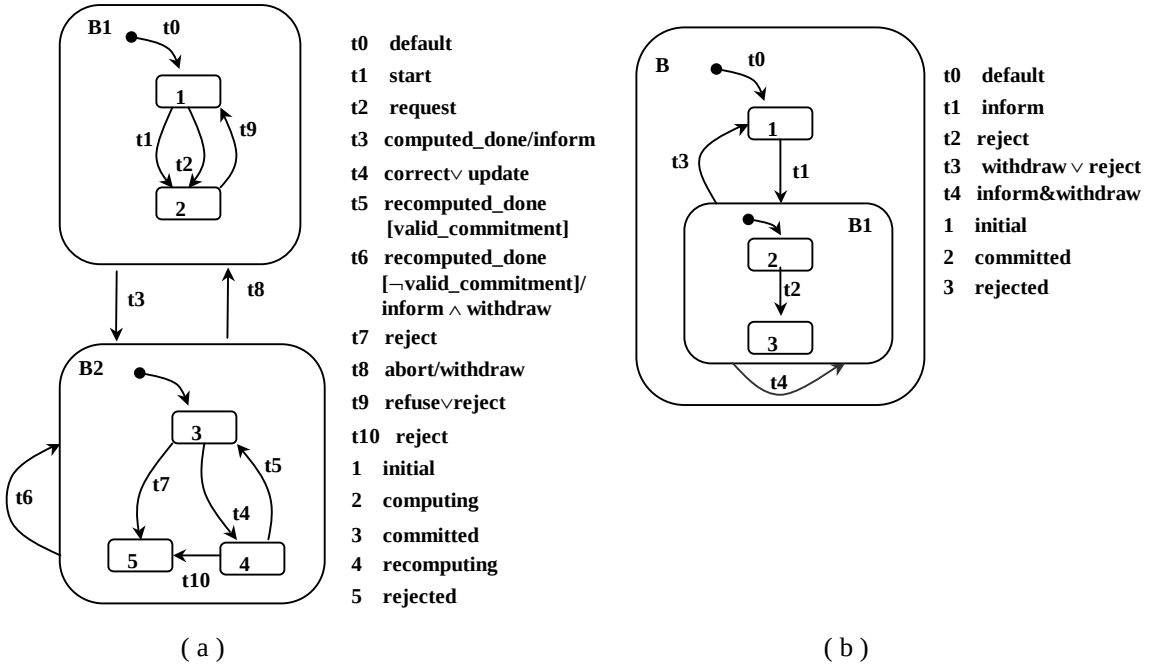


Figure 5.7: Push model, (a) producer, (b) consumer

to its consumer.

- After an *update* event is issued by a producer when the producer is in the state *committed*, the producer reasons again. The producer has the following choices. First, when the producer finishes its reasoning successfully, the producer informs its consumer about new results and withdraw the old results. Second, the producer may withdraw the old results. Third, the consumer may send a message to reject the old results if it cannot succeed to execute its reasoning or tasks.
- After an *abort* event is issued by a producer, the producer will withdraw the results. The consumer will go back to its *initial* state.

Pull model. The producer accepts any request from its consumer and responses its consumer about the request (see Figure 5.8).

- When the producer receives the consumer's *request*, the producer goes to state 2 and reasons. After that, the producer has the following choices. First, after the producer

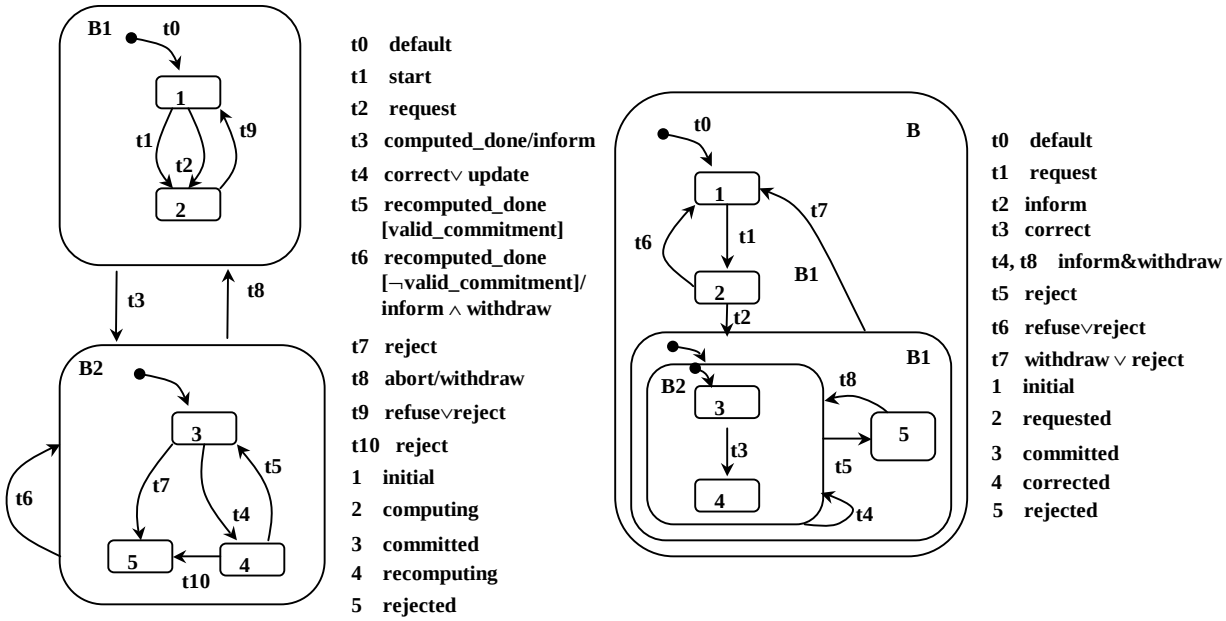


Figure 5.8: Pull model, (a) producer, (b) consumer

finishes its reasoning successfully, it informs the consumer about the results. Second, the producer may refuse the consumer's request. Third, the consumer may cancel the request by sending *reject* message.

- When the producer is at the state *committed* and receives the consumer's *correct* information, it goes to state 4 and reasons again. If the producer finishes its reasoning successfully, it informs its consumer about new results and withdraws the old results. The consumer may reject the *correct* request. The producer may withdraw the old results.
- After the consumer sends a *reject* message, the producer may withdraw the results, or inform the consumer about new results and withdraw the old results.

Flexible model. We use parts of the request and informed agent behaviors as construction blocks (B_1 state in each model). We create more flexible agent behaviors for a consumer agent. This can be constructed by exchanging the B_1 in the request agent behavior model and informed

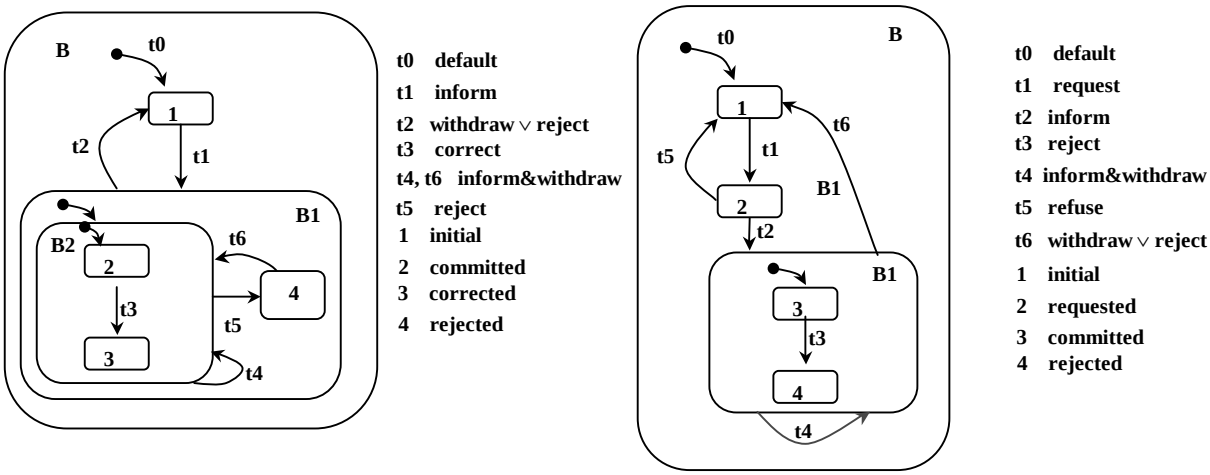


Figure 5.9: Combinations of pull and push models for a consumer

agent behavior model. Two new agent behavior models allow consumer agents with combination behaviors of the pull and push models (see Figure 5.9).

5.4 Tasks

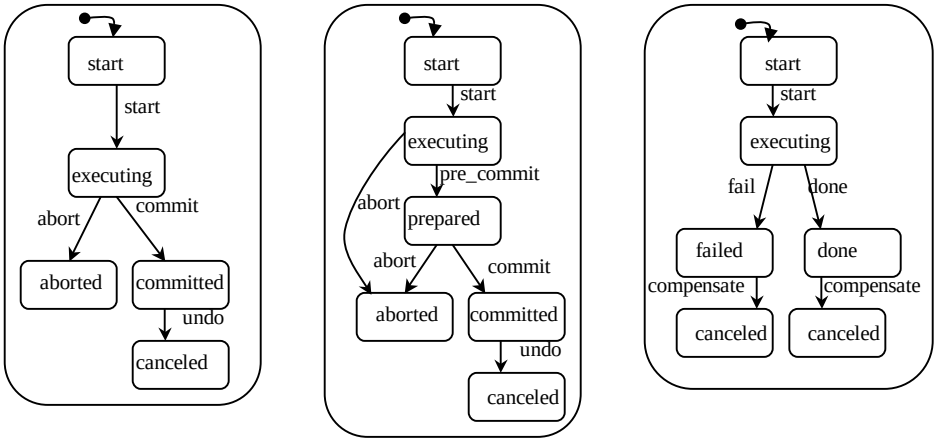


Figure 5.10: Statecharts of typical task executions, (a) transactional, (b) 2PC transactional, (c) non-transactional

Tasks may be transactional, or non-transactional. For a transactional task, we can refine it

to a 2-phase commit task. The executions of tasks are initialized by the control connectors of a workflow. Then the task is assigned to a given agent. After this point, an agent with the capability of executing the task is in charge of the task execution. Tasks may be atomic units of work, or they may be compound tasks by using the control connectors. A task can succeed or fail. If the task fails, it has to be undone. For each task, we consider how to compensate it. An agent can operate a task by the following operations, or generate some task events, for example, do, $redo$, $undo$, $abort$, $computed_done$, $recomputed_done$. Statecharts are used to represent a task execution. Figure 5.10 demonstrates the states and transitions between states for each type of task.

For a transactional task, an agent generates a *start* event to start executing a task. In *executing* state, the task is performed (by do operation). If the task is finished successfully, the agent can commit the task. Otherwise the agent has to abort the transactional task so that the transactional task hasn't any effect. If an agent wants to cancel the task execution after the task has been committed, the agent has to use compensate operations ($undo$).

For two-phase commit task, we increase the *prepared* state. An agent can defer some time to commit, or abort the task. For a non-transactional task, if an agent fails to execute a task, it also has to compensate the task.

Our metacommitment patterns require the agent to execute its task many times. Thus operation $undo$, or $comp$ is used to cancel the task execution.

5.5 Control Connectors

Control connectors describe relationships among tasks. They are important components to connect tasks to generate a big compound task, or a workflow. Here we provide advanced features for definition of semantically rich connectors between tasks, or connectors.

1. First we choose a set of connectors, start, end, linear, and-join, and-fork, conditional, or-join.

2. For revision requirement in our commitment patterns, we allow connectors with reentrance

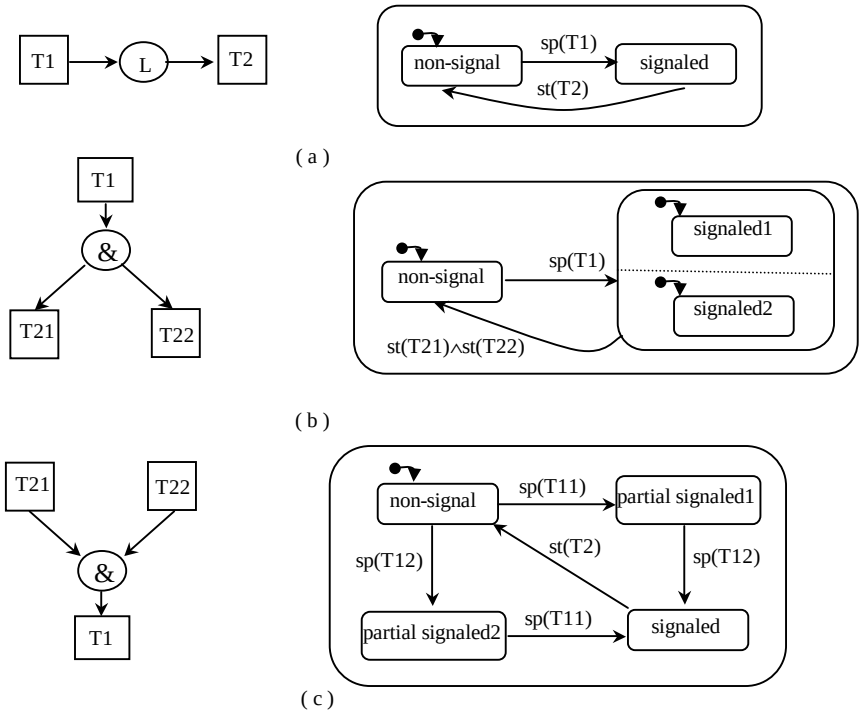


Figure 5.11: Statechart representation of typical connectors, (a) linear, (b) and-join, (c) and-fork

function.

3. We represent control connectors with statechart notions. For each statechart, there are two major states, non-signal and signaling. Once an event triggers a transition from non-signal, we say this connector is activated. When an event triggers a transition from signaling to non-signal. We say that the connector is inactive and waiting for some events, which trigger it to become active again.

Figure 5.11, Figure 5.12 shows our statechart operational semantics for control connectors.

- Start and end connectors are simple. Start connector triggers the task start of a workflow. End connector represents that a workflow ends when some task(s) is finished.
- For a linear connector, a task is initialized after another task is finished. There is a static react $st!(T_2)$ to start T_2 at state $signaled$.

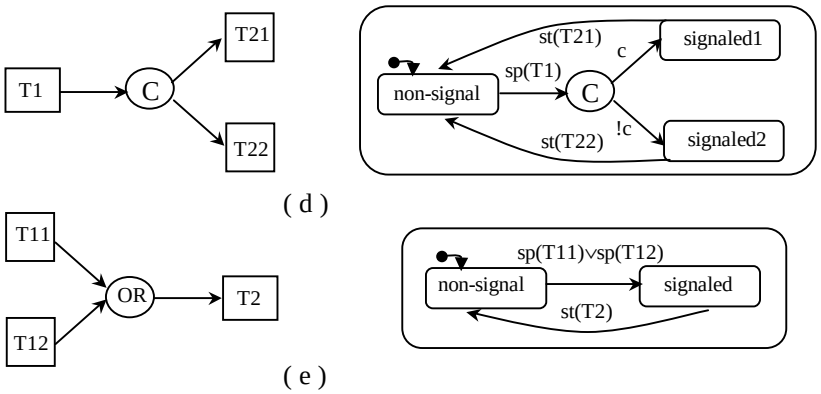


Figure 5.12: Statechart representation of typical connectors, (d) condition, (e) or-join

- For an and-fork connector, after T_1 is performed successfully, T_{21} and T_{22} is ready for execution. There are static react $st!(T_{21})$ to start T_{21} at state $signaled1$ and static react $st!(T_{22})$ to start T_{22} at state $signaled2$.
- For an and-join connector, this is a synchronization connector. T_2 becomes ready for start execution after the end of both T_{11} and T_{12}. There is a static react $st!(T_2)$ to start T_2 at state $signaled$.
- For a conditional connector, there is a condition associated with the connector. After end of T_1, the condition is evaluated and only the task with a true condition branch are ready for start execution. There are static react $st!(T_{21})$ at state $signaled1$ and static react $st!(T_{22})$ to start T_{22} at state $signaled2$.
- For an or-join connector, after the task in either branches is executed successfully, the T_2 becomes ready for start execution. There is a static react $st!(T_2)$ to start T_2 at the state $signaled$.

Chapter 6

Soundness of Operational Semantics

To relate our operational semantics with the temporal logic specifications, we must produce a CTL structure from a statechart so that we can verify whether the points in the CTL structure satisfy the specifications. In Section 6.1, we give some definitions of a statechart. In Section 6.2, we show the pattern composition. In Section 6.3, we propose an approach to derive a CTL structure from a statechart. In Section 6.4, we give a labeling model checker algorithm. In Section 6.5, we prove soundness of agent behavior models. In Section 6.6, we describe the inter-agent and intra-agent pattern operational sequences. In Section 6.7, we give the exceptional handling in general.

6.1 Definition of a Statechart

We use Figure 5.1 as an example to explain the definitions of a statechart.

Definition 4 A *state* is the main element of a statechart. A state may be idle, invoke activities. Each state s has a $type$. For example, C, D, E, F, G, H, I, J are states in the statechart example. ▮

Definition 5 *Activities* are operations performed in a state and are not represented graphically. ▮

Definition 6 A transition label lbl is composed of an event, E, a condition, C, and an action, A, and is written as $E[C]/A$. E and A may be empty and C may be true. ▮

Definition 7 *Triggers* are the dynamic elements of a statechart. Triggers cause state transitions. Events, conditions, or a combination can be triggers. The events are classified into two types: *external* and *internal* events. An external event occurs randomly outside a statechart while an internal event is generated deterministically within a statechart. ▮

Definition 8 A statechart is a tuple (B, S, h, $type$, δ, T), where

- B is a finite set of primitive events, conditions, and actions.
- S is a finite set of state names.
- h: $S \rightarrow 2^S$ is the hierarchy function. It defines the set of children of that state for each state. We have $h(R)$ = $\{I, J\}$ in the statechart example. The hierarchy function generates a tree:
  - There is exactly one state $r \in S$ that has no parent: $\forall\ s \in S : r \notin h(s)$. This is a root state of a statechart. R is the root state of the statechart example.
  - If $s \neq s'$, then $h(s) \cap h(s')$ = $\emptyset$.
- $type$: $S \rightarrow$ {PRIM, AND, OR} is the type function. It gives for each state its type: PRIM-state, AND-state or OR-state.
  - PRIM-states have no children: $\forall\ s \in S$, $type(s)$ = PRIM iff $h(s)$ = $\emptyset$. In the statechart example, $type(I)$ = PRIM.
  - OR-states contain a statechart with one or more sub-states (each of which may again be either PRIM, AND, or OR); the sub-states are connected by transitions. In the statechart example, $type(G)$ = OR.
  - AND-states consists of two or more sub-states graphically separated by dashed lines. This introduces concurrency. In the statechart example, $type(J)$ = AND.
- δ: $S \rightarrow 2^S$ is a default state function. It gives the default sub-states of a state. The function δ satisfies the following requirements for all states s:
  - $\delta(s) \subseteq h(s)$.

- If $type(s)$ = AND, then $\delta(s) = h(s)$.
- If $type(s)$ = OR, then $\delta(s) = s'$, where s' is graphically marked with a small arrow pointing to the state.

In the statechart example, $\delta(G) = \{C\}$.

- T is a finite set of transitions. A transition t is a tuple ($Source$, $Dest$, $Trigger$/ $Action$), where
  - $Source \subseteq S$ is a nonempty orthogonal set of source states. Informally, this means that the state can be in all of source states at the same time. We will define orthogonal below.
  - $Dest \subseteq S$ is a nonempty, orthogonal set of destination states.
  - $Trigger \subseteq B$ is an expression built from the events in E and the conditions C using negation, conjunction, and disjunction. The trigger serves as a condition when the transition may be taken.
  - $Action \subseteq B$ is a possibly empty set of events generated when this transition is taken.

t_1, t_2, t_3, t_4, t_5, t_6 are transitions in the statechart example. ▮

Here $type$ and $hierarchy$ functions for statecharts can represent some relationships among states. However they cannot represent other useful relationships among states of a statechart. For example, we cannot represent relationships between R and C, or D and I by the above definition of $type$ and $hierarchy$ in the statechart example. Thus some new concepts are introduced below. These concepts can also be used to describe basic configurations of a statechart.

For two states s and $s' \in S$, we say that s is an ancestor of s' iff $s' \in h^*(s)$. h^* here is the Kleene closure of h, i.e., $h^* = s \cup h(s) \cup h^2(s) \ldots$. s is *ancestrally related* to s' if s is an ancestor of s', or if s' is an ancestor of s. In the statechart example, J and C are ancestrally related.

For a set $S' \subseteq S$ of states, the *least common ancestor* of S' is the unique state s such that:

- $S' \subseteq h^*(s)$.

- For all $s' \in S$: if $S' \subseteq h^*(s')$, then $s \in h^*(s')$.

Definition 9 Two states s, s' are *orthogonal* (written $s \perp s'$) if they are not ancestrally related, and their *least common ancestor* is an AND-state. When two states are orthogonal, the statechart can be in both of them at the same time. For example, C and E are orthogonal in Figure 5.1. ▮

Definition 10 A set $S' \subseteq S$ of states is *consistent*, if for all s, $s' \in T$, either s and s' are ancestrally related, or $s \perp s'$. ▮

Definition 11 S' is maximally consistent, if for every state $s \in S$-S', $S' \cup \{s\}$ is not consistent. ▮

Definition 12 A *basic configuration* is a maximally consistent set of states. A configuration contains for each state s all states above s in the hierarchy. If $type(s)$ = OR, then it contains exactly one child of s, if $type(s)$ = AND, it contains all children of s. ▮

The *initial configuration* of a statechart is a basic configuration with root state r. In the example, we have the initial configuration $\{R, J, G, H, C, E\}$.

Definition 13 A statechart execution SC_e is a tuple (S_c, T_e, c_0), where:

- S_c is a set of basic configuration states.
- T_e is the set of transitions, $\forall\, t \in T_e$, $t = (S_1, S_2, lbl)$, and $S_1, S_2 \in S_c$.
- c_0 is the *initial configuration.* ▮

6.2 Pattern Composition

Pattern composition is an important concept in object-oriented design and analysis. Here we will use the concept to compose agent behaviors. We will compose different agent behavior models, which support various metacommitment patterns. In practice, these patterns may often be modified, such as updating some states or transitions. Further, to adapt different practice requirements, we also have to introduce some new patterns or remove some old patterns. Thus the agent behavior

models often need to be changed. It isn't convenient to build these models by hand. We propose a procedure to generate agent behavior models from a given set of metacommitment patterns.

Each metacommitment pattern can be represented as a flat statechart. The following procedure is used to generate our agent behavior models from a set of metacommitment patterns.

1. Annotating the states, transitions for each metacommitment pattern.

2. Translating each pattern into a flat statechart.

3. Merging multiple flat statecharts into a statechart to represent the agent behavior.

4. Introducing hierarchy for the statechart to make it a formal agent behavior model.

6.2.1 Annotation and Translation

First we need to carefully define states and transitions of each metacommitment. For a state in a statechart, there may be some activities in it. In order to focus on the investigation of pattern compositions, we ignore illustration of some internal activities of statecharts.

Figure 6.1 shows the statecharts we generate from our commitment patterns. *Entertain request* pattern can be annotated with *initial* and *computing* states, *start, request, refuse* and *reject* transitions. *Notify* pattern can be annotated with *computing* and *committed* states, and *computed_done/inform* transitions. *Entertain-update* pattern can be annotated with *committed* and *recomputing* states, *update* and *correct* transitions. *Renotify* pattern can be annotated with *recomputing* and *committed* states, and *recomputed_done*[$\neg$*valid_commitment*]/*inform*$\wedge$*withdraw* transition. Notice that here we use different states with the same name: *computing. Computing* state is used for reasoning for the first time. *Recomputing* state is used for reasoning for the second or later time. *Dissatisfy* pattern can be annotated with *committed* and *rejected* states, and a *reject* transition. *Abort* pattern can be annotated with *rejected* and *initial* states, and an *abort/withdraw* transition.

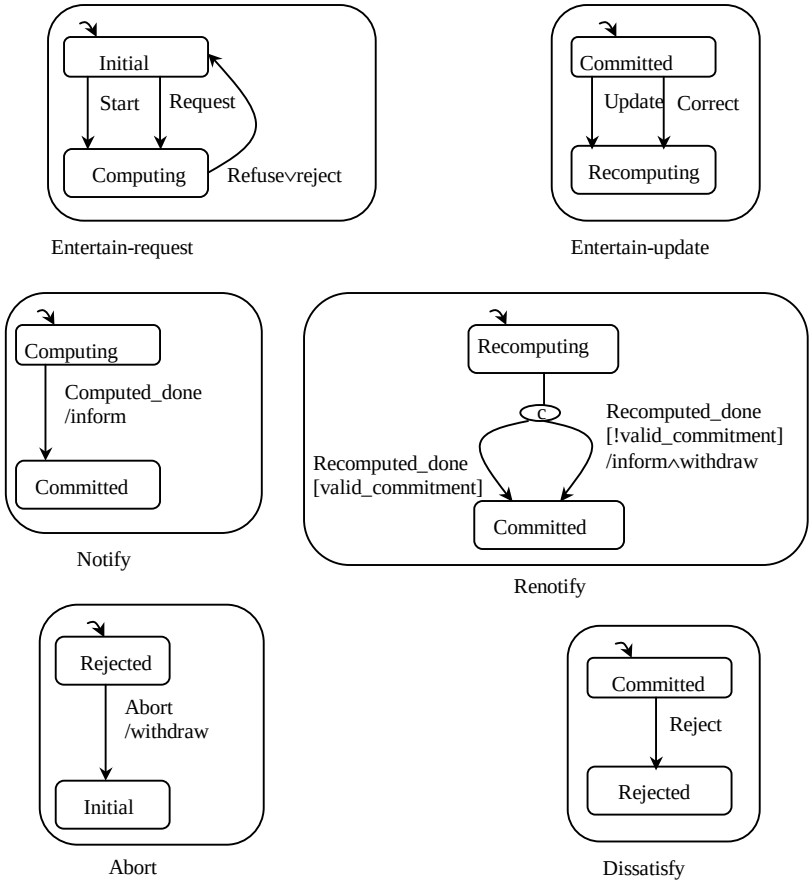


Figure 6.1: Statecharts generated from metacommitment patterns

6.2.2 Merging and Hierarchy

In order to merge a set of flat statecharts into a statechart, we have to introduce some concepts for the composition.

- *Identical state.* If the names of two states in different statecharts are the same, we take the two states to be identical. This is justified because all the patterns are based on the same agent behavior model. For example, *computing* state in the *notify* pattern statechart is the same as *computing* state in the *entertain-request* pattern statechart.

- *Identical transition.* Two transitions are identical if they have the same label, are triggered from the identical state, or lead into the identical state. This is useful for introducing a hierarchy.
- *Meaningful abstraction.* Meaningful abstraction is used as a means for creating the statechart hierarchy so that we can consolidate the related behaviors into one state and separate the unrelated behaviors into different states. For example, we can separate the agent behavior by the times agent reasons. If the pattern's states are related to first time reasoning, we put them in one super-state, and we put other pattern states in another different super-state.
- *Default states and hierarchy function.* After the statecharts are merged we need to introduce some default transitions for the composite statechart. After introducing the hierarchy function, we need to give new hierarchy function for the composite states.

In order to merge two statecharts into a statechart, we use the union of two statecharts. Let Sc = (B, S, h, $type$, δ, T) be the union of two statecharts Sc_1 = (B_1, S_1, H_1, $type_1$, δ_1, T_1) and Sc_2 = (B_2, S_2, H_2, $type_2$, δ_2, T_2).

- $B = B_1 \bigcup B_2$, $S = S_1 \bigcup S_2$, $T = T_1 \bigcup T_2$
- $h = h_1$ if $s \in S_1$, or $h = h_2$ if $s \in S_2$.
- $type = type_1$ if $s \in S_1$, or $type = type_2$ if $s \in S_2$.
- $\delta = \delta_1$ if $s \in S_1$, or $\delta = \delta_2$ if $s \in S_2$.
- For each pair of identical states, s_1 and $s_2 \in S$, add two ϵ-transitions from s_1 to s_2 and from s_2 to s_1.
- Update the hierarchy function for new states.
- Select a state as a default state from the two statecharts.

We can derive the flat statechart by removing the ϵ-transition, just like merging two automata by introducing the ϵ-transition and transforming a non-deterministic finite automaton into a deterministic finite automaton by removing ϵ-transition.

Figure 6.2 shows how to merge two statecharts of *entertain-request* and *notify* pattern. We apply the approach to merge all flat statecharts.

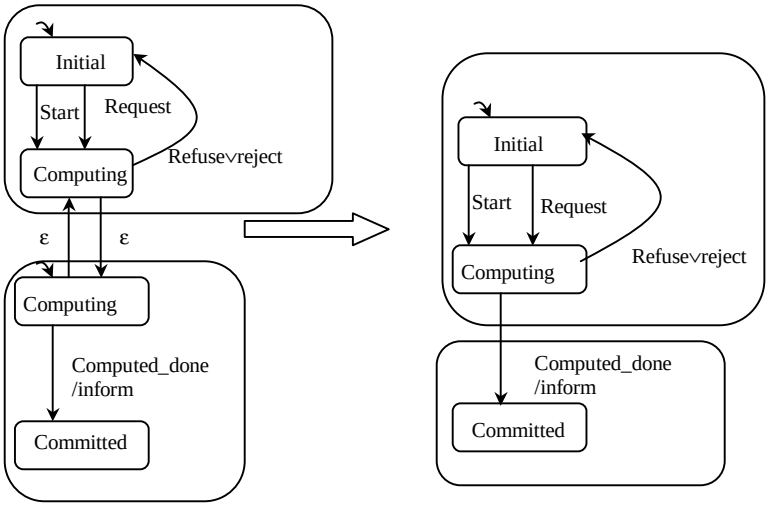


Figure 6.2: Merge two statecharts

So far, we have shown how to compose our metacommitment patterns by merging these flat statecharts of commitment patterns. We obtain a flat statechart without any hierarchy. In practice, however, statechart designs tend to get very large. So the judicious use of hierarchy and orthogonality is crucial to readability and maintainability of the designs. Several considerations go into making a well-designed statechart. They include the consolidation of related behavior, the separation of unrelated behavior, and the introduction of meaningful abstractions. For the agent behavior model, we can separate the behavior of reasoning in the second or later time from that in the first time. We can consolidate these behaviors in respective partitions. A basic agent behavior model shown in Figure 5.2 can be composed from the statecharts shown in Figure 6.1. From temporal logic specification for metacommitment patterns, two observations:

- The *abort* can be generated from any state of state B_2.
- t_6 should be triggered in any state of state B_2 if there is any commitment updated.
- When the system is in recomputing state, we allow the system to receive any reject event.

When we add the three cases, our agent behavior model can react to any request, or correct and it can start its reasoning by itself. This increases the flexibility of the basic agent behavior model.

6.3 CTL Structure Derivation

To relate our operational semantics with the temporal logic specifications, we must produce a CTL structure τ = (Q, R, P, s_0) from our statecharts. It is difficult to prove eventuality with statecharts. An external event may never occur during the execution of a statechart. Thus eventuality of a condition is always falsified if external events are required to reach the condition. To address the problem and capture possible executions of a statechart, we separate points in our CTL structure corresponding to the same basic configuration. Further we introduce a point for each transition. We add additional members of the relationships into R. We provide the following rules for deriving CTL structure from the statechart.

For each transition $t = (S_1, S_2, lbl)$ and lbl = $E[C]/A$, we create a point S' that is between S_1 and S_2. S' has atomic propositions (denoted by $\{E,C\}$), which can be derived from E and C - for reason of space, we don't elaborate this here. Similarly action A generates atomic propositions (denoted by $\{A\}$) in S_2. We place the pair (S_1, S'), (S', S_2) into R. If S_2 is reached from different transitions and has different propositions, we separate the basic configuration S_2 into different points in the CTL structure. If E includes an external event, we place the (S_1, S_1) into R. The propositions in S_1 don't change. This forces an execution from S_1 to S_1 if the external event doesn't occur. If there is a set of activities $\{acts\}$ in S_2, each activity generates an atomic proposition in state S_2. We denote the set of atomic propositions as $\{acts\} \subseteq P(S_2)$.

Now the CTL structure τ = (Q, R, P, s_0) may be derived from a statechart S_c = (S, S_{bc}, T, c_0) by the following rules.

1. We introduce a point $S' \in Q$. $P(S')$ = $\{E,C\}$, $P(S_2)$ = $\{A\} \bigcup \{acts\}$, (S_1, S'), (S', S_2) $\in$ R iff t = (S_1, S_2, lbl) $\in T$, S_1, $S_2 \in S_c$ and $\{acts\}$ are a set of activities in S_2, where lbl = $E[C]/A$.

2. If E consists of an external event in the transition $t = (S_1, S_2, lbl)$, we introduce a relationship, $(S_1, S_1) \in R$.

3. If the basic configuration $s_c \in S_{bc}$ is reached from different transitions and has different propositions generated by the first rule, we separate the s_{bc} into different points.

4. For each point $s \in Q$, name of $s \in P(s)$.

Following the above rules, we give an algorithm to generate a CTL structure from a statechart.

Algorithm 1 *CTL structure generation from a statechart*

1. Start an initial basic configuration s_0, let it as the first point in the CTL structure with proposition $\{$state name of $s_0\}$. Let $s = s_0$.

2. Choose each transition t of s.

   If there isn't any transition available in the initial basic configuration, stop.

   If there isn't any available transition at s, then return.

   Otherwise, check if the transition t is selected.

   - If it is, add binary relation from s to t.
   - Otherwise, choose the transition t and go to next step.

3. Let this transition t as a point with propositions $\{$E,C$\}$. If t contains an external event, add a binary relation from s to s. Add a binary relation from s to t, next go to the basic configuration s', which the transition t will reach.

4. Check if the state s' is already reached.

   - If it isn't, let s' as a new point with propositions $\{A\} \cup \{$acts$\} \cup \{$state name$\}$. Add a binary relationship from t to s'.

- Otherwise, saying it a point v, compare if the labeled propositions of v are the same as the propositions of s'. If they are the same, this transition t goes to the point v. Add a binary relationship from the point t to the point v. Otherwise, add a new point s' with propositions $\{A\}\cup\{\text{acts}\}\cup\{\text{state name}\}$. Add a binary relationship from t to s'.

5. Let's $s = s'$. Go to step 2 recursively. ∎

Theorem 1 This algorithm runs in time $O(|T||S_c|)$, where T is the set of transition and S_c is the set of basic configuration.

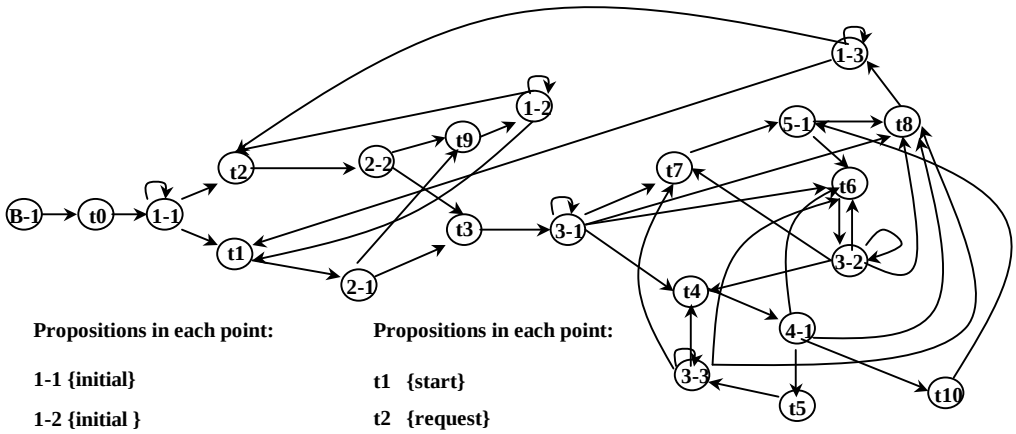
Proof.

In the second step, each basic configuration goes to all its transitions. In the third and fourth step each of these transitions can generate a new point in the worse case. Thus we have the running time $O(|T||S_c|)$ in worse case. ∎

To derive CTL structure from the statechart S_{ca}, we have to know the event type of each transition. *Start*, *update*, *computed_done*, *recomputed_done*, *refuse*, *inform*, *withdraw*, *reject* and *abort* are internal events. *Request*, *correct* are external events. Figure 6.3 shows the CTL structure derived from the statechart of Figure 5.2 by the algorithm 1.

6.4 CTL Model Checker

In Section 4.1, we have introduced the CTL syntax and semantics. In the section, we will present an algorithm to output the points of a given CTL model that satisfies a given formula. Any CTL formula can be written as a set of atomic propositions and/or connectives. These connectives are propositional connectives $\neg$, $\vee$, $\wedge$, $\rightarrow$ and temporal connectives AG, EG, AF, EF, AX, EX, AU, and EU. Moreover, some connectives can be expressed by other connectives. In other words, there exist some equivalent relations among CTL formulas.



Propositions in each point:

1-1 {initial}

1-2 {initial }

1-3 {withdraw, initial}

2-1 {computing}

2-2 {computing}

3-1 {inform, committed}

3-2 {inform, withdraw, committed}

3-3 {committed}

4-1 {recomputing}

5-1 {rejected}

Propositions in each point:

t1 {start}

t2 {request}

t3 {computed_done}

t4 {correct, update}

t5 {recomputed_done, valid_commitment}

t6 {recomputed_done, ¬valid_commitment}

t7 {reject}

t8 {abort}

t9 {reject ∨ refuse }

t10 {reject}

Other point: ∅

Figure 6.3: CTL structure derived from the statechart S_{ca}

Definition 14 Two CTL formula ϕ and ψ are said to be semantically equivalent if any state in any model that satisfies one of them also satisfies the other. We denoted the equivalent relationship between ϕ and ψ as $\phi \equiv \psi$. ▮

Theorem 2 Any CTL formula can be transformed into a semantically equivalent formula, which can be expressed by the set of propositional connectives $\neg$, $\vee$ and temporal connectives AF, EU, and EX.

Proof.

The set of connectives $\neg$, $\vee$ are a set of connectives, which express all the propositional formulas.

$p \wedge q \equiv \neg(\neg\, p \vee \neg\, q)$ (Morgan's law).

$p \rightarrow q \equiv \neg\, p \vee q$.

We recall the CTL semantics. AF, EU, and EX form a set of temporal connectives, which express all formulas that consists of temporal connectives. We have the equivalent relations.

$\mathsf{EF}(f) \equiv \mathsf{E}[\text{True} \cup f]$.

$\mathsf{EG}(f) \equiv \neg\mathsf{AF}(\neg f)$.

$\mathsf{AG}(f) \equiv \neg\mathsf{EF}(\neg f)$.

$\mathsf{AX}(f) \equiv \neg\mathsf{EX}(\neg f)$.

$\mathsf{A}(f_1\ \mathsf{U} f_2) \equiv \neg(\mathsf{E}[\neg f_2 \mathsf{U}(\neg f_1 \wedge \neg f_2)] \vee \mathsf{EG}\neg f_2)$.

Thus we prove the theorem.∎

The model checker algorithm we will present here is a labeling algorithm. We first translate a formula f in terms of the connectives $\vee$, $\neg$, AF, EU, EX. Second we label the points of the CTL structure with smallest sub-formulas of f. Third, we label the points of the CTL structure towards larger sub-formulas by using smaller sub-formulas of f until we label the points of CTL with f.

Algorithm 2 *Labeling model checker algorithm*

Label each point with the formula by the following cases.

1. $f_1 \vee f_2$: label a point with $f_1 \vee f_2$ if the point is labeled either with f_1 or with f_2.

2. $\neg f$: label a point with $\neg f$ if the point is not labeled with f.

3. $\mathsf{AF} f$:

   - If any point is labeled with f, label it with $\mathsf{AF} f$.
   - Repeat: label any point $\mathsf{AF} f$ if all of its successors are labeled with $\mathsf{AF} f$, until there is no change.

4. $\mathsf{E}[f_1\ \mathsf{U} f_2]$:

   - If any point is labeled with f_2, label it with $\mathsf{E}[f_1\ \mathsf{U} f_2]$.
   - Repeat: label any point with $\mathsf{E}[f_1\ \mathsf{U} f_2]$ if it is labeled with f_1 and at least one of its successors is labeled with $\mathsf{E}[f_1\ \mathsf{U} f_2]$, until there is no change.

5. $\mathsf{EX}f$: label any point with $\mathsf{EX}f$ if one of its successors is labeled with f.∎

Theorem 3 The complexity of the labeling model checker algorithm is $O(|f||V|(|V|+|E|))$, Where

- $|f|$ is the number of connectives in a formula.
- $|V|$ is the number of points.
- $|E|$ is the number of binary relationships.

Proof.

We consider the complexity of the algorithm for each case.

- The complexity of the first case is $O(|V|)$ because the algorithm checks if f_1 or f_2 at each point is labeled.
- The complexity of the second case is $O(|V|)$ because the algorithm checks if $\neg f$ at each point is labeled.
- The complexity of the third case is $O(|V|(|V|+|E|))$. We prove it as follows:

  We need to have the complexity $O(|V|)$ to go through each point for the first step.

  We need to have the complexity $O(|V|+|E|)$ in the second step because we label each point and need to check each of its successors. Once a point is labeled $\mathsf{AF}f$, we cannot remove it. We need to repeat maximum $|V|$ time. Let's assume that there is at least a point labeled with $\mathsf{AF}f$ at one time. Otherwise we will stop in the middle if there is any point label changed. We can repeat maximum $|V|$ to label all points.
- For the fourth case, we can use the same proof as that of $\mathsf{AF}f$. It has the complexity of $O(|V|(|V|+|E|))$.
- For the fifth case, we have to check each point and its successors. It has the complexity of $O(|V|+|E|)$.

For any formula, we have to run at most $|f|$ time so that we can get its sub-formulas labeled. Thus the overall complexity of the algorithm is $O(|f||V|(|V|+|E|))$. ∎

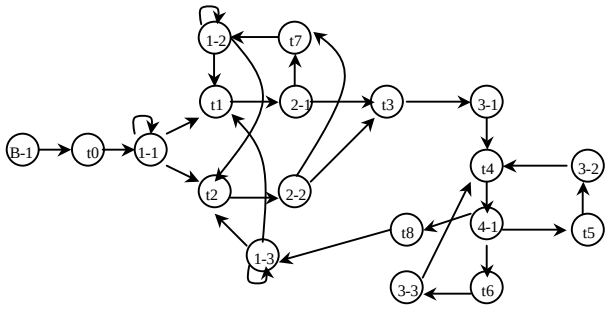
6.5 Soundness

Definition 15 τ,$s \models \phi$ means that the formula ϕ holds at state s in the CTL structure τ. ▮

Definition 16 A statechart S_c is sound with respect to a formula ϕ iff ($\forall\tau$: (S_c generates τ) $\Rightarrow$ $\tau, s_0 \models \phi$). A statechart S_c is sound with respect to a specification $\{\phi_1, \phi_2, \ldots, \phi_n\}$ iff S_c is sound with respect to each ϕ_i. ▮

In the following subsection, we will use the conversation-oriented agent behavior models to check the soundness. The task-oriented models can use the same approach to check.

6.5.1 Soundness for Monitor Agent Behavior Model



Propositions in each point:
- 1-1 {initial}
- 1-2 {initial}
- 1-3 {withdraw, initial}
- 2-1 {computing}
- 2-2 {computing}
- 3-1 {inform, committed}
- 3-2 {committed}
- 3-3 {inform, withdraw, committed}
- 4-1 {recomputing}

Propositions in each point:
- t1{start}
- t2{request}
- t3{computed_done}
- t4 {true}
- t5{recomputed_done, valid_commitment}
- t6{recomputed_done, ¬valid_commitment}
- t7{refuse∨reject}
- t8{abort}
- Other point: ∅

Figure 6.4: The CTL structure derived from the statechart S_{ma}

Figure 6.4 is CTL structure derived from monitor agent behavior model.

Theorem 4 Any agent that follows the monitor agent behavior model is sound with respect to any specification consisting of the patterns {*entertain-request*, *entertain-update*, *notify*, *renotify*}.

Proof.

For monitor agent behavior model S_{ma}, *start*, *refuse*$\vee$*reject*, *computed_done*, *recomputed_done*, *inform*, *withdraw*, and *abort* are internal events. *Request* is external events. We apply Algorithm 1 to S_{ma}. We obtain the CTL structure, which has 19 points and 29 binary relationships (see Figure 6.4). We apply Algorithm 2 to the CTL structure.

First, *notify* has antecedent condition with *computed_done*. The point t_3 is labeled with *computed_done*, and its successor 3-1 labeled with AF*inform* because 3-1 has been labeled with *inform*. t_3 can be labeled with *computed_done* and AF*inform*. For other points, we can labeled with $\neg$*computed_done*. Further, each point can be labeled with *computed_done* $\rightarrow$ AF*inform*. Thus we have τ,$t_0 \models$ *notify*.

Second, *renotify* has antecedent conditions with *recomputed_done*$\wedge\neg$*valid_commitment*. The point t_6 is labeled with *recomputed_done*$\wedge\neg$*valid_commitment*, and its successor 3-3 labeled with AF[*inform*$\wedge$*withdraw*] because 3-3 is labeled with *inform* and *withdraw*. Thus t_6 can be labeled with AF[*inform*$\wedge$*withdraw*], and *recomputed_done*$\wedge\neg$*valid_commitment*. Point t_5 is labeled with *recomputed_done* and *valid_commitment*. So it can be labeled with $\neg$(*recomputed_done*$\wedge\neg$*valid_commitment*). Other points can be labeled with $\neg$(*recomputed_done*$\wedge\neg$*valid_commitment*). Thus we have each point labeled with *recomputed_done*$\wedge\neg$*valid_commitment* $\rightarrow$ AF[*inform*$\wedge$*withdraw*]. So we have τ,$t_0 \models$ *renotify*.

Third, *entertain-request* has the antecedent condition with *start* or *request*. The point t_1 is labeled with *start* and t_2 is labeled with *request*. Let's consider first antecedent condition *start*. t_7 can be labeled with AF[*reject*$\vee$*refuse*] because t_7 is labeled with *reject*$\vee$*refuse*. t_3 can be labeled with AF*notify* because we have proved that t_3 is labeled with *notify*. So 2-1 can be labeled with AF[*notify* $\vee$*reject*$\vee$*refuse*]. Furthermore, we can label t_1 with AF[*notify*$\vee$*reject*$\vee$*refuse*] and *start*. Thus we can label t_1 with *start* $\rightarrow$ AF[*notify* $\vee$ *reject* $\vee$ *refuse*]. For other points, we can label them with $\neg$*start*. Thus we can label each point with *start* $\rightarrow$ AF[*notify*$\vee$*reject*$\vee$*refuse*].

In the same way, we can check each point with *request* $\rightarrow$ AF[*notify*$\vee$ *reject*$\vee$*refuse*].

Thus we prove τ,$t_0 \models$ *entertain-request*.

Fourth, *entertain-update* hasn't any explicit antecedent condition. The recomputation is in $4-1$ because it is labeled with *recomputing*. t_5 and t_6 can be labeled with AF*renotify* because they are labeled with *renotify*. t_8 can be labeled with AF*withdraw* because its successor 3-3 is labeled with AF*withdraw*. So we can label 4-1 and t_4 with AF[*renotify*$\vee$*withdraw*]. That means that *renotify*, or *withdraw* eventually occurs after recomputing. For other points, we can label them with $\neg$*correct* or $\neg$*update*. Thus, we can have each point labeled with *correct* $\rightarrow$ AF[*renotify*$\vee$*withdraw*] and *update* $\rightarrow$ AF[*renotify*$\vee$*withdraw*]. Thus we show τ,t_0 $\models$ *entertain-update*.

Thus we prove the theorem. ∎

In order to monitor some situations of a system, the monitor agent needs to report or notify any change to its listener(s). The following formulas represent some properties of the agent to inform any change.

- τ,$t_0$$\models$*entertain-update*.

  It is the liveness properties. It represents that an agent entertains the update.

- τ,$t_0$$\models$*renotify*.

  It is the liveness properties. It represents that an agent notifies any change.

6.5.2 Soundness for Reactive Agent Behavior Model

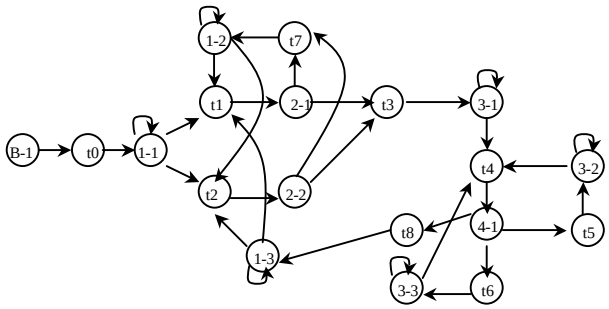
Theorem 5 Any agent that follows the reactive agent behavior model is sound with respect to any specification consisting of the patterns {*entertain-request*, *entertain-update*, *notify*, *renotify*}.

Proof.

For reactive agent behavior model S_{ra}, *start*, *refuse*$\vee$*reject*, *computed_done*, *recomputed_done*, *inform*, *update*, *withdraw*, and *abort* are internal events. *Request* and *correct* is external events. We apply Algorithm 1 to S_{ra}. We obtain the CTL structure, which has 19 points and 32 binary relationships (see Figure 6.5). We apply Algorithm 2 to the CTL structure.

We can use the same method to have the following properties as that we have used in S_{ma}.

- τ,$t_0$$\models$*notify*.



Propositions in each point:
1-1 {initial}
1-2 {initial}
1-3 {withdraw, initial}
2-1 {computing}
2-2 {computing}
3-1 {inform, committed}
3-2 {committed}
3-3 {inform, withdraw, committed}
4-1 {recomputing}

Propositions in each point:
t1{start}
t2{request}
t3{computed_done}
t4 {correct, update}
t5{recomputed_done, valid_commitment}
t6{recomputed_done, ¬valid_commitment}
t7{refuse∨reject}
t8{abort}
Other point: ∅

Figure 6.5: The CTL structure derived from the statechart S_{ra}

- τ,$t_0$$\models$*renotify*.
- τ,$t_0$$\models$*entertain-request*.

entertain-update has antecedent conditions *update* and *correct*. t_5 and t_6 can be labeled with AF*renotify* because they are labeled with *renotify*. t_8 can be labeled with AF*withdraw* because its successor ls labeled with AF*withdraw*. So we can label 4-1 and t_4 with AF[*renotify*$\vee$*withdraw*]. Thus we can label t_4 with *update* $\rightarrow$ AF[*renotify*$\vee$*withdraw*] and *correct* $\rightarrow$ AF[*renotify*$\vee$*withdraw*]. For other points, we can label them with $\neg$*correct* or $\neg$*update*. So can have each point labeled with *correct* $\rightarrow$ AF[*renotify*$\vee$*withdraw*] and *update* $\rightarrow$ AF[*renotify*$\vee$*withdraw*]. Thus we show τ,t_0 $\models$ *entertain-update.*

Thus we prove the theorem. ∎

A reactive system needs to response to some stimuli. In the reactive agent behavior model, it can response the following request.

1. Every time a *request* stimulus occurs, eventually there is a response. We have the formula.

$\tau, t_0 \models \mathsf{AG}[\mathit{request} \rightarrow \mathsf{AF}[\mathit{inform} \vee \mathit{refuse} \vee \mathit{reject}]]$

This formula can be derived by the following formulas.

- $\mathsf{AG}[\mathit{request} \rightarrow \mathsf{AF}[\mathit{computed_done} \vee \mathit{refuse} \vee \mathit{reject}]]$
- $\mathsf{AG}[\mathit{computed_done} \rightarrow \mathsf{AF}\mathit{inform}]$

2. Every time a *correct* stimulus occurs, eventually there is a response. We have the formula.

   $\tau, t_0 \models \mathsf{AG}[\mathit{correct} \rightarrow \mathsf{AF}[(\mathit{inform} \wedge \mathit{withdraw}) \vee \mathit{withdraw}]]$

   This formula can be derived by the following formulas.

   - $\mathsf{AG}[\mathit{correct} \rightarrow \mathsf{AF}[\mathit{recomputed_done} \vee \mathit{withdraw}]]$
   - $\mathsf{AG}[\mathit{recomputed_done} \rightarrow \mathsf{AF}[\mathit{inform} \wedge \mathit{withdraw}]]$

The following properties represent an agent to inform any change. For details, you can find in Section 6.5.1.

- $\tau, t_0 \models$ *entertain-update*
- $\tau, t_0 \models$ *renotify*

6.5.3 Soundness for Basic Agent Behavior Model

Theorem 6 Any agent that follows agent behavior model is sound with respect to any specification consisting of the patterns {*entertain-request*, *entertain-update*, *dissatisfy*, *notify*, *renotify*, *abort*}.

Proof.

For basic agent behavior model S_{ca}, *start*, *refuse*$\vee$*reject*, *computed_done*, *recomputed_done*, *inform*, *update*, *withdraw*, and *abort* are internal events. *Request* and *correct* is external events. We apply Algorithm 1 to S_{ca}. We obtain the CTL structure, which has 22 points and 42 binary relationships (see Figure 6.3). We apply Algorithm 2 to the CTL structure.

We can use the same method to have the following properties as that we used in S_{ma}.

- $\tau, t_0 \models$ *notify*.

- τ,$t_0$$\models$*renotify.*
- τ,$t_0$$\models$*entertain-request.*

Abort has antecedent conditions *abort.* 1-3 can be labeled with AF*withdraw* because 1-3 is labeled with *withdraw*. So t_8 can be labeled with *abort* and AF*withdraw*. Furthermore, t_8 can be labeled with *abort* $\rightarrow$ AF*withdraw*. Other points can be labeled with $\neg$*abort.* So each point can be labeled with *abort* $\rightarrow$ AF*withdraw*. Thus we prove τ,$t_0$$\models$*abort.*

Dissatisfy has antecedent condition *reject.* There are three points, t_7, t_9, and t_{10} with the antecedent condition. t_9 is a special point, which doesn't have any associated commitment and goes to an initial point. There isn't any necessary to withdraw anything. So we think that the antecedent condition is satisfied. We have t_6 labeled with *renotify*, and t_8 labeled with *abort.* So they can be labeled with AF*renotify* and AF*abort* respectively. So 5-1 can be labeled with AF[*renotify*$\vee$*abort*]. Furthermore, t_7 and t_{10} can be labeled with AF[*renotify*$\vee$*abort*]. So we can label t_7 and t_{10} with *reject* $\rightarrow$ AF[*renotify*$\vee$*abort*]. Other points can be labeled with $\neg$*reject.* We can label each point with *reject* $\rightarrow$ AF[*renotify* $\vee$*abort*]. Thus we prove τ, t_0 $\models$ *dissatisfy.*

Entertain-update has antecedent condition *update* or *correct.* We have known t_6 and t_5 labeled with *renotify*, t_{10} labeled with *dissatisfy* and t_8 labeled with *abort.* So we can label 4-1 with AF[*renotify*$\vee$*dissatisfy*$\vee$*abort*]. Furthermore, we can label t_4 with *update* $\rightarrow$ AF[*renotify*$\vee$*dissatisfy*$\vee$*abort*], or *correct* $\rightarrow$ AF[*renotify*$\vee$*dissatisfy*$\vee$*abort*]. Other points can be labeled with $\neg$*correct*, or $\neg$*correct*. So each can be labeled with *update* $\rightarrow$ AF[*renotify*$\vee$*dissatisfy*$\vee$*abort*], or *correct* $\rightarrow$ AF[*renotify*$\vee$*dissatisfy*$\vee$*abort*]. Thus we prove τ,$t_0$$\models$*entertain-update.*

Thus we prove the theorem. ∎

The basic agent behavior model has all of these properties of the monitor and reactive agent models. It has the following properties.

Every time a *reject* stimulus occurs, eventually there is a response. We have the formula.

τ, t_0 $\models$ AG[*reject* $\rightarrow$ AF[(*inform*$\wedge$*withdraw*)$\vee$*withdraw*]]

This formula can be derived by the following formulas.

- AG[*reject* $\rightarrow$ AF[*recomputed_done*$\vee$*withdraw*]]
- AG[*recomputed_done* $\rightarrow$ AF[*inform*$\wedge$*withdraw*]]

6.6 Intra-agent and Inter-agent Pattern Operational Sequences

An agent behavior model consists of a set of composable metacommitment patterns. It enforces the agent to behave in a coherent way by following the model. The agent doesn't lose its operational control. It is a good way for us to specify the interaction by using metacommitment patterns, then go to the details for each pattern. It is ideal that an agent interacts in the following operational sequences in process interoperation.

An agent A_1 uses *notify* pattern to inform some results of another agent A_2. A_2 will entertain the request, notify its consumer and go to its committed state. When A_1 uses *renotify* pattern to inform some updated results of A_2, A_2 will use *entertain-update* pattern to entertain the request, renotify its consumer agent, and then go to committed state again. However in a real scenario, it is not like that case. It often encounters the cases, such as new updated result arrival earlier, out-of-order received results, communication delay, or an agent execution delay because of the revision and exception. For example, A_2 has received the results from A_1 who uses *renotify* pattern before A_2 hasn't finished the *entertain-request* pattern operation. An agent must assure that it entertains the latest request. Thus A_2 must entertain the latest updated results before A_2 notifies its consumer. Thus we need to have some mechanism to enforce these constraints. In order to consider the general case, we consider the basic agent behavior model. Here we represent the operational sequence of patterns. If one pattern $pattern_1$ operates before another pattern $pattern_2$, we represent it as: $pattern_1 \prec pattern_2$.

First, an agent must enforce its internal execution sequence. It is easy that we can give the operational sequence of metacommitment patterns following the basic agent behavior model. The operational sequences are as follows:

- *entertain-request* $\prec$ *notify*

- *notify* $\prec$ *abort*
- *notify* $\prec$ *dissatisfy*
- *entertain-update* $\prec$ *renotify*
- *entertain-update* $\prec$ *abort*
- *entertain-update* $\prec$ *dissatisfy*
- *renotify* $\prec$ *abort*
- *renotify* $\prec$ *dissatisfy*
- *renotify* $\prec$ *entertain-update*
- *dissatisfy* $\prec$ *renotify*
- *dissatisfy* $\prec$ *abort*

Because an agent can send an updated information to another agent at any time, who cannot predict which one is the latest one. The agent will make its decisions based on observing the received information and its internal execution.

Many applications require us to identify the cause and effect relationships. If e is a cause of f, then e must occur before f in physical time. Thus, the causally precedence relation is a subset of *occurred-before* relation. We will construct a mechanism in distributed systems that gives us information about the causally precedence relation. Multi agent systems are loosely-coupled message-passing distributed systems without any shared memory or a global clock. Availability of a logic clock during distributed computation makes it easier to solve the MAS problem. We do assume that communication is reliable, don't assume that the message are pre-ordered. Each agent A_i maintain a vector $V_i[1...n]$, where $V_i[j]$is the logical clock of A_i and describes the logical time progress in A_j. Each process A_i executes the following algorithm.

Algorithm 3 Logical clock generation:

1. Rule for any initial state:

   $V_i[j]$ = 0 $\forall$ j = 1,...n;

2. Rule for an internal event produced by A_i:

   $V_i[i] = V_i[i] + +;$

3. Rule for a *send* operation

   $V_i[i]$ is sent as part of message.

4. Rule for a *receive* operation from A_k to A_i

   $V_i[j] = max(V_i[j], V_k[j]);$ ∎

The algorithm is described by the initial conditions and actions taken for each event. It satisfies the following property: for any two events x and y, $x \rightarrow y \Leftrightarrow$ timestamp $(x) <$ timestamp (y).

The following three relations are defined to compare two vector timestamps, v_x and v_y:

- $v_x \leq v_y \Leftrightarrow \forall\, i\colon v_x[i] \leq v_y[i]$
- $v_x < v_y \Leftrightarrow v_x \leq v_y$ and $\exists\, i\colon v_x[i] < v_y[i]$
- $v_x \parallel v_y \Leftrightarrow \neg(v_x \leq v_y) \wedge \neg(v_x \leq v_y)$

Although timestamps are often applied on the events, or messages, here we consider timestamps to operate on metacommitment patterns. We have the following mechanism to assure that an agent always entertains the latest information when there are revisions and out-of-order patterns. If a pattern is for interaction, the pattern is associated with a message. The message is assigned with a logical clock; otherwise a pattern is assigned a logical clock by the agent who operates the pattern. Thus there is a logical clock associated with the end of each pattern. Figure 6.6 illustrates how to handle these revision problems.

1. A_1 sends results to A_2 by using the *notify* pattern. A_2 starts to entertain the results. A_1 sends its updated result again before A_2 finishes its *entertain-request* pattern execution. Thus A_2 has to entertain the latest information again before it sends its execution results to its consumer.

2. A_1 sends results and updated results to A_2 by using notify and renotify pattern respectively. A_2 receives the two results in different order. The updated results that A_1 sends by *renotify*

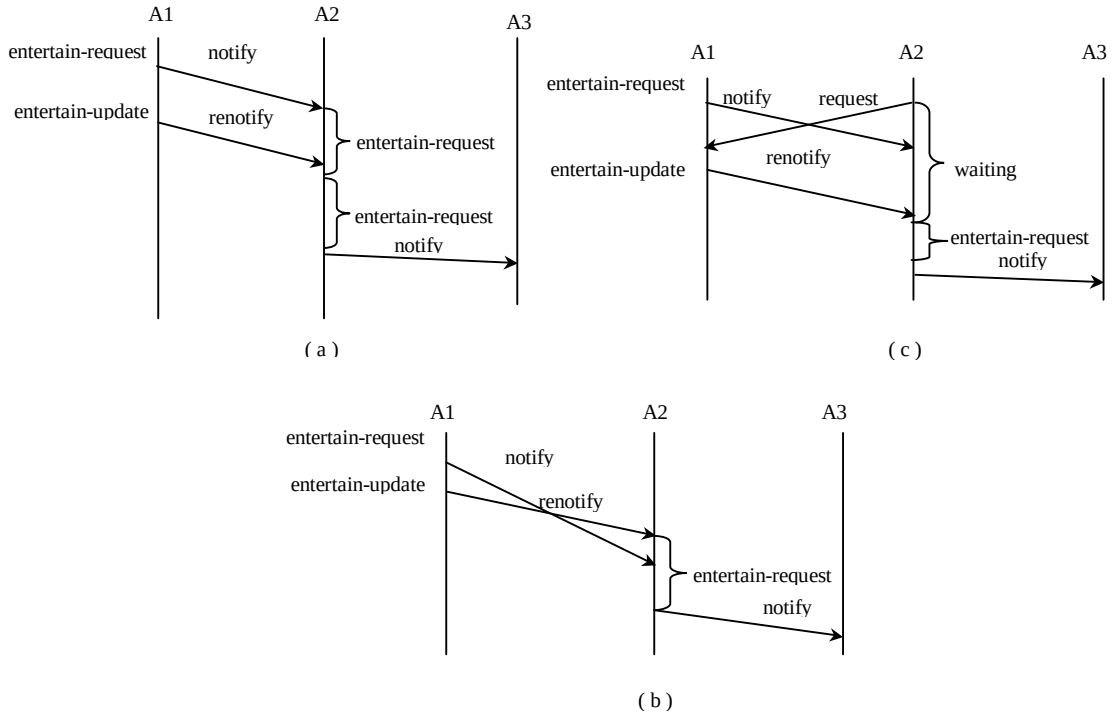


Figure 6.6: Inter-agent interaction cases

pattern are received earlier than the results A_1 sends by *notify*. A_2 will entertain the updated results and ignore the old results because for each received information there is a timestamp associated with it.

3. If A_2 requests some information from A_1, it will only entertain the received information, whose timestamp is later than the request's timestamp.

On the other hand, if a producer agent receives the request and correct request, it always checks the latest one and entertains the latest request with later logical clock.

6.7 Exception Handling

The interaction among agents causes the unanticipated exceptions. We need to handle the exception by using the interaction approaches. There is a set of exceptional categories. Let's show how

metacommitment patterns to handle these exceptions.

Revisions. This occurs when the inputs to an agent change after it has already declared its results or a request for new information is made. The agent who may have previously committed to some result may then have to ensure that its consumers too obtain the benefit of the new information. In such a case, the original producer may renotify the consumer thereby entering into a new commitment to the consumer. From the consumer's perspective, the receipt of such a notification would be an exception, because the consumer would already have processed the original version and forwarded any results to its consumers. The consumer should entertain this update.

Revisions are extremely common in most real applications of workflow technology, yet are not included in traditional metamodels. In our case, revision exceptions arise, because our framework makes the agents persistent even after the given instance of the workflow has been processed by them. If the agents are around, they can continue to observe their environment or communicate with other agents, either of which might cause them to change their mind and then to propagate their revisions further.

Our proposed approach to handling revisions is simply to require the producers to use the *renotify* consumer pattern so they send out updates and require the consumers to use the *request* message to request for new information in its request behavior model.

Incorrectness. This occurs when the inputs received by an agent from any source are simply incorrect. In this case, the incorrectness can be determined by the receiver without reference to inputs received from another source—this is what distinguishes incorrectness from inconsistency.

The inputs cause the consumer to repeatedly fail in its execution, thereby leading to its dissatisfaction. In this case, the producers must act to satisfy the consumer. Consequently, handling incorrectness exceptions involves the *dissatisfy* pattern. Since a revised output is desired, we would also typically see *dissatisfy* and *renotify* patterns in the producer, or the *reject* primitives in the consumer.

However, handling this exception can give rise to a new exception(revision) if modifications can be made. Alternatively, if the modifications cannot be made, some other exception may be raised depending on the specific circumstances.

Incompleteness. This exception occurs when an agent receives fewer inputs than it needs to execute the appropriate capability. We emphasize that incompleteness cannot be removed in the design of the workflow. A common reason for the occurrence of this exception is that the capabilities may have several method signatures. Even if the inputs are sufficient for some of the signatures, they may not be sufficient for other signatures. Usually, under special circumstances, the agent will need to try signatures other than the simplest ones, for which purposes the inputs it received are incomplete. Handling incompleteness requires metacommitment pattern of *notify* the consumer in the producer and usually also *request* primitive in its agent request behavior model in the consumer.

Ad hoc request. A task agent, during the persistent execution of a workflow discovers that it needs additional information to execute properly. It then issues that request to an agent who possesses the capability to supply that information. The requestee then supplies that information maybe by reexecuting appropriately. The challenge here is simply that the agent be able to follow a control path that diverges from whatever workflows in which it is engaged. Ad hoc requests can be handled through the *request* primitive of the agent request behavior model.

Handling ad hoc requests could cause further exceptions that must then be handled according to appropriate patterns.

Failure. Sometimes, an agent can be forced to give up when the inputs are such that the underlying database system is unable to produce an acceptable answer.

If a consumer is repeatedly dissatisfied until it decides that it is not worth trying further, it can announce that the task it was attempting cannot succeed. The decision to declare failure may be based on all available method signatures having been attempted and resulted in failure (e.g.,

because the underlying database has been corrupted). The decision can also be based on the agent's knowledge of the importance of the task to the entire workflow. Much application-specific reasoning can go into this decision, but in the end when the agent has made the decision, we only show how the agent may announce it to its colleagues.

If the failure is along a conditional branch, the rest of the workflow may survive. In the worst case, however, a failure can result in the termination of the entire workflow. The pattern to handle this exception is the *entertain-request*, or *abort* pattern.

Chapter 7

Architecture, Implementation, and Application

In Section 7.1, we introduce the architecture and implementation for our system. In Section 7.2, we show three application examples.

7.1 Architecture and Implementation

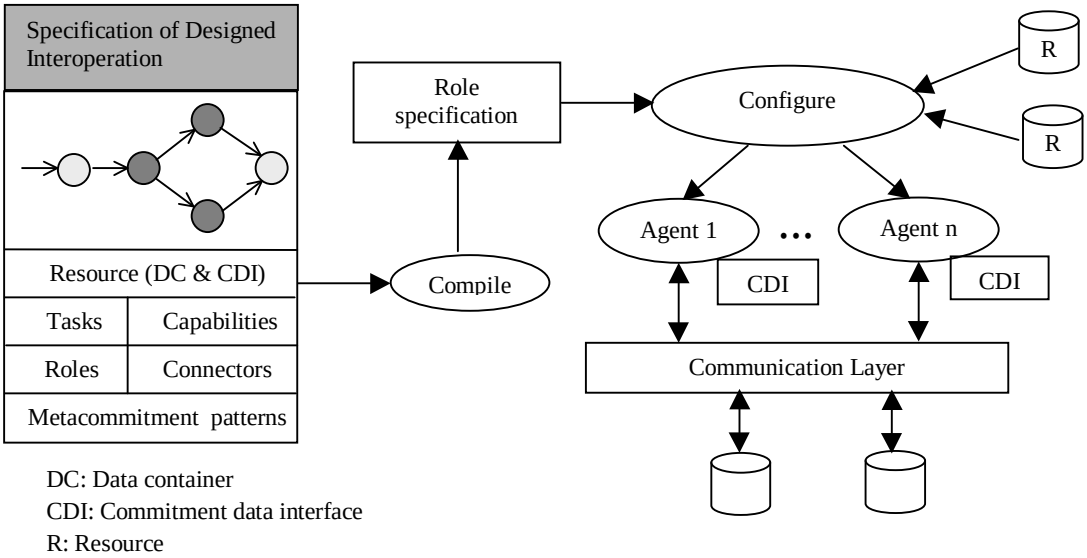


Figure 7.1: System architecture

Figure 7.1 shows our system architecture. Initially, a specification for the desired interoperation is created. This is then compiled into a set of executable rules for each role. The commitment data interface supports access to legacy data. Lastly, the system is configured by having the agents adopt appropriate roles. The agents then enact the specifications.

7.1.1 Specification

Our metamodel translates naturally into a set of document type definitions (DTDs) for the Extensible Markup Language (XML). XML is an emerging standard for exchange of information. We use our DTDs as an intermediate representation between a graphical interface and the reasoning system. We exploit XML tools, including an object interface for XML documents, to validate and parse the process interoperation specifications.

7.1.2 Commitment Data Interface

Recall from Section 3.1 that domain-level commitments are not explicitly captured in the model. However, for the computations to proceed, the agents must have memory of their commitments to others and others' commitments to them. And storing all the domain-level commitments might seem to require significant storage and overhead.

However, in real-life settings, the situation is much better. This is because in real-life organizations, even with current approaches (i.e., before agents are introduced), some record is kept of the various commitments that the parties enter into. For example, a travel reservation system would store a lot of data reflecting commitments by airlines, hotels, and passengers.

Consequently, our architecture includes a *commitment data interface* (CDI). The CDI relates commitments to the underlying information resources. For example, CDI would capture the schema of the (say, relational) database in which all the bookings are stored. Using the CDI, agents can retrieve and modify their domain-level commitments from the appropriate information resource. The domain-level commitments themselves, therefore, add no storage overhead.

The applicable metacommitment patterns must additionally be represented. These could be stored in separate business-rule repositories, which are also becoming common in enterprise information architectures.

7.1.3 Execution

Our execution framework is through a set of agents, each written in Jess, the Java Expert System Shell [24]. Jess is a well-regarded, open source Java-based forward-chaining inference engine. A programmer can define rules and assert facts. Jess evaluates these rules and executes any actions embedded in them, such as asserting more facts or invoking the desired capabilities. Jess evaluates the rules against the facts, executing any rule all of whose antecedents are true, potentially asserting more facts, and executing still more rules until quiescence.

Jess offers seamless integration with Java: it can be invoked from or invoke Java methods. This simplifies the implementation of a system such as ours. An agent is realized as a combination of Java and Jess. The Jess engine is embedded in a Java thread. The former reasons about the interactions with other agents while the latter facilitates these interactions by providing the means of accessing the external world.

7.2 Applications

7.2.1 Travel Planning

Example 6 We now describe how our system executes on our running example. The traveler orders a trip from the travel agent, who requests bookings from the airline and the hotel (notify pattern). The airline and hotel clerks begin processing (entertain-request pattern). Both airline and hotel succeed and notify their consumer, the traveler. But the traveler is not satisfied, because the hotel is too far from the airport. She sends a dissatisfaction event to the travel agent along with the reason. The travel agent sends an update to the hotel (renotify-consumer pattern), but not to the airline, because the old commitment is still valid. The hotel clerk accepts the update (entertain-update

pattern), and books the traveler at the airport location. He sends an updated confirmation to the traveler (renotify pattern). The traveler is satisfied this time and the activity instance concludes. However, the different parties don't forget what they did in case there are further changes. In the following, we apply exceptional-handling to our example in more details.

Revision If the prospective traveler changes his mind and suggests a different date or destination city, he renotifies the travel agent about the revision on account of his commitment to renotify when a revision exception occurs. The revision exception is thus detected. It is completely handled when the travel agent entertains this update on account of his entertain update commitment. The travel agent, in turn, may propagate the effect of revision if (1) the effects are significant and lead to the violation of a prior commitment by the travel agent and (2) the travel agent is subject to the same commitment patterns.

Incorrectness. If the airline agent cannot find a flight on the day of travel despite repeated trials (trying different airlines at different times of the day) he can either convey his failure to the customer or convey his dissatisfaction directly to the travel agent. In the case of failure, the customer, in turn, conveys her dissatisfaction to the travel agent. In both cases, when dissatisfaction is conveyed, a *reason* for the dissatisfaction is included. The reason states the constraints whose violation caused the dissatisfaction. Here, the reason is that the date of travel is not acceptable.

The travel agent must, therefore, provide a new date as he is bound by the commitment to satisfy his consumer. As the date was provided when the travel agent was requested to make the reservation, he propagates the dissatisfaction back (incidentally, also to the customer in this case). The customer then changes the date. Once the date is changed the incorrectness exception has been handled.

Incompleteness. If the customer provides the date of travel and the city but not the state and if two or more cities across different states have the same name, the workflow enters a state of incompleteness exception. This is handled as the travel agent requests for more information (name

```
            .
            .
            .
<commitment>
   <pattern> renotify consumer </pattern>
   <debtor> Travel agent </debtor>
   <creditor> Airline clerk </creditor>
   <condition> flight-info-unchanged </condition>
</commitment>
            .
            .
            .
<commitment>
   <pattern> satisfy consumer </pattern>
   <debtor> Travel agent </debtor>
   <creditor> Customer </creditor>
   <condition> no-outstanding-complaint </condition>
</commitment>
            .
            .
            .
```

Figure 7.2: A snippet of our XML file for Example 3

of state) and the customer entertains the request and sort of completes the information by notifying his consumer (the travel agent).

Ad hoc request. The traveler may decide to check whether the given hotel has a shuttle service to the selected airport. Or, the travel agent may do so when suggesting that the traveler rent a car upon arrival. The hotel agent should be able to accommodate this request even though it is not part of the workflow as stated.

Failure. If a traveler asks for a ticket below a certain price and only wishes to travel on a particular day, the airline agent may declare failure after trying the flights for that day and finding that all the cheap fares are sold out. ▮

For our running example, we first specify the trip order interoperation as an XML file (Figure 7.2) that conforms to our metamodel expressed as a DTD. Next we parse the XML file (using the SAX API) into a W3C-compliant document object model (DOM) tree (in Java). The parsed DOM objects are then compiled into Jess rules for each agent. Because these agents may be located on different platforms, we use JATLite [32] as the underlying communication layer. Even for an example this simple, a variety of exceptions were identified and successfully handled using our metacommitment patterns without the need for human intervention.

7.2.2 Contract Net Protocol

Example 7 The manager and contractor agents in CNP can follow basic agent behavior model. The manager has two threads, for a call for bids and for task assignment. The contractor has also two threads, for proposing a bid and for performing a task. For conventional CNP, we have the following:

- The manager initializes a thread. It sends out request for a call for bids. To satisfy the request by *entertain-request*, the contractor initializes a thread to reason whether to bid.
- If contractor decides to bid, it send a bid to the manager (*notify*).
- The manager chooses the optimal bid and sends a task assignment to the willing contractor. The willing contractor satisfies *entertain-request* for tasks by performing task assignment.
- When the task results are available, the contractor notifies the manager.

To allow revisions in CNP, we have the following:

- If the manager changes its the number of widgets, it can send the corrected information to the contractor (*renotify*).
- The contractor reasons again (*entertain-update*). The contractor may request to increase the price. If so, it *renotifies* the manager.

To handle exceptions in CNP, we have the following:

- If the contractor does not finish its task by the deadline, it can withdraw the task assignment and cancel its commitment by using *abort*. ▮

7.2.3 Distributed Database Interoperability

In order to coordinate database interoperability, some existing methods such as two-phase commitment are used to solve the problem. A coordinator should be introduced. However two-phase approach cannot solve the exception problem. Thus some researchers suggest three-phase commitment. This make the approach very complicated. For our approach, we introduce agents for coordinating database interoperability. We don't have to have a coordinator. Our suggested approach has the following advantage. First, agents interact with each other to achieve database interoperability by using a set of well-defined commitment patterns. We can handle exceptions and revisions more naturally than traditional approaches. Second, there is no need to have central control and each database preserves its autonomy. Third, our approach demonstrates what the commitments are between agents. It is used to distinguish our commitments from database commitment. It is a good example to demonstrate our agent approach.

Agents exchange their information, or knowledge through interactions with one another. Researchers represent the information by using some information exchange formats such as XML, KIF, or first order logic. The information is domain-dependent. Agents communicate with each other by using a communication language. Relational database stores information by using the tuple space. Let's consider the relational model. Under this model, a database tuple provides a means to represent the information that agents exchange with each other. The database schema provides a basis for the represented commitment.

There are several aspects that we have to consider in our approach. Figure 7.3 and Figure 7.4 show the architecture for distributed database interoperability and an example for an agent approach of distributed database interoperability

- *Predicate representation.* We will develop a means to represent the predicates. The

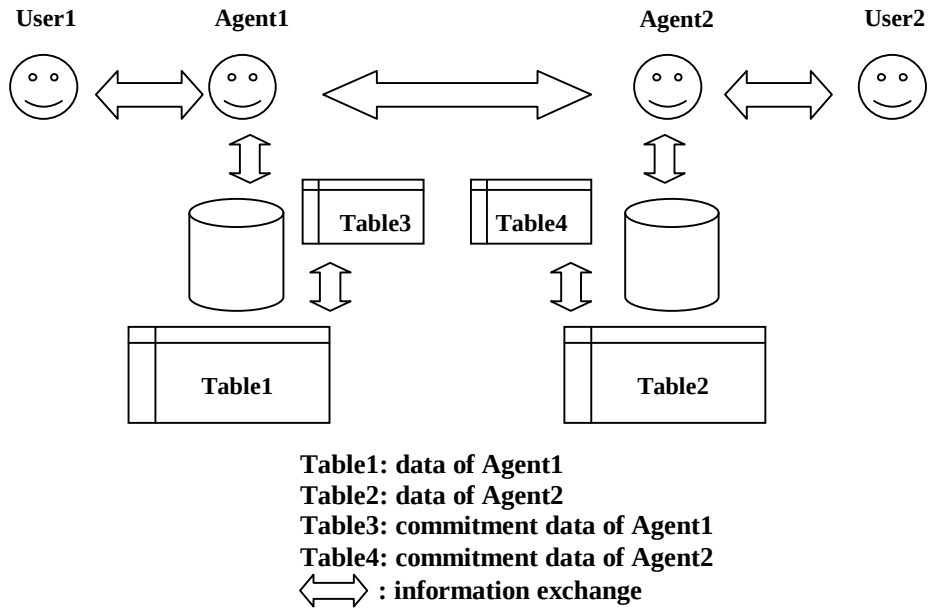


Figure 7.3: Architecture for distributed database interoperation

followings are some examples for the representation,

- $query(A = a, B = ?, C = ?, ...)$.

  This is used to represent a query to ask for B, C, and ... given that A = a.

- $insert/update(A = a, B = b, C = c, ...)$.

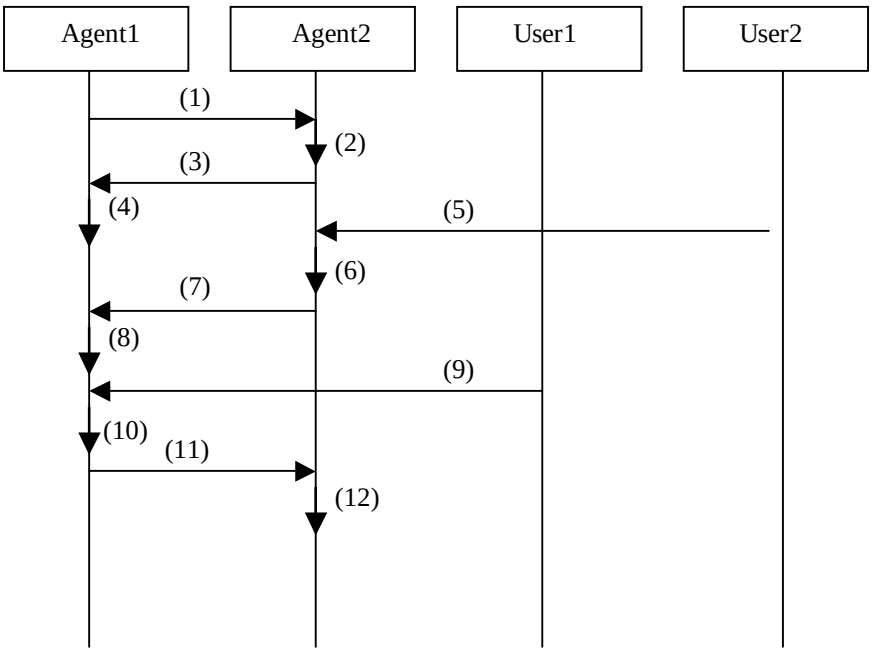
  This is used to represent a query to ask for *insert*, or *update* A=a, B=b, C=c, and ...

- $delete(A = a, B = b, C = c, ...)$.

  This is used to represent a query to ask for *delete* these items when $A = a$, $B = b$, $C = c$, and ... from tuple space.

- *Represent commitment between agents.* There are two kinds of commitment used here. One is incoming commitments that are created by other agents. Other is outgoing commitments that are sent to other agents. Database table can be used to represent the commitments among agents. We can represent commitment table as (debtor, creditor, group, attributes in predicate). It can be saved in database tuple space.

- *Represent the commitment patterns.* Each pattern will trigger some database transactions. Thus we will specify which transactions are in the commitment patterns. These transactions consist of some SQL commands for operating in database information of a practical application as well as commitment information between agents. For example, When $agent_2$ receives a request ($request(Agent_1, Agent_2, query(A = a, B = ?, C = ?))$), it can use $entertain_request$ pattern. This pattern uses SQL command SELECT to find the B, C given that $A = a$ from the table with attributes A, B, and C.



(1) request(A1,A2, pred(A=a, B=?, C=?))
(2) entertain_request(A2,A1, pred((A=a, B=?, C=?))
(3) notify(A2,A1, pred(A=a, B=b, C=c))
(4) entertain_request(A2,A1, pred((A=a, B=b, C=c))
(5) request(U2, A2, pred(A=a, B=b, C=c, E=e))
(6) entertain_request(A2,U2, pred(A=a, B=b, C=c, E=e))
(7) renotify(A2,A1, pred(A=a', B=b', C=c'), pred(A=a, B=b, C=c))
(8) entertain_update(A1,A2, pred(A=a', B=b', C=c'), pred(A=a, B=b, C=c))
(9) request(U1, A1, remove(A=a, B=b, C=c))
(10) entertain_request(U1,A2, pred(A=a, B= b, C= c))
(11) dissatisfy(A1,A2, pred(A=a, B=b, C=c))
(12) abort(A2, A1, pred(A=a, B=b, C=c))

Figure 7.4: Example for an agent approach of distributed database interoperability

Chapter 8

Related Work

Software agents, transaction and interoperability standards are related to our work. There is a large of literature in these areas. In Section 8.1 we overview some interoperability standards. In Section 8.2, we review current efforts in applying software agents for process interoperation. In Section 8.3, we introduce current transaction models and trends of workflow systems.

8.1 Interoperability Standards

8.1.1 Existing Workflow Interoperability

Abstract Interface from WfMC. WfMC [14] describes workflow interoperability as the abilities of two or more workflow engines to communicate and interoperate in order to coordinate and execute workflow process instances across those engines. It defines a service provider and service requester model of workflow interoperability. The workflow instances of service provider are invisible to the service requester. The standard defines interaction operations by which service requester can instantiate and control the processes of service provider and service provider can feed back the status reports to service requester. These interactions consist of requesting and responding about creating process instance, setting process attributes, starting process enactment, notifying status change, and getting process attributes, etc. There are three interoperability scenarios between workflow engines: direct interactions between two or more workflow engines, engine interactions between the same

enactment systems, engine interactions between different enactments in the same workflow management systems. Three interoperability models are chained, nested and paralleled between two or more workflow instances. A nested sub-process provides information back to their parent process instance enacted on the workflow engine that requests their invocation, while a chained sub-process is one in which the parent process starts a process instance on another workflow engine, but takes no further interest in its progress. The chained sub-processes are used more within organizations where they are treated as members of a set of trusted processes deployed to support business operations. Nested sub-processes are more likely to be used in those business contexts where synchronization and coordination are used for operations of the business process or where it is important that the parent process tracks or maintains operational control over the operations of the sub-process. It is notable that the standard lacks the interaction scenarios of engine-to-engine in different workflow management systems, which often occur between organizations.

OMG's Workflow. Interoperability specification of OMG workflow management facility defines interfaces, protocols and shared services from implementing distributed workflow applications, which can be performed by business objects and supporting runtime interactions with workflow components. This provides the base for future evolution of the WfMC standard. It adopts an object-oriented view of workflow management. Workflow components take care of the overall process logic of a business process, enable monitoring of a workflow execution and support the combination of a business components into workflow applications. It is an implementation of interface 4 protocol of workflow reference model of WfMC on the top of CORBA. These interactions are used CORBA IIOP. In response to OMG's workflow management facility, jointFlow [51] specification is submitted by 19 companies. The standard focuses on specification of workflow business components and doesn't address the general aspects of business component management. The interfaces are defined on CORBA API. The CORBA application approach appears workable within the organization, it is not practical for inter-organization integration. This standard supports the concepts of assigning resources to an activity, however, the details of identifying the resources to be assigned

are not defined. JointFlow provides get_result operation to return process data. It doesn't support to extract partial results.

SWAP. SWAP is proposed by Netscape, HP, Sun. SWAP proposal is presented to IETF for purpose of integrating work providers across the Internet. SWAP [6] attempts to define an HTTP-based workflow interaction protocol to integrate work providers, asynchronous services across the Intranet/Internet. It provides operations to initiate, monitor, and control workflow instances. An observer can be notified when instance statuses are changed during normal execution or when exceptions occur. All parameters about process interaction are represented by XML and encoded in HTTP1.1 protocol. The resources represent generic services. There are three fundamental interfaces, which form the core of resource model. They are ProcessInstance, ProcessDefintion, and Observer. The processInstance is actual performer of the service. Every invocation of the service creates a new process instance resource, each with a unique URI. The process instance can be given a reference to an observer resource. The activityObserver forms a bridge between the process instances that are part of the sub process instances that are doing work externally. This allows nested process instance.

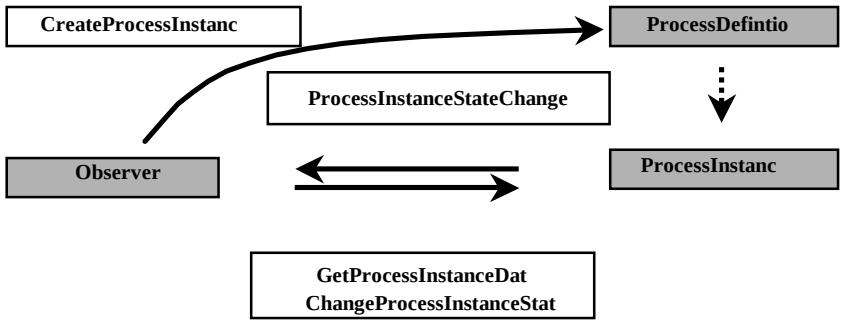


Figure 8.1: Logical resource model (cited from WfMC)

Wf-XML. The Wf-XML [15] evolves from the above specifications. Wf-XML defines message contents between process instances, and interoperability contract among different vendors using XML and is planning to use various transport protocols such as HTTP, e-mail, CORBA IIOP and

COM/DCOM. The interoperability contract addresses the issues of connectivity, security, the context data format and constraints. It defines a logical resource model to accomplish interaction among different operation groups. The primary operation groups for interoperability are named process definition, process instance, and observer. Process definition group represents the description of a service's basic functions and is uniquely identifiable by interoperability services or service requesters. Instances of a service are created by this resource. Process Instance group of operations represents an actual enactment of a given process definition and will have its own resource identifier. A new instance will create for each enactment. Each process instance is unique and exists only one time. Once created, a process instance starts and eventually is completed or terminated. The observer group provides a mean by which a process may communicate with it about the statuses such as its completion or termination. Figure 8.1 shows the logical resource model. Wf-XML draft1.0 is only at the beta status. It only defines a minimum interoperability message sets. Some operations such as retrieving a list of instance, subscribing and unsubscribing process instance and notifying aren't defined in this specification. Because DTD is used as implementation reference and optional data validation, concise occurrences and data types aren't specified in XML.

The proposed standards will help interoperation between workflow engine-to-engine communications and allow interoperation across vendor's workflows, but not across organizations. They lack the enough suggestions for handling exceptions or terminations. Other facilities such as resource registries, resource locating, negotiation, process monitoring are necessary to be addressed or introduced from other standards.

8.1.2 Interoperation with Existing Systems

A number of existing applications such as Lotus Notes, SAP R/3, e-mail and legacy applications have successfully applied to e-commerce applications. Workflow systems are necessary to interoperate or integrate with them for e-commerce processes between organizations or across organizations. CrossFlow is such a system to support the requirements.

Seven European institutes form the consortium of the CrossFlow ESPRIT project [37]. The

project aims at developing information technology for advanced process support in dynamic virtual organizations with contract-based service trading. CrossFlow provides high-level support for workflow in dynamically formed virtual organizations by abstracting services and offering advanced cooperation support. Virtual organizations are dynamically formed by contract-based matchmaking between service providers and consumers.

In a virtual organization, a partner does not know full operational details of other partners due to their autonomy. It can obtain its partners' data and processes by a defined abstraction of their operations. This abstraction can be considered to address heterogeneous environment problems existing in its partners. CrossFlow defines the interaction between organizations not in terms of their workflow systems, but on an abstraction level that can be any application. CrossFlow addresses three areas of advanced co-operation support functionality to complement basic workflow interoperability. Quality of service monitoring allows to track the progress of outsourced services, both online during service execution and offline to provide aggregate information. Levels of control enactment provide means for high-level cross-organizational transaction management and consumer-controlled process control over the outsourced services. Flexible change control allows for dynamic changes to execution paths of services during their execution.

The basis for cooperation in virtual organizations is the contract. The encapsulated service and cooperation support services can be completely specified in this contract. Partially defined contracts are used by service providers to advertise their services and by service consumers to search for services. The gateways are used to connect WfMS of service requester and the system used by service provider, to homogenize the differences, and to add functionality to WfMS and other systems when required.

8.1.3 Existing Languages and Protocols

Although some efforts from standard organizations are under development, interoperability specifications are still far from meeting the interoperability need, especially those that operate in virtual business environment. Currently some languages, middleware and implementation infrastructures

are used as interoperability standards. The Internet is expected as a common ground of multi-organization interoperability.

XML. The extensible markup language (XML) is a data format for structured document interchange on the Web. It provides a framework for tagging structured data by allowing developers to define an unlimited set of tags bringing great flexibility. XML resembles and complements HTML. XML describes data such as city name, temperature. HTML defines tags that describe how the data should be displayed such as with a bulleted list or a table. Document type definitions (DTD) may accompany an XML document, essentially defining the rules of document, such as which elements are present and the structural relationships between the elements. DTD helps to validate the data. XML schema draft suggests to support user-defined data types, richer occurrence models and inheritance mechanisms. This promotes XML to e-commerce application. XML brings so much power and flexibility to web-based applications that it provides a number of benefits to developers such as being able to do more meaningful searches. XML is emerging as a standard for self-describing data exchange on the Internet. Its power lies in its extensibility and ubiquity. Web will use XML to encode information and services with meaningful structure and semantics that computer can understand. E-commerce will use XML to place orders, make reservations, and schedules shipments, etc. There are many XML document formats in different partners. For example, if two trading partners use different versions or formats for a purchase order, they cannot interoperate.

CBL. CBL was created to address the above issue. CBL (common business library) [47] promotes business interoperability by representing common business definitions (e. g., companies, services and products) and interaction (e. g., catalog searches, purchase-order processing and inventory updates) using an extensible, public collection of predefined XML components. It is the first overlap in XML specifications for vertical industry initiatives to facilitate interoperability across industries and marketplace. CBL helps the spontaneous commerce between trading partners without custom

integration or prior agreement on specific industry wide standards. The CBL is designed to encourage development and use of generic XML document models. The library consists of information models for various concepts, including:

- Business descriptions, such as companies, services, and products
- Business forms, such as catalogs, purchase orders, and invoices
- Standard measurements, such as date, time, location and classification codes.

CBL is used as building blocks to implement the basic business forms used in X12 EDI transactions, as well as OTP, OBI, and other emerging Internet standards. eCo is such an architecture using CBL for e-commerce interoperability. It uses document service architecture to describe the application interfaces to business services. This is a nice architecture. The trading partners need to agree only on the structure, content, and sequencing of business documents they exchange, not on API details. This makes interoperation simple.

OBI. The OBI (open buying on the Internet) is an e-commerce standard, which has been specified by OBI Consortium. OBI is "an open, flexible framework for business-to-business Internet commerce solutions" [48]. It is intended for the procurement of high-volume, low-dollar indirect goods and services that account for 80% of most organizations' purchasing activities. It is expected to streamline the non-mission critical procurement processes of organizations by specifying a standard set of roles that OBI-compliant selling and buying parties must conform to. Attendees will be able to create requisitions and follow their progress through the procurement cycle with input from multiple vendors. Exchange of OBI order requests and OBI orders is accomplished via HTTP over SSL. Contents of OBI Order requests and OBI orders are based on ANSI ASC X.12's 850, a standard for EDI purchase order. Currently OBI is working on using XML as content formats.

tpaML. Trading partner agreement markup language [53] is a B2B transaction specification. IBM has designed a trading partner agreement markup language governing electronic contracts. The tpaML specification uses XML to define and implement electronic contracts. The foundation of

tpaML is the trading partner agreement (TPA), which defines how trading partners interact at the transport, document exchange and business protocol layers. A TPA contains the general contract terms and conditions, participant roles (buyers, sellers), communication and security protocols and business processes (valid actions, sequencing rules, etc.). XML-based TPA documents capture the essential information upon which trading partners must agree in order for their applications and business processes to communicate. tpaML is a complementary technology to ebXML, the electronic business XML initiative, which is a joint effort of the United Nations/CEFACT and OASIS to establish a global framework for the exchange of electronic business data.

OTP. OTP (open trading protocol) is a protocol for the development of software products that will permit to produce interoperability for the electronic purchase that is independent of the chosen payment mechanism. OTP encapsulates the payment with the offers/invoice/receipts for payment and delivery. OTP defines a number of different type OTP transactions such as purchase, refund, value exchange, authentication, withdrawal, and deposit. These transactions involve trading roles, exchanges, and components. Typical trading exchanges are offer, payment, delivery, and authentication.

8.1.4 Discussion

Currently the main trend towards process interoperation architectures can be generalized as XML-based document exchange architectures, which are used to address these technical challenges such as heterogeneity and adaptation. Business documents independent of different platforms represent a more intuitive and flexible way to access services than programming business process APIs. They facilitate spontaneous commerce between trading partners without prior agreements on specific industry-wide standards. SWAP and Wf-XML specify process interaction information between workflows by using XML. A number of organizations agree on common business languages. Current marketplaces or portals such as an eCo architecture are such an example. It consists of some facilities such as common catalog, search, negotiation, and monitor tools. Common ontology for

vertical industry sectors is being derived.

Although these XML-based approaches have shown promising results, they are still too limited to support process interoperation. There is no XML standard for process definition. Tedious translation between process definition languages and based-XML documents are still needed. Wf-XML and SWAP don't adopt the related XML standards for process interoperability. Wf-XML only provides limited operations for process interoperation. Interoperation between organizations doesn't follow a simple nesting of request and response. For example, a retailer may modify an order after it has placed the order.

Although the existing standards specify business components, business transactions, exchange documents as well as interoperation protocols, they only refer to the infrastructure or some low-level synchronization interactions. They provide little support for handling exceptions and revisions that are crucial in an e-commerce context. Although it is easier to interconnect companies in terms of the documents they exchange, organizational issues aren't addressed properly. Workflow management systems are well-suited to manage process within organizations. They lack the tools to control and manage the outsourced tasks.

8.2 Software Agents

Standard organizations and companies are making efforts in infrastructure and information exchange formats, which are the foundation of process interoperations. However the high-level process interoperation, such as process negotiation and process coordination needs to be addressed by organization cooperation. It is a key factor that organizations provide high service quality.

These participants of the process interoperations are human beings and organizations. In the e-commerce context, the cooperation is automatically, or semi-automatically achieved by software agents. Software agents can be though of as persistent active objects in open environments. The key aspects of agents are their autonomy and abilities to perceive, reason, and act in their environments, as well as to socially interact and communicate with other agents. What makes agents interesting is

that they can form multi-agent systems. Multi agent can be used to mediate users, organizations to automate a number of the commerce processes.

In conventional workflow systems, a program task has a designed execution life span, it exists only during the execution time and cannot receive messages before and after that tasks. On the contrary, software agents have memory and knowledge and work on multi tasks and multi business process simultaneously. Distributed object systems have the encapsulation but not the sophisticated reasoning required for social interaction, and distributed processing systems deal with the distributed aspect of the domain but not with the autonomous nature of the components. These two properties make agents distinct.

8.2.1 Commitments

Dignum and van Linder [18] propose a framework for social agents based on dynamic logic. Agents interact each other based on deontic relationship among agents. They use a subset of primitives of the common agent communication language to denote the semantic content of messages. They employ commitments or obligations to enforce agent action. We represent our message primitives related to commitments among agents. The commitments are allowed to update and cancel. Dignum and van Linder deal with the motivational attitudes of agents and define many concepts such as wishes, goals, intentions, and commitments or obligations. This complements our approach. We focus more on operational semantics. The reasoning results represented by predicates with data values as the arguments trigger the interaction.

The *language for action* metamodel involves commitments as well [65] and is applied in the ActionWorkflow tool. This metamodel uses *loops* representing a four-step exchange between a *customer* and a *performer*: (a) a request from the customer, (b) negotiation by the two about the task, (c) actual performance of the task, and (d) evaluation of the performance by the customer. A step may potentially be nested with other loops. The loop metamodel has some limitations. It only considers two actors at a time, and does not explicitly consider the surrounding organizational structure. It cannot easily accommodate modifications or revocations of the commitments. We

provide richer interaction among roles. For example, patterns such as *entertain-update, renotify, dissatisfy,* and *abort,* enable an agent to change its mind and handle exceptions, etc.

8.2.2 Negotiations

Negotiation can be defined as a kind of decision-making process where two or more parties jointly search a space of possible solutions with the goal of reaching a consensus [26]. Protocols and strategies must support the negotiation process. A protocol defines the rules of negotiation process to encourage customers and suppliers to negotiate fairly and avoid the fraud. A strategy defines a plan of action to maximize the utility of participants. Agent negotiations can be traced back to the contract net protocol. Limitations of original contract net protocol have been addressed by Sandhlm [54]. We are interested in automating negotiation process using Agents.

Kasbah [13] is a web-based system where the users create autonomous agents that buy and sell goods or services on their behalf. It provides buyers with one of three negotiation strategies: anxious, cool-headed, and frugal. Selling agents try to sell themselves by going into the marketplace contacting interesting buying agents and negotiating with them to find the best deal. The marketplace's job is to facilitate interaction between the agents.

OFFERS [5] is an electronic brokering architecture, which uses OMG's common object request broker architecture (CORBA) as a distribution infrastructure. There are three main components: suppliers, customers and e-brokers. A customer can search for a service either directly in the e-catalog of supplier or use the e-broker to search all the e-catalogs of all the suppliers, which are registered with the broker. COBRA is chosen as the communication infrastructure to solve the interoperability problem. The negotiation mechanism via e-brokers employs simple auction implementation.

AuctionBot [66] is a general-purpose Internet auction server at the University of Michigan. Its users create new auctions by choosing from a selection of auction types and then specifying its parameters (such as clearing times, method of resolving tie bids, and number of sellers permitted).

Buyers and sellers can then bid according to the auction's multilateral distributive negotiation protocols. One important feature is that it provides an application programming interface for users to create their own software agents to autonomously compete in AuctionBot marketplaces.

Agent-based process management system (APMS) [49] suggests a service-oriented model to meet the requirements of open distributed enterprises. It supports the decentralizations of responsibility and local autonomy, within and cross enterprises. APMS uses negotiation process to seek a mutual agreement and commitment on the delivery of a service. Quality, time and cost are three typical parameters, which would form basis for negotiation. Negotiation processes are used to improve deployment of resources and enactment of process tasks. APMS represented the interests of different partners of an organization as software agents. The inherent autonomy of software agents enables these parts to retain their autonomy of information and control, and to enable different organizations to automate their interaction in the management of a common business process. Negotiation language protocol defines the valid ordering for sending and receipt of primitives during negotiation between agents.

Negotiation should allow multi strategies [13, 5, 66]. It should also support multi attributes [5, 49] other than the price. This may find incentive for both parties in negotiation space. Each agent should have itself API [66] to specify its strategy. Contracts are used as an agreement between supplier and consumer agents during the negotiation process. The contracts contain a list of agreed values of parameters, which may establish the terms and conditions for delivery of service. These contracts restrain the action of agents. In a realistic business process an agent will require to negotiate for multi contracts simultaneously. Thus agents are constructed to be multi-threaded so that it can support multiple negotiations for different services concurrently. Agent also can execute multiple services or tasks simultaneously based on contracts during negotiation processes. When tasks and services fail, agents can perform correct actions and try to resolve the failure.

8.2.3 Process Coordination and Agent Cooperation.

Coordination has been defined as the process of managing dependencies between activities [4]. E-commerce environment is an open environment. Agents are represented as autonomous entities. An agent operates in that environment with certain goal and takes a number of actions to affect that environment. The problems arising here are what action the agent takes and in which order or time the agent takes the action. Making coordination difficult is that the agent often has an incomplete knowledge and the environment changes dynamically.

In MOPPET [2] project, the agent based workflow is suggested to achieve buying and selling process in an electronic marketplace. Agents can be represented as interface agents, schedule agents, task agents, recovery agents and facilitator agents. These agents interact to perform process inter-operations. They propose an electronic marketplace architecture to address process interoperation. XML is used as process interaction language. KQML is used as agent communication language.

Sycara et al. [61] consider the problem of matchmaking in heterogeneous agents. Their approach essentially involves matching agents based on their capabilities. This complements our approaches because finding agents with the right capabilities is an important sub-problem for us. A possible synthesis is an enhanced approach that will also consider the metacommitments that the agents are willing to support.

Barbuceanu et. al. [4] proposed a conceptualization of the coordination task around the notion of structured conversions among agents. They explicitly represented coordination knowledge. A finite state machine model is used to represent the conversion plan. KQML-based language is used to describe agent interaction as multi conversion selection, conversion execution and event handling. They apply the concepts to multi-agent supply chain systems for manufacture enterprises.

Barbuceanu and Fox [3] describe a language for specifying coordination among agents. Their approach involves finite state representations of an entire conversation. While their approach is quite effective in coordinating agents, it leaves open the question of how the given conversation is acquired. In their case, the desired protocol is kept as a central model that controls the participating

agents. By contrast, in our approach, we provide general agent behavior models to autonomously coordinate the activities by agent interaction. There is no central model. In specifying agent behavior, we use a statechart to represent agent behavior, not using flat finite state machine, which causes state exposition problem when there are a large number of states in finite state machine. For verification purposes, we use temporal logic to verify our behavior model.

Although preserving the autonomy of participating components is crucial, unrestrained autonomy may lead to undesirable consequences. In [29], they use sphere of commitment (SoCom) to restrain the autonomy and to enforce the process coherence in open environment. SoCom is a multi-agent system that agents constitute. It serves in the context of commitments among those agents. A SoCom is typically associated with a set of resources and authorities over them. They apply SoCom to virtual enterprise where SoCom coordinates sellers to provide on-time, cheap, high-quality services to buyers.

Klein and Dellarocas exploit a knowledge base of generic exception detection, diagnosis, and resolution expertise [36]. Specialized agents are dedicated to exception handling. This approach is complementary to ours; special roles could be included in our approach with commitments by other roles. The advanced decision environment for decision tasks (ADEPT) project considered workflow management [30]. This project emphasized negotiation among agents. However, the underlying notion of commitments doesn't allow contextual nesting, as in our approach.

Traditional workflow has controlled by predefined rules, saying control flow. It is inflexible and isn't appropriate for this dynamic e-commerce environment. Thus agent social interaction can be used to attack the above problems. However it is hard to specify interaction processes completely and the processes are very likely to change dramatically during the life cycle of application. Thus we need an increasing specification process to define process interaction. Sometime agents need to interact with users to revise the process specification to meet the new requirement. Agents cooperate each other to achieve process coordination. Process coordination results in agent cooperation. Workflow can control agent interaction and multi agents can cooperate for supporting e-commerce process.

8.2.4 Process Management

E-commerce business processes operate in highly dynamic and uncertain operation environments. Monitoring for critical events and responding appropriately are necessary when such events are detected. Automatic load balancing is essential to ensure effectiveness and timelines. Within organization, full tracking of process execution and auditing capabilities are required for controlling and monitoring processes. However between organizations, only partial tracking of process execution is allowed. Traditional workflow management system addresses process management more within organization.

BT agent-enhanced workflow [33] provides an agent layer combined with commercial workflow management systems. The agent layer has responsibility for provisioning phase of business process management and enacting phase of handling workflow executing. The agents are used to cope with dynamic changes in resource levels and task availability. The agent-enhanced workflow provides an agent broker that can translate between the local ontology of a workflow agency and global ontology of inter-operating systems. JKQML is used as an inter-agent communication language. FIPA ACL achieves the inter-broker communication with the process interchange format as a content language.

In [56], agents are used to manage overall resource levels in a call center application. Agent enhanced workflow is allowed to use third party workflow tool. This explores the possibility of adding an agent layer with the added value of providing mechanisms for proactively handling and repairing workflow failures, while retaining the user's confidence in the workflow tools they were familiar with. Agent enhanced workflow is achieved by combining a layer of agents with a commercial workflow system. The agent layer is given responsibility for the provisioning phase of business process management, whilst the underlying workflow system handles process execution. In the case of a process failure the agents are employed to renegotiate the flow of work and redistribute the work accordingly. This may be done either reactively or proactively according to the failure type.

In e-commerce environment, process management requires new functions for inter-organization

operations. These tasks are more complicated computation. Thus intelligent, distributed, autonomous software agents are used to improve the management of business process under the control of a workflow management system. Agents can learn the knowledge from process execution and other partner's process. Data mining technology can be used to acquire the important information from the audit data. Queuing theory and stochastic Petri net can be used to evaluate the performance in different levels such as turnaround time, response time, failure rates of tasks, agents and resources etc.

8.2.5 Discussion

Negotiation, process coordination, and process management are indispensable to process interoperation. They should come together in the e-commerce applications. Negotiation processes make business processes rational, especially in e-commerce environment. This is ignored in current workflow systems. In addition, negotiation can be used as a market mechanism to coordinate processes. Agents negotiate to reach contracts. Process management then assigns the agents and resources to perform the processes. Agents fulfilled contracts during process coordination. Agents may revise their contracts such as decommitting contracts during process execution. Agents need to renegotiate to reach new contracts. There isn't any mechanism to modify the processes during process execution.

There are two ways to model multi-agent workflow systems in an e-commerce context. First, agents can represent process participants and resources. They can interact to negotiate and to cooperate each other at run time. Second, agents associated with a workflow engine can interact with other agents associated with other engines. The agents augment existing workflow systems to improve their flexibility. Agents are used to automate business process and improve QoS of process interoperations, and process management.

Agent inherent autonomy property can help retain the autonomy of interaction parties. In order to make right decisions, agents need to interact with each other or interact with users. Agents acquire the knowledge from the interactions. In negotiation process, bidder agents know the current

high bid price by interaction. In process coordination, agents know what action they will take also by interaction. Agents should interact each other in high level. Agents can act as different roles to adapt the dynamic environments. How to specify these roles to satisfy different requirements is an important work. These specifications should allow an increasing specification. This makes agents more powerful.

8.3 Transactions

Traditional transaction models are too strict for advanced e-commerce applications such as workflow systems that are of long duration. The ACID properties aren't satisfied. Several advanced transaction models have been proposed. In the sage model, a transaction consists of a number of sub-transactions to be executed in sequence and committed independently. When a sage aborts, it has to compensate for the committed components by executing their corresponding compensation transactions. Flexible transaction works in the context of heterogeneous multi-database environments. Flexible transaction provides alternative execution paths. If a sub transaction is aborted, then a different sub transaction can be submitted in the hope that it will be successful. Workflow systems resemble to the advanced transaction models that are multi database transactions. Thus modeling workflow and related exception handling are considered to extend these advanced transaction models. Some work in modeling workflow systems is motivated by transaction models [11, 38].

8.3.1 Advanced Transaction Model Workflow

However, multi-database transactions are data centric. They utilize extended and relaxed transaction models to preserve data consistency. Workflow is generally application centric. They tend to focus on control flow. Workflow management systems address a much richer set of requirements such as organization aspects, user interface, monitoring, and accounting. Organization issue is often ignored in these models. In the electronic commerce applications, it is necessary to be addressed because multi parties are involved. In many aspects workflow models are a superset of advanced transaction

models.

There are several products and a huge body of literature on workflows. The work on rule-based systems for workflows is pertinent. A lot of it, however, deals with the lower-level aspects of workflow management. Several approaches include rules for handling exceptions [59, 9], but they do not capture the bigger patterns of long-lived interactions, as studied here.

The CREW approach provides one of the best recent approaches from the database community [34]. Kamath and Ramamritham develop a rich model for workflow that includes failure handling and enables *opportunistic* rollback, restricting the extent of the compensation activities. They consider the interactions among the different tasks. CREW can accommodate interactions among workflows, including the important cases where a workflow feeds information to another, and can change its results under some circumstances. Like CREW, we consider higher-level abstractions, but unlike CREW, we also consider abstractions that relate to the organizational aspects of a workflow. Some of our patterns are designed to accommodate opportunism in rollback, e.g., by renotifying consumers of updates only when necessary.

WIDE [11] is a distributed architecture for workflow management. WIDE uses enhanced database technology as the basis for its workflow management system and focuses on extended transaction management. It provides reactivity through the use of a decoupled rule execution model. Rule execution is divided into scheduling and interpretation of rules. Triggers inserted into the database management system cause events to be noticed. The scheduler inspects the event database and matches the event to the rules, which are then recorded in a to-execute list and interpreted for execution. WIDE is distinguished by its exploration of distributed workflow management system, and its sophisticated transactional capabilities. It adopts a two-layered transactional model: the upper layer provides global transactions with loose semantics and the lower provides strict, more conventional, and transactional semantics. The two layers are orthogonal, providing a possible approach to long-lived transactions.

METEOR (Managing End To End OpeRations) [38] provides a range of products, including the METEOR Designer, the WEBWORK enactment system, and ORBWork, all grounded in the activity

based paradigm. Their ORBWork prototype exploits CORBA for inter-component communications. It is a fully distributed implementation of workflow server functionality. It addresses the problem of compound transactions through a two-phase commitment coordinator, and provides a platform for experimentation with algorithms related to workflow. Some of the METEOR work was done in the context of healthcare applications.

8.3.2 Exception Handling

Transaction models provide well-defined failure semantics in the sense of concurrency control and sophisticated recovery features. Many works have been done to merge advanced transaction technology and workflow management systems to support business processes with well-defined semantics and recovery features [1]. Kammath et. al. [35] gives a good overview to deal with logical failures. They suggested opportunistic compensation and re-execution based on correctness and performance perspective. This approach is rule-based with a set of high-level building blocks. They deal with intra-workflow and inter-workflow exceptions in data inconsistencies that are mainly the lower level exceptions. Ludwig [39] considers organizational responsibility for cost in the case of outsourced sub-process termination. A contract is employed to govern the activities, which are performed outside the organization. The termination cost is defined in the contract, which is agreed among organizations. This approach deals high-level terminations, which can be as a complement for handling cross organization exceptions.

8.3.3 Atomic Transaction Extension

Atomic transactions form the cornerstone of modern transaction processing theory. Nancy Lynch et. al. [40] wrote a textbook about atomic transactions, which is a tremendous resource for implementing atomic transaction process systems. J. D. Tygar [62] suggested NetBill, which links goods delivery and payment into a single atomic transaction. An atomic transaction protocol is grouped by classical transactional and non-transactional processes together into a unit of work that reflects

the semantics and behavior of business tasks. NetBill provides three levels of atomic transactions: money atomicity, goods atomicity and certified delivery. Money atomicity affects the transfer of funds from one party to another. This atomicity is a basic level of atomicity that each electronic commerce protocol should satisfy. Goods atomicity protocols include money atomicity, and also affect an exact transfer of goods for money. Certified delivery atomic protocols allow both transacting parties to prove exactly which goods is delivered. These advanced transaction approaches need to integrate in the interoperability of business process.

8.3.4 Transactions as Actions

Since electronic commerce transactions are characterized by highly communicative character, some researchers use language/action perspective as a deep structure analysis. Hans Weigand et. al. [64] proposes to use a layer model of patterns to model electronic commerce processes. Speech acts is the most elementary unit with the communication between the subjects. A transaction is defined as a smallest possible of acts that leads to a certain deontic state, in other words an obligation or/and authorization, or an accomplishment. For example, the combination of customer requests to deliver a product and the promise of supplier actually to deliver it. A workflow is a set of related transactions aiming at some goal. Action workflow approach [41] describes workflow as a loop consisting of four phases, which starts with a proposal, a request from a customer, following by an agreement between the customer and a performer. After the performer has executed the promised action, the customer can declare to the performer that transaction was successful. Contracts describe the agreement as mutual obligations and authorizations between the consumer and the performer. It is thought as constraints, which enforced fulfillment of the contracts. Scenario describes the interaction between several contracts that run between several parties. The layered approach can be used to specify e-commerce applications in an increasing process. Deontic logic provides means to specify and verify communication-based transactions.

8.3.5 Discussion

E-commerce focuses increasingly on inter-organization communication. As the number of trading partners and sophistication of commerce applications increase, the new need to reconsider transaction models increases. There are two kinds of transactions: database transactions and action-based transactions. Database transactions are used to deal with data consistency. Action-based transactions are considered to specify communication issues. Action-based transactions uses loop representing a workflow. It only allows to considering two roles at one time. It doesn't explicitly consider the surrounding organization structures.

Workflow implementations of business process can be not only transactional processes, or classical transactions, but also non-transactional processes. Transactions as activity implementations frequently appear when the business models represent one of the core businesses (such as an order) of an enterprise. Non-transactional activity implementations are frequently found within support processes (such as accounts). Transactions in the business-to-business e-commerce are usually long-lived that involve pre-sale (negotiations, commitments, contracts), sale (order, fund exchange, shipping, logistics, and tracking) and post-sale (customer satisfaction). Transactional semantics such as atomicity and isolation in strict sense are not practical in workflow systems. Advanced transaction concepts such as compensation can be mapped to workflow domain in terms of a compensating task. Thus workflow should borrow ideas from advanced database technology. Data and communication correctness need to be verified before process execution. Exception handling only deals with low level. High-level exceptions aren't properly addressed.

Chapter 9

Conclusions and Future Research

9.1 Conclusions

Our approach synthesizes conventional software engineering techniques (namely, statecharts and process modeling) with artificial intelligence techniques (namely, agents and commitments) to develop a powerful approach for interoperation among autonomous, decentralized entities. We develop a means to declaratively characterize the different metacommitment patterns for agent interaction and show how different behavioral models for agents can support a given set of metacommitment patterns. This would yield general principles for the design of agents for interoperation. We apply agent interactions to process interoperation. Traditional workflow approaches only cover some simple interoperation. They couldn't accommodate the exceptions and revisions challenges in face to autonomy and heterogeneity. We show how our agent approach solves these challenges.

- *Metamodel for process interoperation.* We apply software engineering approaches to specify process interoperation. Sequence diagram are used to achieve requirement analysis. Class diagram (commitment-based metamodel for interoperation) and sequence diagram with the commitment patterns are used for design stage. Then the state diagrams such as statecharts are used to represent an operational model for interoperation.
- *Agent metacommitment patterns.* An important part for process interoperation is agent interaction in an agent approach. We abstract agent interaction scenarios into a set

of common interaction patterns, the metacommitment patterns. These patterns invoke commitments among agents. It can be used as some reusable and domain-independent components to represent the interaction dependency and enforce agent interaction.

- *Agent metacommitment pattern formalization.* We develop a means to formalize agent metacommitment patterns by Computation Tree Logic. CTL is used as the formal declarative semantics for the metacommitment patterns. We link some communication primitives to CTL notations and the commitment operations. Based on different purpose, we gave two kinds of formulizations for task-oriented and conversation-oriented metacommitment patterns. Task-oriented patterns are used to link to task executions of a workflow.
- *Operational semantics.* Nowadays statecharts are a well-accepted means to specify dynamic behavior of software systems. They are a part of the Unified Modeling Language (UML). We apply statecharts to specify agent behavior models. A set of sub-statecharts is developed from metacommitment patterns. We compose these different sets of the sub-statecharts into different agent behavior models. We also apply statcharts to specify different types of task execution models. In order to incorporate requirements of the revision and exception handling, we specify each task execution model with compensation handling. We give operational semantics of a set of control connectors with the reentrance feature.
- *Soundness.* We suggest an algorithm to derive CTL structure from a statechart. We apply CTL structure to verify the agent behavior models by the labeling model checker algorithm. We prove each model satisfies some specification and has some additional properties, such as liveness, responseness. We also give the operational sequence of inter-agent and intra-agent pattern execution. Intra-agent pattern execution follows the agent behavior models. Inter-agent pattern executions need to observe the logical clock to ensure correct operational sequences.

- *Prototype system and application.* We develop a means to represent the metamodel by using XML. XML DTD is used to check XML document validation. We developed API to parse the XML and create objects of an instance of a practical application's metamodel. We define agent interaction message format, map the metacommitment patterns and control connectors to a set of Jess rules. We apply our metacommitment patterns to the contract net protocol, distributed database system and travel planning system. We use the travel planning as a prototype system example

9.2 Future Research

A number of extensions are possible to this work. We envision the following ways in which the ideas developed in this thesis can be enhanced.

- On traditional side, we have applied software engineering approaches to the multiagent system from requirement, design, verification, and implementation. We used the UML notations, such as Class diagram, Sequence diagram, Statecharts, and so on. There are some enhancements in UML notations about sequence diagram, such as conditional and concurrency [50]. However sequence diagram doesn't provide a means to integrate a set of sequence diagrams, which are used to specify different interaction scenarios. Statecharts provides the syntax and semantics approaches to integrate or compose a set of sub-statecharts. Thus additional approaches are required to automatically translate or convert among these notations, for example, from sequence diagrams and class diagram to statecharts. These statecharts can be integrated, or composed into a large stachart. This makes it easy to verify some properties of a system.
- On the AI side, we have applied commitments of agents to abstract the interaction scenarios into the commitment patterns. We can use these commitment patterns to handle the exceptions and revisions during agent interactions. These patterns can be used to specify the dependency of agent interaction. Enhanced representations for teams and

organizations are required that would better accommodate the challenges of coherent, long-lived, complex activities in information environments and elsewhere. An agent can play a role, which could nest in another role when a team or organization is formed. There is a requirement to introduce some patterns about the agreement, or negotiation, and composition to complement our metacommitment patterns.

Bibliography

[1] Gustavo Alonso, Divyakant Agrawal, Amr El Abbadi, Mohan Kamath, Roger Günthör, and C. Mohan. Advanced transaction models in workflow contexts. In *International Conference on Data Engineering*, 1996.

[2] Sena Arpinar, Asuman Dogac, and Nesime Tatbul. An open electronic marketplace through agent-based workflows: MOPPET. *Journal on Digital Libraries*, 3(1), 2000.

[3] Mihai Barbuceanu and Mark S. Fox. COOL: A language for describing coordination in multi agent systems. In *Proceedings of the International Conference on Multiagent Systems*, pages 17–24, 1995.

[4] Mihai Barbuceanu and Mark S. Fox. Coordinating multiple agents in the supply chain. In *Proceedings of Fifth Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises*, pages 134–142, 1996.

[5] Martin Bichler, Carrie Bean, and Arie Segev. OFFER: A broker-centered object framework for requisitioning. In *Electronic IFIP conference*, 1998.

[6] Gregory Alan Bolcer and Gail E. Kaiser. Collaborative work (SWAP): Leveraging the web to manage workflow. *Internet Computing*, 3(1):85–88, 1999.

[7] Alan Bond and Les Gasser, editors. *Readings in Distributed Artificial Intelligence*. Morgan Kaufmann, San Francisco, 1988.

[8] Omran A. Bukhres and Ahmed K. Elmagarmid, editors. *Object-Oriented Multidatabase Systems: A Solution for Advanced Applications.* Prentice-Hall, 1996.

[9] Fabio Casati, Paul Grefen, Barbara Pernici, Giuseppe Pozzi, and Gabriel Sánchez. WIDE workflow model and architecture. TR 96.050, Dipartimento di Elettronica e Informazione, Politecnico di Milano, 1996.

[10] Cristiano Castelfranchi. Commitments: From individual intentions to groups and organizations. In *Proceedings of the International Conference on Multiagent Systems*, pages 41–48, 1995.

[11] Stefano Ceri, Paul Grefen, and Gabriel Sanchez. WIDE - a distributed architecture for workflow management. In *RIDE Conference*, 1997.

[12] B. Chaib-draa, B. Moulin, R. Mandiau, and P. Millot. Trends in distributed artificial intelligence. *Artificial Intelligence Review*, (6):35–66, 1992.

[13] Anthony Chavez and Pattie Maes. Kasbah: An agent marketplace for buying and selling goods. In *Proceedings of the 1st International Conference on the Practical Application of Intelligent Agents and Multiagent Technology (PAAM'96)*, 1996.

[14] Workflow Management Coalition. Interface 4 - interoperability abstract specification, 1996. URL: http://www.aiim.org/wfmc/mainframe.htm.

[15] Workflow Management Coalition. Interoperability Wf-XML binding, 2000. Pre-submission Draft, Version: 1.0.3, URL: http://www.aiim.org/wfmc/mainframe.htm.

[16] Philip R. Cohen and Hector J. Levesque. Intention is choice with commitment. *Artificial Intelligence*, 42:213–261, 1990.

[17] Randall Davis and Reid G. Smith. Negotiation as a metaphor for distributed problem solving. *Artificial Intelligence*, 20:23–109, 1983.

[18] Frank Dignum and Bernd van Linder. Modelling social agents: Communication as action. In *Intelligent Agents III: Agent Theories, Architectures, and Languages*, pages 205–218. Springer-Verlag, 1997.

[19] E. Allen Emerson. Temporal and modal logic. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 995–1072. North-Holland, Amsterdam, 1990.

[20] Herbert B. Enderton. A mathematical introduction to logic. Academic Press, 1972.

[21] Tim Finin, Richard Fritzson, Don McKay, and Robin McEntire. KQML as an agent communication language. In *Proceedings of the International Conference on Information and Knowledge Management*. ACM Press, 1994.

[22] Foundation for intelligent physical agents (FIPA) specification, 1998. www.fipa.org.

[23] Martin Fowler. *UML Distilled: Applying the Standard Object Modeling Language*. Addison-Wesley, Reading, MA, 1997.

[24] Ernest J. Friedman-Hill. Jess, the Java expert system shell, 1997. herzberg.ca.sandia.gov/jess.

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

[26] Hebert Guttman, Alexandros G. Moukas, and Pattie Maes. Agent-mediated electronic commerce: A survey. *Knowledge Engineering Review*, 1998.

[27] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[28] David Harel and Anmon Naamad. The STATEMATE semantics of statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, 1996.

[29] Anuj K. Jain, Manuel Aparicio IV, and Munindar P. Singh. Agents for process coherence in virtual enterprises. *Communications of the ACM*, 42(3):62–69, 1999.

[30] N. R. Jennings, P. Faratin, M. J. Johnson, T. J. Norman, P. O'Brien, and M. E. Wiegand. Agent-based business process management. *International Journal of Cooperative Information Systems*, 5(2&3):105–130, 1996.

[31] Heecheol Jeon, Charles Petrie, and Mark R. Cutkosky. JATLite: A java agent infrastructure with message routing. *IEEE Internet Computing*, (2), March/Apirl 2000.

[32] Heecheol Jeon, Charles Petrie, and Mark R. Cutkosky. JATLite: A Java agent infrastructure with message routing. *IEEE Internet Computing*, 4(2):87–96, March 2000.

[33] Donald Judge, Brian Odgers, John Shepherdson, and Zhan Cui. Agent enhanced workflow. *BT Technical Journal*, 16, 1998.

[34] Mohan Kamath and Krithi Ramamritham. Failure handling and coordinated execution of concurrent workflows. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 334–341, 1998.

[35] Mohan U. Kamath and Krithi Ramamritham. Failure handling and coordinated execution of concurrent workflows. In *Proceedings of 14th International Conference on Data Engineering*, 1998.

[36] Mark Klein and Chrysanthos Dellarocas. Exception handling in agent systems. In *Proceedings of the 3rd International Conference on Autonomous Agents*, pages 62–68, Seattle, 1999.

[37] J. Klingemann, J. Wasch, and K. Aberer. Adaptive outsourcing in cross-organizational workflows. In *the 11th International Conference on Advanced Information Systems Engineering (CAiSE'99), Heidelberg, Germany*, pages 417–421, 1999.

[38] Narayanan Krishnakumar and Amit Sheth. Managing heterogeneous multi-system tasks to support enterprise-wide operations. *Distributed and Parallel Databases*, 2(3):155–186, 1995.

[39] Heiko Ludwig. Termination handling in inter-organisational workflows - an exception management approach. In *Proceedings of the Seventh Euromicro Workshop on Parallel and Distributed Processing, Funchal*, 1999.

[40] N. Lynch, M. Merritt, W. Weihl, and A. Fekete. *Atomic Transactions.* Morgan Kaufmann, San Mateo, CA, 1994.

[41] R. Medina-Mora, T. Winograd, R. Flores, and F. Flores. The action workflow approach to workflow management technology. In *Proc. Of 4th Conf. On Computer Supported Cooperative Work (CSCW'94)*, 1994.

[42] Raúl Medina-Mora and Kelly W. Cartron. ActionWorkflow^R in use: Clark County department of business license. In *Proceedings of the 12th International Conference on Data Engineering (ICDE)*, pages 288–294, February 1996.

[43] Elliott Mendelson. Introduction to mathematical logic. D. Van Nostrand, 1979.

[44] Jürgen Müller. Negotiation principles. In N. R. Jennings and G. M. P. O'Hare, editors, *Foundations of Distributed Artificial Intelligence.* 1996.

[45] Gary J. Nutt. Using workflow in contemporary IS applications. TR CU-CS-663-93, University of Colorado, August 1993.

[46] Hyacinth Nwana, Divine Ndumu, Lyndon Lee, and Jaron Collis. Zeus: A tool-kit for building distributed multi-agent systems. *Applied Artifical Intelligence*, 13(1), 1999.

[47] OASIS. Common business language, 1999. URL: www.veosystems.com/xml/cbl/cbl.html.

[48] OBI. Open buying on the internet consortium, 1999. Technical Specifications release 2.0, URL: http://www.openbuy.org.

[49] P. D. O'Brien and M. E. Wiegand. Agent based process management: applying intelligent agents to workflow. *The Knowledge Engineering Review*, 13(2):1–14, 1998.

[50] James Odell, H. Van Dyke Parunak, and Bernhard Bauer. Extending uml for agents. In *AOIS Worshop at AAAI 2000*, 2000.

[51] OMG. Workflow management facility, 1998. OMG Document Number: bom/98-06-07, URL: http://www.aiim.org/wfmc/mainframe.htm.

[52] A. Pnueli and M. Shalev. What is in a step: On the semantics of statecharts. *Lecture Notes in Computer Sciecne*, 526:244–264, 1991.

[53] Martin Sachs and John Ibbotson. Electronic trading-partner agreement for e-commerce, 2000. Pre-submission Draft, Version: 1.0.3, URL: http://www.oasis-open.org/cover/tpa.html.

[54] T. Sandholm and V. Lesser. Issues in automated negotiation and electronic commerc: Extending the contract net framework. In *Proceedings of the First International Conference on Multiagent Systems*, 1995.

[55] John R. Searle. *Speech Acts*. Cambridge University Press, Cambridge, UK, 1969.

[56] J. W. Shepherdson, S.G. Thompson, and B.R. Odgers. Cross organisational workflow co-ordinated by software agents. *Work Activity Co-ordination and Collaboration*, 13(2):94–96, 1999.

[57] Yoav. Shoham. Agent-oriented programming. *Artificial Intelligence*, 60(1), 1993.

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

[59] Munindar P. Singh and Michael N. Huhns. Automating workflows for service provisioning: Integrating AI and database technologies. *IEEE Expert*, 9(5):19–23, October 1994.

[60] Mukesh Singhal and Niranjan G. Shivaratri. *Advanced Concepts in Operating Systems*. McGraw-Hill, 1994.

[61] Katia Sycara, Jiangui Lu, Matthias Klusch, and Seth Widoff. Matchmaking among heterogeneous agents on the Internet. In *Proceedings of the AAAI Spring Symposium on Intelligent Agents in Cyberspace*, 1999.

[62] J. D. Tygar. Atomicity in electronic commerce (invited paper). In *ACM/IEEE 21st Conference on Principles of Distributed Computation*, 1996.

[63] Mahadevan Venkatraman and Munindar P. Singh. Verifying compliance with commitment protocols: Enabling open web-based multiagent systems. *Autonomous Agents and Multi-Agent Systems*, 2(3):217–236, September 1999.

[64] H. Weigand, E. Verharen, and F. Dignum. Dynamic business models as a basis for interoperable transaction design. *Information Systems*, 22(2):139–154, 1997.

[65] Terry Winograd and Fernando Flores. *Understanding Computers and Cognition: A New Foundation for Design.* Addison-Wesley, Reading, MA, 1987.

[66] Peter R. Wurman, William E. Walsh, and Michael P. Wellman. Flexible double auctions for electronic commerce: Theory and implementation. *Decision Support Systems*, 24:17–27, 1998.