Agent-Oriented Requirements Engineering Using the *ConGolog* and *i** Frameworks

By

Xiyun Wang

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in

THE FACULTY OF GRADUATE STUDIES

DEPARTMENT OF COMPUTER SCIENCE

YORK UNIVERSITY

TORONTO, ONTARIO

©Copyright by Xiyun Wang, 2001

Abstract

Agent-oriented approaches are becoming more popular in software engineering, both as architectural frameworks and as modeling formalisms for requirements engineering and design. In this thesis, two agent-oriented modeling frameworks, *i** and *ConGolog*, will be used together for requirements engineering. The *i** framework has been developed for the early stages of requirements engineering and supports the modeling of social dependencies between agents with respect to tasks and goals both functional and non-functional. *ConGolog* is an agent-oriented process modeling framework that is very expressive and fully formal. It can be used to model complex processes involving loops, nondeterminism, concurrency, and multiple-agents and can accommodate incompletely specified models. It is well adapted to late requirements engineering and early design stages of system development, when detailed alternative process designs have to be specified and compared.

This thesis develops a methodology involving the combined use of *i** and *ConGolog* for agent-oriented requirements engineering. We identify steps in the requirements engineering process and how the *i** and *ConGolog* models of a system/domain need to be refined at each step, as well as map out the relationships between corresponding elements of the *i** and *ConGolog* models. The methodology developed is tested on a meeting scheduling application and a mail-order business application. The resulting methodology is compared to related work such as the *KAOS* method and proposals involving the combined use of *i** and *ALBERT-II*.

Acknowledgements

First, I want to thank my thesis supervisor, Professor Yves Lespérance. He put a lot of efforts to guide me and discuss questions with me, which led to the successful completion of my thesis.

I also want to thank Professor Richard Paige. He reviewed my thesis proposals and gave me feedback with comments, which led to the improvement of my work.

I also want to thank Professor Jonathan Ostroff and Henry Kim for reviewing my thesis and giving me good comments.

Finally, I want to thank my parents. They always encourage me to make achievements in my work. Their support is the key to my success in my study.

The study in the Computer Science Graduate Programme is a treasure experience in my life-time. Thank you all who gave me supports from my heart.

Contents

Abstract .... **iv**

Acknowledgements .... **v**

List of Tables .... **x**

List of Figures .... **xi**

1 Introduction **1**

1.1 The Problem 1

1.2 The Approach 3

1.3 Overview of the Thesis 5

2 Related Work **6**

2.1 Software Engineering: Definitions and Stages 6

2.2 Requirements Engineering: Definitions and Phases 8

2.3 Multiagent Systems 10

2.4 *ALBERT-II* 13

2.5 *KAOS* 15

2.6 Agent-Oriented Methodologies 17

3 Foundations **20**

3.1 The *i** Modeling Framework 20

3.1.1 The Strategic Dependency (SD) Model 22

3.1.2 The Strategic Rationale (SR) Model 31

3.1.3 Discussion 39

3.2 The *ConGolog* Modeling Framework 40

3.2.1 Introduction 40
3.2.2 Modelling a Domain 41
3.2.3 Modelling Domain Dynamics...42
3.2.3.1 The Situation Calculus Language 42
3.2.3.2 Domain Dynamics Specification in the Situation Calculus...................45
3.2.3.3 Summary...47
3.2.4 Modelling Domain Processes in *Congolog* 47
3.2.5 Analysing Domain Specifications Using *ConGolog* Tools..........................52
3.2.6 Summary/Discussion...54

4 A Methodology for the Combined Use of the *i and *ConGolog* Frameworks 55**

4.1 SR Diagram Annotations 57
4.1.1 Composition Annotations..58
4.1.2 Link Annotations 61
4.2 Operationalizing Dependencies in the *i** SR model 64
4.3 The Annotated *i** SR Diagram 72
4.4 Mapping Rules 73
4.4.1 Mapping Rules for Nodes..73
4.4.2 Mapping Rules for Links..76
4.4.3 Mapping Dependencies 83
4.5 A Methodology for the Combined Use of the *i** and *ConGolog* Frameworks 84

5 Case Study I: A Meeting Scheduling Process 89

5.1 Building the Strategic Dependency (SD) Model...90
5.2 Building the Strategic Rationale (SR) Model 95
5.3 Building the Annotated *i** SR Model 102
5.3.1 Suppressing Unnecessary Information...103
5.3.2 Operationalizing the Dependencies 106
5.3.3 Relativizing the Goals that Cannot Always Be Achieved..........................113

5.3.4 Filling out Process Details using Decompositions and Annotations 114
5.4 Developing the Initial *ConGolog* Model 125
5.4.1 The Initial *ConGolog* Model for `Initiator` 125
5.4.2 The Initial *ConGolog* Model for `MeetingScheduler` 130
5.4.3 Specifying the Domain Dynamics 135
5.5 Validating the *ConGolog* Model by Simulation... 141
5.5.1 Specifying a System Instance 142
5.5.2 Simulation Examples 143
5.5.3 Discussion.. 146
5.6 Refining the *i** and *ConGolog* Model Based on Validation Results.................. 147

6 Case Study II: A Mail-Order Business Application 152

6.1 Building The Strategic Dependency (SD) Model 153
6.2 Building The Strategic Rationale (SR) Model 157
6.3 Building The Annotated *i** SR Model 160
6.3.1 Suppressing Unnecessary Information.. 161
6.3.2 Operationalizing Dependencies 162
6.3.3 Relativizing the Goals That Cannot Always Be Achieved 172
6.3.4 Filling out Process Details Using Decomposition and Annotations............ 172
6.4 Developing the Initial *ConGolog* Model 180
6.4.1 The Initial *ConGolog* Model for `StockClerk` 184
6.4.2 Specifying the Domain Dynamics 186
6.5 Validating the *ConGolog* Model by Simulation... 188
6.5.1 Specifying a System Instance 188
6.5.2 Simulation Examples 191
6.6 Refining the *i** and *ConGolog* Models Based on Validation Results 198
6.6.1 Modifying the *ConGolog* Model and Corresponding Parts of the *i** Model—
An Example 198
6.6.2 The Process Alternatives for the Example 200

7 Discussion....203

7.1 An Evaluation of the Methodology 203

7.2 Issues in Mapping *i** to *ConGolog*....206

8 Conclusion....209

8.1 Contributions....209

8.2 Comparison to Related Work....211

8.3 Future Work 213

Bibliography 216

Appendix A: Modeling the Meeting Scheduling Process ....A—1

A-1 The *ConGolog* Model for `Participant` .... A—1

A-2 Successor State Axioms for Actions .... A—2

A-3 Actions and Fluents. .... A—7

A-4 Obtaining Simulation Traces under Unix .... A—9

A-5 The Simulation Trace for Example 3 in Section 5.5 .... A—11

A-6 The Whole *ConGolog* Model for the Meeting Scheduling Process.... A—13

A-7 The Initial *ConGolog* Model for `MeetingScheduler` .... A—28

A-8 The Precondition Axioms for Actions.... A—30

Appendix B: Modeling the Mail-Order Business Process....B—1

B-1 Obtaining Simulation Traces under Unix .... B—1

B-2 The Simulation Trace for Example 5 in Section 6.5 .... B—2

B-3 The Simulation Trace for Example 6 in Section 6.6. .... B—4

B-4 The *ConGolog* Model for `OfficeClerk` .... B—5

B-5 The *ConGolog* Model for `BankClerk`.... B—6

B-6 The *ConGolog* Model for `Customer` .... B—7

B-7 The Whole *ConGolog* Model for the Mail-Order Business Process .... B—8

List of Tables

Table 3.1 Constructs for processes in *ConGolog* 49

Table 3.2 Constructs for conditionals in *ConGolog* 50

Table 4.1 *ConGolog* operators associated with composition annotations 79

Table 4.2 *ConGolog* constructs associated with link annotations 90

List of Figures

Figure 3.1 The SD model for a simple meeting scheduling process 23
Figure 3.2 The SR model for a simple meeting scheduling process 33
Figure 4.1 Sequence annotation applied to a group of decomposition links 58
Figure 4.2 Concurrency annotation applied to a group of decomposition links 59
Figure 4.3 Alternative annotation applied to a group of decomposition links............ 60
Figure 4.4 Prioritized concurrency annotation applied to a group of decomposition links 60
Figure 4.5 While-loop annotation attached to a single decomposition link 61
Figure 4.6 For-loop annotation attached to a single decomposition link 62
Figure 4.7 Interrupt annotation attached to a single decomposition link 63
Figure 4.8 If annotation attached to a single decomposition link 63
Figure 4.9 Pick annotation attached to a single decomposition link 64
Figure 4.10 SR diagram for the task dependency before operationalization.............. 66
Figure 4.11 SR diagram for the task dependency after operationalization................. 67
Figure 4.12 SR diagram for the goal dependency before operationalization 68
Figure 4.13 SR diagram for the goal dependency after operationalization 69
Figure 4.14 SR diagram for the resource dependency before operationalization 70
Figure 4.15 SR diagram for the resource dependency after operationalization.......... 71
Figure 4.16 The mapping for an agent node in the annotated SR diagram 74
Figure 4.17 The mapping for a role/position node in the annotated SR diagram 75
Figure 4.18 The mapping for a goal node in the annotated SR diagram.................... 75
Figure 4.19 The mapping for a task node in the annotated SR diagram 76
Figure 4.20 E.g. task node with task decomposition links in the annotated SR diagram 77

Figure 4.21 The mapping for the SR diagram of Figure 4.20.................................. 77
Figure 4.22 Task node `t` with task decomposition links in the annotated SR diagram 78
Figure 4.23 The mapping for the task decomposition of Figure 4.22 78
Figure 4.24 E.g. goal node with goal decomposition links in the annotated SR diagram 80
Figure 4.25 `m_achieve(g)` for the goal node g of Figure 4.24 81
Figure 4.26 Goal node `g` with goal decomposition links in the annotated SR diagram. 82
Figure 4.27 `m_achieve(g)` for the goal node `g` of Figure 4.26 82
Figure 5.1 A Strategic Dependency model for the meeting scheduling process 92
Figure 5.2 An initial Strategic Rationale model for the meeting scheduling process 97
Figure 5.3 The second version of the *i** SR model for the meeting scheduling process 104
Figure 5.4 The third version of the *i** SR model for the meeting scheduling process 105
Figure 5.5 The SR diagram for the dependencies between the meeting initiator and the MS 107
Figure 5.6 The SR diagram after operationalizing the dependencies of Figure 5.5.. 107
Figure 5.7 The SR diagram for the dependencies between the MS and the participants 109
Figure 5.8 The SR diagram after operationalizing the dependencies of Figure 5.7.. 110
Figure 5.9 (a) The SR diagram for the initiator after operationalizing dependencies..... 111
Figure 5.9 (b) The SR diagram for the MS after operationalizing dependencies 112
Figure 5.9 (c) The SR diagram for the participant after operationalizing dependencies 113
Figure 5.10 (a) The annotated SR diagram for the meeting initiator 116

Figure 5.10 (b) The annotated SR diagram for the MS 118
Figure 5.10 (c) The annotated SR diagram for the participant 122
Figure 5.11 The initial *ConGolog* model for the initiator 126
Figure 5.12 The mapping for the role node `Initiator` 127
Figure 5.13 The mapping for the goal node `MeetingBeenScheduledIfPossible` 127
Figure 5.14 (a) The mapping for the task node `TryScheduleMeeting` 128
Figure 5.14 (b) The mapping for the task node `LetSchedulerScheduleAMeeting` 128
Figure 5.15 The mapping for the agent node `MeetingScheduler` 131
Figure 6.1 The Strategic Dependency model for the mail-order business process ... 155
Figure 6.2 The initial Strategic Rationale model of the mail-order process 158
Figure 6.3 The second version of the SR model for the mail-order process 161
Figure 6.4 The third version of the SR model for the mail-order process 163
Figure 6.5 Dependencies between roles played by `StockClerk` agent 165
Figure 6.6 SR diagram for the dependencies of Figure 6.5 after operationalization 165
Figure 6.7 SR diagram for dependencies between `BankClerk` and `OrderProcessor` after operationalization 168
Figure 6.8 SR diagram for the dependencies between `Customer` and `OrderProcessor` after operationalization 169
Figure 6.9 The SR model for the mail-order process after operationalizing dependencies 171
Figure 6.10 (a) The annotated SR diagram for the agent `Customer` agent 172
Figure 6.10 (b) The annotated SR diagram for positions `Bank` and `BankClerk` 173
Figure 6.10 (c) The annotated SR diagram for agent `StockClerk` with its three roles 174
Figure 6.10 (d) The annotated *i** SR diagram for the agent `OfficeClerk` 175

1 Introduction

1.1 The Problem

With the advent of the Word-Wide Web and electronic commerce, trends in software are towards open systems, more integration across applications, and systems that can adapt to change. In response to this, many developers are starting to adopt agent-oriented architectures, where a system is composed of agents, autonomous entities that can interact in flexible ways. For instance, agents can interact through negotiation, while working towards their goals and reacting to changes in the environment [JW98]. To support the development of agent-based systems, suitable software engineering methods and tools are required. So far, most efforts in this area have been directed at the design phase of software development. In this thesis, we focus mainly on the requirements engineering phase.

Requirements engineering (RE) studies what goals are to be accomplished by the system to be built, how those goals should be operationalized into services and constraints, and how the responsibilities for the resulting requirements can be assigned to agents such as humans, devices, and software. The processes involved in RE include domain analysis, requirements elicitation, specification, and assessment, negotiation about requirements and the documentation and evolution of requirements [VanL00]. Much of requirements engineering research has taken as its starting point the initial requirements statements, which express the client's wishes about what the system should do, and which are often ambiguous, incomplete, inconsistent, and usually expressed informally, such as in natural language text. The objective of requirements engineering (RE) is to produce a requirements document that resolves these problems and is suitable for developers to start developing systems.

RE is becoming more and more important because as the earliest stage of software development, it is being acknowledged as the crucial stage for successful software development, successful subsequent deployment, and ongoing evolution of the system [Boe81]. Recent surveys have confirmed that RE is becoming recognized as an area of utmost importance in software engineering research and practice [Davis90].

Requirements engineering (RE) started with the study of what the system should do, i.e., late-phase requirements analysis, which focuses on the specification of requirements and their completeness, consistency, and automated verification. Most existing requirements modeling frameworks are proposed for this late phase of requirements engineering and they can help the modeler in making requirements precise, complete, and consistent. Most impose some degree of structure and formality (from box-and-arrow diagrams to logical formalisms) [Bubenko80] [DUHLPR86]. However considerably less attention has been given to supporting the activities that precede the formulation of the initial requirements [Bubenko95]. A good understanding of the organizational context and rationales (the "whys") of a system that leads up to system requirements is very important for the successful development of the system. If one doesn't understand why things are done the way they are, one is likely to automate outdated processes and miss the opportunity to innovate. In choosing among alternative processes for the system, the modeler must be able both to describe relationships and to propose and argue about solutions from a strategic perspective. The early-phase requirements analysis focuses on how the desired system will meet its goals and accomplish its tasks, why the system must be developed, what alternatives can be proposed, what the relationships between various actors or stakeholders are, and how the interests of actors can be achieved. The emphasis here is on understanding the "whys" that underlies system requirements rather than on the precise and detailed specification of "what" the system should do [YM94B].

Early-phase RE activities have traditionally been done informally and without much tool support. As the complexity of the problem domain increases, it is evident that tool

support will be needed to leverage the efforts of the requirements engineer. A considerable body of knowledge would be built up during early-phase RE. This knowledge would be used to support reasoning about organizational objectives, system-and-environment alternatives, implications for stakeholders, etc. It is important to retain and maintain this body of knowledge in order to guide system development, and to deal with change throughout the system's lifetime.

There are many formal requirements modeling languages and frameworks for late-phase requirements analysis, for instance *KAOS* [DVF93], *ALBERT-II* [DuBois95], etc. Early-phase requirements analysis on the other hand, has generally been done informally and without much tool/formalism support. The *i** framework [YDDM97] was developed for early-phase requirements analysis. It provides an informal diagram-based notation that supports the modeling of social dependencies between agents and how process design choices effect agents' goals. But *i** is not a formal language and it has limited support for describing complex processes. *ConGolog* [DLL00] [LKMY99] is a formal language for process specification and agent programming. It supports the formal specification of complex multiagent systems and provides a tool for process simulation. But it lacks features for modeling the rationale behind design choices. The two frameworks complement each other and it would be good to have a methodology for using them in combination.

1.2 The Approach

In this thesis, the combined use of the *i** and *ConGolog* frameworks for requirements analysis is investigated. The *i** framework [Yu95B] was proposed to provide the kinds of modeling features and reasoning capabilities that might be appropriate for early-phase requirements engineering. It introduces an ontology and reasoning support features that are substantially different from those intended for late-phase RE. *i** views processes as involving social actors who depend on one another for goals to be achieved, tasks to be performed, and resources to be supplied. It has also been represented in the conceptual

modeling language Telos [MYBJK91]. *i** provides two kinds of models: the Strategic Dependency (SD) model and the Strategic Rationale (SR) model. The SD model is used to describe the dependency relationships between actors. The SR model is used to describe how goals and tasks are decomposed within actors and how dependencies with other actors relate to this. But *i** is not a formal logic-based language and has limited support for describing complex processes. *ConGolog* [DLL97] [DLL00] can model the detailed dynamics of processes, and supports the validation of process specifications using simulation and automated reasoning techniques. It also supports formal specifications and complex process descriptions. The modeler even can reason about processes with only a partial description of the real-world state. But *ConGolog* lacks features for modeling the motivations, intents, and rationales behind processes. Thus, we can assert that the frameworks are complementary.

In our combined methodology, the *i** framework will be used to model different alternatives for the desired system, analyze and decompose the functions of the different actors, and model the dependency relationships between the actors and the rationales behind process design decisions. The *ConGolog* framework will be used to formally specify the system behavior described informally in the *i** model. The *ConGolog* model will provide more detailed information about the actors, tasks, processes, and goals in the system, and the relationships between them. Complete *ConGolog* models are executable and this will be used to validate the specifications by simulation. To bridge the gap both syntactic and semantic between *i** and *ConGolog* models, an intermediate notation involving the use of process specification annotations in *i** SR diagrams will be introduced. We will also propose a set of mapping rules that constrains the modeler to map the elements of the annotated SR diagram to appropriate *ConGolog* entities and ensures that the two models are consistent. Finally we propose a methodology for the combined use of the *i** and *ConGolog* frameworks for early to late phase requirements engineering. We then illustrate the use of the methodology on two examples: a meeting scheduling application and a mail-order business process. The goal of the combined use

of the *i** and *ConGolog* frameworks is to exploit the advantages of these two frameworks to provide a better tool for early to late phase requirements analysis.

1.3 Overview of the Thesis

In chapter 2, we discuss related work on frameworks for requirements engineering and agent-oriented software engineering in recent years. We review what software engineering, requirements engineering, and agents are and introduce different frameworks and techniques used in these areas. In chapter 3, the two frameworks that we will use: *i** and *ConGolog*, are presented. We describe the ontologies of the frameworks and their components and features, and discuss how they are used. In chapter 4, we develop a methodology for the combined use of the *i** and *ConGolog* frameworks for requirements engineering. A set of process specification annotations are defined to elaborate the *i** SR model. With these, one can produce an annotated SR model that bridges the gap between the SR model and the *ConGolog* model. We require the analyst to define a mapping between elements of the annotated SR model and corresponding elements of the *ConGolog* model. To ensure the consistency of the mapping, we define a set of mapping rules. In chapter 5 and chapter 6, two case studies, a meeting scheduling application and a mail-order business application, will be presented to test this methodology. We show how models can be validated by simulation. In chapter 7, we discuss the benefits of our approach and examine some issues that remain to be resolved. In chapter 8, we review the contributions of the thesis and discuss what further research could be done.

2 Related Work

In this chapter, we start by providing some basic background on software engineering and requirements engineering, and discuss research trends in the latter area. Then, we introduce agent-oriented computing and the notions of agent and multiagent systems. Then, we review work on two RE frameworks that are more closely related to our work, *ALBERT-II* and *KAOS*. We also survey work on agent-oriented design methodologies. The frameworks that we use in our approach, *i** and *ConGolog*, are presented in detail in chapter 3. We compare our approach to closely related work in chapter 7.

2.1 Software Engineering: Definitions and Stages

Software Engineering (SE) is the term used to describe software development that respects the following principles: helping organizations involved in software development in making sure that the software is developed according to accepted industry practices, with good quality control, adherence to standards, and in an efficient and timely manner.

In [Leach2000], Leach argues that SE lies at the heart of the computer technology revolution. SE is described as "the application of engineering techniques to develop and maintain software that runs properly and its constructed in an efficient manner". SE may involve the following activities: problem analysis, requirements, specification, design, coding, testing and integration, installation and delivery, documentation, maintenance, quality assurance, and training. The software produced should satisfy qualities such as being efficient, reliable, usable, modifiable, portable, testable, reusable, maintainable, interoperable, and correct.

Davis says that SE is the application of scientific principles to: "(1) the orderly transformation of a problem into a working software solution and (2) the subsequent maintenance of that software until the end of its useful life" [Davis90]. The software development cycle typically involves following stages:

- *Requirements*: analyzing the current problem and proposing a complete specification of the desired external behavior of the software system to be developed.
- *Design*: in preliminary design, one decomposes the software system into its actual constituent components, and then repeatedly decomposes those components into smaller and smaller sub-components until the sub-components are small enough to be solved by a person easily; detailed design defines and documents algorithms for each component that will be realized as code.
- *Coding*: transforms the algorithms defined during detailed design into a computer-understandable language.
- *Testing*: first, in unit testing, one checks each coded module of a sub-component for the presence of bugs and ensures that each module behaves according to its specification as defined during detailed design; then in integration testing, one interconnects sets of previously tested modules to ensure that the sets behave as well as they did when independently tested, and ensure that each component integrated from those sub-components behaves according to its specification defined during preliminary design; finally, in system testing, one checks the entire software system embedded in its actual hardware environment to ensure that it behaves according to the requirements specifications.
- *Delivery*, *production*, and *deployment*: after the testing stage, the software and the hardware it runs on should be delivered and become operational for the client.
- *Maintenance* and *enhancement*: the maintenance and enhancement processes are actually a full development life cycle. If a coding change is made, then the coding and the three subsequent testing stages must be performed. If a design change is made, the design, coding, and three subsequent testing stages must be performed. If a requirement change has occurred then all stages must be performed.

Leach [Leach2000] also describes four basic models of the software development life cycles: the waterfall model, the rapid prototyping model, the spiral model, and the market-driven model. Royce [ROY70] first used the term “waterfall model” to characterize the series of software engineering stages.

Many modeling languages have been proposed both for requirements analysis and specification, such as *ALBERT-II* [DUDZ95] [DuBois97], *KAOS* [DVF93], etc., and for system design such as *UML* [RUJB99] [UML98], *BON* [WN95], and *Z*-notation [Spivey92].

2.2 Requirements Engineering: Definitions and Phases

Boehm [Boe81] points out that requirements analysis and specification is a crucial stage to ensure the correctness and cost- and time-effectiveness of the system development in all the stages of software development. A “requirements” is defined as “something required; something wanted or needed” [Web84]. The IEEE standard 729 defines it as “(1) a condition or capability needed by a user to solve a problem or achieve an objective; (2) a condition or capability that must be met or possessed by a system … to satisfy a contract, standard, specification, or other formally imposed document.” [IEEE83]

According to [Davis95], requirements engineering (RE) is the set of activities including “eliciting or learning about a problem that needs a solution, and specifying the external behavior of a system that can solve that problem”. The aim of RE is to develop a requirements specification, a precise set of concordant descriptions of the requirements, a set that the parties, developer and stakeholders, can agree upon [REQ97].

RE is a crucial part of software engineering. It helps the software engineer in exploring, understanding, documenting, and refining the needs and expectations of clients for a desired system to the extent that the engineer can develop an implementation. RE is also important because requirements errors are often made, and not detecting these errors

early may lead to significant software costs. Also the resulting software may not satisfy user's real needs if requirements errors are not corrected. The later in development life cycle that a software error is detected, the more expensive it will be to repair. It is better to detect the errors at the requirements analysis stage. There are some automated tools can be used to detect a significant number of errors in an approved software requirements specification [Leach2000].

Recent work in this area has emphasized the need for an engineering approach, where models and languages, methods and tools are employed to assist in the RE effort. Empirical studies of software development projects have also confirmed the crucial importance of domain knowledge and requirements analysis. For example, it has been estimated that an error that is not identified and corrected in the requirements phase can cost a hundred times more to correct in subsequent phases [Boe81].

The survey paper for RE of Lamsweerde [VanL00] presents a brief history of the main concepts and techniques developed to support the RE task. It argues that recent research on RE is concerned with the identification of the goals to be achieved by the envisioned system, the operationalization of these goals into services and constraints, and the assignment of responsibilities for the resulting requirements to agents such as humans, devices, and software. The processes involved in RE include domain analysis, requirements elicitation, specification, and assessment, negotiation about requirements, and their documentation, and evolution. In [JP84], Jarke and Pohl discuss the current research challenges for RE.

Castro *et al.* divide the stage of requirements analysis and specification into two sub-stages: early requirements and late requirements [CK00]. The *early requirements stage* focuses on studying the organizational setting, understanding the software development problem, and specifying an organizational model which includes relevant actors, their

goals, and their inter-dependencies. The *late requirements stage* is concerned with specifying the system of interest within its operational environment, along with its relevant functions and qualities.

According to Yu, this early phase of requirements engineering can be just as important as that of refining initial requirements to a requirements specification [Yu97]. First, system development involves many assumptions about the embedding environment and domain. Poor understanding of the domain is a primary cause of project failure. To have a deep understanding about a domain, one needs to understand the interests, priorities, and abilities of the various actors and players, in addition to having a good grasp of the domain concepts and facts. Second, users need help in coming up with initial requirements in the first place. As technical systems increase in diversity and complexity, the number of technical alternatives and organizational configurations made possible bring out many options. A systematic framework is needed to help developers understand what clients want and help users understand what technical systems can do. Third, it is well known that changes to requirements are a major source of problem. Traceability is an important part in software engineering. Having an understanding of organizational issues would allow software changes to be traced all the way to the originating source, i.e., the organizational changes that leads to requirements changes. Finally, having well-organized bodies of organizational and strategic knowledge would allow such knowledge to be shared across domains at this high level, deepening understanding about relationships among domains.

2.3 Multiagent Systems

The concept of an agent is becoming more and more important in the areas of Artificial Intelligence (AI) and Software Engineering (SE). Progress in SE over the past two decades has primarily been made through the development of increasingly powerful and natural abstractions with which to model and develop complex systems. Increasingly, many computer systems are being viewed in terms of autonomous agents. Agents are

being thought of as the next generation model for engineering complex distributed systems. Agents are also being used as a framework for bringing together the component AI sub-disciplines that are necessary to design and build intelligent entities.

Also agent-oriented approaches are also becoming more popular in modeling formalisms for requirements engineering and design. It is natural to view the users of a system, the organizations in which they are involved, and even system components as agents that have knowledge, goals, intentions, etc. This allows the designer to explain or predict their behavior even when there is little information about their internal structure [LS99].

The term "agent" can be defined as "an encapsulated computer system that is situated in some environment, and that is capable of flexible, autonomous action in that environment in order to meet its design objectives" [Wooldridge97]. Agents are typically thought to have at least four basic attributes: *Autonomy* — they are not controlled directly by humans or others, *cooperativeness* — they communicate and work together, *perceptiveness* — they perceive their environment and react to it, and *pro-activeness* — they exhibit goal-directed behavior [WJ95].

In [WJ96] [WJ95], the theoretical and practical issues associated with the design and construction of intelligent agents are addressed. Work on agents is divided into three areas: "agent theory", which deals with the question of what an agent is and how to represent and reason about the properties of agents using mathematical formalisms, "agent architectures", which is concerned with software engineering models of agents, i.e., how to construct software or hardware systems that will satisfy the properties specified by agent theorists, and "agent languages", which are software systems for programming and experimenting with agents.

In [Wooldridge98], Wooldridge summarizes why agents are perceived to be an important new trend in software engineering and then reviews the various techniques and

formalisms that have been developed for engineering agent-based systems. New distributed internet computing applications tend to be open systems, there is more integration across applications, and systems that can adapt to change. Agent-oriented architectures are adopted into the SE area to help dealing with these new issues. A multiagent system is composed of agents, autonomous entities that can interact in flexible ways, for instance through negotiation, while working towards their goals and reacting to changes in the environment [JW98].

[JSW98] provides an overview of research and development activities in the field of autonomous agents and multiagent systems. It aims to identify key concepts and applications, and indicate how they relate to one-another.

In [JW00], Jennings *et al.* argue that agent-oriented techniques represent a significant new means of analyzing, designing, and building complex software systems. They have the potential to significantly improve current practice in software engineering and to extend the range of applications that can feasibly be tackled. The paper specifically argues that: (i) the conceptual apparatus of agent-oriented systems is well-suited to building software solutions for complex systems and (ii) agent-oriented approaches represent a genuine advance over the current state of the art for engineering complex systems. The major issues raised by adopting an agent-oriented approach to software engineering are highlighted and discussed.

Lespérance and Shapiro [LS99] discuss why agent-orientation is important for RE. According to them, "the specification of agents in terms of their mental states (beliefs, goals, commitments, etc.) allows modeling at a higher level of abstraction." It may be possible to illustrate or forecast agents' behavior even when their internal control structure is not explicitly known by us through describing these "mental attitudes" of agents. "Mental attitudes" can help us understand how the agents will react to changes made in their environment or organization. Also "representing communication as various

types of speech acts being performed by agents abstracts over the form and mechanism of messages" when modeling the organization or environment where a system runs. Moreover, models can absorb the analyses of multiagent cooperative problem-solving and integrate representations of social relations and rules into themselves. Finally, the authors argues that RE tools can use implementation techniques for agent-oriented frameworks to obtain more powerful and effective modeling and analysis techniques. There have been many proposed methodologies for analyzing, designing, and building multiagent systems [IG98]. We discuss agent-oriented methodologies in section 2.6.

2.4 *ALBERT-II*

ALBERT-II [DUDZ95] [DuBois97] (an acronym for Agent-Oriented Language for Building and Eliciting Real-Time requirements) is an agent-oriented requirements engineering framework that was developed by Dubois and others at the University of Namur. It is designed for the purpose of modeling functional requirements related to distributed heterogeneous real-time cooperative systems [DUP94] [DUDZ95] [ZDD98]. The design of the language has focused on three aspects: agent-orientation, formality, and expressiveness. The language models agents and their properties: internal states, responsibility for actions, perception of the environment, etc. Agents can be grouped into classes or societies, and this "object-oriented" approach can help in structuring large specifications [DUDU94]. Typical patterns of constraints are identified that can support the analyst in writing complex and consistent formulas. Cooperation constraints are specified to model how agents interact with each other in order to fulfil the overall goal.

ALBERT-II models a system as a collection of agents [YDDM97]. These agents interact with each other in order to accomplish tasks for the system. Each agent is specified in terms of its state structure including what it knows, the actions it can perform, and various types of constraints on them. The actions associated with an agent change or maintain its own state and/or the states of other agents. Action perception and state perception constraints specify what an agent can see about other agents. Action

information and state information constraints specify what an agent shows to other agents. Internal constraints specify the internal behavior of agents, the effects of their actions and trigger conditions for them. Cooperation constraints specify how agents interact with other agents. The constraints in *ALBERT-II* are specified in a typed temporal first-order logic, which support statements about agents' knowledge. *ALBERT-II* provides a graphical component for describing system structure and a textual component for constraints specifying admissible behaviors of agents through logical formulas.

ALBERT-II is a successor to the *ALBERT* language [DUDU94], a formal language based on the concept of 'agent' (seen as a specialization of the 'object' concept) in terms of which one may express real-time requirements as well as 'non-functional' requirements related to the reliability and security aspects of agents. In [DUDU94], the *ALBERT* language is presented and its usage is illustrated through a computer-integrated manufacturing case study. Some methodological guidelines are proposed to help to the analyst in the incremental elaboration of a complex requirements document.

A support tool for *ALBERT-II* is discussed in [Dubois98]. Recent work on *ALBERT-II* has dealt with its formal semantics and theorem proving with PVS [CRFS98], animation and scenarios [HD98] [HHJ98], applications to computer integrated manufacturing [DUYP98], and conceptual reasoning [ZDD98].

Bissener studied the combined use of the *i** and *ALBERT-II* frameworks for requirements engineering in [Bissener97]. He proposed a methodology that deals with organizational, and non-functional, as well as functional requirements. The functional requirements are specified in *ALBERT-II* and the organizational issues and non-functional requirements are specified in *i**. A mail-order business process is used to validate the methodology. The example is similar to the one we will study in chapter 6.

In [YDDM97], Yu *et al.* show how two agent-oriented frameworks, *i** and *ALBERT-II* can be used together for requirements engineering in cooperative information systems. *ALBERT-II* specifies requirements formally through states and actions, and information and perception. *i** help understanding and redesigning organizational processes through strategic relationships and rationales. This combined use is tested on a small banking example, which helps understanding how the requirements process may iterate between these two levels of modeling in order to obtain a requirements specification.

In [DUYP98], the combined use of *i** with *KAOS*, Timed Automata [MMT91], and *ALBERT-II* is investigated. *KAOS* is used for reasoning about the system's goals, *ALBERT-II* for formally specifying the system's requirements, and *Timed Automata* for modeling the system's internals. The *i** framework is used for linking the various formal models and for providing a "high level" model in terms of which organizational issues are captured. Organizational goals are identified and analyzed using the complementary techniques of *i** and *KAOS*.

2.5 *KAOS*

The *KAOS* framework [DVF93] [VLDDD91] is a goal-oriented framework for requirements engineering, developed at the University of Louvain; *KAOS* stands for Knowledge Acquisition in automated Specification. It focuses on the formal modeling of functional and non-functional requirements in terms of goals, constraints, assumptions, objects, events, actions, agents, etc. *KAOS* aims at supporting the whole process of requirements elaboration — from the high-level goals that should be achieved by the composite system to the operations, objects, and constraints to be implemented by the software part of it. The framework has a specification language, an elaboration method, and meta-level knowledge used for local guidance during method application. The framework also addresses issues in requirements acquisition, i.e., goal-directed [DVF93] [VLDM95] [DV96], scenario-directed [VLW98], viewpoint-directed strategies [VanL98], and the reuse of requirements specifications [ML97]. The *KAOS* framework

uses a multi-paradigm language. This language has an external semantic net layer for capturing goals, constraints, agents, objects, and actions, together with their links, and an inner formal assertion layer that includes a real-time temporal logic for the specification of goals and constraints.

[DVF93] focuses on the requirements acquisition task where a global model for the specification of the system and its environment is elaborated. The importance of concepts that are currently not supported by most other formal specification languages, such as goals to be achieved, responsibilities of agents to be assigned, alternatives to be negotiated, etc. is discussed. The paper presents elements of a general approach to requirements acquisition developed in the context of the *KAOS* project [VLDDD91] [VLDD91]. The driving forces behind this approach are the reuse of domain knowledge and the application of machine learning technology [VanL91]. Two learning strategies have been adapted to the context of requirements acquisition: *learning-by-instruction*, where the learner conducts the acquisition process by using meta-knowledge about the kind of knowledge to be acquired, and *learning-by-analogy*, where the learner retrieves knowledge about some "similar" system to map it to the system being learned. The overall approach taken in *KAOS* has three components: (1) a conceptual model for acquiring and structuring requirements models, with an associated acquisition language; (2) a set of strategies for elaborating requirements models in this framework; (3) an automated assistant to provide guidance in the acquisition process according to such strategies.

In [VLDM95], the authors show how the *KAOS* goal-directed methodology is used for requirements analysis for a distributed meeting scheduler system. The following issues raised by this case study are addressed: goal identification, the “de-idealization” of unachievable goals, the handling of interfering goals, the impact of early formal reasoning, the merits of early reuse of abstract descriptions and categories, requirements

traceability, the need to link requirements to retractable assumptions, and the potential benefits of hybrid acquisition strategies.

[DV96] proposes an approach for goal refinement and operationalization that aims at providing constructive formal support while hiding the underlying mathematics. The principle is to reuse generic refinement patterns from a library structured according to strengthening/weakening relationships among patterns. They can be used for guiding the refinement process and for pointing out missing elements in a refinement. Some frequent refinement patters are discussed and their use is illustrated through a variety of examples.

In [DDMV98], a tool called GRAIL for supporting the use of the *KAOS* analysis and specification framework is introduced. Its kernel combines a graphical view, a textual view, an abstract syntax view, and an object base view of specification.

2.6 Agent-Oriented Methodologies

The development and spread of agent technologies has brought out significant interest in agent-oriented methodologies and modeling techniques in the last few years. A number of specifically agent-oriented methodologies have been proposed for complex software system development. A valuable survey of agent-oriented methodologies is done in [IG98]. Most of this work deals with the design stage of system development.

Some agent-oriented approaches, such as [Burmeister96], [KGR96], and [KG97], are based on existing object-oriented modeling techniques or methodologies. They extend and adapt the models and define a methodology appropriate for agent-based systems. Some approaches just extend other methodologies and modeling techniques from the area of software and knowledge engineering, or provide formal and compositional modeling languages that are suitable for the verification of system structures and functions.

In [Deloach99], Deloach presents a methodology and system modeling language for multiagents systems. The goal is to integrate the methodology and language into an automated multiagent system synthesis system. The integrated system formally verifies and generates multiagent systems that are correct by construction. The methodology includes domain level design, agent level design, component design, and system design. The language is called agent modeling language AgML. It uses four types of diagrams to define high-level features of multiagent systems.

[WJK99] presents a methodology for agent-oriented analysis and design. The methodology is intended to allow an analyst to go systematically from a statement of requirements to a design that is sufficiently detailed to be implemented directly. In applying the methodology, the analyst can move from abstract to concrete concepts. The methodology encourages a developer to think of building agent-based systems as a process of organizational design. There are two main categories of concepts: abstract and concrete concepts. Abstract concepts are used to model a system as a "society" or "organization". The system is decomposed into a set of roles and these roles can be instantiated by actual individuals. Analysis at this abstract concept level is to develop an understanding of the system and its structure, i.e., an organization as a collection of roles, that stand in certain relations to one another, and that take part in systematic, institutionalized patterns of interactions with other roles. A role model and an interaction model are built through this analysis stage. The role model specifies key roles in the system, and their schema. A role is defined by three attributes: responsibilities, permissions, and protocols. The interaction model captures the dependencies and relationships between the various roles in the organization. The analysis processes can be iterated. The design process is to transform the abstract models derived during the analysis stage into concrete models at a sufficient low level of abstraction that they can be easily implemented. The steps of design process are as follows: create an agent model, develop a service model, and develop an acquaintance model. The agent model identifies the agent types that make up the system. The agent instances will belong to these types.

The service model identifies the main services that will be associated with each agent type. The acquaintance model documents the acquaintances for each agent type. A case study involving a system for business process management is described. A revised version of this methodology called *Gaia* is described in [WJK00].

In [DW00], the authors propose the multiagent systems engineering methodology, a seven-step process that guides a designer in transforming a set of requirements into a successively more concrete sequence of models. By analyzing the system as a set of roles and tasks, a system designer is naturally led to the definition of autonomous, pro-active agents that coordinate their actions to solve the overall system goals.

Other methodologies for analyzing, designing, and building multiagent systems include *DESIRE* presented in [BD95] and *CoMoMAS* presented in [KG97]. There is also work on extending *UML* to deal with agent-based systems [OBP2000].

3 Foundations

Castro *et al.* [CK00] catalogue software development into four stages: *early-phase requirements*, i.e., understand the problems, *late-phase requirements*, i.e., specifying the desired system within its operational environment, *architectural design*, i.e., identifying the system's global architecture, and *detailed design*, i.e., specifying the behavior of each architectural component in detail. In our methodology, we focus on the early-phase requirements analysis using the *i** framework and the late-phase requirements analysis and preliminary design using the *ConGolog* framework.

3.1 The *i** Modeling Framework

The *i** framework [Yu95B] is an agent-oriented modeling framework. It was developed by Eric Yu at the University of Toronto for modeling and analyzing organizations to help support requirements engineering and business process reengineering. The framework is presented in detail in [Yu95B]. It has also been presented in the context of different application domains, including information systems requirements engineering [Yu93], business process reengineering [YM93] [YM94], and software process modeling [YM94A].

In [Yu95B] [Yu97] [YM93] [YM94C] [YML96], the definition of the *i** framework is outlined. "The framework is called *i**, as it attempts to articulate a notion of "distributed intentionality"" [Yu95B]. The framework is used for modeling intentional and strategic actor relationships. It consists of two main components—the *Strategic Dependency* (*SD*) model and the *Strategic Rationale* (*SR*) model [YM94D]. The SD model describes relationships between actors. The SR model describes the possible alternatives that the

actors can choose to accomplish their goals/tasks. This helps the modeler understand the existing processes and propose alternatives towards a new process design that better satisfies the business's objectives.

The notion "intentional actor" is a very important concept in *i**. Intentional characteristics such as goals, beliefs, capabilities, and commitments are assigned to actors in their organization environment. Actors depend on each other to accomplish goals, perform tasks, and supply resources. By depending on others, an actor becomes capable to accomplish goals/tasks that it may be unable to accomplish by itself. Meanwhile an actor becomes "vulnerable" if the other actors, on which it depends, do not participate the dependency. Actors are "strategic" because they are concerned about their "opportunities" and "vulnerabilities" in the process [Yu95B].

The *i** framework has a lot of useful features for the early stages of requirements engineering. It can be used to describe the "why" of a process — the motivations, intents, and rationales behind the activities and entities. It also can be used to support the modeling of social dependencies between agents regarding tasks and goals both functional and non-functional [Yu95B]. All alternative process designs can be explored and informal comparative analyses can be performed. A support tool also has been developed for *i** [YML96].

Next we will describe the concepts in the SD model and SR model offered by the *i** framework in detail. Our presentation is mainly based on [Yu95B].

3.1.1 The Strategic Dependency (SD) Model

"The *Strategic Dependency* (SD) model is a network of dependency relationships among actors" [Yu95B]. The SD model provides an "intentional" description of a process by modeling the dependency relationships among actors. It is used to explore the motivations and intents behind the whole set of activities and information flows in a process. Compared to conventional, non-intentional models, the SD model provides the modeler with a better basis to study the implications of a process because of its richer set of modeling constructs.

Specifically, "A Strategic Dependency (SD) model consists of a set of nodes and links. Each node represents an "actor", and each link between two actors indicates that one actor depends on the other for something in order that the former may attain some goals. We call the depending actor the depender, and the actor who is depended upon the dependee. The object around which the dependency relationship centers is called dependum" [Yu95B].

Before discussing the features of the SD model, let us look at our first example involving a simple meeting scheduling process within which the initiator (a role) and the participant (a role) are the only two actors; the initiator tries to schedule a meeting with the participants by contacting them himself. The process proceeds as follows: the initiator requests the participants to send their available dates, then he proposes a meeting date to the participants, and then the participants agree or reject the proposed meeting date. A SD model for this example appears in Figure 3.1 (adapted from the SD model for a more complicated meeting scheduling process in [Yu97]).

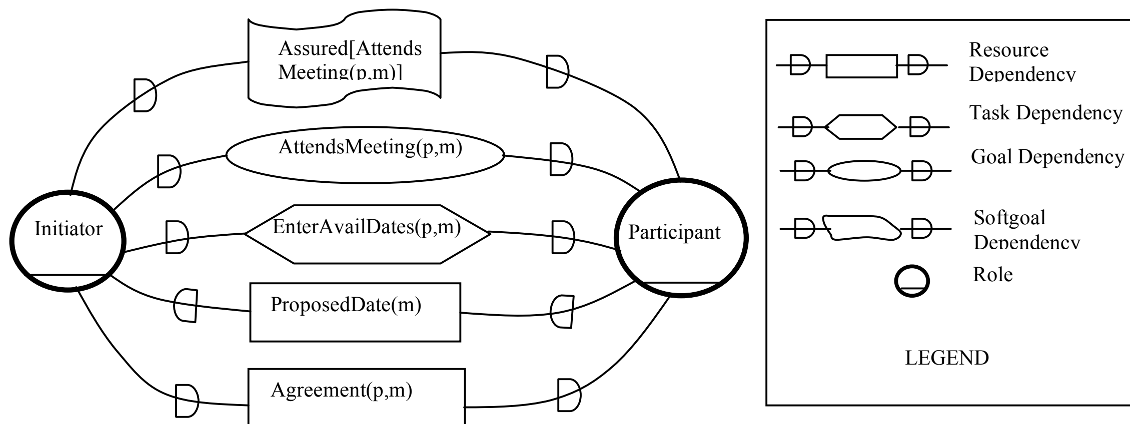


Figure 3.1 The SD model for a simple meeting scheduling process

In Figure 3.1, the actor nodes are roles `Initiator` and `Participant`. There are five links between these two actor nodes. For example, the initiator depends on the participant's attendance at the meeting. This is modeled as a dependency `AttendsMeeting(p,m)`, where `p` refers to the participant, and `m` refers to the meeting. The initiator is the depender, the participant is the dependee, and the goal `AttendsMeeting(p,m)` is the dependum.

- **Actors, Agents, Roles, and Positions**

"An *Actor* is an active entity that carries out actions to achieve goals by exercising its know-how" [Yu95B]. In Figure 3.1, the initiator is responsible for organizing a meeting by communicating with the participants. The participants are responsible for agreeing or rejecting a proposed meeting date and attending the meeting once they agree.

Furthermore, an actor can be specialized into notions of agents, roles, or positions [YM94A]. Here let us look at our second example of a mail-order business process first. There are three actors that participate in this process: a customer, who makes orders, a mail-order company, who processes the orders, and the bank, who is responsible for

processing banking transactions. The mail-order company might have two sub-parts: the stock clerk and the office clerk. The stock clerk could play both the shipping processor and stock informant roles.

"An *agent* is an actor with concrete, physical manifestations, such as a human individual" [YM94A]. It has dependencies that apply no matter what roles he/she/it happens to be playing. For example, John might be the real office clerk, an agent who also maintains the real stock of items; others depend on him to maintain the real stock. "A *role* is an abstract actor. Dependencies are associated with a role when these dependencies apply regardless of who plays the role" [YM94A]. For example, an order processor role might depend on a stock informant role to provide information about the stock for an ordered item; the order processor doesn't care who is playing this stock informant role. "A *position* is intermediate in abstraction between a role and an agent. It is a set of roles typically assigned jointly to one agent" [YM94A]. For example, the bank actor can be specialized as a position that covers the roles of providing customer's account information, debiting money from a customer's account, and crediting the money into the company's account. Roles, positions and agents can also have subparts. An agent can occupy several positions, and play different roles and also can be part of another agent. A position can cover several roles and be a part of another position.

By classifying actors into three classes, the SD model provides a way to separately identify those dependencies that are associated with a role/position/agent, as opposed to those that are associated directly with a concrete actor. Also modeling and analysis would be more efficient and accurate when the distinctions among the various specialized actors — agents, roles, and positions, are introduced.

- **Dependencies**

"A *dependency* is intentional if the dependum is somehow related to some goals or desires of the depender" [Yu95B]. In Figure 3.1, the dependency of the initiator's depending on the participants' attendance of the meeting is intentional because if an important participant doesn't show up the meeting, the initiator might fail to achieve his goal of organizing the meeting. "By depending on another actor for a *dependum*, an actor (the *depender*) is *able* to achieve goals that it was not able to do without the dependency, or not as easily or as well". This brings *opportunities* for selecting the process of the system. One actor might want to accomplish some goal/task by having another actor (the *dependee*) do it because he gets some benefit by using the other actor's efforts. At the same time, the depender becomes *vulnerable* because if the dependee fails to supply the dependum, the depender might not be able to accomplish its goals/tasks any more.

For example, in Figure 3.1, the initiator's dependency on the participants' agreement to a meeting date is related to his goal of arranging a meeting with all participants. Without the agreement to the meeting date from all participants, the initiator cannot really schedule the meeting. Also if a participant cannot provide his agreement to a proposed meeting date (e.g., he is on vacation), then the initiator is vulnerable and cannot proceed with the meeting scheduling. But in some cases, such as when a manager wants to arrange a meeting with his subordinates, the manager (initiator) might be able to command all participants to agree to the meeting date without depending on them. This brings out another opportunity for the process of scheduling a meeting.

- **Dependency Types**

There are four types of dependencies in *i**: *goal*-, *task*-, *resource*-, and *softgoal*-dependencies [Yu95B] [Yu97]. "In a *goal-dependency*, the depender depends on the dependee to bring out a certain state in the world" [Yu95B]. The dependee can decide how he will achieve the goal. Meanwhile the depender is able to assume that the

condition or state of the world will hold through a goal dependency, but becomes vulnerable since the dependee may fail to bring about the condition. For example, in Figure 3.1, the initiator depends on the participant to attend the meeting. This is be modeled as a goal dependency `AttendsMeeting(p,m)`; `Initiator` is the depender, `Participant` is the dependee, and the goal `AttendsMeeting(p,m)` is the dependum, which is put inside an oval. The initiator becomes vulnerable when a participant cannot achieve the goal `AttendsMeeting(p,m)`. For example, if a participant is on vacation, then the initiator might have to cancel the meeting because the attendance of this participant is very important for the meeting.

"In a *task-dependency*, the depender depends on the dependee to carry out an activity" [Yu95B]. The task-dependency specifies how the depender depends on the dependee to complete a certain task through some activities. The dependum is how the task is to be performed, but not why. The depender is vulnerable since the dependee may fail to perform the task. The dependee might be not able to perform the task or might decide not to perform the task even when it is able to, e.g., if it decides that there are more important things for it to do due to other commitments. For example, in Figure 3.1, the initiator depends on the participant to send its available dates for the meeting. This is modeled as a task dependency `EnterAvailDate(p,m)`; `Initiator` is the depender, `Participant` is the dependee, and the task `EnterAvailDate(p,m)` is the dependum, which is put in a diamond. The initiator is vulnerable if the participant refuses to enter his available dates because he might dislike attending this kind of meeting. The initiator will be hurt in this case and may have to cancel the meeting.

"In a *resource-dependency*, one actor (the depender) depends on the other (the dependee) for the availability of an entity (physical or informational)" [Yu95B]. By having this dependency, the depender is able to use the resource provided by the dependee and meanwhile becomes vulnerable if the resource is not provided. For example, in Figure

3.1, the participant depends on the initiator to propose a meeting date. This is modeled as a resource dependency `ProposedDate(m)`, where `Participant` is the depender, `Initiator` is the dependee, and the resource `ProposedDate(m)` is the dependum, which is put inside a rectangle. The participant is vulnerable if the initiator cannot provide a proposed meeting date suitable for his schedule; then he cannot attend the meeting.

"In a *softgoal dependency*, a depender depends on the dependee to perform some task that meets a softgoal" [Yu95B]. Softgoal dependencies are related to the notion of non-functional requirements (or quality requirements). They involve goals that can be satisfied to various degrees, and needs to be optimized. By identifying alternatives and having the depender choose an alternative, the goals could be clarified during the process of trying to achieve them. Usually, the dependee provides the alternatives, but the decision of choosing an alternative is made by the depender. Through this dependency, the depender gains the opportunity of having the goal condition satisfied, but becomes vulnerable in case the dependee fails to have the condition satisfied. These types of relationships cannot be expressed or distinguished formally in the non-intentional models that are used in most other requirements modeling frameworks.

In Figure 3.1, the initiator depends on the participant to assure him of his attendance at the meeting. How the participant's attendance should be assured is decided by the initiator. An email notification or a phone call might be enough. This is modeled as a softgoal dependency `Assured[AttendsMeeting(p,m)]`. `Initiator` is the depender, `Participant` is the dependee, and the softgoal `Assured[AttendsMeeting(p,m)]` is the dependum, which is put inside a flag-shape. The initiator might have to cancel the meeting if it is not assured of an important participant's attendance.

These four types of dependencies also describe how either side of the dependency makes decisions to having the dependency supplied, thus indicating who will take care of problems when they arise. For a goal dependency, the dependee makes decisions on how to achieve the goal. For a task dependency, the depender makes decisions that specify how to perform the task. For a resource dependency, because a resource is the result of some deliberation-action process, it is assumed that there are no open issues or decisions to be made. For a softgoal dependency, the depender makes the final decision, but lets the dependee determine what the alternatives are. Note that there is similarity between these notions and the notion of "*design by contract*" [Meyer91] used in software engineering.

- **Dependency Strength**

The SD model also classifies the degree of strength of dependencies. A stronger dependency means that the depender is more vulnerable and the dependee will make a greater effort in trying to provide the dependum. The depender is likely to take actions to decrease vulnerability in such case. There are three degrees of strength of dependencies in the SD model: *open* (uncommitted), *committed*, and *critical* [YM94D]. "In an *open dependency*, a depender would like to have the dependum goal achieved, task performed, or resource available, so that it could be used in some course of actions" [Yu95B]. If the dependum is not supplied, the depender's goals would be affected somehow but the consequences would not be serious. "In a *committed dependency*, the depender has goals which would be significantly affected — in that some planned course of action would fail — if the dependum is not achieved" [Yu95B]. In a case where a series of actions will be performed and cannot be reversed, the depender might have to investigate this case significantly. Because of its vulnerability, a depender would be concerned about the *viability* of the committed dependency, i.e., whether there is a viable way to supply this dependency. Meanwhile the dependee will try its best to supply the dependum, e.g., by making sure that is own dependencies are *viable*. "In a *critical dependency*, the depender has goals which would be serious affected — in that all known courses of action would

fail — if the dependum is not achieved" [Yu95B]. In this case, the depender would be concerned not only about the viability of the immediate dependency, but also about the viability of the dependee's dependencies, and the dependee's dependee's dependencies and so on [Yu95B] [Yu97].

- **Knowledge Management**

The SD model has been embedded into a formal conceptual modeling framework Telos that allows for the effective usage and management of the potentially large amount of knowledge involved when modeling real work processes [MYBJK91].

- **Formal Characterization of the SD Model**

The SD model has been presented informally using descriptive text, a graphical notation, and illustrative examples. In [Yu95B], Yu has developed a somewhat formal characterization of the SD concepts in terms of agent modeling concepts developed in AI. This involves the following notions: (1) *Routines* — "A interconnected collection of process elements serving some purpose for an agent is called *routine*" . A routine is the primary vehicle through which an agent can accomplish what it wants. The internal characterization of an agent centers on the routines held by the agent, and the elements that make up the routine. (2) *Ability* — when an agent has a routine that can achieve a certain goal, then it has an *ability* to achieve the goal. (3) *Workability* — *workability* means that an agent believes some routine would work, even though the routine is incompletely specified or known; during strategic reasoning, an agent is content to reduce a solution to a level at which all components are workable. (4) *Commitments* — *commitments* means that if an agent is able to achieve a goal and committed to doing so, then the goal is *workabl*e for the agent. Commitments thus provide an abstraction that allows workability to be judged without having to know about the routines used to achieve the goal. (5) *Vulnerability* — the characterization of *vulnerability* is based on the

extent to which the *workability* of a goal would affect the *workability* of the *routine* in which the goal is supposed to serve.

- **Analysis Methods Based on the SD Model**

The SD model can be analyzed based on the following features [Yu95B] [Yu97]: (1) *Opportunity and Vulnerability*: The chains of dependencies in a SD model can help us to explore the expanded possibilities that are accessible to an actor. This brings out the opportunities for the process. An actor could also use the dependency network to determine how it might be adversely by these dependencies. This brings out the vulnerability of the actor. By enlisting the dependees, a depender seeks opportunities of a process and can achieve what would otherwise be unachievable. By matching the dependencies from dependers and those from dependees, one can explore opportunities that are available to these actors. Classification and generalization hierarchies facilitate the matching of dependums [Yu95B]. (2) *Commitment*, *Assuranc*e, *Insurance*: These mechanisms contribute to revising a dependency and to lessening vulnerability. A *commitment* is implementable if there is some way for the depender to cause some goals of the dependee to fail, i.e., if there is a reciprocal dependency. The dependee has to try to supply the dependum for the depender to avoid its goals not being satisfied. *Assurance* means that there is some evidence that the dependee will deliver the dependum, apart from the dependee's claim. *Insurance* mechanisms reduce the vulnerability of a depender by reducing the degree of dependency on a particular dependee. By having more that one dependee for the same dependum, a depender can increase the chances of a dependum being achieved.

The SD model provides a formal representation of the nodes and links in a dependency network, thus allowing for analysis based on network topology, i.e., chain analysis, loop analysis, and node analysis [Yu95B]. It helps the modeler gain a deeper understanding of a process and identify what is the stake, for whom, and what impacts are likely if a

dependency fails. But it just provides a description about why a process is structured in a certain way and doesn't explicitly model the depender's internal goals and desires. It does not really support the process of suggesting, exploring, and evaluating alternative solutions [Yu97]. The Strategic Rationale (SR) model of *i** addresses these issues.

3.1.2 The Strategic Rationale (SR) Model

"In the *Strategic Rationale* (SR) model, the rationales behind process configurations can be explicitly described, in terms of process elements and relationships among them" [Yu95B]. The SD model provides one level of abstraction for describing organizational environments and their embedded systems. It shows external (but nevertheless intentional) relationships among actors, while hiding the intentional constructs within each actor. The SD model can be useful for understanding organizational and systems configurations as they exist, or as proposed new configurations. During early phase RE, however, one would also like to have more explicit representation and reasoning about actors' interests, and how these interests might be addressed or impacted by different system-and-environment configurations, existing or proposed. The SR model is proposed to provide a more detailed level of modeling by looking "inside" actors to model internal intentional relationships. Intentional elements (tasks, goals, resources, and softgoals) appear in the SR model not only as external dependencies, but also as internal elements linked by means-ends and task-decompositions relationships. The SR model elaborates on the relationships between actors as described in the SD model.

The SR model provides a way of modeling stakeholder interests, and how they might be met, and the stakeholder's evaluation of various alternatives with respect to their interests. Task-decomposition links provide a hierarchical description of intentional elements that make up a *routine*. The means-ends links in the SR model provides understanding about why an actor would engage in some tasks, pursue a goal, need a resource, or want a softgoal. From the softgoals, one can tell why one alternative maybe

chosen over others. For example, in meeting scheduling, availability information is collected so as to minimize the number of rounds of interaction with the participants.

[MYCY99] [YM94] describe goal-oriented analysis using the *i** framework and the Strategic Rationale (SR) model of *i**. [YM94A] [YM97] discuss the applications of *i** framework in software processes modeling. [DUYP98] [Yu95A] [YDDM95] discuss different approaches to modeling organizational work and also give examples of how *i** may be used in combination with other modeling or specification languages in software development. [YM94D] presents some formal axioms for the *i** model.

The SR model is a graph consisting of four types of nodes, *goal*, *task*, *resource*, and *softgoal* nodes, and two types of links, *means-ends links* and *task-decomposition* links. "*Task*-decomposition links provide a hierarchical description of elements that compose a task" [Yu97]. Means-ends links specify how a goal may be achieved. They provide information about why an actor would perform a task, pursue a goal, need a resource, or want a softgoal. From the softgoals, the modeler can tell why one alternative may be chosen over others. For example, the SR model of Figure 3.2 elaborates on the SD model of Figure 3.1; we explain its basic elements and features next.

- **Nodes**

There are four types of *nodes*, based on the same types as for dependum types in the SD model — *goal*, *task*, *resource*, and *softgoal* nodes.

"A *goal* is a condition or a state of affairs in the world that the actor would like to achieve" [Yu95B]. How the goal is to be achieved is not specified, allowing alternatives to be considered. For example, in Figure 3.2, `MeetingBeScheduled` is a goal node that represents the goal of the meeting initiator that a meeting be scheduled.

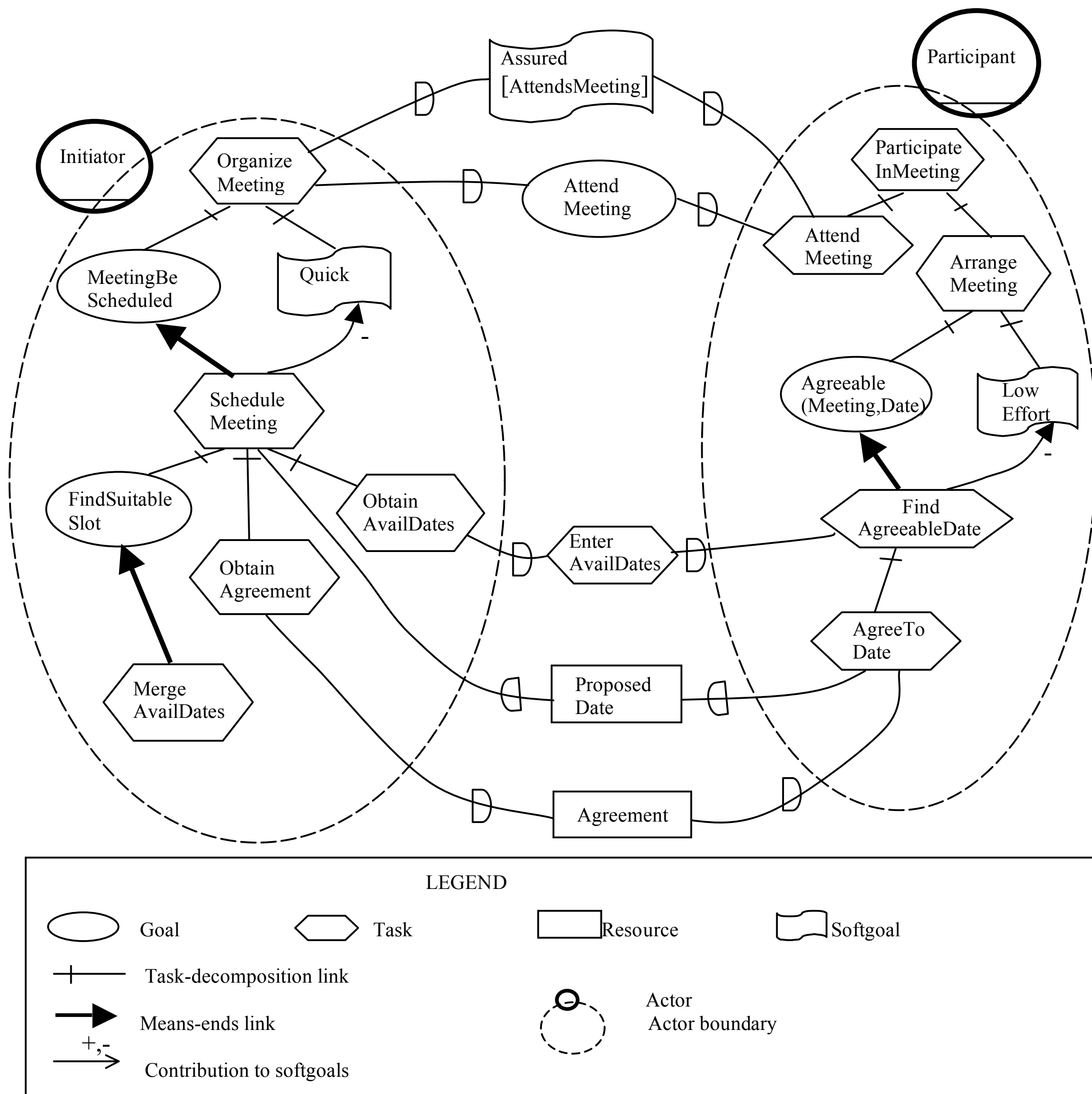


Figure 3.2 The SR model for a simple meeting scheduling process

"A *task* specifies a particular way of doing something" [Yu95B]. When a task is specified as a sub-component of a super task, this restricts the super task to include that particular course of action. For example, in Figure 3.2, `ScheduleMeeting` is a task node that represents the task of the meeting initiator to schedule a meeting.

"A *resource* is an entity (physical or informational) that is not considered problematic by actor" [Yu95B]. The main concern is whether it is available and from whom if it is an external dependency. For example, in Figure 3.2, `ProposedDate` is a task node that represents information about the proposed meeting date.

"A *softgoal* is condition in the world which the actor would like to achieve, but unlike in the concept of (hard-) goal, the criteria for the condition being achieved is not sharply defined a priori, and is subject to interpretation" [Yu95B]. If a softgoal is a subcomponent in a task decomposition, it serves as a quality goal for that task, and thus guides or restricts the selection among alternatives in further decomposition of that task. For example, in Figure 3.2, `Quick` is a softgoal node that represents the condition that the initiator wants the scheduling of the meetings to be done as quickly as possible.

- **Links**

There are two main classes of *links*: *means-ends links* and *task decomposition links*. There are several types of means-ends links. A means-ends link represents a relationship between an end — a goal to be achieved, a task to be accomplished, a resource to be produced, or a softgoal to be satisfied — and a means for attaining the end. The means is usually in the form of a task, since the notion of task represents how to do something. In the graphical notation, the arrowhead points from the means to the end. For example, in Figure 3.2, a means-ends link involves in the task node `MergeAvailDates` which is the means and the goal node `FindSuitableSlot` which is the end; a thick arrow connects the means to the end. This means-ends link represents the fact that the meeting initiator finds a suitable date slot by merging all available dates.

There are two common means-ends link types involving softgoals. A softgoal-task link has a softgoal as the end, and a task as the means. Softgoal-task links are shown as curved arrows in the graphical notation. Links involving softgoals require an extra

attribute to indicate the type of contribution — positive or negative, enough or not enough. The sofgoal-softgoal link permits the development of a means-ends hierarchy of softgoals, until eventually some softgoals are addressed by linking to tasks. For example, in Figure 3.2, inside the participant role, a softgoals-task link involves the end — a softgoal `LowEffort` and the means — a task `FindAgreeableDate`; a curved arrow starts from the means to the end and the contribution attribute is represented by a negative sign "-". This means that the participant's finding agreeable dates himself increases his efforts.

At an actor boundary, an incoming dependency link is also an implicit means-ends link, with the dependum being the "end". Other means-ends link types are possible as a result of combining other element types for the means and for the end.

A task node is linked to its sub-components nodes by task decomposition links. There are four types of task decomposition links — subgoal, subtasks, resource, and softgoal; each corresponds to the four types of sub-component nodes. These links also can connect up with dependency links in the SD model, when the reasoning goes outside an actor's boundary. An outgoing dependency link is usually also a task-decomposition link, the dependum being a sub-element of the task decomposed into. For example, in Figure 3.2, the task `ScheduleMeeting` can be decomposed into four sub-components: a subtask of obtaining available dates `ObtainAvailDates`, a subgoal of finding suitable date slot `FindSuitableDateSlot`, a resource dependum which provides a proposed date for a meeting, and a subtask of obtaining agreement from participants `ObtainAgreement`. Every sub-component is connected to the decomposed task by a line. The task `ScheduleMeeting` with its task-decomposition represents the fact that in order to schedule a meeting, the initiator first obtains available dates from all participants, then finds the suitable date slot for the meeting, then proposes a meeting date to all participants, and then get an agreement on the proposed date (the ordering is

not represented in the SR diagram). The task-decompositions describe how a process operates and what the rationales behind it are.

Each task-decomposition link can be *open*, *committed*, or *critical*. *Committed* means that the agent believes that the associated routine will fail if this element fails. *Open* means that the routine would be affected, but would not necessarily fail. If the link is an outgoing dependency link, the link can also be critical. *Critical* means that the agent believes there is no other way to succeed. There may also be constraints among the components of a task, such as temporal relationships that are not shown in the graphical notation, but appear in the formal language notation Telos [MYBJK91].

- **Routines and Rules**

"A *routine* is a subgraph in the SR graph with a single link to a "means" node from each "end" node" [Yu95B]. A routine therefore represents one particular series of actions among the multiple alternatives presented at each "OR" node. The notion of a routine is used to refer to one process and its rationales. Routines typically have connections to other actors through dependency links in the SD model. The means-ends links in a SR model are shown as embedded in particular context. They are rationales. However these links can be seen as application of more generic relationship which says that whenever you have some element as an end, you can use some other element as a means to that end. Yu calls the generic principle a *rule* [Yu95B]. A rule consists of an applicability condition, a means, and an end. A means-ends link is an application of a rule in a context in which the agent believes the applicability condition to hold.

- **Analysis Methods for the *i** SR Model**

The SR model provides a powerful set of concepts for modeling and analyzing processes. First, the SR model helps in understanding the "*whys*" as well as the "*how*" behind processes. Conventional process models view a process as activities with flows between

them and provide a non-intentional view of a process. They leave out the motivations and the rationales behind the process and do not consider alternatives. Using means-ends links, the SR model can provide a view of process that is goal-oriented (i.e., intentional). By being agent-oriented, the SR model investigates where intentional processes are coming from and proceed towards. One can ask a how question, i.e., how can this goal (node) be achieved, seeking a means to the desired end. One also can ask a why question, seeking to discover the end for which the current node is the means. By being able to express "whys" and "how", the model gives a deeper understanding based on means-ends reasoning. One can see that there are alternatives, and that actors have choice. One can thus better forecasts the implications of change.

Second, the SR model also provides task decomposition and composition. Many modeling frameworks have incorporated the composition/decomposition features so that description of processes can be hierarchical. But the SR model allows task decomposition to include different types of components, not just a decomposition of activities into sub-activities. In a non-intentional context, activities are merely carried out. There is no notion of success or failure, or goal-achievement with different means. Under SR, one assumes an intentional, strategic modeling context. Thus, one can classify task components by the degree of openness or uncertainty. A goal means that one expects there can be different ways of achieving it – alternatives come out. A task means there are constraints on how to perform it. Quality concepts that constrain the selection among alternatives are represented by softgoals. At the bottom of the means-ends hierarchy in the SR model, task elements can still be goals or tasks. Process execution would require further problem solving at run-time.

Third, the SR model supports process analysis and design activities. In analyzing a process, one can examine the network of links involved. Moreover, one can do analysis that is of strategic concern at the actor level: whether an actor knows how to do

something, whether it will work, how well it will work, and why the agent believes it will work. In design, one can systematically explore alternatives, by seeking means to ends. Several additional concepts that enlarge the analytical power of the SR model are: (1) *Ability* — does the actor have a process for accomplishing the goal? (2) *Workability* — is the process going to work? (3) *Viability* — how well will it work? (4) *Believability* — what evidence is there to confirm or disconfirm that it will work? Let us sketch how these notions are used in analysis (the examples used here are from [Yu97]). In a meeting scheduling process, when a meeting initiator has a routine to organize a meeting, he is said to be *able* to organize a meeting. One that is able to organize one type of meeting is not necessarily able to organize another type of meeting. Given a *routine*, we can analyze it for *workability* and *viability*. Organizing meeting is *workable* if there is workable routine for doing so. To determine workability, we have to inspect the workability of each element. For example, can the initiator obtain availability information from participants, find agreeable dates, and can obtain an agreement from participants. If workability of an element cannot be confirmed by the actor, the element needs to be further elaborated. If the subgoal `FindSuitableSlot` is not primitively workable, it needs to be elaborated in terms of a particular way for achieving it. For example, one possible means for achieving it is to do an intersection of the availability information from all participants. If this task is confirmed to be workable by the initiator, then the `FindSuitableSlot` goal node would be workable. A task can be workable by way of external dependencies on others. The workability of `ObtainAvailDates` and `ObtainAgreement` are evaluated in terms of the workability of the commitments of meeting participants to provide availability information and agreement.

A *routine* that is *workable* is not necessarily *viable*. Although computing intersection of time slots by hand is possible, it is slow and error-prone. Potentially good slots may be missed. When softgoals are not satisfied, the routine is not viable. Note that a routine which is not viable from one actor's view may be viable from another actor's view. For

example, a routine where the initiator does all the work for scheduling meetings may be viable for participants, if the resulting meeting dates are convenient, and the meeting arrangement efforts do not involve too many interruptions; but it may not be viable for the initiator. The assessment of workability and viability is based on many *beliefs* and *assumptions*; these can be provided as *justifications* for the *assessment*. The *believability* of the rationale network can be analyzed by checking the network of justifications for the beliefs. For example, the argument that finding agreeable dates by merging available dates is workable may be justified with the assertion that the initiator has been doing it this way for years and it works. The evaluation of such goal graphs can be supported by graph propagation algorithms following a qualitative reasoning framework [DuBois95] [Yu95B].

The SR model allows us to raise ability, workability, and viability as issues that need to be addressed, and by using means-ends reasoning, these issues can be addressed systematically, resulting in new configurations that can be evaluated and compared. Means-ends rules that encode know-how in the domain can be used to suggest possible alternatives. Issues for stakeholders that are cross-impacted may be discovered during this process, and can be raised so that tradeoffs can be made. Issues are settled when they are deemed to be adequately addressed by stakeholders. Once settled, one can then proceed from the descriptive model of the *i** framework to a prescriptive model that would serve as the requirements specification for systems development. Believability can also be raised an issue, and then assumptions would have to be justified.

3.1.3 Discussion

*i** is designed for early-phase requirements engineering and focuses on capturing the rationales from various choices made for the system. Most of approaches discussed in chapter 2 are either for late-phase requirements to produce a precise, complete, and unambiguous requirements specification, such as *ALBERT-II*, *KAOS*, etc., or for system

design to specify the functional components at a detailed level, such as *Z*-notation, *UML*, *BON*, etc. However, *i** is a graphical notion that is somewhat informal. It has an axiomatic semantics, but it is somewhat abstract, being based on notations such as "having a routine". It is also limited in its ability to represent complex processes and does not support simulation and verification. By combining it with *ConGolog*, we can address these limitations.

3.2 The *ConGolog* Modeling Framework

3.2.1 Introduction

ConGolog [DLL97] [DLL00] [LKMY99] is an *agent-oriented* process-modeling framework that is very expressive and fully formal. It is well adapted to the late-requirements-engineering and early-design stages of system development, when detailed alternative process designs have been specified and need to be compared. A process simulation tool has already been developed for process model validation, and verification methods and tools are being developed. The language has been used to model various multiagent applications [DLL00] [LKMY99] [LLRU97] [LTJ98] [SLL97] (e.g., meeting scheduling, feature interaction resolution). It also has been used as an implementation language for agent systems [Tam98] [LKMY99] [LLRU97] [LTJ98] [SLL97].

ConGolog is based on a logical formalism, i.e., the *situation calculus*. The *situation calculus* [MH79] is a first-order language for representing dynamically changing worlds. The version of the situation calculus used in *ConGolog* is described in [DLL00] [LKMY99] [Reiter91]. The *ConGolog* framework can be used to model complex processes involving loops, nondeterminism, concurrency, and multiple-agents. It is an extension of *Golog* [LRLLS97]. In *ConGolog*, the effects of actions in a dynamic domain are specified in a logical framework; this supports modeling even in the absence of complete information. The behavior of agents in the domain is specified in a concurrent

process language whose semantics is defined in the same logical framework. Because of its logical foundations, *ConGolog* can accommodate incompletely specified models, either in the sense that the initial state of the system is not completely specified, or in the sense that the processes involved are nondeterministic and may evolve in any number of ways. These features are especially useful when one models business processes and open-ended real-world situations.

3.2.2 Modeling a Domain

In *ConGolog*, an application domain is modeled logically so as to support reasoning about the specification. A *ConGolog* model of a domain includes two components. The first component is a specification of the *domain dynamics*, i.e., how to model the state, what is the initial state of the domain, what actions can be performed, when the actions can be performed, and what their effects are. This component is specified in purely declarative way, in the situation calculus or in a high-level language called the *Golog* Domain Language (GDL). The full syntax and semantics of GDL are defined in [LRLLS97]. In this thesis, we use the encoding of the situation calculus used by the *Prolog* implementation of *ConGolog* to specify the domain dynamics rather than GDL. The second component of a *ConGolog* domain model is a specification of the *processes* that are unfolding in the domain, i.e., the behavior of the agents involved in the domain. To support the modeling of domains involving complex processes, this component is specified procedurally in the *ConGolog* process description language. In this thesis, we use the notation of the *Prolog* implementation of this process language to specify process. Both components have formal semantics defined in the situation calculus. Various mechanisms for reasoning about properties of a domain have been implemented using this situation calculus semantics.

3.2.3 Modeling Domain Dynamics

The first component of a *ConGolog* model is a specification of the dynamics of the domain and of what is known about its initial state. The situation calculus can be used to specify this component.

3.2.3.1 The Situation Calculus Language

In the situation calculus, we imagine the world as starting out in a particular initial situation or state, and evolving into various other possible situations through the performance of actions by various agents. We use the simple meeting scheduling process as our example to introduce the situation calculus and the *ConGolog* framework. A dynamic domain is modeled in terms of the following entities:

- *Agents*: The agents involve in the modeled system. For example, in the simple meeting scheduling process, the initiator and the participants are the agents who will be involved in the process of scheduling a meeting.

- *Primitive Actions*: In the situation calculus, all changes to the world are the results of named primitive actions that are performed by some agent in the system. Actions are denoted by function symbols and are also first-order terms that take the agent and possibly other parameters. For example, in our simple meeting scheduling process, the action `SendAvailDates(Participant,Ini,Meeting,Availdates)` represents `Participant` sending his available dates `Availdates` to the initiator `Ini` regrading `Meeting`. The preconditions and effects of primitive actions are specified by axioms. We discuss this later.

- *Exogenous Actions*: In order to complete the simulation of the system of interest, some actions may have to be performed by agents outside the system who are not modeled in detail; those actions are called exogenous actions. For example, the action

occupyDateFromParticipant(Participant,Date) is an exogenous action that represents that someone outside the meeting scheduling system books the given date on the participant's time schedule. Exogenous actions like ordinary primitives must be formalized by axioms. In the simulation tool, exogenous actions can be randomly generated.

- *Situations*: A possible world history, which is a sequence of actions, is represented by a first order term called a situation. The constant "s0" is used to denote the initial situation in which no action has been executed. There is a distinguished binary function symbol do and a term do(a,s) denotes the situation which results from action a being performed in situation s. For example:

```
do(SendAvailDates(Participant,Initiator,Meeting,AvailDates),S)
```

denotes the situation after Participant has sent his available dates AvailDates to Initiator regarding Meeting in situation S.

The sequence of actions in a history and the order in which they occur are obtained from a situation term by reading off its action instances from right to left. For example, "$do(a_3,do(a_2,do(a_1,s0)))$" represents the history where a_1, a_2, and then a_3 are performed starting in the initial situation "s0". For example, in the meeting scheduling domain, we might have the situation term:

```
do(obtainAvailDates(initiator1,yves),
do(obtainAvailDates(initiator1,jeff),
do(scheduleMeeting(initiator1,[jeff,yves],[11,12]),S0)))
```

This denotes the situation where first initiator1 commands the scheduling of a meeting with jeff and yves on dates Feb. 11 or Feb. 12; then initiator1 makes

a request to obtain his available dates from `jeff`, and then requests to obtain his available dates from `yves`.

- *Fluents*: Situations are described in terms of fluents. In the situation calculus, a relation of interest to the modeler whose truth-value changes from situation to situation is called a predicate fluent and is denoted by a predicate symbol taking a situation term as its last argument. This makes the dependence of the value of the fluent on the situation explicit. There are three types of fluents: predicate fluents, functional fluents, and defined fluents. For example, we might write the following to assert that a fluent holds in a situation:

```
holds(sentAvailDates(Participant,Initiator,Meeting,AvailDates),
                                                          do(A,S))
```

This says that after the action `A` has been performed in situation `S`, `Participant` has sent his `AvailDates` to `Initiator` regarding `Meeting`; the fluent `sentAvailDates(Participant,Initiator,Meeting,AvailDates)` has become true.

Similarly, functions whose value varies from situations to situations are called “functional fluents". For example,

```
holds(val(participantTimeSchedule(Participant),DateList),S)
```

states that `Participant`’s time schedule, the value of the functional fluent `participantTimeSchedule(Participant)`, is `DateList` in situation `S`. The value of `participantTimeSchedule(Participant)` will change when some action occupies a date from `Participant`’s time schedule.

One can also introduce defined fluents, which are defined in terms of the primitive fluents, We don't need to specify how these derived fluents are affected by actions since this can be deduced from their definitions.

3.2.3.2 Domain Dynamics Specification in the Situation Calculus

The dynamics of a domain are specified using three kinds of axioms that specify when actions can be performed, what the effects of performing the actions are, and what the initial state of the system is:

- *Action Precondition Axioms*: These axioms state the conditions under which a primitive action can be performed. They use the predicate `poss(a,s)`, which means that action `a` is possible in situation `S`. For example, in the simple meeting scheduling process, we have the following precondition axiom:

```
poss(acceptAgreement(Participant,Initiator,Date,Meeting),S)
≡ DateIsFree(Date,Participant,S)
```

In the notation of the *Prolog* implementation we write:

```
poss(acceptAgreement(Participant,Initiator,Date,Meeting),S)
:- holds(DateIsFree(Date,Participant),S)
```

This means that the action where `Participant` agrees to meet on `Date`, `acceptAgreement(Participant,Initiator,Date,Meeting)`, is possible in situation `S` if and only if `Date` is free for `Participant` in situation `S`. The modeler must provide a precondition axiom for each primitive action. From now on, we use the *Prolog* implementation's notation.

- *Successor State Axioms:* These axioms specify how fluents are affected by the actions in the domain. For example, in the simple meeting scheduling domain, we might have the following successor state axiom:

```
holds(sentAvailDates(Participant,Initiator,M,AvailDates),do(A,S))
:- A = sendAvailDates(Participant,Initiator,M,AvailDates),
   holds(sentAvailDates(Participant,Initiator,M,AvailDates),S)
```

  This means that `Participant` has sent his available dates to `Initiator` regarding the meeting `M` in the situation that results from action `A` being performed in situation `S` if and only if action `A` is that of `Participant`'s sending his dates to `Initiator` or if `Participant` had already sent his available dates to `Initiator` regarding the meeting `M` in situation `S`. Successor state axioms can be generated automatically from a specification of the effects of the actions if we assume that they specify all of the ways that the value of the fluent may change. A tool that does this is described in [LKMY99]. As we see below, successor state axioms provide a solution to the frame problem [MH79] [Reiter91]. A domain specification must include a successor state axiom for each primitive fluent.

- *Initial Situation Axioms:* These axioms specify the initial state of the modeled system. The process of the system starts from the initial situation specified by these axioms. For example, in our meeting scheduling process, we have an initial situation axiom, `holds(val(participantTimeSchedule(yves),[11,12]),s0)`, meaning that `yves` has meetings on Feb. 11th and 12th in the initial state.

- *Dealing with the frame problem*: The sort of logic-based framework we have described allows incomplete information about a dynamic domain to be specified. But this creates difficulties in reasoning about action and change. Effect axioms state what must change when an action is performed, but do not specify what aspects of the domain remain unchanged. One way to address this is to add frame axioms that

specify when fluents remain unchanged by actions. For example, the initiator's obtaining available dates does not cause an agreement to be accepted:

```
holds(not(agreementAccepted(Participant,Initiator,Date,Meeting)
                                                              ,S)
∧ A=obtainAvaildates(Initiator,Participant,Meeting)
⊃ holds(not(agreementAccepted(Participant,Initiator,Date,Meeting)
                                                         ,do(A,S))
```

The frame problem arises because the number of these frame axioms is very large, in general, of the order of 2×A×F, where A is the number of actions and F is the number of fluents. This complicates the task of axiomatizing a domain and can make automated reasoning extremely inefficient. To deal with the frame problem, *ConGolog* uses a solution proposed in [Reiter91]. The basic idea behind this is to collect all effect axioms about a given fluent and make a completeness assumption, i.e., assume that they specify all of the ways that the value of the fluent may change. A syntactic transformation can then be applied to obtain a successor state axiom for the fluent.

3.2.3.3 Summary

The *ConGolog* Framework has been extended in [SLL97] [SL01] to support the modeling of agent mental states (knowledge and goals) and the effects of communication and perception acts on them. It would be interesting to use these extensions for RE, but we leave this for future work.

3.2.4 Modeling Domain Processes in *ConGolog*

As mentioned earlier, a *ConGolog* domain model includes a second component that describes the processes unfolding in the domain. The process of a system is specified procedurally in the *ConGolog* framework. The main procedure will specify the whole system's behavior. Every agent also has a corresponding *ConGolog* procedure to

represent its behavior in the system. The *ConGolog* framework includes constructs for conditionals, loops, nondeterminism, concurrency, etc. Communication between agents will be represented by actions as that one agent performs which affect certain fluents; the other agent has access to these fluents and then can continue the process. This component is specified in a procedural sub-language where the actions can be composed into complex processes.

- **Constructs in the *ConGolog* Process Specification Language**

The *ConGolog* process specification language provides constructs for processes listed in Table 3.1.

In Table 3.1, there are three nondeterministic constructs: `σ$σ2` nondeterministically chooses between processes σ1 and σ2; `pi(variable,σ)` and `pi(ListOfVariables,σ)` nondeterministically picks a binding for the variables in `ListOfVariables/variable` and performs the process σ for this binding of `ListOfVariables/variable`. σ@ means performing σ zero or more times. Note that `for(var,ListOfVariables, varList, σ)` is an abbreviation for `pi(varList,for(var,ListOfValues, σ))`.

Concurrent processes are modeled as interleavings of the primitive actions involved. The primitive actions themselves are viewed as atomic and cannot be interrupted. A process may become blocked when it reaches a primitive action whose preconditions are false or a wait action ϕ? whose condition ϕ is false. Then execution of the system may continue provided another process executes next. In σ1#>σ2, σ1 has higher priority than σ2, and σ2 may only execute when σ1 is done or blocked. σ! is like nondeterministic iteration `σ@`, but the instances of σ are executed concurrently rather than in sequence. Finally, an interrupt `==>(varList, ϕ,σ)` has a list of variables `varList`, a trigger condition ϕ, and a body σ. If the interrupt gets control from higher priority processes and the

condition ϕ is true for some binding of the variables, the interrupt triggers and the body σ is executed with the variables taking these values. Once the body completes execution, the interrupt may trigger again. With interrupts, it is easy to write process specifications that are reactive in that they will suspend whatever task they are doing to handle given conditions as they arise.

PrimActName(ArgList)	primitive action
ϕ?	Wait for condition ϕ
[σ1,σ2,… ,σn]	Sequential execution of programs σ1, σ2,…,σn
if (ϕ, σ1,σ2)	If condition ϕ is true, execute σ1, otherwise execute σ2
while(ϕ, σ)	While condition ϕ is true, repeatedly execute of program σ
for(var,ListOfValue,σ)	For each x in ListOfValues, execute σ with var=x
ProcName(ArgList)	Procedure call
σ1$σ2	Nondeterministic choice between programs σ1 and σ2
σ1#=σ2	Concurrent execution of programs σ1 and σ2 with equal priority
σ1#>σ2	Concurrent execution of programs σ1 and σ2 with σ1 having higher priority
σ!	Concurrent iteration
σ@	Nondeterministic iteration
Pi(variable,σ) Pi(ListOfVariables,σ)	Nondeterministic choice of arguments
==>(varList,ϕ,σ)	Interrupt
No_op	do nothing

Table 3.1 Constructs for processes in *ConGolog*

The construct for procedure definition in *ConGolog* is as follows:

```
proc(name(Parameters), σ)
```

It defines a procedure with `Parameters` and the body σ. Procedure definitions are global.

Constructs for conditionals in *ConGolog* are listed in Table 3.2.

`and(ϕ1,ϕ2)`	Conjunction
`or(Condition1,Condition2)`	Disjunction
`Condition1 --> Condition2`	Implication
`not(Condition)`	Negation
`some(variable,Condition)` `some(varList,Condition)`	Existential quantification
`val(Fluent_name(ArgList),Value)`	Atomic formula involving in functional fluent
`Fluent_name(ArgList)`	Atomic formula with predicate fluent
`true`	Always true

Table 3.2 Constructs for conditionals in *ConGolog*

Note that in the implementation we use ordinary *Prolog* variables for parameters in procedure definitions, Elsewhere, that is in `pi`, `some`, and `for`, the variables bound by the construct are *Prolog* constants.

- **Specifying a System in *ConGolog***

The whole system is specified by the main procedure. Usually, main executes a sub-process for each agent in the domain. For example, a system with an initiator and a single participant running concurrently would be defined as follows:

```
proc(main,
         initiator_behavior#=
         participant_behavior
      ).
```

The behavior of the initiator would be specified in the `initiator_behavior` procedure. This agent might perform the following activities: ordering the scheduling of a meeting, obtaining available dates from participants, finding the suitable dates for participants by merging the available dates with the proposed meeting dates, proposing a suitable date (requesting the agreement on the date and waiting for an answer to the request). These activities are performed in sequence because each activity depends on what has been done earlier. We can use the following procedure to specify the initiator's sequentially performing these activities:

```
proc(initiator_behavior,
  [  orderScheduleMeeting,
     obtainAvailableDates,
     findSuitableDateSlot,
     proposeAMeetingDateForAgreement,
     answerReceived?
  ]
).
```

`orderScheduleMeeting` represents the initiator's ordering scheduling a meeting, `obtainAvailableDates` represents the initiator's obtaining the available dates from the participants, etc.

The behavior of the participant would be specified by the `participant_behavior` procedure. Here, the participant is essentially reactive and will passively answer requests from the initiator whenever a request is made; so we specify its behavior using interrupts. This agent has following responsibilities: sending available dates when requested, acknowledging the occupation of a date by an outside agent, and answering a request for agreement to a meeting date. Each of these is handled by an interrupt, and they are executed concurrently with equal priority. We can write the following procedure to specify this behavior for the participant:

```
proc(participant_behavior,
     ==>(requestedSendAvailableDates, sendAvailableDate)
     #=
     ==>(requestedOccupyDates, acknowledgeOccupyDate)
     #=
     ==>(requestedAgreement, answerAgreement)
).
```

The first interrupt will ensure that whenever the initiator has requested the participant's available dates, the participant will proceed to send his available dates. To ensure that the interrupt triggers only once, sending the available dates should make `RequestedSendAvailDates` false. The other interrupts work in a similar way.

We will present more complicated processes in chapters 5 and 6. A formal semantics for the *ConGolog* process description language has been defined within the Situation Calculus [DLL00]. The semantics is a type of structural operational semantics defining executions as sequences of transitions over configurations involving a situation and a program to be executed; see [DLL00] for details.

3.2.5 Analysing Domain Specifications Using *ConGolog* Tools

Simulation is a useful method for validating domain models and comparing process alternatives. A tool for incrementally generating execution traces of *ConGolog* process specifications has been developed. This tool can be used to check whether a model executes as expected in various conditions by investigating the action trace shown by the simulation. There is also graphical viewer to support displaying the action traces and querying the fluent values [LKMY99].

The simulation tool is based on *Prolog* implementation of the *ConGolog* framework. The *ConGolog* interpreter, which takes a *ConGolog* domain specification and a process specification, generates execution traces that satisfy the process specification given the domain theory. The interpreter uses the domain theory in evaluating test and checking

whether action preconditions are satisfied as it generates the execution traces. The *Prolog* implementation of the interpreter is described in detail in [GLL00].

In our simulation work, we don't use the graphical viewer to show the simulation of *ConGolog* process specification, since using the viewer requires additional specifications to describe how fluent values are to be displayed. But the process execution can be stepped through and exogenous events can be generated at random according to a given probability distribution.

The *Prolog* implementation of the *ConGolog* framework is fairly efficient and can be used for both simulation and for deploying actual applications when one provides implementations for the actions used. However, the current implementation is limited to specifications of initial situation that can be represented as logic programs, which are essentially closed-world theories. This is a limitation of the logic programming implementation, not the *ConGolog* framework.

One may be interested in verifying that the processes in a domain satisfy certain properties. For example, in a mail-order business process, we may be interested in showing that no order is ever shipped before payment is processed. The *ConGolog* framework supports this through its logic-based semantics. A discussion of how *ConGolog* supports verification appears in [LKMY99]. A user-assisted verification tool that can handle arbitrary *ConGolog* theories, including incompletely specified initial situations and specifications of agents' mental states is being developed. Due to time limitations, we don't address the use of verification in our thesis and leave it for future work.

3.2.6 Summary/Discussion

ConGolog is a fully formal and very expressive language. The situational calculus is the logical foundation of the *ConGolog* framework that supports its use in verification. Unlike many other formalisms, it supports simulation given sufficient information about the initial situation, and complex system behaviors are easy to specify using its rich procedural language.

ConGolog can be used both for late-phase requirements engineering and early-phase system design. The analyst can exploit its modeling features and perform simulation and verification based on its logical semantics. But *ConGolog* cannot address issues such as why the process is the way it is, i.e., the motivations, intentions, and rationales behind the activities. If one does not understand why things are done the way they are, one is likely to pick unsatisfactory alternatives for the system of interest, or simply automate outdated processes and miss the opportunity to innovative in redesigning processes. In this thesis, we investigate how *i** and *ConGolog* can be used together to address these issues in requirements engineering.

4 A Methodology for the Combined Use of the *i** and *ConGolog* Frameworks

The *i** SR diagram notation allows many aspects of processes to be represented. It can be used to model why the process is the way it is, what are the motivations, intents, and rationales behind the activities and entities, what are other innovative alternative solutions to the process, and what are the relationships among the participants of the system. These relationships are strategic in the sense that each party is concerned about opportunities and vulnerabilities. But the *i** SR diagram notation is somewhat imprecise and the models produced are often incomplete. For instance, it does not specify whether the subtask in a task decomposition link has to be performed once or several times, whether the subtasks/subgoals are to be performed concurrently, alternatively, or sequentially, and under what conditions they should be performed. In a *ConGolog* model, the process must be completely and precisely specified (although non-deterministic processes are allowed). We need to bridge this gap. To do this, we will introduce a set of *annotations* to SR diagrams that allow the missing process information to be specified. The defined annotations will allow the modeler to specify detailed information about the behavior of every agent, role, and position in the SR model, i.e., what are the conditions for a task to be performed/a goal to be achieved, how are the different tasks/goals to be composed to produce the full behavior of an actor, etc. We also require the modeler to operationalize dependencies between actors, i.e., clarify what interaction (e.g., requests, replies, etc.) has to occur between the actors to have the dependum supplied.

The result of this is an *annotated SR diagram*, a model in an intermediate notation between the initial *i** SR model and the desired *ConGolog* model. In this annotated SR diagram, dependencies are operationalized and all tasks/goals are decomposed into

subtasks/subgoals using the introduced annotations until the process specification is clear enough for the modeler to obtain a corresponding *ConGolog* model. This annotated SR diagram specifies how a process actually proceeds at a detailed level. Obtaining this annotated *i** SR model will help the modeler gain a deeper understanding of the requirements of the system.

We also want to have a tight mapping between the annotated SR diagram and the *ConGolog* model, one that specifies which parts of each model are related and what entity in annotated SR diagram is corresponding to what entity in the *ConGolog* model. This allows us identify which parts of the *ConGolog* model need to be changed when the SR model is modified and vice versa. So we will require the modeler to define such a mapping. We want to ensure that the mapping respects the semantics of both frameworks, so we define a set of *mapping rules* that define what mappings are allowed. The mapping rules help ensure consistency between the annotated *i** SR model and the corresponding *ConGolog* model. The modeler has to respect the mapping rules and map entities in the annotated *i** SR model into appropriate elements in the *ConGolog* model. Finally, after this, we introduce our methodology for combined use of the *i** and *ConGolog* frameworks.

Let us outline the structure of this chapter.

In section 4.1, we discuss the definitions of two types of SR diagram annotations: composition and link annotations. Composition annotations are applied to a group of decomposition links in task/goal decompositions. They help in clarifying whether the subtasks/subgoals are performed sequentially, concurrently, or are the alternative ways to achieve the super-task/super-goal. Link annotations are associated to a single decomposition link connecting a super-task/super-goal with a subtask/subgoal. They

specify under what condition the associated subtask/subgoal should be performed and whether it should be performed once or repeatedly.

Then in 4.2, we discuss the operationalization of dependencies. By adding communication actions into the depender and the dependee, such as the depender's requesting for a dependum, the dependee's waiting for the request, the dependee performing tasks/goals to provide the dependum, etc., the process by which a goal/task/resource dependency is fulfilled is clarified.

In section 4.3, we discuss the steps involved in producing the annotated SR diagram that can be mapped into a *ConGolog* model.

Then in 4.4, we discuss the mapping rules that are used to ensure that entities in an annotated SR diagram are mapped into appropriate entities in a *ConGolog* model and that the models are consistent. Two types of mapping rules are defined: SR node mapping rules and SR link mapping rules. The former ensures that nodes in the annotated SR diagram are mapped into appropriate entities in the *ConGolog* model. The latter ensures that the process of accomplishing the decomposed task/goal is correctly mapped into entities in the *ConGolog* model.

Finally, in 4.5, our methodology for the combined use of the *i** and *ConGolog* frameworks is introduced and the steps of applying the methodology are specified.

4.1 SR diagram Annotations

Two types of annotations are defined: composition annotations and link annotations. Composition annotations are applied to groups of decomposition links in the SR model. These annotations clarify how the linked subtasks/subgoals are to be composed in order to perform the super-task/super-goal, i.e., show whether these subtasks/subgoals are

performed concurrently, sequentially, whether they are alternatives, etc. Link annotations are applied to single decomposition links connecting a super-task/super-goal with its decomposed subtasks/subgoals. These annotations describe under what conditions a subtask/subgoal is to be performed and whether it should be done once or repeatedly. The annotations help the modeler map the annotated *i** SR model into an explicit *ConGolog* process model.

4.1.1 Composition Annotations

The composition annotations are applied to groups of decomposition links in the annotated SR model. These annotations clarify the relationships among the subtasks/subgoals and their composed super-task/super-goal. There are four types of composition annotations: sequence annotation "`;`", alternative annotation "`|`", concurrency annotation "||", and prioritized concurrency annotation ">>".

- **The Sequence Annotation "`;`"**

The sequence annotation is used to specify that the subtasks/subgoals involved in a decomposition are to be performed in sequence in order to accomplish their composed super-task/super-goal.

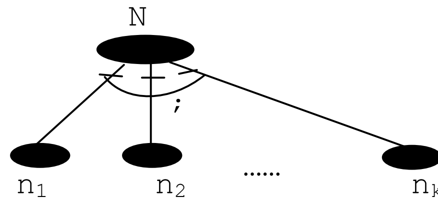


Figure 4.1 Sequence annotation applied to a group of decomposition links.

For example, in Figure 4.1, the super-task/super-goal `N` is decomposed into its subtasks/subgoals n_1, n_2, ... , and n_k and the sequence annotation "`;`" is put on this group of decomposition links. This means that the subtasks/subgoals n_1, n_2, ..., and n_k are to be performed sequentially left to the right to accomplish the super-task/super-goal `N`. The

sequence annotation "`;`" will be mapped into the sequence operation "`,`" provided by *ConGolog*.

The sequence annotation is taken to be the default annotation on a group of decomposition links and is often left out. When no annotation appears on a group of decomposition links, the sequence annotation is assumed.

- **The Concurrency Annotation "||"**

The concurrency annotation is used to specify that the subtasks/subgoals involved in a task/goal decomposition to which the annotation is applied are to be performed concurrently.

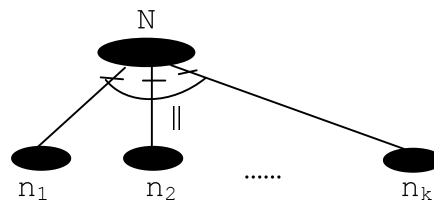


Figure 4.2 Concurrency annotation applied to a group of decomposition links.

For example, in Figure 4.2, the super-task/super-goal `N` is decomposed into its subtasks/subgoals n_1, n_2, ..., and n_k and the concurrency annotation "||" is put on the group of decomposition links. This means that the subtasks/subgoals n_1, n_2, ..., and n_k are to be performed concurrently to accomplish the super-task/super-goal `N`. The concurrency annotation "||" will be mapped into the concurrency operation "`#=`" provided by *ConGolog*.

- **The Alternative Annotation "|"**

The alternative annotation is used to specify that the subtasks/subgoals involved in a task/goal decomposition to which the annotation is applied are different alternative ways of accomplishing the super-task/super-goal.

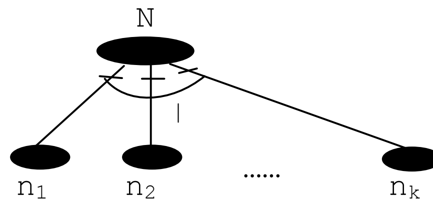


Figure 4.3 Alternative annotation applied to a group of decomposition links.

For example, in Figure 4.3, the super task/goal N is decomposed into its subtasks/subgoals n_1, n_2, ..., and n_k and the alternative annotation "|" is put on the group of decomposition links. This means that any one of the subtasks/subgoals n_1, n_2, ..., and n_k can be selected as an alternative to accomplish the super-task/super-goal N. The alternative annotation "|" will be mapped into the alternative operation "$" provided by *ConGolog*.

- **The Prioritized Concurrency Annotation ">>"**

The prioritized concurrency annotation is used to specify that the subtasks/subgoals involved in a task/goal decomposition are to be performed concurrently in decreasing order of priority.

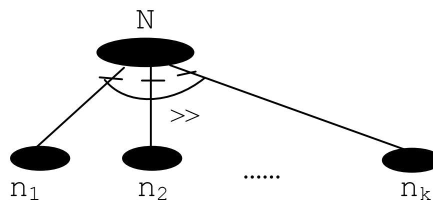


Figure 4.4 Prioritized concurrency annotation applied to a group of decomposition links.

For example, in Figure 4.4, the super-task/super-goal N is decomposed into its subtasks/subgoals n_1, n_2, ..., and n_k and the prioritized concurrency annotation ">>" is put on the group of decomposition links. The subtasks/subgoals n_1, n_2, …, and n_k are to be performed concurrently in order to accomplish the super-task/super-goal N, and the subtask/subgoal n_1 has higher priority than n_2, n_2 has higher priority than n_3, etc. This

means that n_2 will only be executed when n_1 is blocked waiting for some condition, n_3 will be only executed when both n_1 and n_2 are blocked, etc. The prioritized concurrency annotation ">>" will be mapped into the prioritized concurrency operation "#>" provided by *ConGolog*.

4.1.2 Link Annotations

A link annotation is applied to a single decomposition link connecting a super-task/super-goal with one of its subtask/subgoal. These link annotations are used to specify that the linked subtask/subgoal must be performed/achieved repeatedly and/or under some condition. In the absence of an annotation on a single decomposition link, it is assumed that the subtask/subgoal must always be performed exactly once. The modeler uses the link annotations to specify how the process works in detail.

There are five types of link annotations: while-loop annotation `*while(condition)`, for-loop annotation `*for(variable,listOfValue)`, and interrupt annotation `*whenever(variableList,condition)`, which are iteration link annotations, and if annotation `if(condition)` and pick annotation `pick(variablelist, condition)`, which are non-iteration link annotations.

- **The While-Loop Annotation: `*while(condition)`**

The while-loop annotation `*while(condition)` is used to state that the linked subtask/subgoal should be performed repeatedly while `condition` is true.

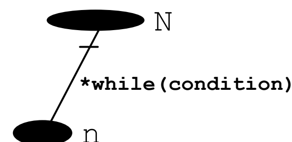


Figure 4.5 While-loop annotation attached to a single decomposition link.

For example, in Figure 4.5, the while-loop annotation `*while(condition)` is attached to the link between the super-task/super-goal `N` and the subtask/subgoal `n`. This means that the subtask/subgoal `n` should be performed repeatedly while `condition` is true in order to accomplish the super-task/super-goal `N`. When `condition` in the while loop annotation becomes false, the repetition terminates. The `condition` is tested before each iteration. This annotation is mapped into the "while-loop" construct provided by the *ConGolog* framework, and its semantics is the standard one for "while loops".

- **The For-Loop Annotation : `*for(variable, listOfValues)`**

This annotation is used to specify that the subtask/subgoal is to be accomplished for each element of the list `listOfValues`. The `variable` can be used to refer to the value of the element in the subtask/subgoal.

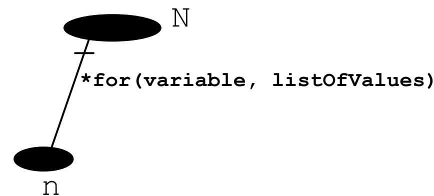


Figure 4.6 For-loop annotation attached to a single decomposition link.

For example, in Figure 4.6, the for-loop annotation `*for(variable,listOfValues)` is attached to the link between the super-task/super-goal `N` and the subtask/subgoal `n`. This means that the subtask/subgoal `n` must be performed for every member of the list `listOfValues` in sequence (left to right) in order to complete the super-task/super-goal `N`. This annotation is mapped into the "for-loop" construct provided by the *ConGolog* framework.

- **The Interrupt Annotation : `*whenever(variableList, condition)`**

This annotation is used to specify that the subtask/subgoal must be performed/accomplished whenever there are values for the variables in `variableList`

for which the `condition` has become true. (the variables may be parameters of the subtask/subgoal).

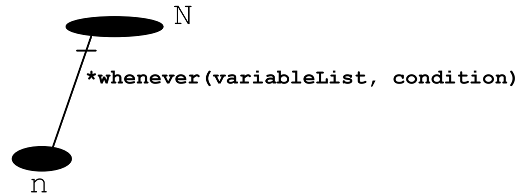


Figure 4.7 Interrupt annotation attached to a single decomposition link.

For example, in Figure 4.7, the interrupt annotation `*whenever(variableList,condition)` is attached to the link between the super-task/super-goal `N` and the subtask/subgoal `n`. This means that the subtask/subgoal `n` will be triggered whenever the `condition` becomes true for some bindings to the variables in the `variableList`; then the subtask/goal `n` must be performed for these bindings of the variables in the list. Once the subtask/subgoal `n` has finished, the interrupt can be triggered again when the `condition` becomes true again. This annotation is mapped into the "interrupt" construct provided by the *ConGolog* framework.

- **The If Annotation: `if(condition)`**

This annotation is used to specify that the linked subtask/subgoal is to be accomplished only if the `condition` is true.

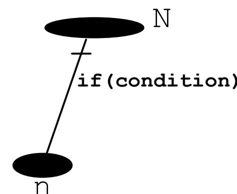


Figure 4.8 If annotation attached to a single decomposition link.

For example, in Figure 4.8, the if annotation `if(condition)` is attached to the link between the super-task/super-goal `N` and the subtask/subgoal `n`. Only when the condition is true, the subtask/subgoal `n` will be performed one time in order to complete the super-

task/super-goal `N`. If this condition is not true, the subtask/subgoal `n` will not be performed and the process of the system skips this subtask/subgoal `n` and proceeds to the other subtasks/subgoals. This annotation is mapped into the "if" construct provided by the *ConGolog* framework, and its semantics is the standard one for "if".

- **The Pick Annotation: `pick(variableList, condition)`**

This annotation is used to specify that the subtask/subgoal must be accomplished for some values of the variables in the `variableList` that satisfy the `condition`.

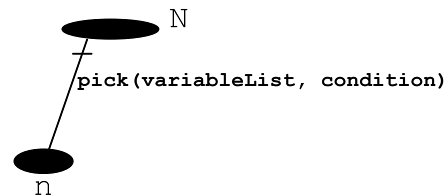


Figure 4.9 Pick annotation attached to a single decomposition link.

For example, in Figure 4.9, the pick annotation `pick(variableList,condition)` is attached to the link between the super-task/super-goal `N` and the subtask/goal `n`. This means that in order to complete the super-task/super-goal `N`, the subtask/subgoal `n` must be performed for some binding of the variables in the `variableList` that satisfies the given `condition`. This annotation is mapped into the construct `pi(variableList,[condition?,task/achieve_goal]` provided by the *ConGolog* framework, where `task/achieve_goal` is the procedure corresponding to the subtask/subgoal `N`.

4.2 Operationalizing Dependencies in the *i** SR model

The dependencies between agents, roles, and positions in the *i** SR model indicate that the depender depends on the dependee to accomplish one of his tasks or goals or to supply some resource. The *i** model generally abstracts over the details of the associated interaction between the agents (requests and communication acts), while the *ConGolog*

model focuses on the operational aspects rather than the strategic/social aspects. We believe that it is not necessary to represent the dependency relationship per se in the *ConGolog* model, but the associated operational elements need to be represented; they are an important part of the process performed by the agents. So we require the modeler to operationalize the dependencies in the SR diagram, i.e., specify the tasks to be performed by the depender and dependee in the interaction that ensure that the dependum is supplied.

The details of how a dependency is operationalized depend on the particulars of the case. It is up to the modeler to specify this. For example in some cases, the depender and dependee will both be involved in the activities to supply the dependum. First, the depender has to request the dependee to provide this dependum. The dependee has to wait for the request from the depender and then perform a task/achieve a goal to supply the dependum. Then the depender has to wait for the dependee to send him confirmation of having supplied the dependum. All these activities will have to be introduced into the SR model. In other cases, the dependee performs the task/achieves the goal/supplies the resource without the depender having to request it. Then, we can simply view the task (goal) involved in the dependency as a subtask (subgoal) of the task (goal) node in the dependee where the dependency terminates. If the depender must wait for the dependency to be fulfilled, this wait action should be represented as a task in the depender. If the depender must also make a request to get the dependee to fulfill the dependency, then this request should be represented as a task in the depender. In some case, the depender and dependee may have to engage in a complex dialogue to have the dependum supplied, and the protocol for this can be specified. Resource dependencies can be operationalized as task dependencies where the task is to supply the resource.

As part of our methodology, the modeler is required to disambiguate the decomposition links and operationalize the dependencies in the SR diagram. We call the result an *annotated SR diagram*. Softgoals and the associated dependencies and links may also be

dropped from the diagram, since they are usually not part of the resulting system's processes. Alternative ways of achieving goals or performing tasks that are not considered for simulation in the *ConGolog* model may also be dropped.

There are four types of dependencies in the initial *i** SR model, i.e., task/goal/resource/softgoal dependencies. Consider a generic case of a dependency between agents, roles, or positions, where the depender and the dependee are both involved in the activity to achieve this dependum. Suppose that first the depender has to request the dependee to provide this dependum when he thinks it is necessary. Second, the dependee has to wait for the request from the depender and then performs a task/achieve a goal to supply the dependum. Third the depender has to wait the dependee to send him confirmation of having supplied the dependum. Let us show how different types of dependencies are operationalized in this generic case. We do not discuss the operationalization of softgoal dependencies here since they will be dropped from the annotated SR model. But when softgoal dependencies are reformulated into hard-goal dependencies, they can be operationalized as goal dependencies.

- **Operationalizing a Task Dependency**

A task dependency between actors, roles, or positions indicates that the depender depends on the dependee to perform a task in order to accomplish his task/goal. For example, in Figure 4.10, the depender depends on the dependee to provide a task-dependum. The dependency relates two nodes, n_1 in the depender and n_2 in the dependee. The nodes may be tasks or goals.

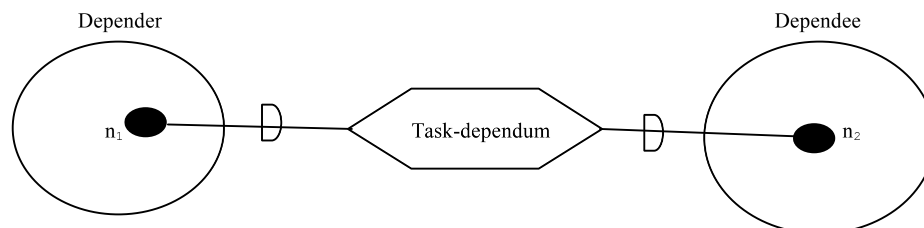


Figure 4.10 SR diagram for the task dependency before operationalization.

We suppose that the interaction between the actors that takes place to get the task-dependum supplied is: the depender requests the dependee to accomplish the task-dependum and then waits for the task-dependum to be performed by the dependee; the dependee waits for the request and then performs the task.

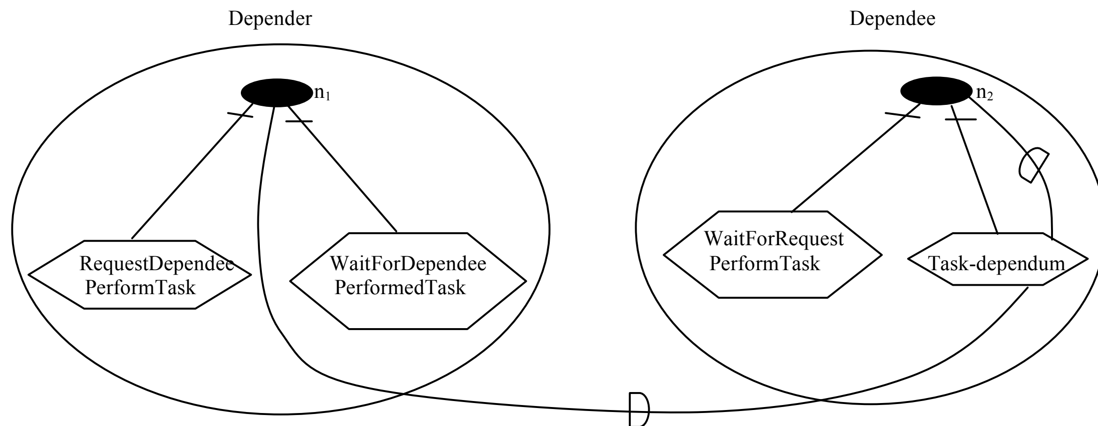


Figure 4.11 SR diagram for the task dependency after operationalization.

The result of operationalizing the task dependency of Figure 4.10 is shown in Figure 4.11. First, the depender requests the dependee to perform the task dependum when it wants to accomplish the task/goal n_1, i.e., the task node `RequestDependeePerformTask`. Second, when the dependee is in the process of performing the task/goal n_2, it will wait for the request from the depender, i.e., the task node `WaitForRequestPerformTask`, and then performs the task when the request is received, i.e., the task node `Task-dependum`. Finally the depender has to wait for the dependee to complete the task-dependum in order to complete n_1, i.e., the task node `WaitForDependeePerformedTask`. In doing the operationalization, we move the task-dependum inside the dependee as a subtask of the task/goal node n_2 because the dependee will perform it. We also add other necessary interaction tasks in the depender and dependee to complete the process of supplying the task-dependum.

In other cases, it may not be necessary for the depender to make a request and for the dependee to wait for the request, and it may not be necessary for the depender to wait for the dependee to complete the task-dependum before continuing with its remaining process. The modeler is responsible for specifying the process to be followed.

- **Operationalizing a Goal Dependency**

A goal dependency between actors, roles, or positions shows that the depender depends on the dependee to achieve a goal. The depender may become vulnerable if the goal fails to be achieved by the dependee.

For example, in Figure 4.12, the depender depends on the dependee to achieve a goal-dependum. The dependency relates two nodes, n_1 in the depender and n_2 in the dependee. The nodes may be tasks or goals.

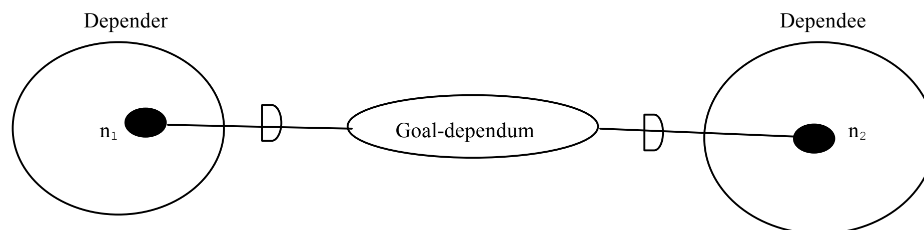


Figure 4.12 SR diagram for the goal dependency before operationalization.

We suppose that the interaction between the actors that takes place to get the goal-dependum supplied is: the depender requests the dependee to achieve the goal-dependum and then waits for the goal-dependum to be achieved by the dependee; the dependee waits for the request and then achieves the goal.

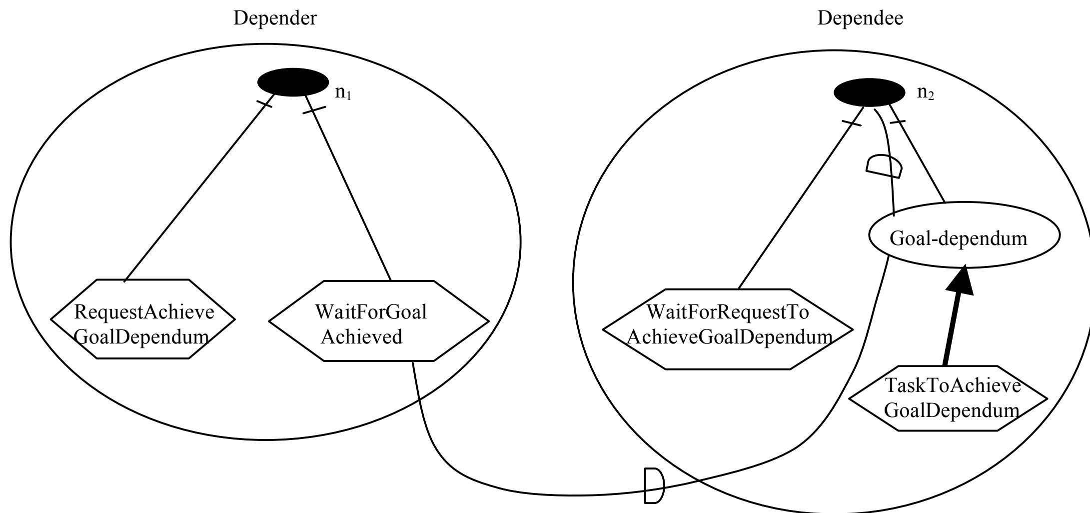


Figure 4.13 SR diagram for the goal dependency after operationalization.

Then, when we operationalize the goal dependency of Figure 4.12, we get the diagram shown in Figure 4.13. First, the depender requests the dependee to achieve the goal-dependum when it wants to accomplish the task/goal n_1, i.e., the task node `RequestAchieveGoalDependum`. Second, in the process of performing the task/goal n_2, the dependee must wait for a request from the depender, i.e., the task/goal node `WaitForRequestToAchieveGoalDependum`, and then perform a task to achieve the goal-dependum, i.e., the task node `TaskToAchieveGoalDependum`. Finally, the depender has to wait for the goal-dependum to be achieved by the dependee in the process of completing its task/goal n_1, i.e., the task node `WaitForGoalAchieved`. In doing this operationalization, we move the goal-dependum inside the dependee as a subgoal of the node n_2 because the dependee will achieve it. We also add other necessary interaction tasks in the depender and dependee to clarify the process of supplying this goal-dependum.

In other cases, it may be not necessary for the depender to make a request and/or for the dependee to wait for the request, and/or for the depender to wait for the dependee to

achieve the goal-dependum. The modeler is responsible for specifying the process to be followed.

- **Operationalizing a Resource Dependency**

A resource dependency between actors, roles, or positions indicates that the depender depends on the dependee to supply some resource in order to accomplish his task/goal. For example, in Figure 4.14, the depender depends on the dependee to supply a resource-dependum. The dependency relates two task/goal nodes, n_1 in the depender and n_2 in the dependee.

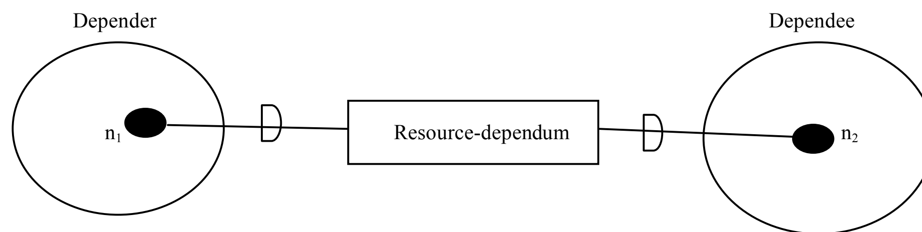


Figure 4.14 SR diagram for the resource dependency before operationalization.

We suppose that the interaction between the actors that takes place to get the resource-dependum supplied is: the depender requests the dependee to supply the resource-dependum and then waits for the resource-dependum to be supplied by the dependee; the dependee waits for the request and then supplies the resource.

Then, the resource dependency of Figure 4.14 is operationalized into the SR model shown in Figure 4.15.

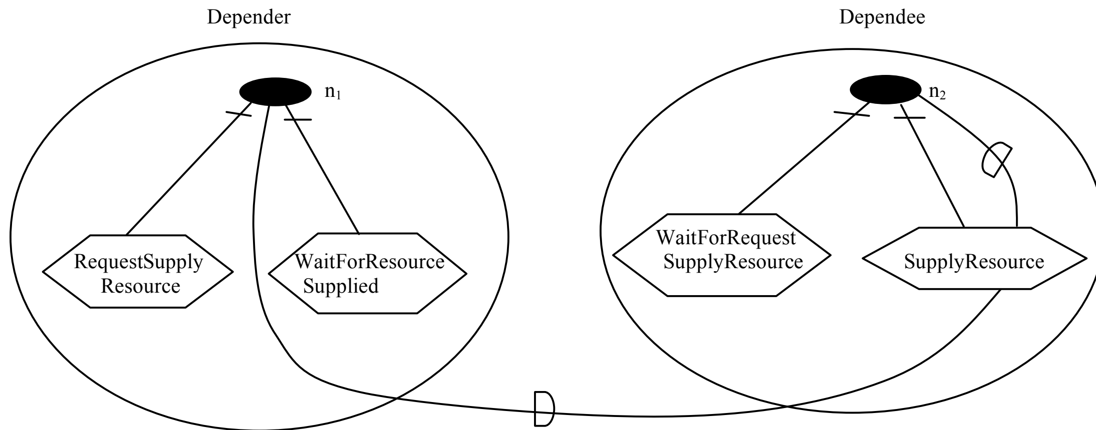


Figure 4.15 SR diagram for the resource dependency after Operationalization.

First, the depender requests the dependee to supply the resource in order to complete its the task/goal n_1, i.e., the task node `RequestSupplyResource`. Second, in order to accomplish the task/goal n_2, the dependee must wait for the request from the depender, i.e., the task node `WaitForRequestSupplyResource`, and then performs a task to supply the resource when the request is received, i.e., the task node `SupplyResource`. Finally the depender has to wait for the dependee to complete the task to get the resource-dependum supplied in order to complete n_1, i.e., the task node `WaitForResourceSupplied`. In doing this operationalization, we move the resource dependum inside the dependee as a subtask `SupplyResource` of the node n_2 because the dependee will perform this subtask to supply the dependum. We add other necessary interaction tasks in the depender and the dependee to complete the specification of the process of supplying the resource-dependum.

In other cases, it may be not necessary for the depender to make a request and/or for the dependee to wait for the request first, and the remaining process still will continue. The modeler is responsible for specifying the process to be followed.

Not all the dependencies will be viewed in the general case as what we described above. In some cases, the dependee performs the task/achieves the goal/supplies the resource without the depender having to request it. Then, we can simply view the task (goal) involved in the dependency as a subtask (subgoal) of the task (goal) node in the dependee where the dependency terminates. As we have seen, resource dependencies can just be treated as task dependencies where the task is to supply the resource. If the depender must wait for the dependency to be fulfilled, this wait action should be represented as a task in the depender. If the depender must also make a request to get the dependee to fulfill the dependency, then this request should be represented as a task in the depender.

4.3 The Annotated *i** SR Diagram

The annotated *i** SR diagram is developed based on the original *i** SR model by employing the defined annotations and operationalizing the dependencies. The objective here is to produce a sufficiently detailed *i** model that can be mapped into a *ConGolog* specification, so that simulation can be performed.

In producing the annotated *i** SR diagram, the modeler must perform the following steps:

- Softgoals and the related links are suppressed.
- Task/goal nodes and dependencies that are not significant to the alternative process to be simulated are suppressed.
- Dependencies are operationalized.
- Goals that cannot always be achieved are weakened or relativized.
- Processes in the alternative(s) that has been selected are modeled in detail. Groups of decomposition links are annotated by composition annotations as necessary. Single decomposition links are annotated by link annotations as necessary.

The above steps must generally be performed in the order given. Of course, sometimes the modeler may want to go back and refine/revise the model; then he has to complete again all the steps that follow the one where the change is made. It takes a lot of effort to

modify the initial SR model to obtain the annotated SR model. But this is beneficial for requirements analysis and the resulting system requirements will be better specified. We will see some examples of this in chapters 5 and 6.

4.4 Mapping Rules

The modeler must define a mapping `m` from the elements of the annotated *i** SR diagram to entities in the *ConGolog* model. We define mapping rules to ensure consistency between the annotated SR model and the *ConGolog* model. The mapping must respect the rules, which arise from the semantics of the two formalisms. There are two types of mapping rules: node mapping rules and link mapping rules. This can be viewed as providing a formal semantics for annotated SR diagrams by mapping them into *ConGolog*, which already has a formal semantics.

In [Yu95B], Yu develops a semantics for *i** by representing *i** notation elements in the Telos conceptual modeling language [MYBJK91] and providing axioms for some *i** notions. We believe that the semantics obtained through our mapping rules is mostly consistent with Yu's semantics, but we haven't tried to prove this. Our semantics is more detailed and formal than Yu's, but it does not try to capture all of *i**. We discuss this more in detail in chapter 7.

4.4.1 Mapping Rules for Nodes

We define mapping rules for each of the five types of nodes in the annotated *i** SR model, i.e., agent nodes, goal nodes, task nodes, role nodes, and position nodes. These ensure that the nodes are mapped into appropriate *ConGolog* entities.

- **Mapping Rule for Agent Nodes**

If `n` is an agent node, then `m(n)=<a, behavior_a>`, i.e., a pair where `a` is a term denoting a *ConGolog* agent (a sub-sort of the sort "other" in the situation calculus) and

behavior_a is a *ConGolog* procedure representing the behavior of the agent. See Figure 4.16 for a graphical representation of the mapping. We use m_agent(n) to refer to the agent a and m_behavior(n) to refer to the agent behavior behavior_a. This rule is applied to all the agents in the system of interest. For agents outside the system, it is not necessary to apply the mapping rule and model them using *ConGolog* agents. Instead we can use the exogenous actions provided by *ConGolog* to simulate the behavior of the outside agents.

In Figure 4.16, the *i** agent Agent is mapped into two elements in the *ConGolog* model: the agent agent_name and the behavior of the agent agent_behavior procedure.

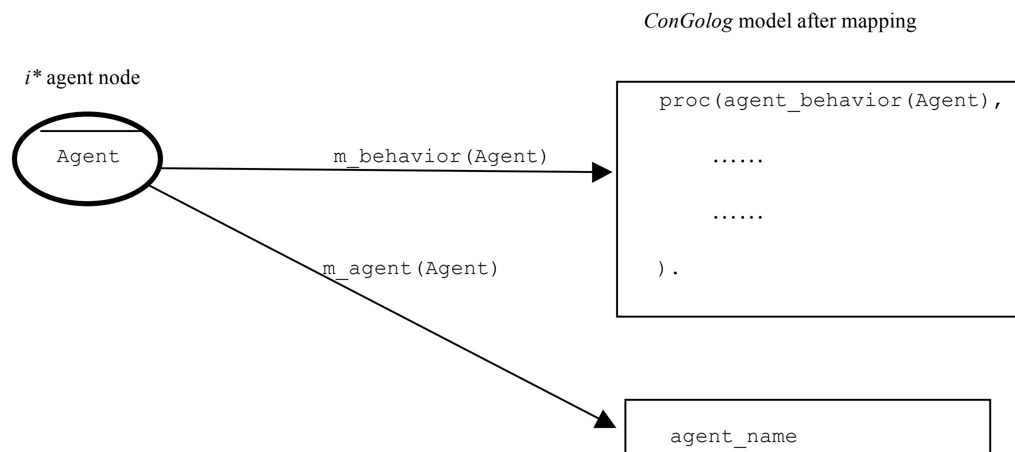


Figure 4.16 The mapping for an agent node in the annotated SR diagram

- **Mapping Rule for Role and Position Nodes**

If n is a role or position node, then m(n) is a *ConGolog* procedure. This procedure is intended to model the behavior of agents playing that role or holding that position. We show the mapping graphically in Figure 4.17.

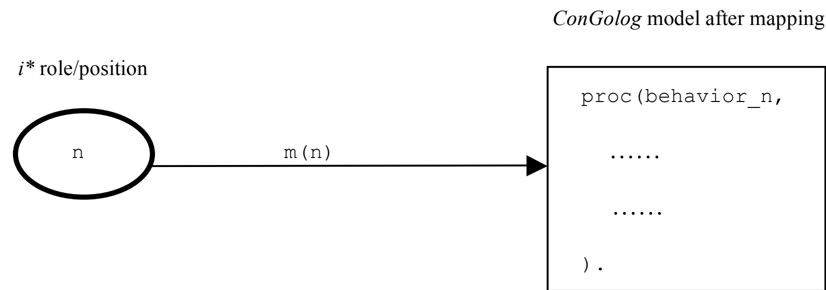


Figure 4.17 The mapping for a role/position node in the annotated SR diagram.

- **Mapping Rule for Goal Nodes**

If `g` is a goal node, then `m(g)=<φ, achieve_g>`, where φ is a *ConGolog* fluent corresponding the goal `g`, either primitive or defined, and `achieve_g` is a *ConGolog* procedure containing means to achieve the goal `g` in which the modeler is interested. `achieve_g` has the post-condition that φ holds, i.e., its body ends with the test "φ?". We use `m_fluent(g)` to refer to the fluent φ and `m_achieve(g)` to refer to the procedure `achieve_g`. See a graphical representation of the mapping for a goal node `g` in Figure 4.18. The goal `g` is mapped into two elements in the *ConGolog* model: the fluent φ and the `achieve_g` procedure.

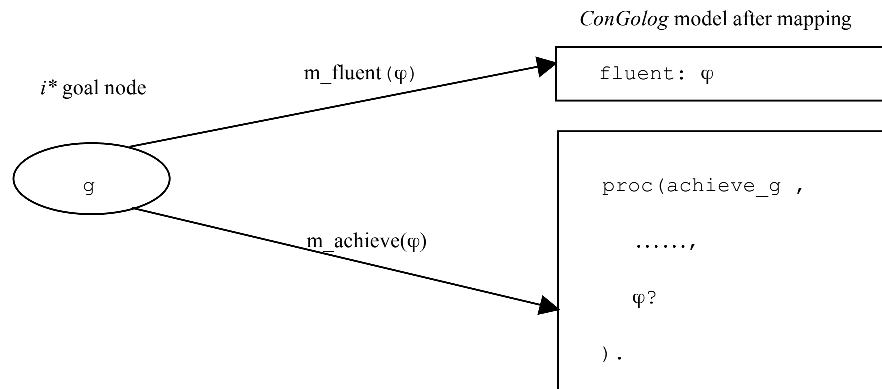
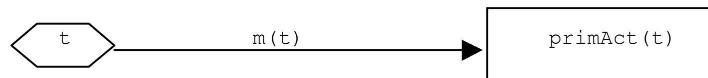


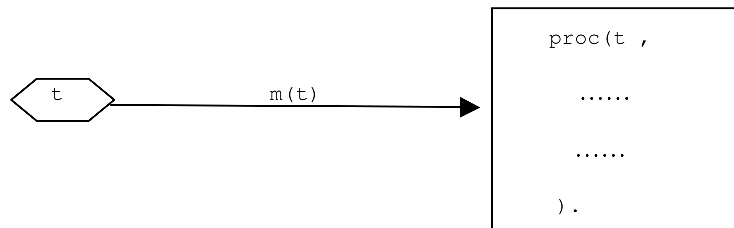
Figure 4.18 The mapping for a goal node `g` in the annotated SR diagram.

- **Mapping Rule for Task Nodes**

If `t` is a task node, then `m(t)` is either a *ConGolog* procedure (complex action) or primitive action. We show the mapping in Figure 4.19; (a) shows a task node mapped into a primitive action and (b) shows a task node mapped into a *ConGolog* procedure.



(a) A task node mapped into a primitive action.



(b) A task node mapped into a *ConGolog* procedure.

Figure 4.19 The mapping for a task node in the annotated SR diagram.

4.4.2 Mapping Rules for Links

There are mapping rules for each of the two types of links in the *i** SR model, i.e., task decomposition links and means-ends links (or goal decomposition links).

- **Mapping Task Decomposition Links**

A task decomposition in the annotated *i** model involves a super task and the decomposed subtasks/subgoals which are connected to the super-task by decomposition links. Composition link annotations must be applied to the group of decomposition links. Link annotations can also be attached to every single decomposition link between the super-task and a subtask/subgoal. For example, consider a task node `t` with its task decomposition links shown in Figure 4.20.

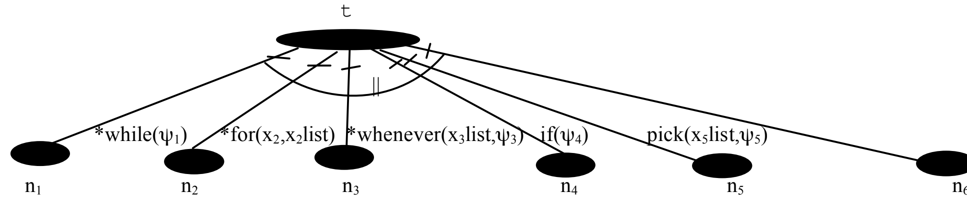


Figure 4.20 E.g. task node with task decomposition links in the annotated *i** SR diagram.

In this example, task node `t` is decomposed into subtask nodes n_1, n_2, n_3, n_4, n_5, and n_6; a concurrency composition annotation "||" is applied to the group of decomposition links; link annotations are attached to each single decomposition link between `t` and n_i except for n_6. According to the mapping rules below, node `t` must be mapped into the *ConGolog* procedure of Figure 4.21.

```
proc(  tp(procedurevariablelist),
             while(ψ1,  m(n1))
             #=
             for((x2, x2list,[], m(n2))
             #=
             ==>(x3list, ψ3, m(n3))
             #=
             if((ψ4, m(n4))
             #=
             pi(x5list, [ψ5?, m(n5)])
             #=
             m(n6)
       ).
```

Figure 4.21 The mapping for the SR diagram of Figure 4.20

The procedure `tp` is the procedure **`m(t)`** corresponding to the behavior of task node `t`. "`#=`" is the *ConGolog* operator for concurrency, which is required by the composition annotation "||". $m(n_i)$ is the result of mapping the task node n_i (for $i=1,2,\ldots,6$). The element that corresponds to the link $t \leftarrow n_i$ accompanying with a link annotation β is the invocation of $m(n_i)$ under the conditions represented by the link annotation β. For

example, the link $t \leftarrow n_1$ accompanied by the link annotation `*while(ψ1)` is mapped into the invocation of `while(ψ1,m(n1))` inside the procedure `tp`, which means that the subtask m(n_1) is repeatedly performed while the condition ψ_1 is true.

Mapping Rule for Task Decomposition Link:

The general mapping rule for task decomposition links is as follows. Consider a general task decomposition shown in Figure 4.22.

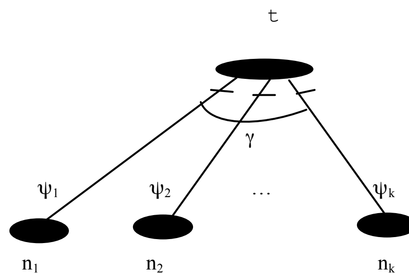


Figure 4.22 Task node `t` with task decomposition links in the annotated *i** SR diagram.

If `t` is a task node that is decomposed into nodes $n_1, n_2, \ldots, n_k$ by task decomposition links, where a composition annotation γ is applied to the group of decomposition links and link annotations ψ_i are applied to the single decomposition link between `t` and n_i, then the mapping for the task node `t`, `m(t)`, is a procedure of the form shown in Figure 4.23:

```
proc(  tp(parameters),
       m(ψ1)(m_proc(n1))
       m(γ)
       m(ψ2)(m_proc(n2))
       m(γ)
       ......
       m(γ)
       m(ψk-1)(m_proc(nk-1))
       m(γ)
       m(ψk)(m_proc(nk))
).
```

Figure 4.23 The mapping for the task decomposition of Figure 4.22.

Here the procedure tp is m(t) corresponding to the behavior of task node t. $m(\gamma)$ is the operator in the *ConGolog* model that corresponds to the composition annotation γ, either the concurrency operator "||", the prioritized concurrency operator ">>", the sequence operator ";", or the nondeterministic choice of action operator "|". $m_proc(n_i)$ is the mapping result of a task node $m(n_i)$ if n_i is a task node or the mapping result of a goal node $m_achieve(n_i)$ if n_i is a goal node. $m(\psi_i)$ is the operator or control structure in *ConGolog* that corresponds to the link annotation ψ_i. The element of the *ConGolog* model that corresponds to the link $t \leftarrow n_i$ accompanied with a link annotation ψ_i is the invocation of $m(n_i)/m_achieve(n_i)$ in m(t) according to the mapping conditions represented by $m(\psi_i)$ of the link annotation ψ_i. If there is no link annotation, then the invocation has no condition.

The *ConGolog* operators associated with composition annotations are shown in Table 4.1.

Composition annotation	*ConGolog* Operator
||	#= : Concurrency
>>	#> : Prioritized concurrency
|	$: Nondeterministic
;	, : Sequence

Table 4.1 *ConGolog* operators associated with composition annotations.

The *ConGolog* control structures associated with link annotations are shown in Table 4.2.

Link annotations	*ConGolog* control structures
t <—while(ψ)— n	`while(ψ, m_proc(n))`
t <—for(variable, valueOfList)— n	`for(variable,valueOfList,[],m_proc(n))`
t <—whenever(variableList,ψ)— n	`==>(variableList,ψ, m_proc(n))`
t <—if(ψ)— n	`if(ψ, m_proc(n))`
t <—pi(variableList,ψ)— n	`pi(variableList,[ψ, m_proc(n)])`

Table 4.2 *ConGolog* constructs associated with composition annotations.

- **Mapping Goal Decomposition (Means-Ends) Links**

A goal decomposition in the annotated *i** model involves a super goal and the decomposed subtasks/subgoals which are connected to the super-goal by decomposition links. Composition link annotations must be applied to the group of decomposition links. Link annotations can also be attached to every single decomposition link between the super-goal and one of its subtask/subgoal if applicable. For example, consider a goal node `g` with its goal-decomposition links shown in Figure 4.24.

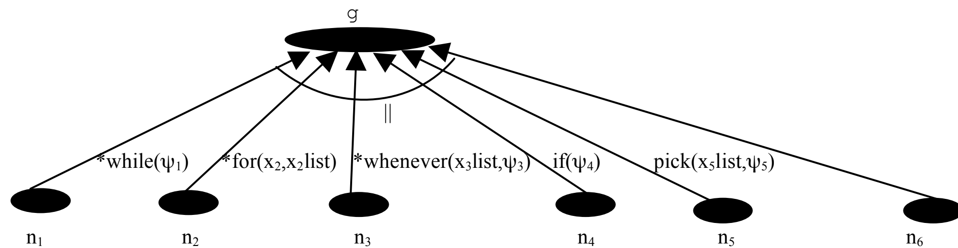


Figure 4.24 E.g. goal node with goal decomposition links in the annotated *i** SR model.

In this example of Figure 4.24, the goal node g is decomposed into task nodes $n_1, n_2, n_3, n_4, n_5,$ and n_6; a concurrency composition annotation "||" is applied to the group of decomposition links; link annotations are applied to each single decomposition link between g and n_i except for n_6. According to the mapping rules for a goal node, node g must be mapped into a *ConGolog* fluent g, either primitive or defined, and a *ConGolog* procedure achieve_g, which is the means to achieve the goal g. achieve_g has the post-condition that g holds, i.e., its body ends with the test "g?". According to the mapping rules for goal-decomposition links below, achieve_g must be a *ConGolog* procedure of the form shown in Figure 4.25:

```
proc(achieve_g(procedurevariablelist),
        while(ψ1,  m(n1))
        #=
        for((x2, x2list,[], m(n2),true)
        #=
        ==>(x3list, ψ3, m(n3))
        #=
        if((ψ4, m(n4))
        #=
        pi(x5list, [ψ5?, m(n5)])
        #=
        m(n6)
        g?
).
```

Figure 4.25 m_achieve(g) for the goal node g the SR diagram of Figure 4.24

The procedure "achieve_g" is the procedure **m_achieve(g)** corresponding the means to achieve the goal g. "#=" is the *ConGolog* operator for concurrency, which is required by the composition annotation "||". $m(n_i)$ is the result of mapping the task node n_i $(i=1,2,\ldots,6)$. The element that corresponds to the link $t \leftarrow n_i$ accompanying with a link annotation **a** is the invocation of $m(n_i)$ under the conditions represented by the link annotation. For example, the link $t \leftarrow n_1$ accompanying with the link annotation $*while(\psi_1)$ is mapped into the invocation of $while(\psi_1, m(n_1))$ inside the

procedure `achieve_g`, which means that the subtask `m(n1)` is repeatedly performed while the condition ψ_1 is true.

Mapping Rules for Goal Decomposition Links:

The general mapping rules for goal decomposition links is as follows. Consider the general goal decomposition in Figure 4.26.

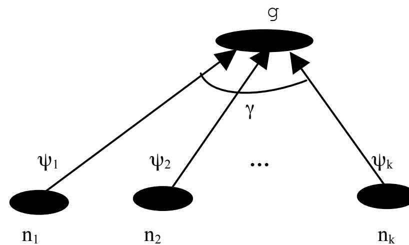


Figure 4.26 Goal node `g` with goal decomposition links in the annotated SR model.

If `g` is a goal node that is decomposed into nodes n_1, n_2, ... , n_k by goal decomposition links, where a composition annotation γ is applied to the group of decomposition links and a link annotation ψ_i is attached to the single decomposition link between `g` and n_i, then `m_achieve(g)` is a procedure **`achieve_g`** of the form shown in Figure 4.27:

```
proc(  achieve_g(procedurevariablelist),
             m(ψ1)(m_proc(n1))
             m(γ)
             m(ψ2)(m_proc(n2))
             m(γ)
             ......
             ......
             m(γ)
             m(ψk-1)(m_proc(nk-1))
             m(γ)
             m(ψk)(m_proc(nk))
             g?
      ).
```

Figure 4.27 `m_achieve(g)` for the goal node `g` in the SR diagram of Figure 4.26.

Here $m(\gamma)$ is the operator in the *ConGolog* model that corresponds to the composition annotation γ, either the concurrency operator "||", the prioritized concurrency operator ">>", the sequence operator ";", or the nondeterministic choice of action operator "|". $m_proc(n_i)$ $(i=1, 2, 3, ..., k)$ is the mapping result of a task node $m(n_i)$ if n_i is a task node, or the mapping result of a goal node $m_achieve(n_i)$ if n_i is a goal node. The link annotation ψ_i is mapped into *ConGolog* control structures such as iteration, conditions, etc. So $m(\psi_i)(m_proc(n_i))$ corresponds to an embedding of a call to the procedure associated with node n_i within the control structure associated with the link annotation ψ_i. The element of the *ConGolog* model that corresponds to the link $g \leftarrow n_i$ accompanied with a link annotation ψ_i is the invocation of $m(n_i)$/$m_achieve(n_i)$ in achieve_g according to the mapping conditions represented by $m(\psi_i)$ of the link annotation ψ_i. If there is no link annotation, then the invocation has no condition. The procedure achieve_g has the post-condition that g holds, i.e., its body ends with the test "g?", which means that the achieve_g is the means to achieve the goal g.

In our examples, goal decomposition always involves or-decomposition, which means any of the subtasks is an alternative way to achieve the goal. Then alternative composition annotation "|" is applied the group of or-decomposition links.

4.4.3 Mapping Dependencies

We do not actually map dependencies into elements of the *ConGolog* model. Instead, we assume that the modeler has made explicit the operational aspects of the dependencies during the operationalization stage, and that the result does not involve dependencies. So we don't need any new mapping rules to deal with dependencies. We just handle the decomposition links that arise from operationalizing the dependencies using the existing link mapping rules.

From a practical point of view, the modeler has to be knowledgeable enough to transfer the dependency relationships in the initial *i** SR model into their operationalized form in the annotated *i** SR model. It may be the case that the modeler has to revise the annotated *i** SR diagram for the dependency relationship several times. These revisions will definitely improve the model and are necessary in applying the methodology. We will give several examples of how dependencies are operationalized and mapped in the next two chapters.

4.5 A Methodology for the Combined Use of the *i** and *ConGolog* Frameworks

Our methodology for the combined use of *i** and *ConGolog* frameworks includes seven steps. Every step enriches the model of the system's requirements gradually. In chapter 5 and 6, we will give a meeting scheduling and a mail-order process as the two typical study cases for modeling complex processes.

Step I. Building the Strategic Dependency Model (SD) for the System

The modeler develops a SD model that answers the questions of who is involved in the system, and what intentional dependencies exist between the agents. The SD model specifies the agents, roles, positions, and the intentional dependency relationships between them. This step is performed as shown in [Yu95B].

Step II. Building the Strategic Rationale Model (SR) for the System

As described in [Yu95B], the modeler further analyzes the requirements of the system based on the developed SD model, focusing on identifying the goals, softgoals, and tasks to be accomplished inside agents/roles/positions, and how they can be accomplished. The answers to these questions are specified in the SR diagram for the system. The SR model specifies the tasks, goals and softgoals inside agents, roles, and positions. It also specifies

the decompositions of the tasks/goals, and the contributions to softgoals. Alternative ways of accomplishing tasks/goals are considered. Opportunities and vulnerabilities also can be analyzed based on the SR diagram. The dependency relationships will be specified between nodes inside the related agents/roles/position. The modeler should represent all the important requirements about how and why the system works the way it does in the SR model.

Step III. Building the Annotated Strategic Rationale Model for the System.

The initial SR model built in the previous step contains information about actors, goals, and activities involved in the application of interest and the rationales behind them. Once some process alternatives have been selected, more details need to be provided to allow the SR model to be mapped into a *ConGolog* model. This is done by building the annotated SR model, which includes the following substeps:

(a) Suppressing Unnecessary Information

The annotated SR model focuses on modeling the workflows and communications between actors, and the important activities performed by the actors in a particular process alternative that fulfills the system objectives. Other alternatives can be ignored at this point. To keep the model as simple as possible, we suppress unnecessary information. Softgoals and the links connected to them will be suppressed because they are qualitative goals that are less important for developing a precise process specification and will be not modeled in the *ConGolog* model. Tasks and goals that are part of other alternative processes will also be suppressed in the annotated SR model.

(b) Operationalizing the Dependencies

The dependencies between the actors will be operationalized as described earlier. The task/goal/resource dependencies will be expanded into internal tasks/goals inside the actors that are the means by which the actors fulfill these outside dependencies.

(c) Relativizing Goals that Cannot Always Be Achieved

Goals that cannot always be achieved by the actors are reformulated so as to be achievable. The decompositions of these goals are refined as appropriate.

(d) Filling out Process Details Using Annotations

Decomposition links will be annotated as necessary to specify how subtasks are composed and when or how often they are performed. The link annotations have to be attached to the decomposition link between the super-task/super-goal and its subtasks/subgoals in the annotated SR model. The link annotations are used to specify whether the linked subtask/subgoal must be accomplished repeatedly and/or only under some condition. The composition annotations have to be applied to a group of decomposition links in the annotated SR model to clarify whether the subtasks/subgoals are performed concurrently, sequentially, concurrently with different priorities, or whether they are alternative ways to accomplish the super-task/super-goal.

Step IV. Developing the Initial *ConGolog* Model

The modeler maps elements in the annotated SR model into entities in the *ConGolog* model using the defined mapping rules and builds the initial *ConGolog* model by specifying the actions, fluents, precondition axioms, successor state axioms, the initial state axioms, and the behavior of the agents in the system.

Step V. Validating the *ConGolog* Model by Simulation

The modeler evaluates the *ConGolog* model through simulation. Given a specification of an initial state for the system, the developed *ConGolog* model will be simulated using the interpreter and the results are used to check the correctness of the model. Then, we identify the shortcomings and refine the *ConGolog* mode according to the result of the evaluation. It may be the case that the first annotated $i*$ model represents some elements

incorrectly or specifies the process incompletely. This step will help the modeler find those mistakes and revise the annotated SR model in the next step. The modeler could also use verification to validate the model, since *ConGolog* supports it. This is not done in this thesis. We discuss this briefly in chapter 8.

Step VI. Refining the *i and *ConGolog* Models Based on Validations Results (Iterated Step)**

Whenever the modeler knows that the *i** model or *ConGolog* models has to be modified based on the results of the validation step, he will refine both the *ConGolog* model and the corresponding part of the *i** model. Also by communicating with the client about the current *i** and *ConGolog* models, the modeler can obtain the feedback from the client and revise the *i** model and the corresponding parts of the *ConGolog* model. This brings out new specification of the system of interest. Another case is when that the modeler needs to add new features into the designed system after he finds some missing requirements have to be modeled, such as loops, exogenous actions, etc. He must modify the *i** model and the corresponding part of the *ConGolog* model, and ensure the consistency between these two models. In chapters 5 and 6, we will show how to do modifications.

Step VII. Producing the Requirements Analysis Document

The models and specifications are collected in a document with appropriate explanations and discussion. The results of simulation and verification are also described.

The above steps must generally be performed in the order given. Of course, sometimes the modeler may want to go back and refine/revise the model; then he has to complete again all the steps that follow the one where the change is made.

In our methodology, we also want to have a close connection and traceability between the *i** and *ConGolog* models. We achieved this by introducing annotations in SR diagrams,

so that they could act as an intermediate notation between *i** and *ConGolog*, and by defining mapping rules that enforce a close correspondence between the annotated SR diagram and *ConGolog* models. In fact, one could automatically generate much of the *ConGolog* specification from the annotated SR diagrams. We discuss issues related to mapping *i** into *ConGolog* and our approach in chapter 7.

5 Case Study I: A Meeting Scheduling Process

In this chapter, our methodology for the combined use of the *i** and *ConGolog* frameworks will be applied to our first case study. This case study concerns a process that is used to support the scheduling of meetings. The idea for the example comes from Yu [Yu97]. The initial requirements for this process might be "For each meeting request, to determine a meeting date and location so that most of the intended participants will be able to effectively participate" [Yu97]. In order to simplify the description, we will only consider determining a meeting date in our example.

There are several alternatives for the meeting scheduling process. One of the alternatives is a process includes a computer-based meeting scheduler (MS). Another alternative is a process that does not include a computer-based MS as shown in chapter 3 (Figure 3.1). In the process with a computer-based MS, the participants' time schedule can be stored on the MS computer system or kept by the participants themselves. There are advantages and disadvantages in these different alternatives. By applying the methodology to these different alternatives for scheduling meetings, especially the *i** analysis techniques, the modeler can make appropriate choices between these alternatives.

In our case study, after the *i** analysis, we will select the alternative for the process that involves a computerized meeting scheduler (MS). We will also decide that the participant's time schedule should be maintained by himself. The selected process operates roughly as follows: After receiving a meeting scheduling request from the initiator, the MS would request all the potential participants for information about their availability to meet during a date range provided by the initiator at that time. A set of

dates when the participant is available will be obtained from the participant. The MS tries to find a suitable date based on the available date sets of all participants. The participants will agree to a meeting date proposed by the MS, if this date is still available in their schedule at the time the proposal is received. If all participants agree, they and the initiator are notified of the confirmed meeting date. In the case where there is no date that suits all the participants, the MS notifies the initiator and the participants that it has failed to find a date to schedule the meeting. In the case where the proposed date has been accepted by some of the participants, but where one of the participants has rejected the date because it has been occupied for some other activities, the MS informs all participants who have accepted the proposed date that it cancels the request, and then goes on to propose another date if there is one available. Otherwise it notifies the initiator and all the participants that meeting scheduling has failed. We will develop the *ConGolog* model and simulate the process to validate the correctness of the modeling using our methodology.

5.1 Building the Strategic Dependency (SD) Model

A Strategic Dependency (SD) model of this meeting scheduling process is shown in Figure 5.1. A version of this model was originally developed by Yu and presented in [Yu97]. We have specialized the actors into agents and roles in our version of the SD model shown as Figure 5.1. The SD model of Figure 5.1 specifies the dependencies that actors have on each other, thus providing the modeler with a better understanding of the "whys" behind the process. Then alternatives can be developed to meet the real needs of the organization.

In the model of Figure 5.1, there are four actor nodes: the meeting scheduler (MS) which is an agent node, and the meeting initiator, important participants, and meeting participants, which are role nodes. Each link between these agents/roles represents how one agent/role depends on another for something. For example, when a meeting `m` is to be

scheduled, the initiator depends on participants for attendance at the meeting. In the SD model, this is represented by a dependency link between the role `Initiator` and the role `Participant`. The role `Initiator` is the depender, the role `Participant` is the dependee, and `AttendsMeeting(p,m)` is the dependum.

The MS is a computer system that helps the meeting initiator schedule meetings by interacting with the participants. We treat it as an agent because it is a concrete system in the organization. The meeting initiator's function is to organize meetings for an organization. He can do this by requesting the MS to schedule a meeting or by scheduling a meeting by talking to the participants himself. We consider it as a role because anyone in the organization can play this role and we don't care who is playing this role. The meeting participants and important participants fulfill the functions of answering the requests from the MS or the initiator regarding meetings and attending meetings. We consider them as roles too because the group of participants are not specified and can vary.

As mentioned in section 3.1, there are four types of dependency relationships in our SD model: task-, goal-, resource-, and softgoal- dependency. The dependency types express different kinds of relationships between the depender and the dependee, involving different types of freedom and constraints. Furthermore, there are three degrees of strength of dependencies: *Open*, *Committed*, and *Critical* [Yu95B].

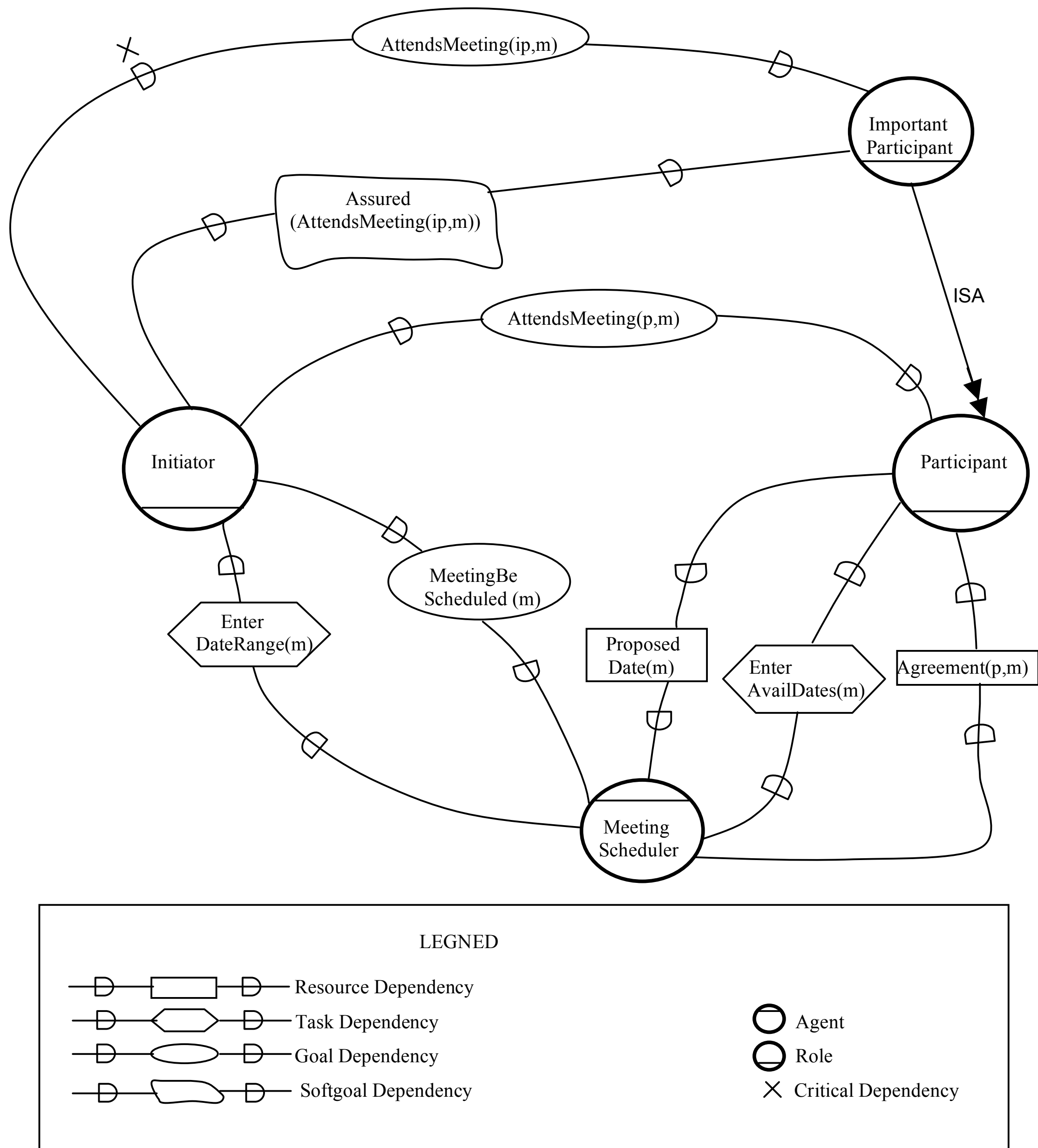


Figure 5.1 A Strategic Dependency model for the meeting scheduling process.

Let us go over the dependencies in the SD model of Figure 5.1. To schedule a meeting, the MS depends on the participants to provide information about their availability for attending the meeting. This is modeled as a task dependency `EnterAvailDates(m)`. It is up to the participants to perform the task of entering their available dates to the MS according to the required procedure (email, phone, etc.). If the participant fails to enter his available dates, the MS becomes vulnerable, but the dependency is not critical for the MS. This is a open dependency, because even if the participants fail to send their available dates, the MS still can propose a date in the meeting date range and ask for agreement from the participants for this proposed date.

The MS also depends on the initiator providing the proposed meeting date range. This is modeled as a task dependency `EnterDateRange(m)` between the initiator and the MS. The meeting initiator has the responsibility to send the meeting date range to the MS in order for the MS to accomplish the task of scheduling the meeting `m`. If the meeting initiator fails to fulfill this dependency, then the MS cannot continue the process of scheduling the meeting `m`. This dependency is a committed dependency, which means that once the meeting initiator has asked for a meeting to be organized, he has committed to providing the meeting date range.

The meeting initiator's dependency on the MS to schedule a meeting is modeled as a goal dependency `MeetingBeScheduled(m)`. It is the MS's responsibility to decide how to reach the goal of having the meeting `m` scheduled. The MS may have various options to reach this goal that will involve alternative processes for scheduling a meeting. For example, the MS could just propose the dates in the meeting date range one by one to all participants, and wait for the participants to reply until it finds an agreeable date on which all participants agree to attend the meeting. This option will require a lot of effort from the participants because they have to check the proposed dates with their time schedule and inform the MS of whether they accept or reject the proposed dates again and again.

Another option is that the MS can first request all the participants to send their available dates, merge these available dates, and then find an suitable date for the meeting from these merged dates. This option will leverage the efforts of the meeting participants because they just need to send their available dates for the meeting once. After these alternatives have been modeled, the clients can choose the one that better meets their needs and intentions. For the rest of our case study, we choose the second option where the MS merges the available date lists to reduce number of interactions with the participants and save time for them.

The meeting initiator depends on the meeting participants to attend a meeting. This is modeled as a goal dependency `AttendsMeeting(p,m)`. It is up to the participant how he attends the meeting. For example, he can take a taxi or drive to the meeting. The meeting initiator also depends on the important participants to attend the meeting, which is modeled as a goal dependency `AttendsMeeting(ip,m)`. This dependency is critical because the attendance of the important participants is required for the meeting to be fruitful. For example, perhaps if the chair of the meeting doesn't show up, then the meeting cannot be held and the initiator will suffer a big loss.

Because the important participants must attend the meeting, the initiator wants to be assured that they will attend. This is modeled as a softgoal dependency `Assured(AttendsMeeting(ip,m))`. It is up to the initiator to decide what measures are enough for him to be assured, e.g., an email or a phone call confirmation. The initiator will be vulnerable if the important participants cannot assure him of their attendance of the meeting `m`. Such softgoal dependencies cannot be expressed in the non-intentional models that are used in most existing requirements modeling frameworks [Yu95B].

The participants depend on the MS to provide a proposed date for a meeting. This is modeled as a resource dependency `ProposedDate(m)`. The MS has to perform some task to provide the resource, i.e., the proposed meeting date. For example, he can send an email to provide the proposed date. The MS also depends on participants to indicate whether they agree to meet on a proposed date. This is modeled as a resource dependency `Agreement(p,m)` between the MS and the participant `p`. It is up to the participant to indicate agreement to a meeting `m` on a proposed date. The participant can accept or reject the request to meet on the proposed date.

Since it captures the intentional dependencies between actors, the SD model can be used to analyze the meeting scheduling process in terms of these intentional relationships [YM94A]. This will help the modeler understand the opportunities and vulnerabilities for the actors. For example, the ability of a computer-based MS to achieve the goal of `MeetingBeScheduled(m)` represents an opportunity for the meeting initiator not to have to achieve this goal by himself. On the other hand, the meeting initiator is vulnerable if the meeting scheduler fails to achieve the goal.

Not that in this chapter, we model the initiator, participant, and important participant as roles. It may be more consistent with *i** concepts to model them as agents, since they are concrete individuals, even though one agent instance (e.g., `yves`) could be an initiator for one meeting and a participant for another. This would also be more consistent with the model in chapter 6.

5.2 Building the Strategic Rationale (SR) Model

In the SR model, a more detailed level of modeling is performed by investigating the activities of the actors and modeling their internal relationships [Yu97]. Intentional elements such as goals, tasks, resources and softgoals are modeled not only as external

dependencies shown in the SD model, but also as internal elements linked by means-ends and task-decomposition relationships and contribution links. These intentional elements express the strategies of every actor and how they try to satisfy their needs and maximize their profits without affecting the success of the whole process. The SR model in Figure 5.2, developed by Yu in [Yu97], elaborates on the relationships between the meeting initiator, the meeting scheduler (MS), and the meeting participants as shown in the SD model in Figure 5.1.

For the meeting initiator, the top level task is to organize a meeting, represented by the internal task node `OrganizeMeeting`. The internal goal node `MeetingBeScheduled` represents the goal that a meeting be successfully scheduled. The internal softgoals nodes `Quick and LowEffort` represent how the initiator wants to arrange the meeting quickly and easily. These softgoals represent whatever quantitative conditions the initiator uses to measure the performance of the processes. The internal task node `ScheduleMeeting` represents a method where the initiator schedules the meeting by himself, and the internal task node `LetSchedulerScheduleMeeting` represents a method where the initiator has the MS to schedule the meeting.

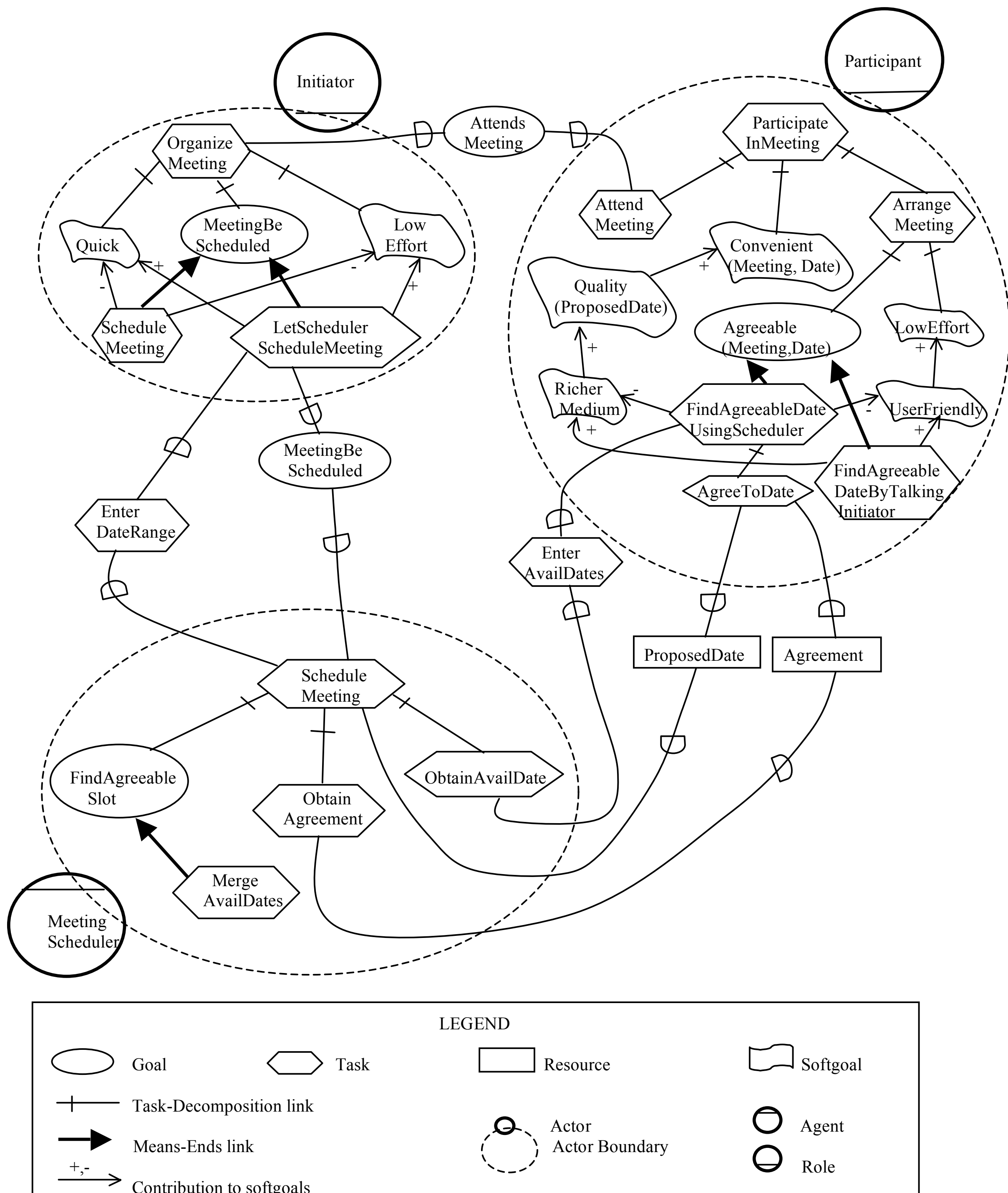


Figure 5.2 An initial Strategic Rationale model for the meeting scheduling process.

Using task decomposition links, the task node `OrganizeMeeting` is decomposed into a subgoal `MeetingBeScheduled`, a sub-softgoal `Quick`, and a sub-softgoal `LowEffort`; there is also a outgoing goal dependency to the participants `AttendsMeeting`. If the subgoal `MeetingBeScheduled` is achieved by the initiator, the goal dependency `AttendsMeeting` is fulfilled by the participants, and the softgoals `Quick` and `LowEffort` are satisfied to a sufficient degree, then the task `OrganizeMeeting` is successfully accomplished. If one of the subtasks/subgoals of the decomposition fails to be accomplished, for example, if a participant fails to attend the meeting, then the task `OrganizeMeeting` fails to be accomplished. To what degree the sub-softgoals need to be satisfied is up to the initiator. The super-task might still be accomplished even if these softgoals are poorly satisfied. These softgoals are introduced to help the client choose one alternative process over another because of their contributions to these softgoals.

In the initiator, the internal goal `MeetingBeScheduled` is or-decomposed into two subtasks: a subtask `ScheduleMeeting` (the initiator does it himself) and a subtask `LetSchedulerScheduleMeeting.` These are two alternative means to achieve the goal `MeetingBeScheduled`. The task `LetSchedulerScheduleMeeting` makes positive contributions to both of the softgoals `Quick` and `LowEffort`, i.e., it will save time and efforts for the initiator. The other alternative `ScheduleMeeting` contributes negatively to these softgoals. So `LetSchedulerScheduleMeeting` will be chosen over the other mean `ScheduleMeeting` to achieve the goal `MeetingBeScheduled.`

Inside the MS, the task `ScheduleMeeting` represents the main task that the `MS` has to perform when it gets a request to schedule a meeting for the initiator. The task `ScheduleMeeting` is decomposed into a subtask of obtaining available dates from participants `ObtainAvailDates`, a subgoal of finding a suitable date slot

FindAgreeableSlot, and a subtask of obtaining agreement from the participants ObtainAgreement (represented as task-decomposition links). Meanwhile the participants depend on the MS to supply a proposed meeting date, which is represented as a resource dependency ProposedDate between the task ScheduleMeeting in the MS and the task AgreeToDate in the participant. The sub-elements of the main task are represented as subgoals, subtasks, or resources depending on the type of freedom of choice the MS has as to how to accomplish these sub-elements. So FindAgreeableSlot is a subgoal which can be achieved by the MS in different ways. On the other hand ObtainAvailDates and ObtainAgreement, both are subtasks that refer to specific ways of accomplishing these tasks. In order to provide the resource dependency ProposedDate, the task ScheduleMeeting needs to perform some tasks to fulfill this dependency requirement. Later we will introduce communication entities into our intermediate notation based on the SR diagram of Figure 5.2 to explicitly show how the resource dependency ProposedDate will be supplied through a specific type of interaction between the depender and dependee. We call this procedure operationalizing the dependencies.

Inside the MS, the goal FindAgreeableDateSlot is to find an agreeable meeting date slot for the participants. Here the only mean to achieve the goal considered is the task MergeAvailDates, i.e., merging all the available dates of participants. Actually, another possible way to achieve this goal might be just choosing all dates in the meeting date range which is proposed by the initiator as the agreeable meeting date slot. This alternative is not considered because if the proposed meeting date range is large, then the rounds of interruption from the MS to the participants are too much. We prefer to get the available dates from all participants and then merge these available dates and the proposed meeting date range to narrow the set of possible agreeable dates as much as possible. This is an example how the SR model can help to analyze possible alternatives for the process and improve its performance.

Inside the participant, the main task is ParticipateInMeeting. This task is decomposed into a subtask AttendMeeting, representing how the participant attends the meeting, a sub-softgoal Convenient(Meeting,Date), meaning that the participant wants the meeting and date are to be convenient to him, and a subtask ArrangeMeeting, representing how the participant will proceed to make meeting arrangements. It is essential that the subtasks AttendMeeting and ArrangeMeeting be accomplished in order to complete the task ParticipateInMeeting. But it is not essential that the proposed meeting date be very convenient. The softgoal Convenient(Meeting,Date) helps the modeler analyze the performance of the different ways to complete the task ParticipateInMeeting and find out the best process for the system.

The task ArrangeMeeting is furthermore decomposed into a subgoal Agreeable(Meeting,Date), representing how the participant wants an agreeable date for the meeting to be selected, and a sub-softgoal LowEffort representing how he wants the meeting arrangements to be easy for him. The softgoal is important but not crucial for the participant. The participant decides how to evaluate the meeting arrangements is easy for him.

The internal goal Agreeable(Meeting,Date) is or-decomposed into two alternative means: a subtask FindAgreeableDateUsingScheduler meaning that the participant uses the meeting scheduler to find an agreeable date for the meeting and a subtask FindAgreeableDateByTalkingInitiator meaning that the participant talks with the initiator to find an agreeable date for the meeting directly. To help choose which alternative is the best, the SR model represents how the choice affects the participant's softgoals. As we can see, performing the task FindAgreeableDateUsingScheduler to achieve the goal Agreeable(Meeting,Date) will produce a less rich medium and not be so user-friendly. As a consequence, this alternative will cause the quality of the proposed date to

be less good and perhaps less convenient to the participant. The participant will also have to put more effort to arrange a meeting. On the other hand, the other alternative of achieving the goal by performing the task `FindAgreeableDateByTalkingInitiator` will involve a richer medium and be more user-friendly. Of course, the choice among the alternatives also involves the initiator. What is the best for the participant may not be the best for the initiator. Later in our further analysis, we choose the alternative `FindAgreeableDateUsingScheduler` over the alternative `FindAgreeableDateByTalkingInitiator` because the initiator wants the MS to find an agreeable date for a meeting instead of having to talk to participants himself. This representation helps the client choose the best alternatives.

Inside the participant, `FindAgreeableDateUsingScheduler` is an internal task, which is decomposed as follows: there is a task dependency `EnterAvaildates`, entering the available dates to the MS, and a subtask `AgreeToDate` of working out an agreement about attending the meeting on a given date. `AgreeToDate` is also involved in two dependencies: the resource dependencies `ProposedDate` and `Agreement` between the MS and the participant. Note that this SR model does not clarify what happens when there is no date on which every participant agrees to attend the meeting. Later, in our intermediate notation, we will refine the SR model to address this.

An important part of the SR diagram is the dependencies between actors. In the SR diagram of Figure 5.2, the dependencies show how one actor depends on another actor, but the diagram does not show how the depender and the dependee interact with each other to fulfill the dependencies. It can be important that the modeler clarifies what is involved in these interactions. Later in section 5.3, we perform the operationalization of the dependencies to clarify the interactions between the depender and dependee nodes in order to supply the dependum.

As we can see, the SR model can help in modeling the interests of actors, how they might be met, and the actors' evaluation of various alternatives with respect to their interests. Task-decomposition links provide a hierarchical description of intentional elements that constitute a routine and means-ends links provide an understanding about why an actor would perform a task, achieve a goal, need a resource, or want a softgoal. The softgoals allow analysis about why one alternative may be chosen over others. For example, availability information in the form of a set of available dates is collected so as to minimize the number of rounds and thus minimize interruptions for the participants.

The SD model and SR model can support the analysis, design, and reasoning performed during early-phase requirements analysis and modeling. In terms of ability, workability, viability, and believability, the *i** framework provides a number of levels of analysis and high level design. [Yu95B]

5.3 Building the Annotated *i** SR Model

In order to produce a sufficiently detailed and precise *i** model that can be mapped into a *ConGolog* specification and allows simulation to be performed, we develop an annotated *i** SR diagram based on the original *i** SR model of Figure 5.2 using the defined annotations and operationalizing the dependencies.

In the annotated *i** SR diagram, the selected alternative, where the meeting scheduler is used to arrange meetings, will be modeled in detail. Softgoals and the related links will be suppressed since they are not central to the task of making the process specification precise. Task/goal nodes and dependencies that are not significant to the selected process will also be suppressed. Dependencies will be operationalized. Goals that cannot always be achieved will be weakened or relativized. Groups of decomposition links will be annotated by composition annotations and single decomposition links will be annotated by link annotations as necessary.

Before we begin the substeps of obtaining the annotated SR diagram, let us clarify which alternative process is selected from the initial SR model of Figure 5.2. The meeting initiator organizes the meeting by using a computerized meeting scheduler, rather than by talking to the participants directly. The participant will only respond to requests from the meeting scheduler; they do not take the initiative.

5.3.1 Suppressing Unnecessary Information

We start by suppressing less important information from the SR diagram. We proceed in two steps.

First, softgoals, softgoal dependencies, and the links related to them are suppressed because they are qualitative goals which are less important for developing a precise process specification and will not be modeled in the *ConGolog* model. As well, tasks and goals inside the actors and the dependencies that are part of other alternative processes are suppressed. We assume that the client wants to use the computer system of the MS to efficiently schedule the meeting for the initiator. For the initiator, the alternative of scheduling meetings by talking with the participants is suppressed. Only the mean `LetSchedulerScheduleMeeting` to achieve the goal `MeetingBeScheduled` appears. The mean `ScheduleMeeting` where the initiator arranges the meeting himself is left out. For the participants, we only show the mean `FindAgreeableDateUsingScheduler` to achieve the goal `Agreeable(Meeting, Date)` because we want the computerized MS to efficiently find an agreeable date for the meeting which is suitable for all participants. The mean `FindAgreeableDateByTalkingInitiator` is suppressed. The resulting SR diagram appears in Figure 5.3.

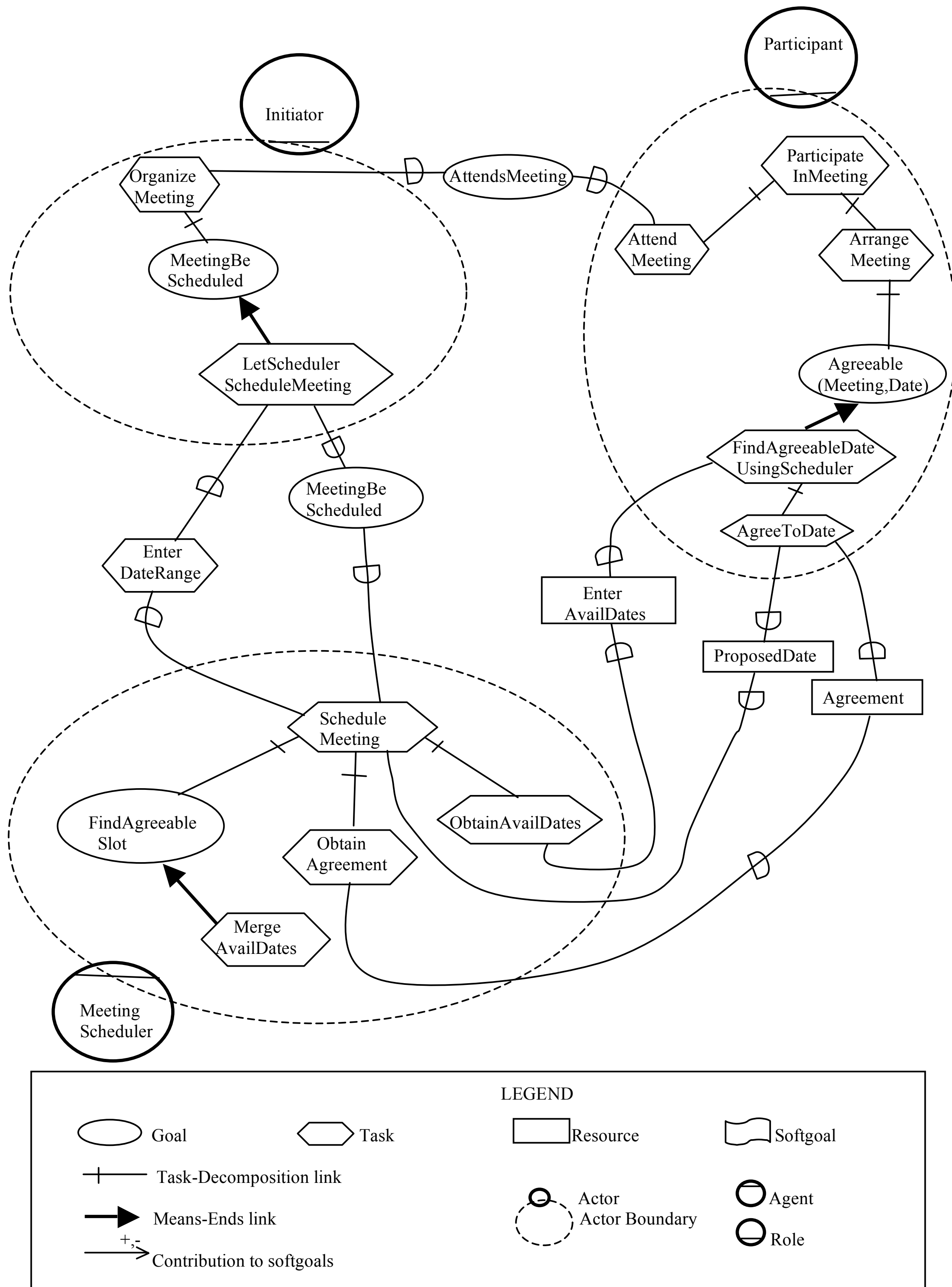


Figure 5.3 The second version of the *i** SR model for the meeting scheduling process.

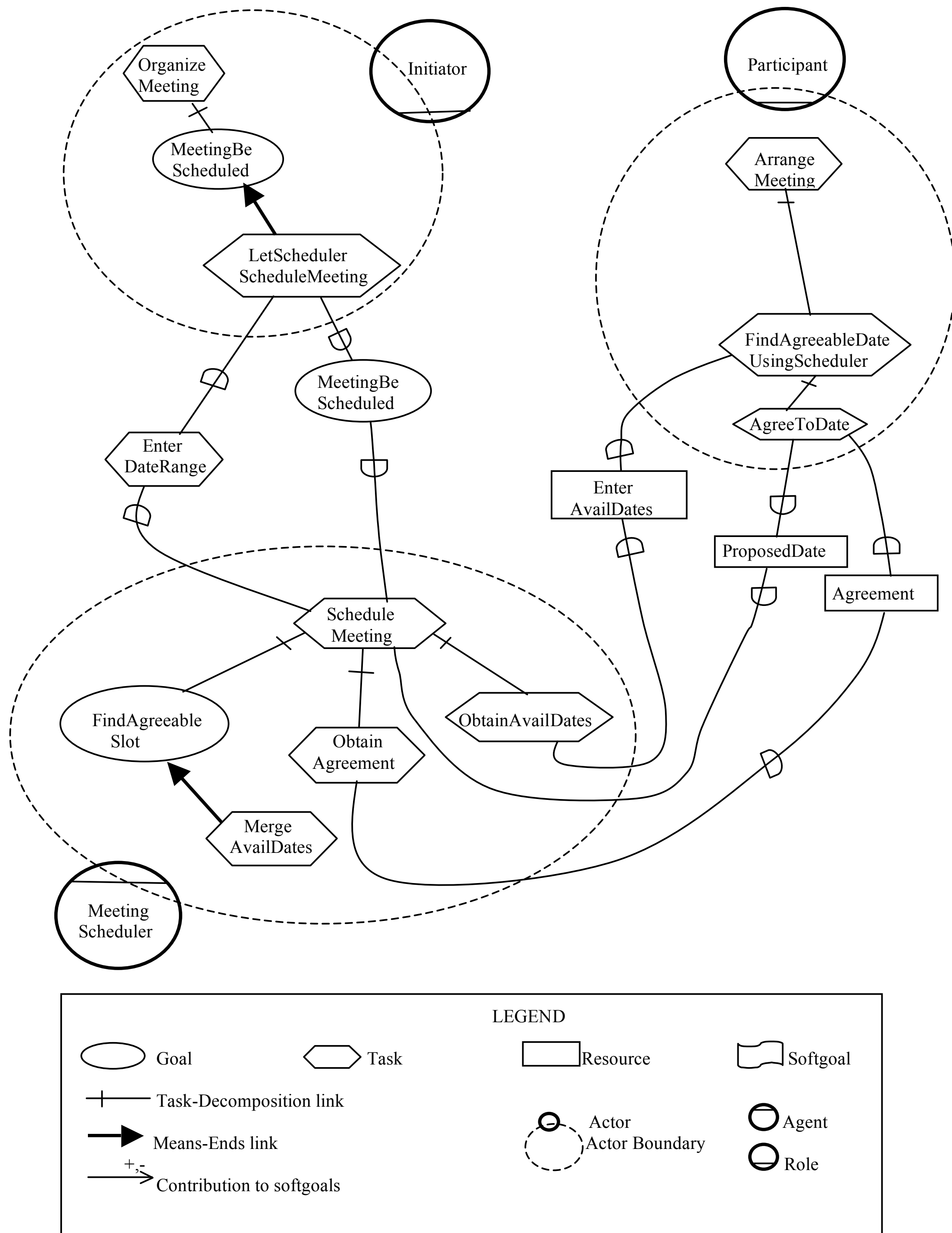


Figure 5.4 The third version of the *i** SR model

for the meeting scheduling process.

In a second step, we suppress other information. In modeling the meeting scheduling process, we want to focus on the meeting being scheduled. Activities such as how the participant participates in a meeting and how he will attend will not be modeled. So we suppress the dependency between the node `OrganizeMeeting` in the initiator and the node `AttendMeeting` in the participant (in Figure 5.3), because this dependency does not relate to the scheduling process. The task nodes `ParticipateInMeeting` and `AttendMeeting` in the participant will be suppressed also because these two tasks are not related to our selected process of how schedule a meeting. The goal node `Agreeable(Meeting, Date)` is also eliminated because we take the participants to just passively answer requests from the meeting scheduler and wait for the meeting scheduler to find an agreeable date. The participant does not actively decide to find an agreeable date by himself. The resulting simplified SR model is shown in Figure 5.4.

5.3.2 Operationalizing the Dependencies

Now, the dependencies between the actors will be operationalized as described earlier. The task/goal/resource dependencies will be represented as internal tasks/goals inside the actors that are the means by which the actors supply these outside dependencies.

First, we operationalize the dependencies between the initiator and the MS in Figure 5.5.

Here, the meeting initiator depends on the MS to schedule a meeting and this is modeled as a goal dependency `MeetingBeScheduled`. The MS depends on the initiator to enter the meeting date range for him and this is modeled as a task dependency `EnterDateRange`. The result of operationalizing the dependencies of Figure 5.5 is shown in the SR model of Figure 5.6.

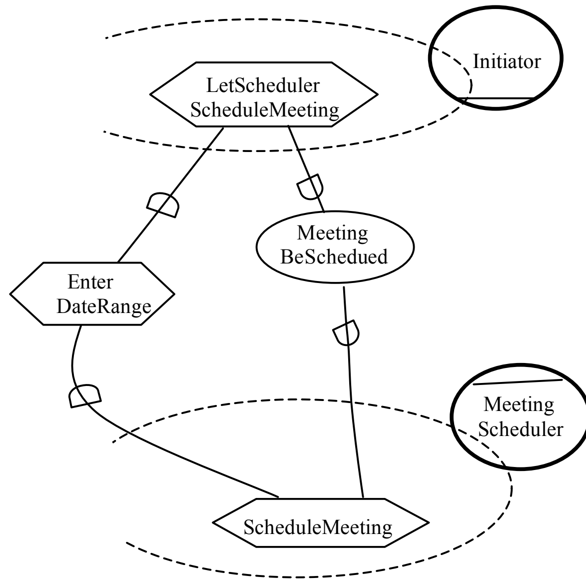


Figure 5.5 The SR diagram for dependencies between the meeting initiator and the MS.

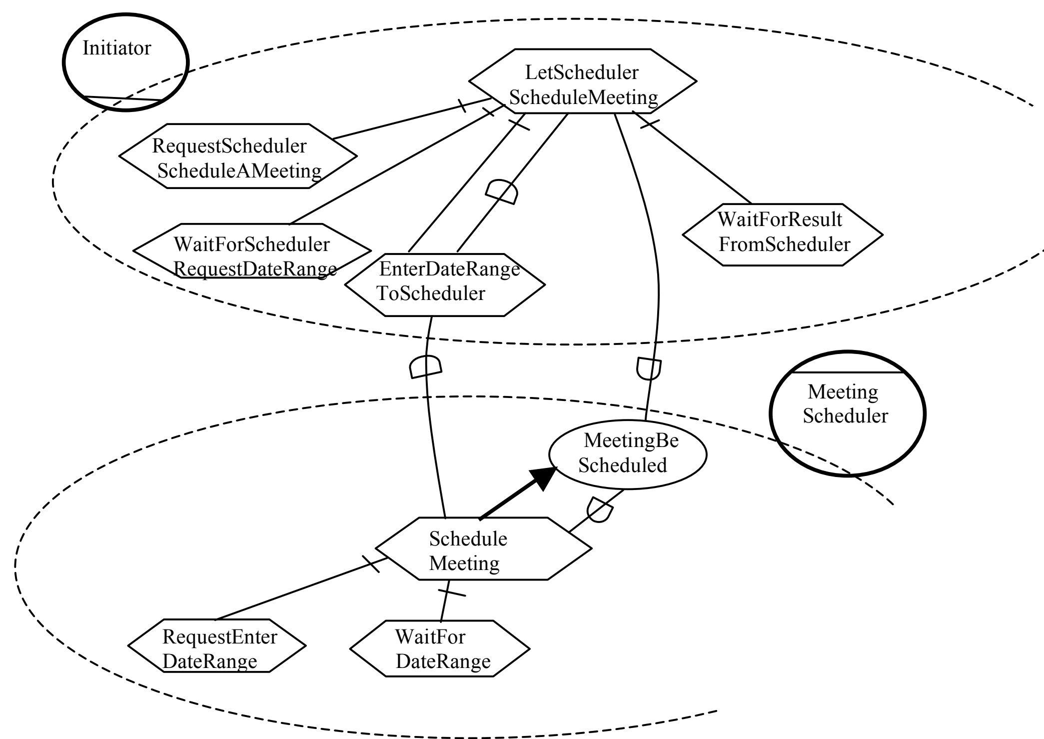


Figure 5.6 The SR diagram after operationalizing the dependencies of Figure 5.5.

To fulfill the task dependency `EnterDateRange`, we suppose that the following activities are to be performed. First the MS has to request the initiator for the date range; this is modeled as a subtask `RequestEnterDateRange` of `ScheduleMeeting` in the MS. Then, the meeting initiator has to wait for the request from the MS; this is modeled as a subtask `WaitForSchedulerRequestDateRange` of `LetSchedulerScheduleMeeting` in the initiator. Then, the meeting initiator performs a task to send the date range; this is modeled as a subtask `EnterDateRangeToScheduler` of `LetSchedulerScheduleMeeting` in the initiator. Finally, the MS has to wait for the date range to be sent and then continues with the remaining process; this is modeled as a subtask `WaitForDateRange` of `ScheduleMeeting` in the MS.

For the goal dependency `MeetingBeScheduled`, we suppose that it is fulfilled as follows. First, the initiator has to request the MS to schedule a meeting; this is modeled as a task `RequestSchedulerScheduleAMeeting` in the initiator, a subtask of `LetSchedulerScheduleMeeting`. Then, the goal `MeetingBeScheduled` is moved into the MS as an internal goal and can be achieved by the task node `ScheduleMeeting` where the dependency terminates, i.e., the task `ScheduleMeeting` becomes the mean to achieve this internal goal. Finally, the initiator has to wait for the result of scheduling the meeting from the MS; this is modeled as a task `WaitForResultFromScheduler` in the initiator, a subtask of `LetSchedulerScheduleMeeting`.

In the SR model of Figure 5.6, the tasks/goals added into actors to have the dependencies supplied are modeled as subtasks/subgoals of the tasks/goals where the dependencies start and terminate. Note that the SR model of Figure 5.6 just shows the part of the SR diagram about the operationalized dependencies. To obtain the complete SR model for the process of scheduling a meeting, the tasks/goals arising from the operationalization have to be related to other tasks/goals inside the actors.

Next, we operationalize the dependencies between the MS and the participants that are shown in Figure 5.7. The SR model of Figure 5.8 shows the result after operationalizing the dependencies of Figure 5.7.

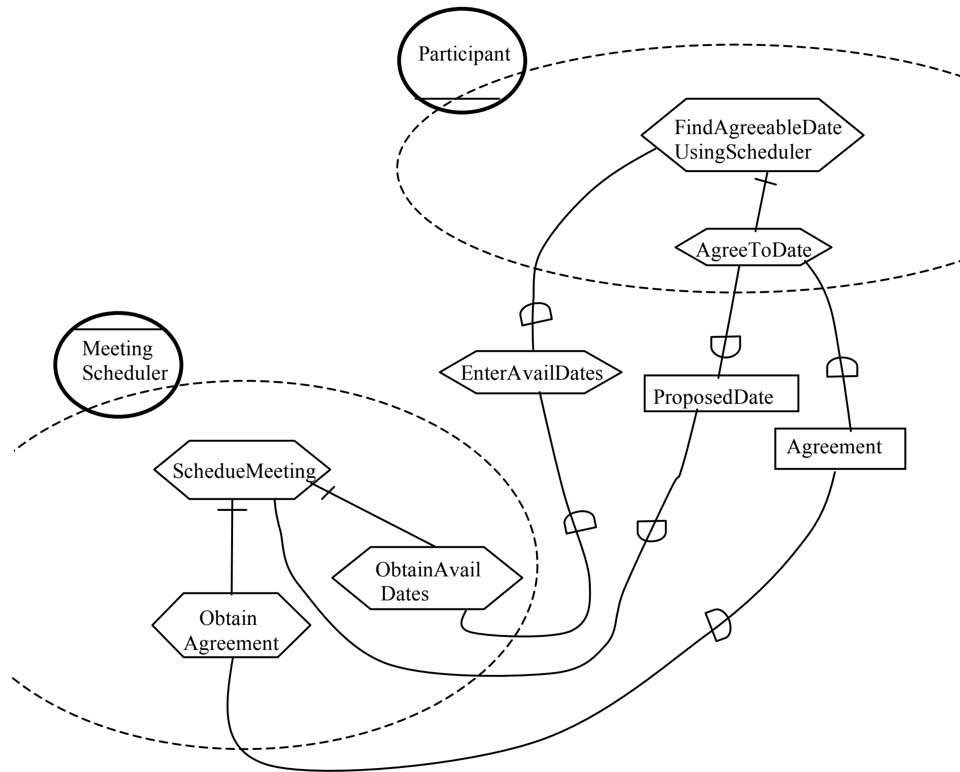


Figure 5.7 The SR diagram for dependencies between the MS and the participants.

We consider the task dependency `EnterAvailDates` first. We operationalize it as follows. First, the MS has to request the available dates from the participants; this is modeled as a task in the MS `RequestAvailDates`. Then, when the participant receives the request he will send his available dates to the MS; this is modeled as a task in the participant `SendAvailableDates`. Then, the scheduler has to wait for all participants sent their available dates; this is modeled as a task in the scheduler `WaitForAllAvailableDates`.

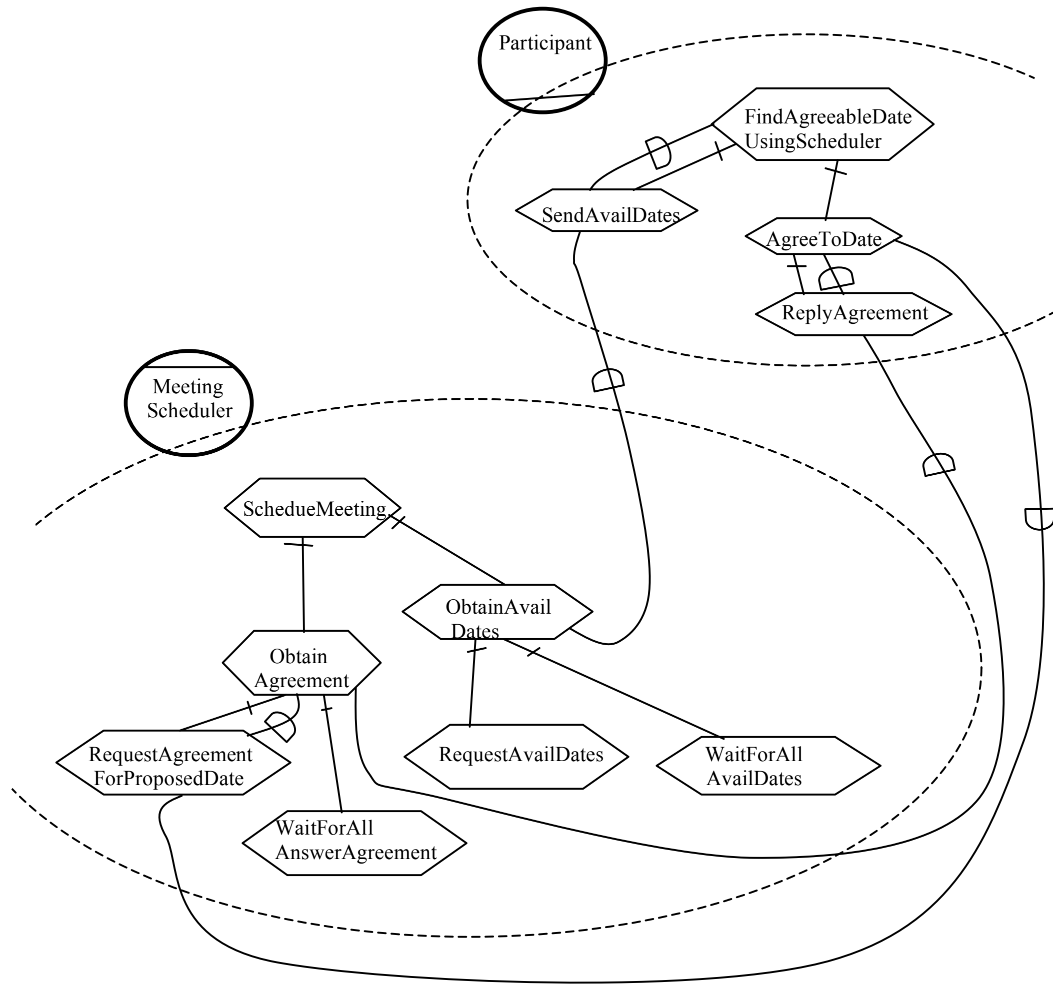


Figure 5.8 The SR diagram after operationalizing the dependencies of Figure 5.7.

Then we consider the resource dependencies `ProposedDate` and `Agreement` together because they are connected to each other in the process of reaching an agreement on a proposed date. We operationalize them as follows. First, the scheduler sends the proposed meeting date to the participant and requests agreement on this proposed date; this is modeled as a task in the MS `RequestAgreementForProposedDate`. Here, we consider sending the proposed date to obtain an agreement as a subtask of the task `ObtainAgreement` because we think that proposing a meeting date is part of the task of obtaining the agreement. Then, when the participant receives the request he will reply

by indicating whether he agrees or rejects it; this is modeled as a task `ReplyAgreement`, a subtask of `AgreeToDate` in the participant. Then, the scheduler has to wait for the replies of all the participants; this is modeled as a task `WaitForAllAnswerAgreement`, a subtask of `ObtainAgreement` in the MS.

In Figure 5.9, we show the SR diagram for the whole process of scheduling a meeting after suppressing unnecessary information and operationalizing all dependencies. In Figure 5.9, the SR diagram of (a) is for the initiator, the one (b) is for the MS, and the one (c) is for the participant.

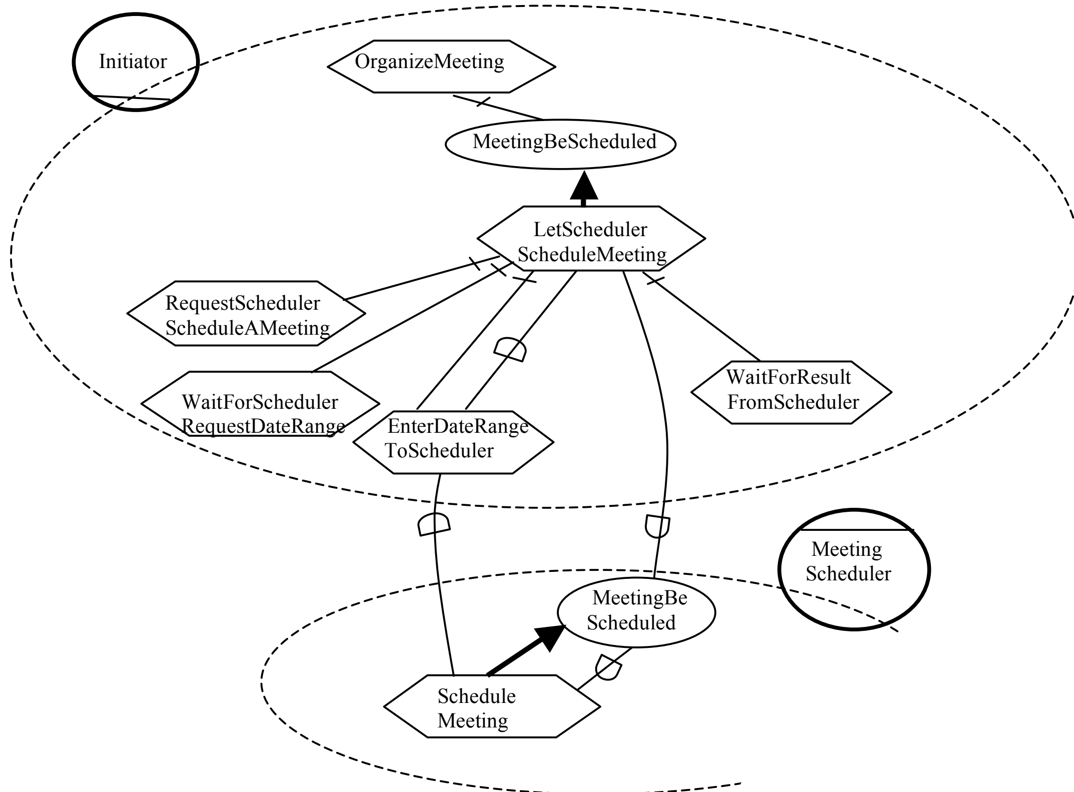


Figure 5.9(a) The SR diagram for the initiator after operationalizing dependencies.

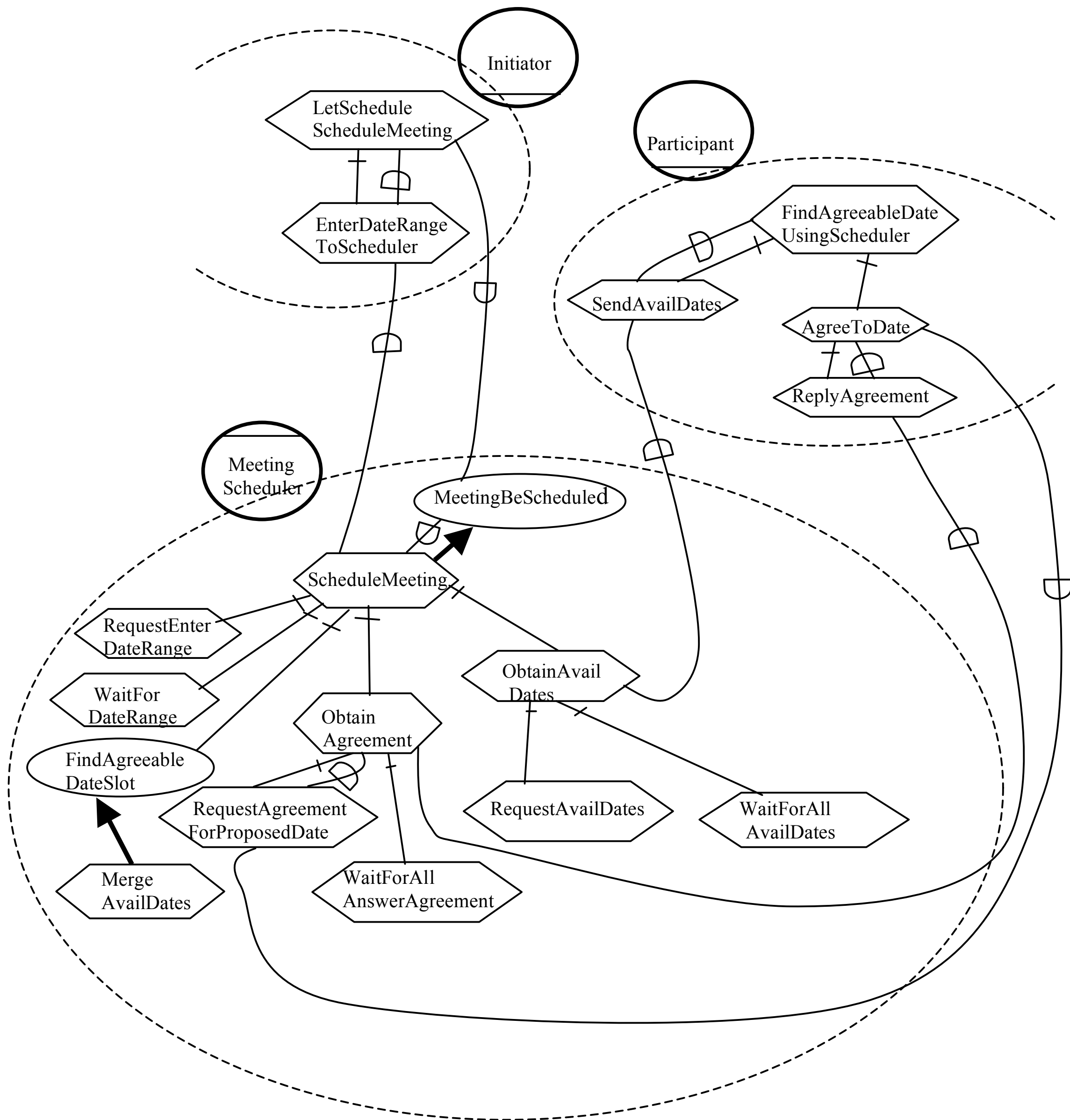


Figure 5.9(b) The SR diagram for the MS after operationalizing dependencies.

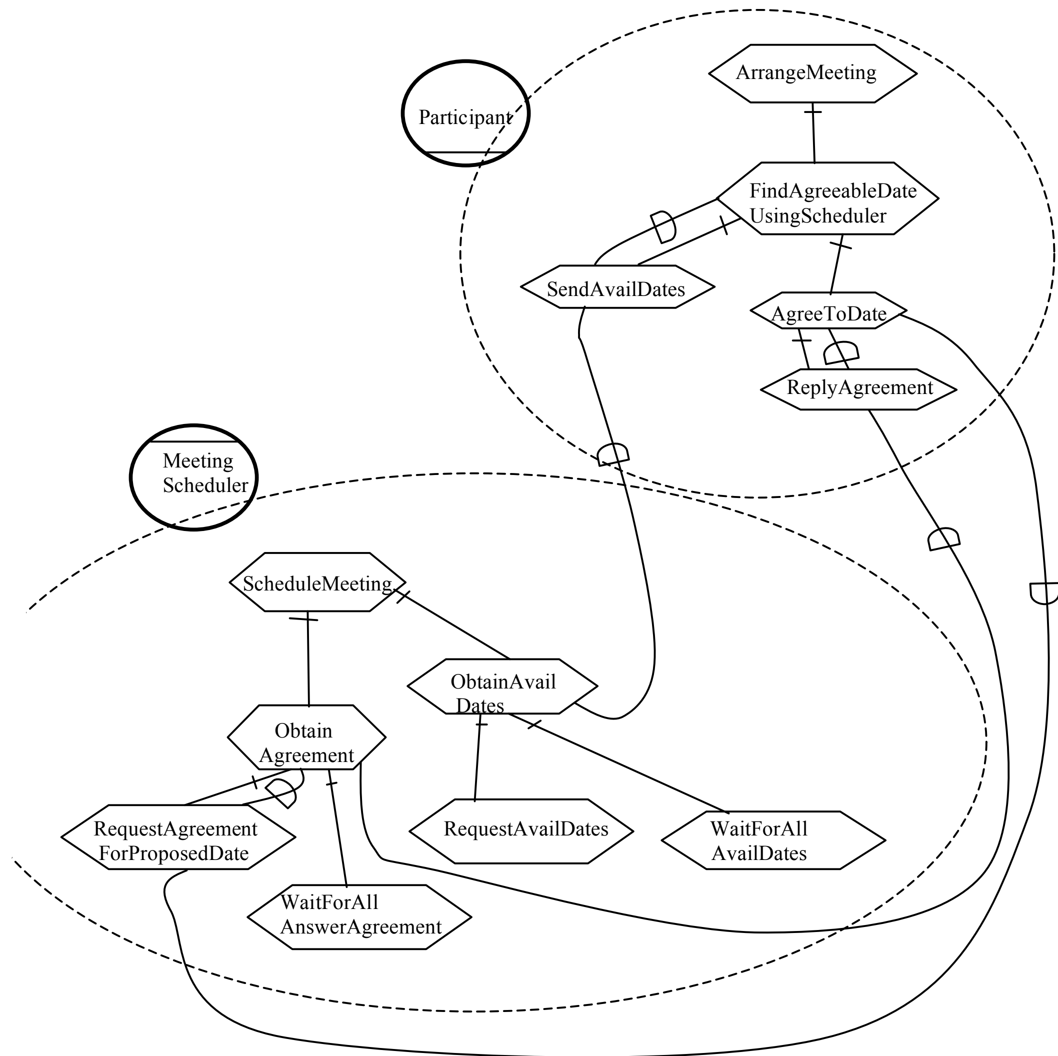


Figure 5.9(c) The SR diagram for the participant after operationalizing dependencies.

5.3.3 Relativizing Goals that Cannot Always Be Achieved

In the next step, goals that cannot always be achieved by the actors are refined into weaker goals that can be achieved. The decompositions related to these goals are modified as appropriate.

Inside the actor `Initiator`, the goal `MeetingBeenScheduled` is achievable when the MS can successfully schedule the meeting. But there are cases where the meeting cannot be scheduled, because there is no meeting date suitable for all participants. If we don't rule out this kind of situation, we have to reformulate the goal into the weaker one `MeetingBeenScheduledIfPossible`. This goal is considered to have been achieved when an attempt to schedule the meeting has been made, successful or not. The super-task `OrganizeMeeting` of this goal is refined into `TryOrganizeMeeting`, which means that the initiator is trying to organize a meeting and the attempt can succeed or fail. The mean `LetSchedulerScheduleMeeting` for the goal remains as before.

Inside the MS, the goal `MeetingBeScheduled` is achievable when the MS successfully finds a suitable date for all participants to attend the meeting. But when there is no date suitable for all participants, the meeting cannot be scheduled and the goal fails to be achieved. So we have to weaken the goal as `MeetingBeScheduledIfPossible`, which is achieved once an attempt to schedule the meeting has been made. The means of achieving this goal is refined into `TryScheduleAMeeting`, allowing for failure. These changes affect some activities of the participants too. The task `ArrangeMeeting` is reformulated into `TryArrangeMeetings` and the task `FindAgreeableDateUsingScheduler` is refined into `TryFindAgreeableDateUsingScheduler`.

After relativizing the goals that cannot be always achievable into weaker goals that can always be achieved, the process specified for scheduling a meeting is more realistic. The SR diagram incorporating these changes appears in Figure 5.10 in the next section.

5.3.4 Filling out Process Details using Decompositions and Annotations

The SR model of Figure 5.9 doesn't show much detail about the process of scheduling meetings. For example, when a meeting cannot be scheduled, the scheduler has to inform

the participants and initiator of this. Also some tasks/goals have to be performed repeatedly or conditionally. For example, if the MS requests all participants to agree to meet on a proposed date and one of the participant rejects the request, then the scheduler has to make a request to cancel the agreement on the proposed date to all the participants who have already agreed. None of this is shown in Figure 5.9. The next step in our RE methodology is to fill out these details by further decomposing the tasks and using annotations to specify control information. The modeler has to analyze every task/goal inside actors to determine how it will be performed. He has to specify in enough detail how the system completes its tasks/goals step by step (in the selected alternative), including what are the conditions to perform tasks/achieve goals, whether the tasks/goals has to be performed repeatedly, etc. Once this has been done, the process can be specified formally in *ConGolog* and the specification can be validated and verified.

An informal description of the selected alternative for the meeting scheduling process was given at the beginning of the chapter. Here we refine the SR diagrams of Figure 5.9 to capture the details of this alternative, resulting in annotated SR diagrams. Let us describe these.

We start with the initiator role. The annotated *i** SR diagram for it appears in Figure 5.10 (a).

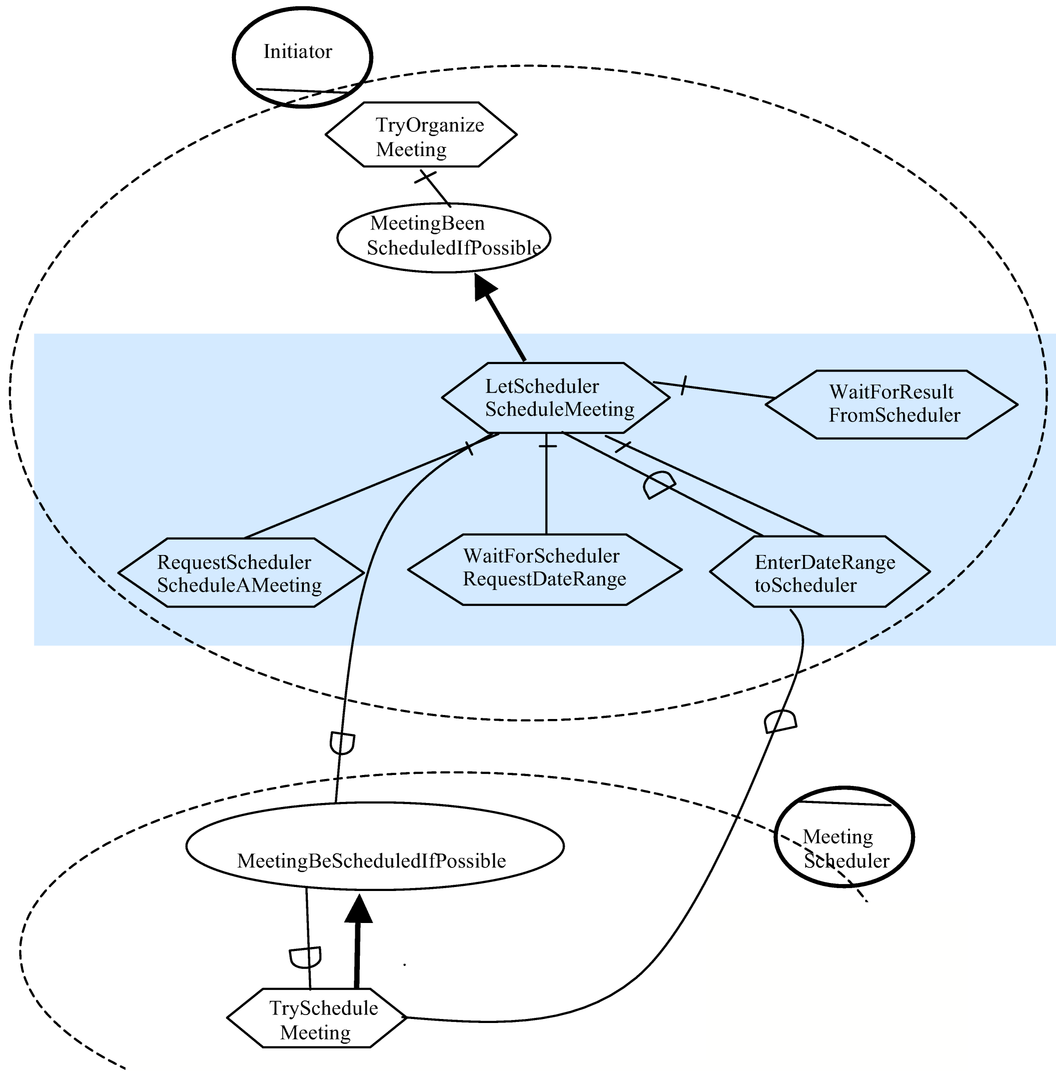


Figure 5.10(a) The annotated SR diagram for the meeting initiator.

The diagram is similar to the diagram Figure 5.9(a), with the top task `TryOrganizeMeeting` and goal `MeetingBeenScheduledIfPossible` relativized as explained in the previous section. The goal `MeetingBeenScheduledIfPossible` is achieved by the mean `LetSchedulerScheduleMeeting`, meaning that the initiator uses the scheduler to

schedule a meeting. The task LetSchedulerScheduleMeeting is decomposed into four subtasks: RequestSchedulerScheduleAMeeting, WaitForSchedulerRequestDateRange, SendDateRangeToScheduler, and WaitForResultFromScheduler. These subtasks are to be performed sequentially, so we can use the default composition annotation ";" (sequence) for the group of four decomposition links. There is no link annotation on single decomposition links because every subtask will be performed exactly once to accomplish the super-task LetSchedulerScheduleMeeting.

Next, let us look at the annotated SR diagram for the MS agent, which appears in Figure 5.10(b).

First, we observe that the MS's role is to try to schedule many meetings, not one, so we add a top task TryScheduleMeetings for the MS. Then, whenever the MS receives a request to schedule a meeting from an initiator, it will want to achieve the goal MeetingBeScheduledIfPossible. The task TryScheduleAMeeting will be the mean used to achieve the internal goal MeetingBeScheduledIfPossible. We use the link annotation *whenever(requestedSchedulingAMeeting) on the link connecting the top task TryScheduleMeetings and the subgoal MeetingBeScheduledIfPossible because only when a request to schedule a meeting has been made, will the subgoal be required to be achieved.

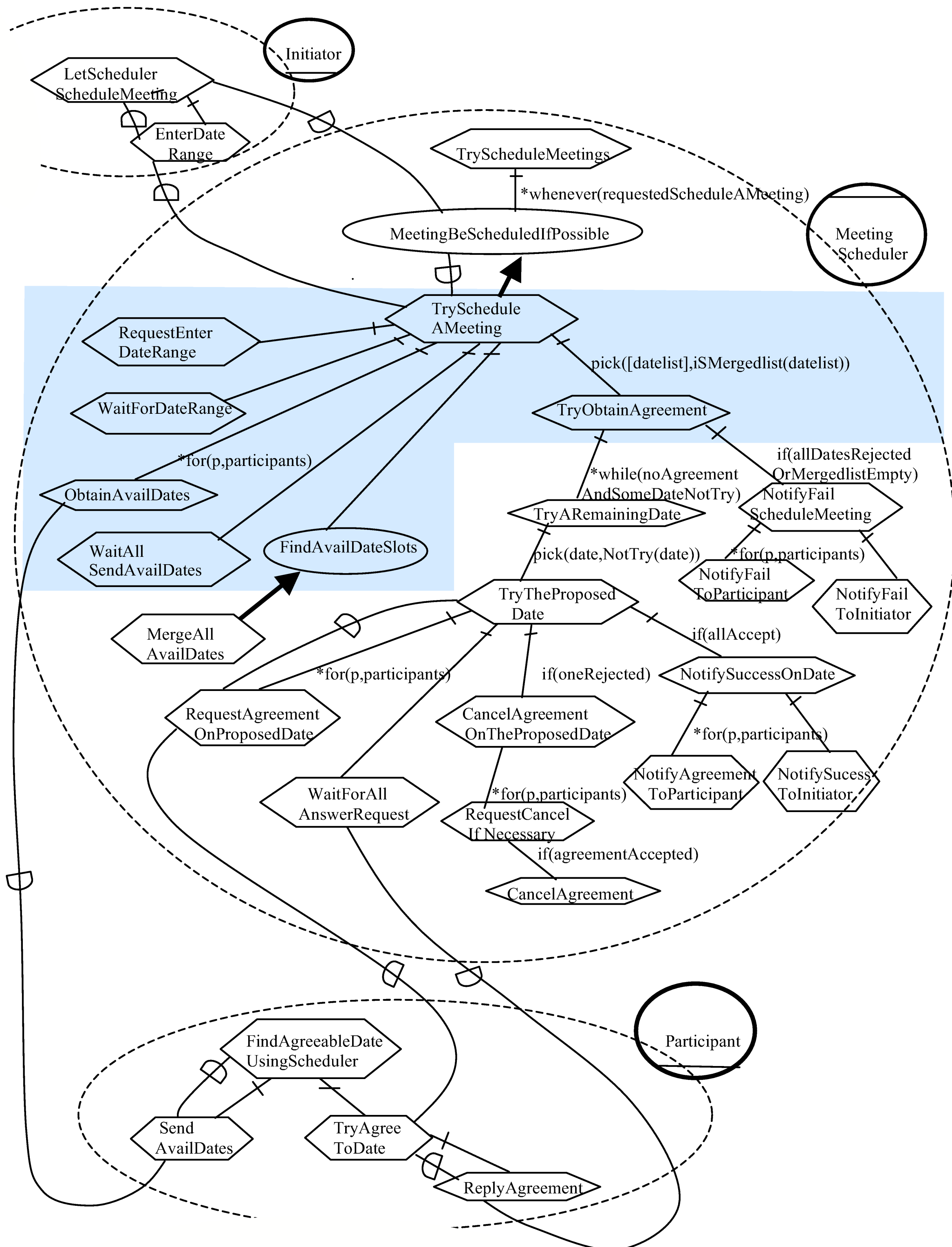


Figure 5.10(b) The annotated SR diagram for the MS.

To perform the task TryScheduleAMeeting, the MS first requests the meeting date range from the initiator, a subtask RequestEnterDateRange, then he waits for the date range to be entered by the initiator, a subtask WaitForDateRange, then he requests all the participants to send their available dates, a subtask ObtainAvailDates, then he waits for all participants to have sent their available dates, a subtask WaitForAllAvailableDates, then he achieves the goal FindAgreeableDateSlot by merging the available dates of participants and the proposed meeting date range of the initiator (we change the goal's name from FindAgreeableDateSlot to FindAvailDateSlots because there may be several dates that are available for all participants), and then he tries to obtain an agreement from all participants on a date from the merged available dates, a subtask TryObtainAgreement. A link annotation pick(datelist,isMergedList(datelist)) accompanies the link connecting the task TryScheduleAMeeting to the subtask TryObtainAgreement, meaning that the MS uses the current merged date list to try to obtain agreement from all participants (this is only to bind the datelist parameter in the subtask TryObtainAgreement). The subtask ObtainAvailDates must be iterated for all the participants, so a link annotation *for(p,participants) accompanies the link connecting it to the task TryScheduleAMeeting. These decomposed subtasks/subgoals of the task TryScheduleAMeeting are performed sequentially and the default composition annotation, i.e., sequence ";", is applied to this group of decomposed links.

The task of trying to obtain agreement on a date, TryObtainAgreement, is decomposed as follows: while there is no agreement and there are still some dates in the merged datelist that have not been tried, then the scheduler tries to obtain an agreement by nondeterministically picking a not-tried date from the datelist. This sub-process is modeled as the subtask TryARemainingDate. If all dates in the

merged date list have been rejected by the participants or the merged date list is empty, then the scheduler notifies the initiator and all the participants that the meeting scheduling has failed; this sub-process is modeled as the subtask `NotifyFailScheduleMeeting`. A link annotation `*while(noAgreementAndSomeDateNotTry)` accompanies the link connecting the task `TryObtainAgreement` to the subtask `TryARemainingDate`, representing the while loop that executes the subtask `TryARemainingDate` for all the possible dates. The link annotation `if(AllDatesRejectedOrMergedlistEmpty)` accompanies the link connecting the task `TryObtainAgreement` to the subtask `NotifyFailScheduleMeeting`, meaning that the subtask is only done when this condition holds. The two subtasks of the task `TryObtainAgreement` are performed sequentially, so the default composition annotation, sequence ";", is applied to the group of these decomposed links.

The task `NotifyFailSchedulingMeeting`, where the scheduler notifies the participants and initiator about his failure to schedule a meeting, is composed into two subtasks: a subtask of notifying a participant of the failure, `NotifyFailToParticipant`, which is performed for all participants as indicated by the link annotation `*for(p,participants)`, and a subtask of notifying the initiator of the failure, `NotifyFailToInitiator`. These two subtasks are performed sequentially and the default composition annotation, sequence ";", is applied to the group of links.

The task `TryARemainingDate` is decomposed as follows: the subtask `TryTheProposedDate` where a date is one be picked from the untried dates; a link annotation `pick(date,notTry(date))` accompanies the link connecting the task `TryARemainingDate` to the subtask `TryTheProposedDate`. The task `TryTheProposedDate` is decomposed into a more detailed subprocess consisting of:

first, requesting agreement on the proposed date from all participants, modeled as the subtask RequestAgreementOnProposedDate, which is accompanied by a link annotation *for(p,participants), meaning that this subtask is to be repeated to all participants; then waiting for all participants to answer the request for an agreement on the proposed date, modeled as the subtask WaitForAllAnswerRequest; then canceling the request for an agreement on the date if one of the participant rejects the agreement, modeled as the subtask CancelAgreementOnProposedDate, which is accompanied by a link annotation if(oneReject); and finally notifying the actors of the agreement on the proposed date if all participants agree to meet on that date, modeled as the subtask NotifySuccessOnDate, which is accompanied with a link annotation if(allAccept). These four subtasks of the task TryTheProposedDate are performed sequentially, so we use the default composition link, sequence ";", on the group of decomposition links.

The subtask CancelAgreementOnProposedDate will request the cancellation of the agreement on the proposed date to all participants who have accepted this agreement. It is decomposed into the subtask RequestCancelIfNecessary, which is accompanied by a link annotation *for(p,participants). RequestCancelIfNecessary is decomposed into the subtask requestCancel accompanied by a link annotation if(agreementAccepted), meaning that only when the participant has accepted the agreement, does the MS need to request the cancellation.

The subtask NotifySucessOnDate is decomposed as follows: notifying all participants of the success of scheduling a meeting on a given date, modeled as the subtask NotifyAgreementToParticipant accompanied by a link annotation *for(p,participants), and notifying the initiator of the success, modeled as the

subtask `NotifySuccessToInitiator`. These two subtasks are performed sequentially, so the sequence annotation “;” is applied to the group of decomposed links.

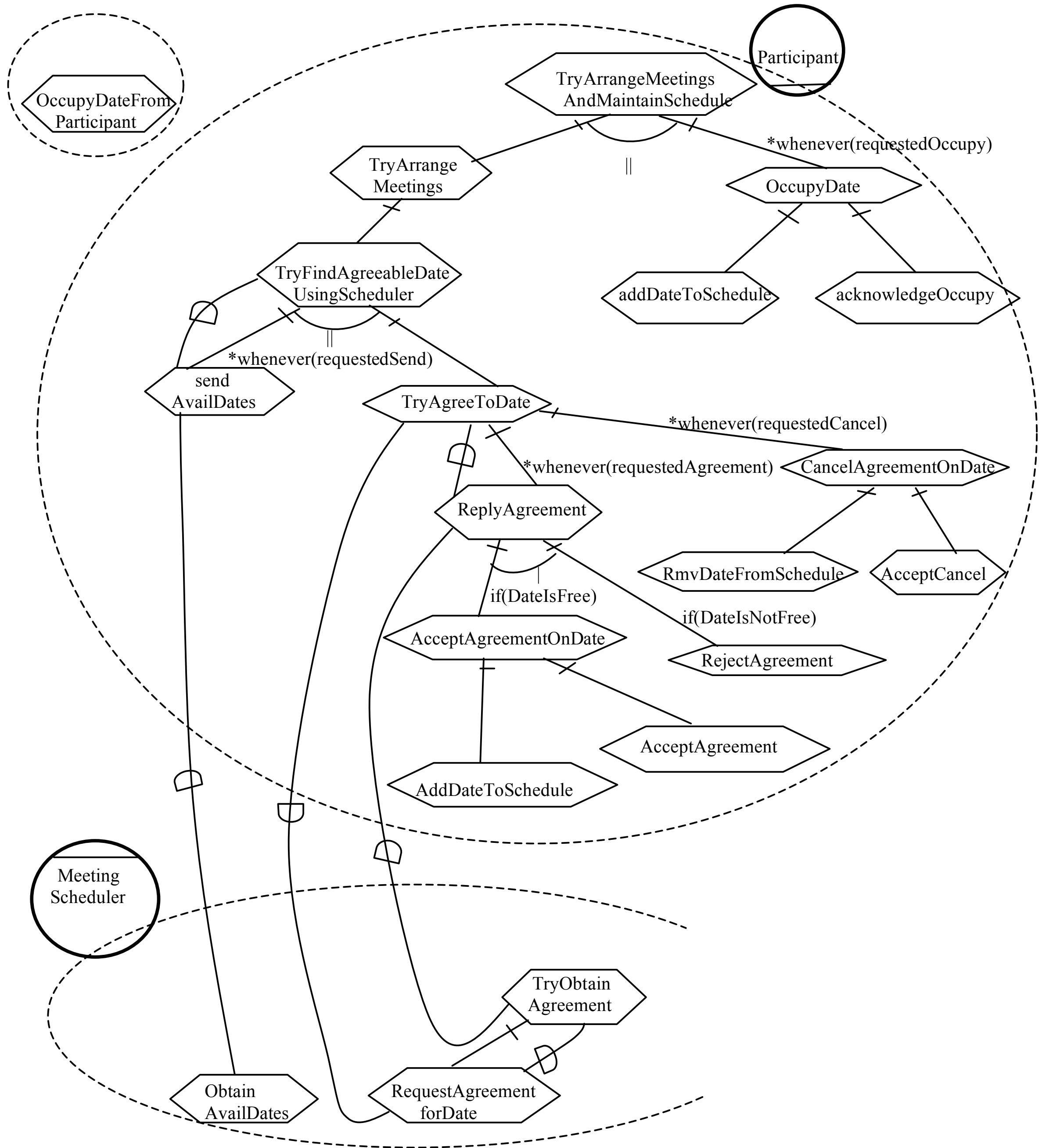


Figure 5.10(c) The annotated SR diagram for the participant.

Finally, let us look at the annotated SR diagram for the participant roles, which appears in Figure 5.10(c).

The participant's role in scheduling meetings is to passively answer requests from other actors and maintain his time schedule. We add a top task `TryArrangeMeetingsAndMaintainSchedule` to model this. This task is decomposed into two subprocesses: a subtask of trying to arrange meetings `TryArrangeMeetings`, and a subtask of reserving/occupying a date on his schedule when requested by an outside actor, modeled as `OccupyDate` with a link annotation `*whenever(requestedOccupy)`. The two subprocesses are performed concurrently, so a concurrency composition annotation "||" is applied to the group of links.

The task `TryArrangeMeetings` can be completed by performing a subtask of finding an agreeable meeting date using the scheduler, where all the participant has to do is to passively answer requests from the scheduler; this is modeled as a subtask `TryFindAgreeableDateUsingScheduler`. This task involves processing requests to obtain available dates, requests to obtain agreement to meet on a proposed date, and requests to cancel an accepted agreement to meet on a proposed date. So `TryFindAgreeableDateUsingScheduler` is decomposed into two subtasks: a subtask of sending the available dates to the scheduler, `SendAvailDates`, with a link annotation `*whenever(requestedSend)`, and a subtask of processing requests regarding meeting date agreements, `TryAgreeToDate`. The two subtasks are performed concurrently, so a concurrency composition annotation "||" is applied to the group of decomposition links. The task `TryAgreeToDate` is decomposed into two subtasks: a subtask of replying to a request for agreement on a proposed date when one is received from the scheduler, modeled as the task `ReplyAgreement`, with a link

annotation *whenever(requestedAgreement), and a subtask of replying to a request to cancel an agreement on a proposed date when one is received, modeled as the subtask CancelAgreementOnDate with a link annotation *whenever(requestedCancel). The two subtasks are performed concurrently, so a concurrency composition annotation "||" is applied to the group of decomposition links.

The task ReplyAgreement will either accept or reject the request to meet on a proposed date from the scheduler. It is decomposed into two subtasks: a subtask of accepting the proposed date if the date is free on the participant's time schedule, modeled as AcceptAgreementOnDate with a link annotation if(dateIsFree), and a subtask of rejecting the proposed date if the date has been occupied on the participant's time schedule, modeled as the task RejectAgreement with a link annotation if(dateIsNotFree). The two subtasks are alternatives, so the alternative composition annotation "|" is applied to the group of decomposition links.

The tasks AcceptAgreementOnDate and CancelAgreementOnDate both involve updating the participant's time schedule and then notifying the scheduler of the agreement or acknowledging the cancellation. So the task AcceptAgreementOnDate is decomposed into two subtasks: a subtask AddDateToSchedule which adds the proposed date to the participant's time schedule, and a subtask AcceptAgreement which notifies the scheduler that the participant accepts the proposed agreement. These two subtasks have to be performed sequentially because the participant has to make sure his time schedule has been updated before he tells the scheduler that he accepts the agreement. So the default (sequence) composition annotation ";" is applied to the group of decomposition links. The task CancelAgreementOnDate is decomposed in a similar way into two subtasks: RmvDateFromSchedule, which removes the meeting date from the participant's time schedule, and AcceptCancel which notifies the scheduler that the participant received his cancellation. These two subtasks have to be performed sequentially because the participant has to make sure his time schedule is

updated before he acknowledge the request of cancellation to the scheduler. So a default (sequence) composition annotation ";" is applied to the group of decomposition links. Note that we need to make sure that the *ConGolog* code on accepting agreement cannot deadlock. The test that the date is free and the `addDateToSchedule` action must be done as a single transition, so that an `occupyDate` cannot happen between them.

The shadowed areas in Figure 5.10 (a) and (b) represent the decompositions of important tasks inside actors. We will use these task decompositions later to explain how the task decompositions can be mapped into the *ConGolog* elements.

5.4 Developing the Initial *ConGolog* Model

Here, the modeler must map entities in the annotated SR diagram into corresponding elements of a *ConGolog* model and complete the development of the *ConGolog* model. We will give a detailed description of how the mapping rules are applied to the entities in the annotated SR models of Figure 5.10 and how the actions, action precondition axioms, successor state axioms, and the initial state axioms are specified in order to build the complete *ConGolog* model.

5.4.1 The Initial *ConGolog* Model for `Initiator`

By applying the mapping rules the elements of the annotated SR diagram of Figure 5.10(a) for the initiator, we obtain the part of the initial *ConGolog* model that specifies the behavior of the initiator, which is shown in Figure 5.11.

```
proc(initiator_behavior(Init,MS),
     tryOrganizeMeeting(Init,MS,peoplelist,datelist)
).

proc(tryOrganizeMeeting(Init,MS,Peoplelist,Datelist),
    achieve_meetingBeenScheduledIfPossible(Init,MS,Peoplelist,Datelist)
).

proc(achieve_meetingBeenScheduledIfPossible(Init,MS,PList,Datelist),
    [  letSchedulerScheduleMeeting(Init,MS,PList,Datelist),
       meetingBeenScheduledIfPossible(Init,MS,PList,Datelist)?
    ]
).

proc(letSchedulerScheduleMeeting(Init,MS,PList,Datelist),
    [
       requestScheduleMeeting(Init,MS,PList),
       waitForSchedulerRequestDateRange(Init,MS,PList,Datelist)?,
       enterDateRangeToScheduler(Init,MS,PList,Datelist),
       waitForSchedulingResultFromScheduler(Init,MS,PList,Datelist)?
    ]
).

proc(enterDateRangeToScheduler(Init,MS,Peoplelist,Datelist),
     pi([meetingID],[
       and(val(skedPeoplelist(meetingID),Peoplelist),
          and(requestedEnterDateRange(meetingID),
             not(dateRangeEntered(meetingID))
     ))?,
     enterDateRange(Init,MS,meetingID,Datelist)  ])
  ).
```

Figure 5.11 The initial *ConGolog* model for the initiator

Let us explain how the mapping is done in detail. First nodes in the SR diagram of Figure 5.10(a) are mapped. There are three types of nodes in Figure 5.10(a) are to be mapped: a role node, a goal node, and some task nodes. Then the decomposition links are mapped. There are two types of decomposition links are mapped: task-decomposition links and goal-decomposition links.

In the annotated SR diagram of Figure 5.10(a), the node `Initiator` is a role node. According to the mapping rule for roles, the role node `Initiator` is mapped into the

ConGolog procedure `initiator_behavior` that specifies that the initiator's behavior is to try to organize meetings. The mapping is shown in Figure 5.12. The `Init` parameter is to be filled by a term that denotes an agent playing the initiator role and the `MS` parameter by a term that denotes its acquaintance, the meeting scheduler agent.

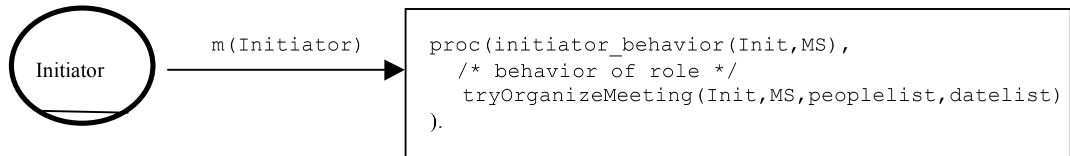


Figure 5.12 The mapping for the role node `Initiator`

In the SR diagram of Figure 5.10(a), there is a goal node `MeetingBeenScheduledIfPossible` inside the `Initiator` role. According to the mapping rule for goals, this node can be mapped into the *ConGolog* procedure `achieve_MeetingBeenScheduledIfPossible` and defined fluent `meetingBeenScheduledIfPossible`. The procedure `achieve_MeetingBeenScheduledIfPossible` contains the mean to achieve the goal `MeetingBeenScheduledIfPossible` and has the post-condition that the fluent holds, i.e., its body ends with the test `meetingBeenScheduledIfPossible?`. Figure 5.13 shows the mapping.

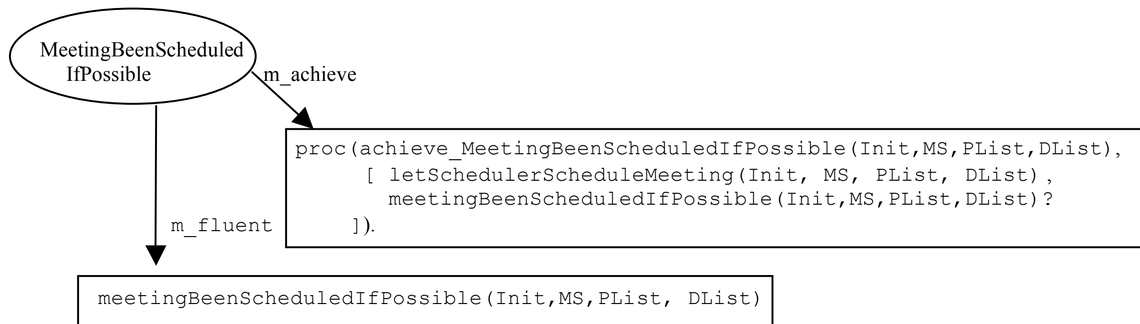


Figure 5.13 The mapping for the goal node `MeetingBeenScheduledIfPossible`

In the SR diagrams of Figure 5.10(a), there are task nodes TryOrganizeMeeting, LetSchedulerScheduleMeeting, RequestScheduleAMeeting, WaitForSchedulerRequestDateRange, EnterDateRangeToScheduler, and WaitForSchedulingResultFromScheduler. We take the task nodes TryOrganizeMeeting and LetSchedulerScheduleAMeeting as examples of how the task nodes can be mapped into elements of a *ConGolog* model. According to the mapping rules for tasks, the task node TryOrganizeMeeting is mapped into the *ConGolog* procedure tryOrganizeMeeting and the task node LetSchedulerScheduleAMeeting is mapped into the *ConGolog* procedure letSchedulerScheduleAMeeting. The mappings are shown in Figure 5.14(a) and (b) respectively.

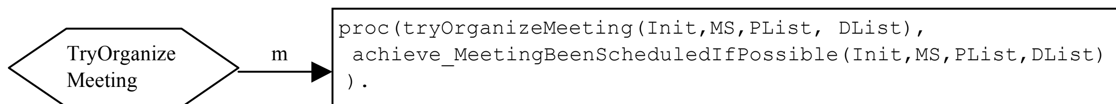


Figure 5.14(a) The mapping for the task node TryScheduleMeeting

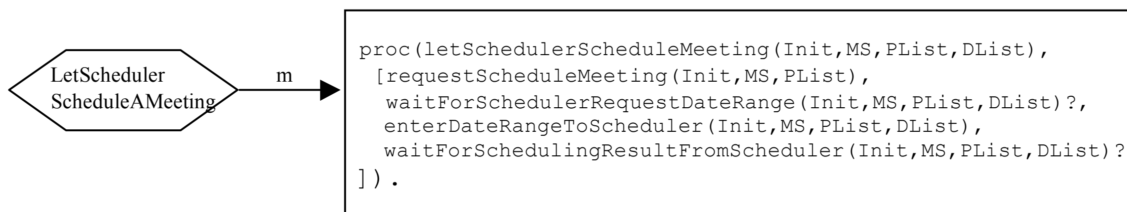


Figure 5.14(b) The mapping for the task node LetSchedulerScheduleAMeeting

Now we explain how task-decomposition links are mapped. In the shadowed rectangle area in the annotated SR diagram of Figure 5.10(a) of Initiator, the task node LetSchedulerScheduleAMeeting is decomposed into four subtasks: RequestSchedulerScheduleMeeting, WaitForSchedulerRequestDateRange, EnterDateRangeToScheduler, and WaitForResultFromScheduler. The default composition annotation, i.e.,

sequence ";", is applied to this group of decomposition links. No link annotation is applied to any single decomposition links which means that every subtask is to be performed exactly once. As we can see in Figure 5.14 (b), `LetSchedulerScheduleAMeeting` is mapped into the *ConGolog* procedure `letSchedulerScheduleMeeting`. The body of the procedure `letSchedulerScheduleMeeting` sequentially invokes the *ConGolog* sub-procedures `requestScheduleMeeting`, `waitForSchedulerRequestDateRange`, `enterDateRangeToScheduler`, and `waitForSchedulingResultFromScheduler` without any condition, exactly once.

In the diagram of Figure 5.10(a), the task `TryOrganizeMeeting` is decomposed into a subgoal `MeetingBeenScheduledIfPossible` and no annotation is associated with this decomposition. So according to the mapping rules for task-decompositions, the *ConGolog* procedure `tryOrganizeMeeting`, mapped from the task `TryOrganizeMeeting`, invokes once the *ConGolog* procedure `achieve_meetingBeenScheduledIfPossible` which is mapped from the goal `MeetingBeenScheduledIfPossible`; we can see this mapping in Figure 5.14(a).

Next we explain how goal-decomposition links (i.e., called means-ends links) are mapped. In the annotated SR diagram for the initiator, the goal node `MeetingBeenScheduledIfPossible` is decomposed into a subtask `LetSchedulerScheduleMeeting`, i.e., the subtask is the only mean to achieve the goal `MeetingBeenScheduledIfPossible`. As we can see in Figure 5.13, this goal node is mapped into the defined fluent `meetingBeenScheduledIfPossible` and the *ConGolog* procedure `achieve_MeetingBeenScheduledIfPossible` whose body has the defined fluent `meetingBeenScheduledIfPossible` as its post-condition, i.e., the body ends with a test `meetingBeenScheduledIfPossible?`.

The procedure `achieve_MeetingBeenScheduledIfPossible` invokes the procedure `letSchedulerScheduleAMeeting` that is mapped from the subtask `LetSchedulerScheduleAMeeting`, which means that the subtask `LetSchedulerScheduleAMeeting` is the preferred mean to achieve the goal `MeetingBeenScheduledIfPossible`.

The goal `MeetingBeenScheduledIfPossible` with its decomposition links and the task `LetSchedulerScheduleMeeting` with its decomposition links have already been explained. Putting all these together, we obtain the *ConGolog* behavior specification for the initiator in Figure 5.11. We discuss the specification of the primitive actions and fluents in section 5.4.4.

5.4.2 The Initial *ConGolog* Model for `MeetingScheduler`

The annotated SR diagram of Figure 5.10(b) for the MS is mapped into the initial *ConGolog* model that appears in full in Appendix A-7.

Let us go over the mapping, focusing on cases that have not been discussed already. The node `MeetingScheduler` is an agent. Following to the mapping rule for agent nodes, it is mapped into a *ConGolog* procedure `meetingScheduler_behavior` specifying the behavior of the meeting scheduler agent and a meeting scheduler agent constant `ms1` (this is assigned to parameter `MS` by the main procedure according the system scenario), shown in Figure 5.15. The procedure invokes the top level task of the agent, `tryScheduleMeetings`.

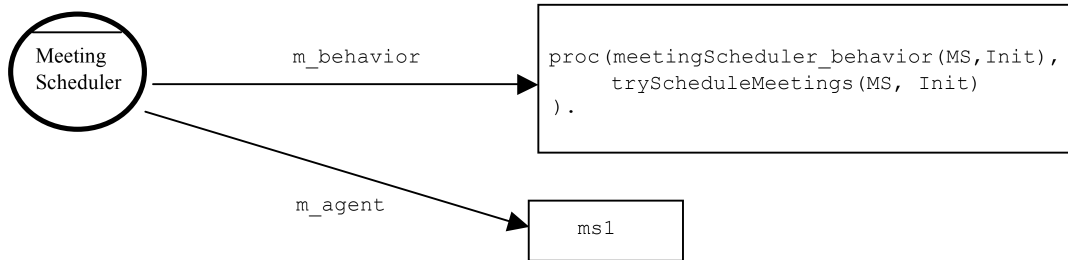


Figure 5.15 The mapping for the agent node `MeetingScheduler`

The top task node `TryScheduleMeetings` is decomposed into a subgoal `MeetingBeScheduledIfPossible`, with the link annotation `*whenever(requestedScheduleAMeeting)` attached to this single decomposition link. Following the mapping rules, the task node is mapped into the following procedure; which invokes the procedure associated with the subgoal `achieve_meetingBeScheduleIfPossible` within an interrupt, the result of mapping the `*whenever` annotation:

```
proc(tryScheduleMeetings(MS,Init),
   ==>([meeting,plist],
        requestedScheduleAMeeting(Init,MS,plist,meeting),
        achieve_MeetingBeScheduledIfPossible(MS,Init,meeting,plist)
      )
).
```

The goal node `MeetingBeScheduledIfPossible` is decomposed into a subtask `TryScheduleAMeeting` without any annotation associated with this decomposition, so it is mapped into the following procedure as for the example of the previous section:

```
proc(achieve_MeetingBeScheduledIfPossible(MS,Init,Meeting,PList),
  [
     tryScheduleAMeeting(MS,Init,Meeting,PList),
     meetingBeScheduledIfPossible(Meeting)?
  ]
).
```

Note how the procedure has the fluent associated with the goal as a post-condition, a final test action. The goal node MeetingBeScheduledIfPossible is decomposed into a task node TryScheduleAMeeting. Accordingly, the above procedure calls the procedure associated with this task.

The task node tryScheduleAMeeting is decomposed into the following subtasks/subgoals: the subtask RequestEnterDateRange, the subtask WaitForDateRange, the subtask ObtainAvailDates, the subtask WaitAllParticipantsSendAvaildDates, the subgoal FindAvailDateSlots, and the subtask TryObtainAgreement. The default composition annotation, sequence ";", is applied this group of decomposition links. The link annotation *for(p,participants) is attached to the link between the subtask ObtainAvailDates and the super-task TryScheduleAMeeting, which means that the MS obtains the available dates from every member p in the list participants. The link annotation pick(datelist,isMergedlist(datelist)) is attached to the link between the subtask ObtainAgreement and the super-task TryScheduleAMeeting, which means that the MS nondeterministically picks up a datelist which is the merged available date list for the proposed meeting (isMergedlist(datelist)), and tries to obtain agreement for a meeting based on this merged list. No link annotation is attached to any other single decomposition links, which means these associated subtasks/subgoals are to be performed exactly once. According to the mapping rules for the task decomposition link, the task node TryScheduleAMeeting with its decomposition links will be mapped into a *ConGolog* procedure tryScheduleAMeeting whose body invokes the mapping results of the subtasks/subgoals sequentially according to the conditions associated with the link annotations. The resulting procedure is shown below.

```
proc(tryScheduleAMeeting(MS,Init,MeetingID,Peoplelist),
  [
     requestEnterDateRange(MS,Init,MeetingID),
     some(datelist,enteredDateRange(MeetingID,datelist))?,
     for(participant,Peoplelist,[],
           obtainAvailDates(MS,participant,MeetingID)
     ),
     waitForAllParticipantSendAvailDates(MeetingID,Peoplelist)?,
     achieve_findAvailDateSlot(MS,MeetingID,Peoplelist),
     pi(xlist,[
        val(allmergedlist(MeetingID),xlist)?,
        tryObtainAgreement(MS,Init,MeetingID,Peoplelist,xlist)
     ])
  ] ).
```

For the goal node `FindAvailDateSlots` and its subtask node `MergeAllAvailDates`, the mapping result is shown as follows:

```
proc(achieve_findAvailDateSlot(MS,MeetingID,Peoplelist),
     [
       mergeAllAvailDates(MS,MeetingID,Peoplelist),
       findAvailDateSlot(MeetingID)?
     ]).

proc(mergeAllAvailDates(MS,MeetingID,Peoplelist),
    pi([datelist],[
        enteredDateRange(MeetingID,datelist)?,
        mergeAll(MS,MeetingID,Peoplelist,datelist)
    ])
).
```

The task node `MergeAllAvailDates` is not decomposed in the annotated SR because the modeler didn't want to get into the details of how to manipulate the date lists. But in the *ConGolog* model, we specify this to obtain an executable model for simulation. In the body of the procedure `MergeAllAvailDates`, the date lists to be merged are bound to `datelist` by the `pi` operator provided by *ConGolog* and then the procedure `mergeAll(MS,MeetingID,Peoplelist,datelist)` is called to recursively merge them. This procedure is defined as follows:

```
proc(mergeAll(MS,MEETINGID,PEOPLELIST,TLIST),
   if(PEOPLELIST=[],
        setAllMergedlist(MS,MEETINGID,TLIST),
        [  pi([f,r],[
               PEOPLELIST=[f|r]?,
               pi( [availdate,templist],
                    [sentAvailDates(MEETINGID,f,availdate)?,
                     interSectionlist(availdate,TLIST,templist)?,
                     mergeAll(MS,MEETINGID,r,templist)
               ])
           ])
        ]
    )
 ).
```

The task `TryObtainAgreement` is decomposed into two subtasks `TryARemainingDate` and `NotifyFailScheduleMeeting`. The default sequence annotation ";" is applied the group of decomposition links. The link annotation `*while(noAgreementAndSomeDateNotTry)` is associated with the link to subtask `TryARemainingDate` and the link annotation `if(allDatesRejectedOrMergedlistEmpty)` is associated with the link to subtask `NotifyFailScheduleMeeting`. The mapping result is as following:

```
proc(tryObtainAgreement(MS,Init,Meeting,PList,Xlist),
  [  while(and(someDateNotTryAndNoAgreement(Meeting,PList,Xlist),
                                                 not(Xlist=[])),
       tryARemainingDate(MS,Init,Meeting,PList,Xlist)
     ),
     if( or(allRejected(Meeting,PList,Xlist),Xlist=[]),
          notifyFailScheduleMeeting(MS,Init,Meeting,PList)
     )
  ]
).
```

The rest of the *ConGolog* model for the meeting scheduler's behavior contains few new features and we will not discuss it. See Appendix A-7 for the specification. We also do not discuss the *ConGolog* model for `Participant`; see Appendix A-1 for the specification.

5.4.3 Specifying the Domain Dynamics

As we are mapping elements of the annotated SR diagram into the *ConGolog* model's process specification, we must also begin to specify the domain dynamics. Primitive actions and fluents are introduced to model aspects of the domain. Precondition axioms and successor state axioms are also specified to model when the actions can be performed and how they affect the fluents. Let us first list the primitive actions, exogenous actions, and fluents we use to model this domain:

- **Primitive actions**

`requestSchedulerScheduleAMeeting(Init,MS,ParticipantList)`

/* `Init` asks `MS` to schedule a meeting with all members in `ParticipantList` */

`requestEnterDateRange(MS,Init,Meeting)`

/* `MS` requests `Init` to enter the date range for `Meeting` */

`enterDateRange(Init,MS,Meeting,Datelist)`

/* `Init` enters `MS` the date range `Datelist` for `Meeting` */

`obtainAvailDates(MS,Participant,Meeting)`

/* `MS` requests the available dates from `Participant` for `Meeting` */

`sendAvailDates(Participant,MS,ReqID,Tlist)`

/* `Participant` sends `MS` his `AvailableDates` regarding `MS`'s request `ReqID` */

`requestAgreement(MS,Participant,Meeting,Date)`

/* `MS` requests `Participant`'s agreement on the proposed `Date` for `Meeting` */

`acceptAgreement(Participant,MS,ReqID,Date)`

/* `Participant` accepts `MS`'s request `ReqID` for agreement to meet on `Date` */

`rejectAgreement(Participant,MS,ReqID,Date)`

/* `Participant` rejects `MS`'s request `ReqID` for agreement to meet on `Date` */

`cancelAgreement(MS,Participant,Meeting,Date)`

/* `MS` requests `Participant` to cancel agreement on `Meeting` on `Date` */

`acceptCancel(Participant,MS,ReqID,Date)`

/* `Participant` accepts `MS`'s request `ReqID` for canceling the meeting on `Date` */

`notifyAgreement(MS,Participant,Meeting,Date)`

/* `MS` notifies `Participant` of the agreement to have `Meeting` on `Date` */

`notifySuccess(MS,Init,Meeting,Participants,Date)`

/* MS notifies Init that it has successfully scheduled Meeting */
/* with all Participants on Date */

notifyFail(MS,Participant,Meeting, Participants)

/* MS notifies Participant that it has failed to schedule Meeting with Participants. */

setAllmergedlist(MS,Meeting,Dlist)

/* MS memorizes Dlist as the merged list of all available dates of all participants for Meeting */

addDateToSchedule(Participant,Date)

/* Participant adds Date into his schedule */

rmvDateFromSchedule(Participant,Date)

/* Participant removes Date from his schedule */

- **Exogenous Actions**

occupyDateFromParticipant(Participant,Date).
/* An unspecified agent outside the system commands Participant */
/* to occupying Date on his schedule. */

- **Predicate Fluents**

requestedSchedulerSkeduleAMeeting(Init,MS,Meeting)
/ * Init has requested MS to schedule Meeting */

requestedEnterDateRange(MS,Init,Meeting)
/* MS has requested Init to enter the date range for Meeting */

enteredDateRange(Init,MS,Meeting,Datelist)
/* Init has entered the data range Datelist for Meeting to MS */

submittedobtain(MS,P,Meeting)
/* MS has requested participant P for his available dates for Meeting */

sentAvailDates(P,MS,Meeting,AvailDates)
/* Participant P sent MS his available date AvailDates for Meeting */

submittedAgreement(MS,P,Meeting,Date)
/* MS has requested participant P to agree to have Meeting on Date */

agreementAccepted(P,MS,Meeting,Date)
/* Participant P has accepted to have Meeting on Date as proposed by MS */

agreementRejected(P,MS,Meeting,Date)
/* Participant P has rejected to have Meeting on Date as proposed by MS */

waitingForAgreeAns(MS,P,Meeting, Date)
/* MS is waiting for participant P to agree to have Meeting on Date */

submittedCancel(MeetingID,P,Date)
/* MS has requested participant P to cancel Meeting on Date */

cancelAccepted(P,MS,Meeting,Date)
/* Participant P has acknowledged the cancellation of Meeting on Date by MS */

agreementReqRcvd(P,ReqID,Date)

/* P has received a request ReqID for agreement to have Meeting on Date */
cancelReqRcvd(P,ReqID,Date)
/* Participant P received a request ReqID for canceling meeting on Date */
cancelReqProc(P,ReqID,Date)
/* Participant P has processed the request ReqID for canceling Meeting on Date */
agreementNotified(MS,P,Meeting,Date)
/* MS has notified participant P all participants agree Meeting on Date */
successNotified(MS,Init,Meeting,Participants,Dlist)
/* MS has notified Init of the successfully scheduling of Meeting */
/* on Dlist with Participants */
failNotified(MS,Init,Meeting,Plist,Dlist)
/* MS has notified Init of his failure to schedule Meeting on Dlist with Participants */
participantDateOccupied(Participant,Date)
/* Date is occupied Participant's schedule */

Predicate fluents generally model the fact that a primitive action or exogenous action in the scheduling process has occurred. For example: the fluent requestedSchedulerSkeduleAMeeting(Init,MS,Meeting) models that fact that a request from the scheduler MS to schedule Meeting has been made by the initiator Init (the primitive action requestSchedulerScheduleAMeeting). Similarly, the fluent submittedObtainAvailableDates(MS,Participant,meeting) models the fact that the MS has made a request to obtain Partcipant's available dates for Meeting (the primitive action obtainAvailDates).

- **Functional Fluents**

allmergedlist(Meeting)
/* Denotes the merged available date list for Meeting */
participantSchedule(Participant
/* Denotes Participant's time schedule (list of busy dates) */
dateRange(Meeting)
/* Denotes the meeting date range for Meeting */
participants(Meeting)
/* Denotes the list of participants for Meeting */
reqParticipant(ReqID)
/* Denotes the participant who the request ReqID was made to */
reqDate(ReqID)

/ * Denotes the date associated with the request `ReqID` made to any participant */

```
reqDlist(ReqID)
```

/* Denotes the date list associated with the request `ReqID` made to any participant */

```
reqMeetingID(ReqID)
```

/* Denotes the meeting associated with the request `ReqID` made to any participant */

```
MeetingCtr
```

/* A counter used by MS to assign meeting IDs to meeting scheduling requests */

```
reqctr
```

/* A counter for assigning IDs to requests made to any participant */

We also have various defined fluents. For example, `AgreemeentAnswered(MS,Participant,Date,Meeting)` models the fact that the `MS` is waiting for `Participant` to agree to the `Meeting` on the `Date`. It is defined in *Prolog* as follows:

```
holds(agreementAnswered(MS,Participant,Date,Meeting),S):-
      holds(agreementAccepted(MS,Participant,Date,meeting),S);
      holds(agreementRejected(MS,Participant,Date,Meeting),S).
```

This means that the defined fluent is true if and only if the `Participant` has accepted or rejected the request to agree to have `Meeting` on `Date`.

We also have `allRejected(MeetingID,Peoplelist,Datelist)`, which models the fact that all dates in `Datelist` proposed for the meeting have been rejected by some of the participants in the `Peoplelist`. It is defined as follows:

```
holds(allRejected(MeetingID,Peoplelist,DateliSt),S):-
      holds(member(Date,Datelist)-->
      oneRejected(MeetingID,Peoplelist,Date),S).
```

A complete list of defined fluents with their definition appears in Appendix A-3.

For each predicate fluent or functional fluent, we specify a successor state axiom that captures how it is affected by the actions in the domain.

For example, the successor axiom for the fluent `occupyAcknowledged(Participant,Date)` is specified as follows:

```
holds(occupyAcknowledged(Participant,Date),do(A,S)):-
      A= acknowledgeOccupy(Participant,Date);
      holds(occupyAcknowledged(Participant,Date), S).
```

This says that a request to occupy `Date` on `Participant`'s schedule has been acknowledged in the situation that is the result of doing action `A` in situation `S` if and only if the action `A` is to acknowledge this occupation, or if the occupation has already been acknowledged in situation `S`.

The successor axiom for the fluent `submittedObtain(MS,Participant,Meeting)` is as follows:

```
holds(submittedObtain(MS,Participant,Meeting),do(A,S)):-
       A = obtainAvailDates(MS,Participant,Meeting);
       holds(submittedobtain(MS,Participant,Meeting),S)
```

This says that a request by `MS` to obtain his available dates from `Participant` has bee made in the situation that is the result of doing action `A` in situation `S` if and only if the action `A` is to make a request by `MS` to obtain the available dates from `Participant`, or if the request has already been made in situation `S`.

The successor axiom for the functional fluent `meetingCtr` is as follows:

```
holds(val(meetingCtr,N),do(A,S)):-
      (A = requestScheduleMeeting(_,_,_),
       holds(val(meetingCtr,M),S), N is M + 1);
      (holds(val(meetingCtr,N),S),
       A \= requestScheduleMeeting(_,_,_)).
```

This says that the meeting counter `meetingCtr` will increase one at the situation that is the result of doing action `A` in situation `S` if and only if the action `A` is to make a request to schedule a meeting otherwise `meetingCtr` remains the same value as that in situation `S`.

All successor state axioms are listed in Appendix A-2.

We also specify an action precondition axiom for each primitive action to indicate the pre-condition to perform an action. For example, the precondition axiom for the action `acceptagreement(Participant, MS, ReqID, Date)` is as follows:

```
poss(acceptAgreement(Participant,MS,ReqID,Date),S):-
     holds(agreementReqRcvd(Participant,MS,ReqID,Date),S).
```

This says that the action can be performed if `Participant` has received a request from `MS` for an agreement to meet on `Date`.

The precondition axiom for the action `occupyDateFromParticipant` is as follows:

```
poss(occupyDateFromParticipant(Participant,Date),S):-
     holds(not(participantDateOccupied(Participant,Date)),S),
     holds(not(agreementAccepted(-,Participant,Date)),S)
```

This says that the action can be performed if `Date` is not occupied from `Participant`'s time schedule and he has not agreed to any meeting on `Date`.

We consider most actions to be always possible in this domain. For example, we have `poss(requestSchedulerScheduleAMeeting(_,_,_),_)` is always possible to be performed. All precondition axioms appears in Appendix A-6.

The last element of the *ConGolog* model is the specification of the initial state. This various depending what sort of scenario one wants to simulate or verify. It will include axioms such as the following:

```
holds(val(participantDateInfo(jeff),[]),s0).
```

This says that `jeff` doesn't have any activities on his time schedule initially.

The complete initial state specification might include the following set of axioms:

```
holds(val(participantDateInfo(paige),[11,12,14]),s0)
        /* Initially paige has meetings on Feb 11, 12, 14*/
holds(val(participantDateInfo(yves),[10,12]),s0)
        /* Initially yves has meetings on Feb 10, 12*/
holds(val(feblist,[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,
                   20,21,22,23,24,25,26,27,28,29]),s0)
        /* Initially the meeting can only be arranged in Feburary */
holds(val(meetingCtr,1),s0)
        /* Initially the value of meeting count is assumed to be l */
holds(val(reqCtr, 1),s0)
        /* Initially the value of request count is assumed to be l */
holds(val(allmergedlist(_),[]),s0)
        /* Initially the merged available date list is assumed to be empty */
```

Appendix A-3 lists all the actions and fluents in the *ConGolog* model of the meeting scheduling process.

5.5 Validating the *ConGolog* Model by Simulation

The modeler can evaluate the *ConGolog* model through simulation. First, we specify an instance of the system and then we run a simulation. By checking and comparing the

results of simulation on different system instances or initial states, we can see whether the system behaves as expected. Unforeseen occurrences may cause revisions in the model.

5.5.1 Specifying a System Instance

As mentioned earlier, complete *ConGolog* models can be executed to run process simulation experiments. To do this, the modeler must first specify an instance of the overall system. We do this by defining a main procedure and the agents' behavior procedures. For example, we may want to study a system instance specified by the following `main` procedure:

```
proc(main,[
 initiator_behavior(initl,msl)#=
 meetingScheduler_behavior(msl,initl)#=
 participants_behavior(yves,msl)#=
 participants_behavior(paige,msl)#=
 ]).
```

Here there is one initiator `init1`, one meeting scheduler `ms1`, two participants `yves` and `paige` in the meeting scheduling process. The sign "`#=`" means that the behavior of the agents execute concurrently (since they are independent from each other).

We also have to specify the details of the agents' behavior for this system instance. For example, we may write the following procedure to specify that the behavior of an initiator who wants to organize two meetings:

```
proc(initiator_behavior(Init,MS),
   [
    tryOrganizeMeeting(Init,MS,[paige,yves],[12,14,15,16,17]),
    tryOrganizeMeeting(Init,MS,[paige,yves],[12,14,15])
    ]).
```

Here, the possible meeting dates are limited to those of a February and represented as integers. This procedure specifying that the initiator `Init` wants to schedule two

meetings using the MS. One is with paige and yves on Feb. 12, 14, 15, 16, or 17. Another is with paige and yves on Feb. 12, 14, or 15.

For the meeting scheduler and participant agents, we use the normal behavior specification as described in the previous section:

```
proc(meetingScheduler_behavior(MS,Init),
   tryScheduleMeetings(MS,Init)
).

proc(participant_behavior(Participant,MS),
 tryArrangeMeetingsAndMaintainSchedule(Participant,MS)
).
```

5.5.2 Simulation Examples

Here, we present some example simulation traces. The modeler must specify the initial state of the system as explained in the previous section. For all our examples, we assume that initially, the time schedule for participant paige is [11, 12, 14], i.e., paige is busy on Feb. 11, 12, and 14, and the time schedule for the participant yves is [10, 12], i.e., yves is busy on Feb. 10, and 12.

Example 1: The initiator init1 wants to schedule a meeting with paige and yves on Feburary 12 or 14, since paige is busy on both of these days, meeting scheduling should fail. The modeler executes the main procedure to obtain a simulation trace. The simulation trace that will be obtained for this instance of the system is as follows:

```
requestScheduleMeeing(inil,msl,[paige,yves])
      /* inil requests msl to schedule a meeting with paige and yves*/
requestEnterDateRange(msl,inil,l)
      /* ms1 requests ini1 to enter the possible date range for the meeting No. 1 */
enterDaterange(ini1,ms1,1,[12,14])
      /* ini1 enters Feb. 12, 14 as the possible meeting dates */
obtainAvailDates(msl,yves,l)
      /* ms1 requests available dates from all participants */
obtainAvailDates(msl,paige,l)
```

```
sendAvailDates(yves,msl,2,[1,2,3,4,5,6,7,8,9,10,13,15,16,17,18,19,20,21
,22,23,24,25,26,27,28])            /* yves sends his available dates */
sendAvailDates(paige,msl,l,[1,2,3,4,5,6,7,8,9,11,13,14,15,16,17,18,19,2
0,21,22,23,24,25,26,27,28])      /* paige sends his available dates */
setAllMergedlist(msl,1,[])
      /* msl finds the merged available dates to be empty */
notifyFail(msl,inil,l,[paige,yves])
      /* msl notifies inil , yves, and paige that it has failed to schedule meeting No. 1 */
notifyFail(msl,paige,l,[paige,yves])
notifyFail(msl,yves,l,[paige,yves])
```

The above trace shows that after obtaining available dates from yves and paige, the MS finds out that the merged date list, which is merged from the lists of available dates of yves and paige and the proposed meeting dates offered by the initiator, is empty. So he cannot schedule the meeting and notifies the initiator initl, yves, and paige of his failure. This shows that the specification produces the expected behavior in this case. It also shows that using the MS to schedule meetings is convenient for the initiator, and that obtaining available dates first also decreases the number of exchanges with the participants. See Appendix A-4 for the complete simulation trace for the example. The simulation shows that even if no date is available, the process will proceed as expected.

Generally, in this validation step of the methodology, we try to find gaps or errors in the specification by simulating the processes. Alternative specifications can be also compared. We could specify an alternative where the initiator schedules meetings by himself and compare the resulting simulation traces.

Example 2: The initiator wants to arrange a meeting with paige and yves on Feb. 12, 14, 15, 16, or 17. The scheduling will still succeed after dealing with the occupation of a date on paige's schedule during the process. The simulation trace obtained is as follows:

```
requestScheduleMeeting(ini1,msl,[paige,yves])
      /* initl requests msl to schedule a meeting with paige and yves */
requestEnterDateRange(msl,inil,1)
```

```
        /* msl requests initl to send the date range for the above meeting no.1 */
enterDateRange(initl,msl,1,[12,14,15,16,17])
        /* initl enters the meeting date to msl regarding the meeting no.1 */
obtainAvailDates(msl,paige,l)
        /* msl requests paige and yves to send their available dates */
obtainAvailDates(msl,yves,l)
sendAvailDates(paige,msl,1,[1,2,3,4,5,6,7,8,9,10,13,15,16,17,18,19,20,2
1,22,23,24,25,26,27,28,29])          /* paige sends his available dates to ms1 */
sendAvailDates(yves,msl,2,[1,2,3,4,5,6,7,8,9,11,13,14,15,16,17,18,19,20
,21,22,23,24,25,26,27,28,29])       /* yves sends his available dates to ms1 */
occupyDateFromParticipant(paige,15),
        /* An outside activity occupies the date of Feb. 15 from paige's free time schedule */
addDateToSchedule(paige,15)    /* paige adds the date Feb. 15 into his busy time schedule */
acknowledgeOccupy(paige,15)    /* paige acknowledges the occupation of the date of Feb.15 */
setAllMergedlist(msl,l,[15,16,17])
        /* msl sets a merged date list for  meeting no.1 as Feb.15, 16 and 17 */
requestAgreement(msl,paige,1,15)
        /* msl requests agreement from paige for meeting No.1  on Feb. 15 */
requestAgreement(msl,yves,1,15)
        /* msl requests agreement from yves for meeting No.1  on Feb. 15 */
rejectAgreement(paige,msl,4,15)        /* paige rejects meeting  No. 1 on Feb. 15 */
addDateToSchedule(yves,15)
        /* yves adds Feb. 15 into his time schedule and accepts the agreement */
acceptAgreement(yves,msl,5,15)
cancelAgreement(msl,yves,1,15),
        /* msl requests yves cancel the agreement to meeting No.1  on Feb.15 */
requestAgreement(msl,paige,1,16)
        /* msl picks up next date in the merged date list Feb.16 and requests agreement again */
requestAgreement(msl,yves,1,16)
addDateToSchedule(yves,16)     /* both of yves and paige accept to meet on Feb.16 */
acceptAgreement(yves,msl,8,16)
rmvDateFromSchedule(yves,15)
        /* yves removes Feb.15 from his time schedule and accept the request to cancel agreement on it */
acceptCancel(yves,msl,6,15)
addDateToSchedule(paige,16),
        /* paige adds Feb.16 into his time schedule for meeting no.1 */
acceptAgreement(paige,msl,7,16)     /* paige agrees to meeting no.1 on Feb 16  */
notifySuccess(msl,initl,l,[paige,yves],l6),
        /* ms1 notifies initl, yves, and paige of successfully scheduling meeting no. 1 */
notifySucess(msl,paige,1,16),
notifySucess(msl,yves,1,16),
```

The above simulation trace shows that although `paige` originally sends his available dates to `ms1` which include Feb. 15, later something happens and occupies the date Feb. 15 from the time schedule of `paige` — The exogenous action `OccupyDateFromParticipant`. `ms1` does not know about that, so it still picks Feb. 15 from the merged date list to request `paige` and `yves` to agree to hold the meeting on Feb. 15. `paige` rejects the meeting on Feb. 15 because now his is unavailable on Feb. 15, while `yves` accepts the agreement and adds Feb. 15 into his busy time schedule. `ms1` finds out that `paige` is no longer available on Feb. 15 because `paige` rejects the request to meet on Feb. 15, so it has to request `yves` to cancel the agreement on Feb. 15. Then `ms1` picks an alternative date Feb. 16 in the merged date list and asks `paige` and `yves` whether they agree to have the meeting on Feb. 16. Both agree and the organization of the meeting concludes successfully. This simulation shows that even when an outside activity occupies a date from a participant's time schedule, the *ConGolog* model continues trying to find a suitable date for all participants to schedule the meeting. This behavior was not captured by the original *i** model.

Example 3: The initiator arranges multiple meetings using the scheduler. There are two meetings are to be scheduled: meeting No. 1 can be on Feb. 12, 14, 15, 16, or 17 with `paige` and `yves`; meeting No. 2 can be on Feb. 12, 14, or 15 with `paige` and `yves`. See appendix A-5 for the simulation trace. It shows that meeting No.1 is scheduled on Feb. 16 and meeting No.2 fails to be scheduled because `paige` is busy on Feb. 12 and 14, and Feb. 15 is occupied by an outside acitivity.

5.5.3 Discussion

The *ConGolog* model is certainly helpful for modeling repetitive processes, complex tasks, goals, and even dependencies. For example, in the meeting scheduling process, how to achieve the goal `FindAvailDateSlot` is not clearly shown in the original model, but in the *ConGolog* model, there is a recursive function to find the agreeable slot to reach the goal `FindAvailDateSlot`. Also in the initial SR model, the possibility

of failing to achieve the goal meetingBeScheduled is not clearly shown, but in the *ConGolog* model, this is handled; if the meeting organization fails, the meeting scheduler will notify the initiator about the failure. On the other hand, the *ConGolog* gives no hints about how to achieve the softgoals. In the meeting scheduling process example, the softgoal LowEffort inside the initiator is not modeled in the *ConGolog* model, but the initial SR model clearly shows how the different tasks can help to achieve this softgoal. The two models really complement each other well.

5.6 Refining the *i** and *ConGolog* Models Based on Validation Results

The above five steps will need to be repeated if errors are found or if aspects of the *i** model and *ConGolog* model do not satisfy the client's needs. Based on the *ConGolog* model and simulation experiments, if the *i** model lacks some part of the desired requirements, modifications to the *i** model will be performed. Similarly if the *ConGolog* model needs to specify additional details or aspects of the *i** model, modifications to the *ConGolog* model will be made. Once a satisfactory model of the required system has been developed, a requirements specification document is produced.

Consider the following example of model revision/refinement. In our meeting scheduling example, the alternative we selected allowed the exogenous action occupyDateFromParticipant which commands the participant to occupy a date for another activity if this date is available at the current time. This action is made by an agent outside the organization. If the client decides that all scheduling will have to be made by the scheduler agent, then this exogenous action can no longer occur. Let us describe how the models would be modified for this case.

- Modifications of the SR Diagram

In the annotated SR diagram of Figure 5.10 (c), we remove the subtask `OccupyDate` (and its subtasks) under the `TryArrangeMeetingsAndMaintainSchedule` task, as well as the concurrency annotation on the decomposition link between them. We also remove the exogenous action/task `OccupyDateFromParticipant`.

- Modifications of the *ConGolog* model

Modification in the *ConGolog* model will be to the parts corresponding to those modified in the annotated SR diagram for the participant. The modification parts are listed as follows:

Two actions `acknowledgeOccupy(Participant,Date)` and `occupyDateFromParticipant(Participant,Date)` will be removed from the *ConGolog* model. Action precondition axioms related to these two actions will be eliminated:

```
poss(acknowledgeOccupy(_,_),_).
poss(occupyDateFromParticipant(Participant,Date),S):-
     holds(not(participantDateOccupied(Participant,Date)),S),
    holds(not(agreementAccepted(-,Participant,Date)),S).
```

Successor state axioms which are related to the above two primitive actions will be modified and the effects of these two actions to these successor state axioms will be eliminated:

```
holds(occupyAcknowledged(Participant,Date),do(A,S)):-
      A= acknowledgeOccupy(Participant,Date);
      holds(occupyAcknowledged(Participant,Date),S).

holds(participantDateOccupied(Participant,Date),do(A,S)):-
      A = occupyDateFromParticipant(Participant,Date);
      holds(participantDateOccupied(Participant,Date),S).
```

We modify the following successor state axioms into those that doesn't consider the effects of those removed two actions (the commented out lines in the shadowed area):

```
holds(val(reqParticipant(ID),Participant),do(A,S)):-
      (A=requestAgreement(_,Participant,_,_),
       holds(val(reqCtr,ID),S));
      (A=cancelAgreement(_,Participant,_,_),
       holds(val(reqCtr,ID),S));
      (A=obtainAvailDates(_,Participant,_),
       holds(val(reqCtr,ID),S));
 /*   (A=occupyDateFromParticipant(Participant,_),
       holds(val(reqCtr,ID),S));*/
       holds(val(reqParticipant(ID),Participant),S).

holds(val(reqDate(ID),Date),do(A,S)):-
      (A=requestAgreement(_,_,_,Date), holds(val(reqCtr,ID),S));
      (A=cancelAgreement(_,_,_,Date), holds(val(reqCtr,ID),S));
 /*   (A = occupyDateFromParticipant(_,Date),
       holds(val(reqCtr,ID),S));*/
       holds(val(reqDate(ID),Date),S).

holds(val(reqctr,N),do(A,S)):-
      (A = requestAgreement(_,_,_,_),
       holds(val(reqCtr,M),S), N is M + 1);
      (A = cancelAgreement(_,_,_,_),
       holds(val(reqCtr,M),S), N is M + 1);
      (A = obtainAvailDates(_,_,_),
       holds(val(reqCtr,M),S), N is M + 1);
 /*(A = occupyDateFromParticipant(_,_),
    holds(val(reqCtr,M),S),N is M + 1);*/
      (holds(val(reqCtr,N),S),
       A \= requestAgreement(_,_,_,_),
        A \= cancelAgreement(_,_,_,_),
       A \= obtainAvailDates(_,_,_).
  /*, A \= occupyDateFromParticipant(_,_)*/
```

The procedure that handles the exogenous actions `OccupyDate` will be removed:

```
proc(occupyDate(Participant,Date),
  /*  Pick  up  the  exogenous  action,  occupy  Date  from  the
      participant's schedule */
 [
        addDateToSchedule(Participant,Date),
        acknowledgeOccupy(Participant,Date)
  ]
)
```

The interrupt that invokes this procedure will be removed from the procedure `tryArrageMeetingsAndMaintainSchedule`:

```
proc(tryArrangeMeetingsAndmaintainSchedule(Participant,MS),
  tryArrangeMeetings(MS,Participant)
/* #=
   ==>([reqID,date,xlist),
       and(val[reqParticipant(reqID),Participant),
        and(val[reqDate[reqID],date),
          and(participantDateOccupied(Participant,date),
            and(val(participantDateInfo(Participant),xlist),
              not(occupyAcknowledeged(Participant,date))
       )))),
       occupyDate(Participant,date) */
 ).
```

The shadowed area is the invocation of the procedure `occupydate` in the procedure `tryArrangeMeetingsAndMaintainSchedule.`

- Simulation after Modification

We suppose that the initial state of the system is exactly the same as in example 2 of section 5.5.2. Like in that example, the initiator wants to arrange a meeting with `paige` and `yves` on Feb. 12, 14, 15, 16, or 17, But now there can be no exogenous action that could occupy a date on `paige` 's schedule. The simulation trace obtained is as follows:

`requestScheduleMeeting(init1,msl,[paige,yves])`

/* `initl` reqeusts `msl` to schedule a meeting with `paige` and `yves` */

`requestEnterDateRange(msl, initl, 1)`

/* `msl` requests `initl` to send the date range for the above meeting no.1 */

`enterDateRange(initl,msl,1,[12,14,15,16,17])`

/* `initl` enters the possible meeting dates to `msl` regarding the meeting no.1 */

`obtainAvailDates(msl,paige,1)`

/* `msl` requests `paige` and `yves` to send their available dates */

`obtainAvailDates(msl,yves,l)`

`sendAvailDates(paige,msl,l,[1,2,3,4,5,6,7,8,9,10,13,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29])` /* `paige` sends his available dates to `ms1` */

```
sendAvailDates(yves,msl,2,[I,2,3,4,5,6,7,8,9,11,13,14,15,16,17,18,19,20
,21,22,23,24,25,26,27,28,29])      /* yves sends his available dates to msl */
setAllMergedlist(msl,1,[15,16,17])
        /* msl set the merged date list for the meeting no.1 as Feb.15, 16 and 17 */
requeStAgreement(msl,paige,1,15)
        /* msl picks up next date in the merged list Feb.15 and requests agreement */
requestAgreement(msl,yves,1,15)
        /* both of yves and paige agree to meet on Feb.15 */
addDateToSchedule(yves,15)
acceptAgreement(yves,msl,2,15)
addDateToSchedule(paige,15)
         /* paige adds Feb.15 into his time schedule for meeting no.1 */
acceptAgreement(paige,msl,1,15)
notifySuccess(msl,inil,l,[paige,yves],l5)
        / * msl notifies initl, yves and paige of the successful scheduling of meeting no.1 */
notifySucess(msl,paige,1,15)
notifySucess(msl,yves,1,15)
```

As we can see, the meeting with paige and yves is scheduled on Feb. 15, because there is no exogenous action that can occupy the date Feb. 15 from paige's time schedule. We also see the process proceeds faster than the one that has exogenous actions. This means that the second version of the meeting scheduling process will save time and efforts if none of the participants accepts activities other than the requests from the MS.

Parts of the source document of the *ConGolog* model are included in Appendix A.

6 Case Study II: A Mail-Order Business Application

In this chapter, our methodology for the combined use of the *i** and *ConGolog* frameworks will be demonstrated on a mail-order business application taken from Bissener's thesis [Bissener97]. In this mail-order business process example, we mainly focus on dealing with roles, positions, the decomposition links inside them, and the dependencies between them. The example raises some new issues for our methodology and we will illustrate how they can be handled. Bissener [Bissener97] developed *i** and *ALBERT-II* models of the example. We will discuss how his work compares to ours later in chapter 8. Our example involves a process that might proceed as follows. A customer submits an order to the mail-order company. The mail-order company first has to check whether there is sufficient stock to serve the order. Then, the company interacts with the bank to debit the customer's account and credit the company's account, removes the ordered item from stock, and ships it to the customer. If there is no stock or the customer does not have sufficient credit to make payment for the order, the order must be rejected and the ordered items have to be returned to stock.

The mail-order business application involves different actors: customers, the mail-order company, and the bank. There would also be different agents playing different roles within the mail-order company. This brings out different alternatives for the system. One alternative is that the mail-order company might have two agents to accomplish its tasks, an office clerk and a stock clerk. The office clerk accepts the orders made by customers, requests the stock clerk to provide stock for the orders, requests the bank to transfer payment for them, and rejects orders when there is no stock or the customer cannot pay. The stock clerk processes the requests for stock information concerning the ordered items

from the office clerk, updates stock quantities, and ships orders. Another alternative for the system is that the mail-order company just has an office clerk who does all the work. Then, the office clerk would not need to request stock information from himself and might just update the stock quantities when he ships an order.

One important issue in designing the process for our application is whether one attempts to acquire stock to fill the order first and then attempts to debit the customer's account or whether one does these activities in the reverse order. Which alternative one chooses may have important implications with respect to cost (perhaps the bank charges for each transaction) or how fast the process can be completed. Later, we discuss how to model these two alternatives and compare them.

Note that what we do here is more a business process modeling/reengineering case study. There is no clearly identified computerized component in the process that is modeled. This shows that our combined methodology can also be used for this kind of work, which is an important part of RE. We could modify the selected process to include a computerized component fairly easily. For example, we could assign the stock informant role, which is responsible for processing stock requests from the office clerk, to a computerized inventory management system agent. Our methodology supports modeling and analysis for both business process reengineering and RE.

6.1 Building the Strategic Dependency Model

Our Strategic Dependency (SD) model of the mail-order business process is shown in Figure 6.1. It is essentially based on the SD model developed by Bissener [Bissener97] with some minor changes. In the SD model of Figure 6.1, there are four agents: `Customer`, `OfficeClerk`, `StockClerk`, and `MailOrderCompany`. `OfficeClerk` and `StockClerk` are parts of `MailOrderCompany`. There are two positions: `BankClerk` and `Bank`, and `BankClerk` is part of `Bank`. There are three

roles: OrderProcessor, ShipmentProcessor, and StockInformant; StockClerk plays the two roles ShipmentProcessor and StockInformant, and OfficeClerk plays the role OrderProcessor. (One could also take Customer to be a role that is played by various concrete agents.)

Within the MailOrderCompany, the OfficeClerk agent accepts orders, requests payments from the bank, and notifies the customer that the order has been accepted or rejected; the StockClerk agent provides stock information to the OfficeClerk, maintains stock, and ships the ordered items. Within the Bank, the BankClerk position processes the transactions which are requested by the OfficeClerk to check customer account information, debit customers' accounts, and credit the company's account.

In figure 6.1, the Customer agent makes orders. We model the customer as an agent because he is the person who actively makes the order and is waiting for the order to be shipped. The mail-order company wants to distinguish the responsibilities of communicating with the customer about his order, processing the stock for the ordered items, and shipping the ordered items. So we identify three roles: OrderProcessor, StockInformant, and ShipmentProcessor. The OfficeClerk agent plays one role, OrderProcessor, which involves accepting the customer's order, requesting stock from the stock clerk, and then requesting the bank to process payment for the order as necessary. The StockClerk agent plays two roles ShipmentProcessor and StockInformant. The StockInformant role processes stock requests from the office clerk. The ShipmentProcessor role ships the order to the customer if the order is accepted. We model the bank and bank clerk as positions because they are institutionalized actors and we do not specify what bank it is and who the bank clerk is. We want the bank to realize the process of transferring the money paid for the orders from the customer's account to the company's account. The position BankClerk within the bank fulfills this responsibility.

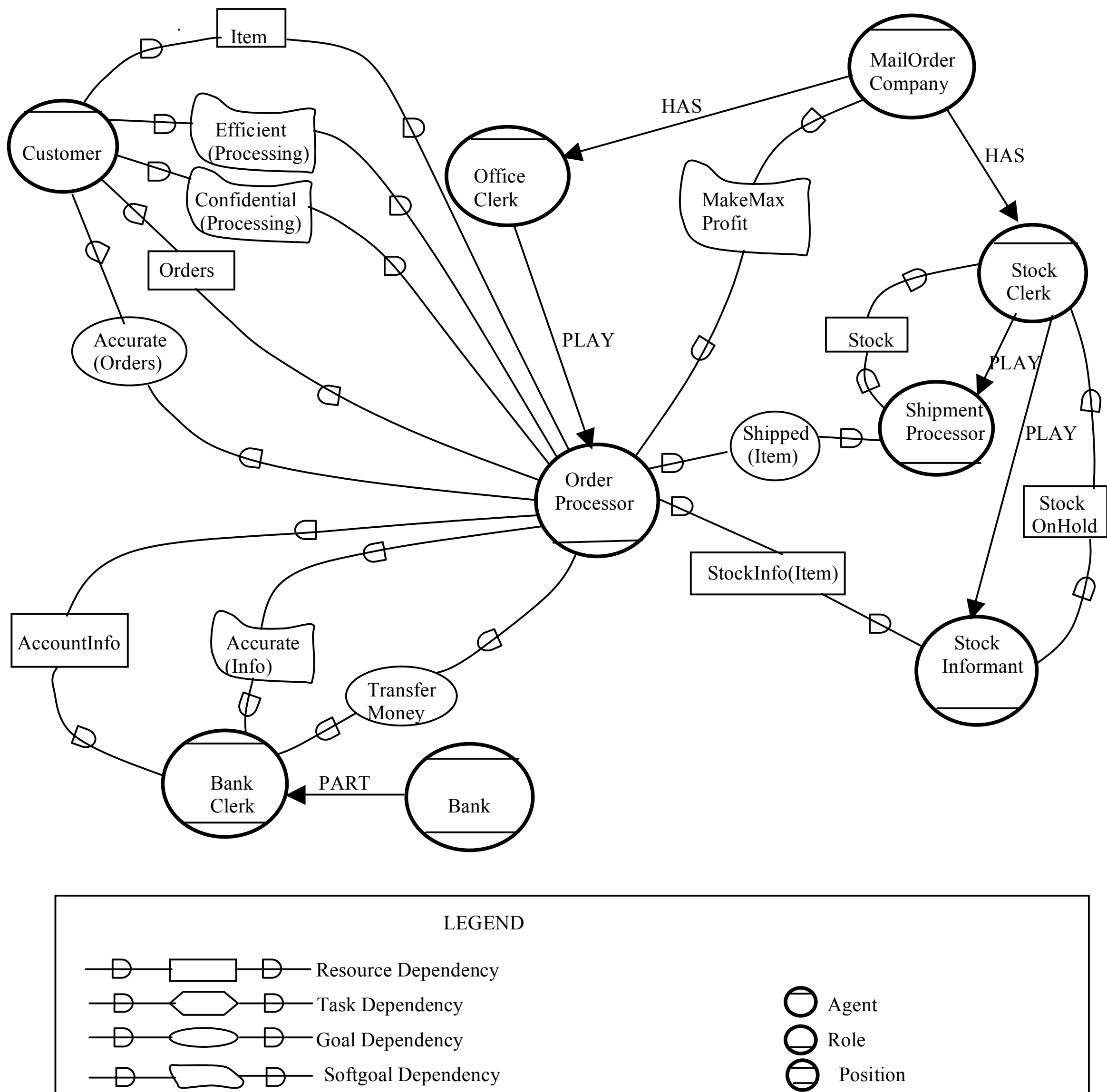


Figure 6.1 The Strategic Dependency model for the mail-order business process.

The dependencies between the agent `Customer` and the role `OrderProcessor` are as follows. The order processor depends on the customer to make an order; this is modeled as a resource dependency `Orders`. On the other hand, the customer depends on the order processor for processing orders confidentially and efficiently; this is modeled as two softgoals dependencies `Efficient(Processing)` and

`Confidential(Processing)`. The order processor depends on the customer to provide accurate orders; this is modeled as a goal dependency `Accurate(Orders)`. The customer also depends on the order processor for obtaining the ordered items; this is modeled as a resource dependency `Item`.

The dependencies between the role `OrderProcessor` and the position `BankClerk` are as follows. The order processor depends on the bank clerk to provide customer account information and check whether the customer has sufficient credit to pay for the order; this is modeled as a resource dependency `AccountInfo`. To avoid shipping orders to customers who cannot pay or rejecting orders from customers who can pay, the order processor depends on the bank clerk to provide the accurate information about the customer's credit; this is modeled as a softgoal dependency `Accurate(Info)`. To get paid by the customer, the order processor depends on the bank clerk to transfer money from the customer account to the company's account. This is modeled a goal dependency `TransferMoney`. The order processor also depends on the shipment processor to ship ordered items; this is modeled as a goal dependency `Shipped(Item)` between them.

Some interesting issues are raised when we have dependencies within a single agent. The mail-order company depends on the order processor to make maximum profit from orders. This is modeled as a softgoal dependency `MakeMaxProfit` between the mail-order company and the order processor. How the mail-order company measures whether this softgoal is achieved is up to the company. The order processor tries to satisfy this softgoal when he processes the orders. The strategy to maximize profit is decided jointly by the mail-order company and the order processor. The order processor depends on the stock informant to provide information concerning the stock on ordered items; this is modeled as a resource dependency `StockInfo(Item)`. The shipment processor depends on the stock clerk to update the stock information for the shipped items; this is modeled as a resource dependency `Stock`. The stock informant also depends on the stock clerk to put the ordered items on hold if the informant accepts an order for them;

this is modeled as a resource dependency `StockOnHold`. The dependency relationships between the roles inside the agent `StockClerk` explicitly show how one role is distinguished from another and what effects one role has on another. Later, when we operationalize the dependencies between roles inside a single agent, some interesting issues will arise.

The SD model of Figure 6.1 models the mail-order business process in terms of intentional relationships among the actors, rather than as a flow of entities. This allows the modeler analyze opportunities and vulnerabilities for the actors. For example, the ability of the stock informant to maintain accurate stock and on-hold information for the whole mail-order company and provide it to other roles/positions/agents represents an opportunity for the other roles/positions/agents. They do not need to maintain stock information by themselves and ensure that their information is consistent. They can just communicate with the stock clerk to obtain stock information and perform transactions on it. On the other hand, if the stock clerk fails to maintain stock information correctly, then the other roles/positions/agents become vulnerable.

6.2 Building the Strategic Rationale Model

The SR model of Figure 6.2 for the application is also closely based on the SR model developed by Bissener [Bissener97]. It elaborates on the relationships between the customer, the mail-order company, and the bank as depicted in the SD model in Figure 6.1. This model provides a more detailed level of analysis and captures the activities inside the actors in order to model internal relationships. Intentional elements such as goals, tasks, resources, and softgoals are modeled as internal elements as well as external dependencies like in the SD model. These intentional elements describe the strategies of actors and how they try to satisfy their needs and maximize profits without affecting the success of the whole process. Internal elements are linked by means-ends, task-

decomposition, and contribution links that model the internal relationships between the intentional elements inside the actors.

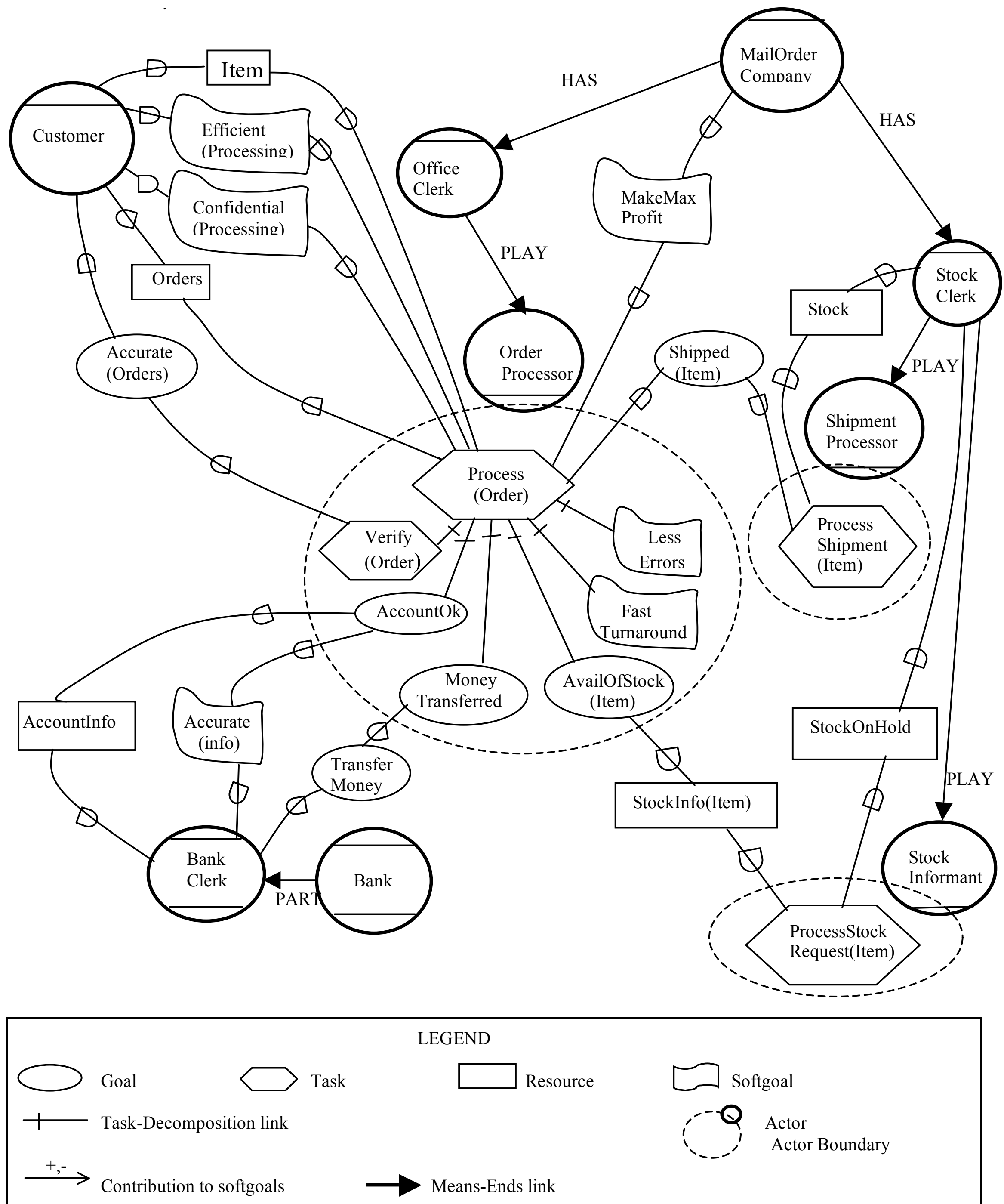


Figure 6.2 The initial Strategic Rationale model of the mail-order process

The SR model of Figure 6.2 focuses on analyzing the internal goals/tasks/resources inside the office clerk and stock clerk agents. These two agents largely control the whole process of processing an order. The customer only initiates the process and the bank clerk passively participates in it. We don't show the internal elements of the customer and bank clerk for now. Later, as our analysis about the process proceeds, we will fill out the SR diagram for the customer and bank clerk as necessary.

For the order processor, his top task is to process the order made by the customer; this is modeled as an internal task `Process(Order)`. This task can be decomposed into a subtask `Verify(Order)`, a subgoal `AccountOk`, a subgoal `AvailOfStock`, a subgoal `MoneyTransferred`, and two softgoals `LessErrors` and `FastTurnaround`. The subtask `Verify(Order)` represents how the order processor checks whether the order that was made is correct or not. This task will fulfill the goal dependency `Accurate(Order)`. The internal goal `AccountOk` models how the order processor wants to ensure that the customer has sufficient credit to pay for the order. This goal depends on the bank clerk to provide accurate account information. In this SR model, how the bank clerk fulfills the dependency is not shown. Later, in our annotated SR diagrams, we will provide complete information about how the bank clerk fulfills the goal dependency. The internal goal `AvailOfStock` models how the order processor wants to make sure that there is sufficient stock for the order to be filled. The order processor depends on the `StockInformant` to provide the stock information resource `StockInfo(Item)` for this order. The internal goal `MoneyTransferred` represents how the order processor wants the payment for the order to be paid by the customer. This goal relies on the bank clerk to achieve the goal dependency `TransferMoney`.

The stock clerk plays two main roles and also provides some dependencies himself. The role `StockInformant` replies to requests from the office clerk for stock information

about ordered items; this is modeled as an internal task `ProcessStockRequest` to provide the stock information. The `StockInformant` also depends on the stock clerk to maintain the resource `StockOnHold`, i.e., keep stock on hold for orders he has been informed of. The role `ShipmentProcessor` handles requests from the office clerk for shipping ordered items; this is modeled as an internal task `ProcessShipment(Item)`. He also depends on the stock clerk to update the real stock information when the item is shipped; this is modeled as a resource dependency `Stock`.

In the SR diagram of Figure 6.2, there is no much detail about how the internal goals and tasks can be achieved and decomposed: how the stock clerk maintains the stock information and the on-hold stock information for the company, how the customer makes an order, and how the bank clerk processes the payments. Later, in our annotated SR diagrams we will decompose tasks and goals to a more detailed level, operationalize the dependencies between actors, and specify the processes precisely. Also we will introduce some additional roles to distinguish between functions of the agents.

The SD model and SR model developed can support analysis, design, and reasoning during early-phase requirements engineering. With the notions of ability, workability, viability, and believability, the *i** framework provide a number of levels of analysis and high level design.

6.3 Building the Annotated *i** SR Model

The SR model of Figure 6.2 provides a sketchy description of the process in the mail order business application. In this section, we want to show how one alternative for the process is chosen and how a detailed description of this process is represented in an annotated SR model.

6.3.1 Suppressing Unnecessary Information

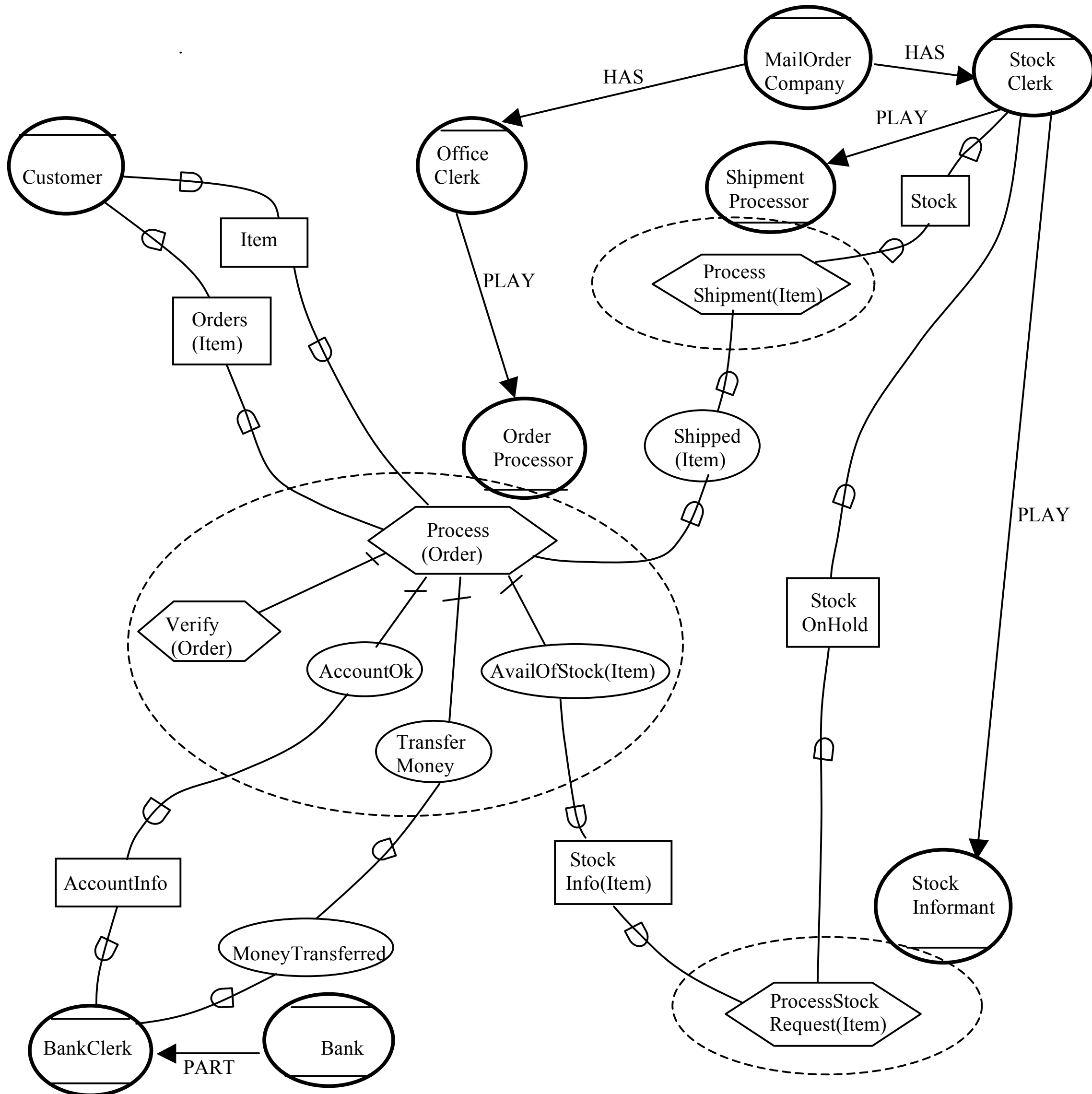


Figure 6.3 The second version of the SR model for the mail-order process.

As described in chapter 4, we start by suppressing less important information from the SR diagram of Figure 6.2. Softgoals, softgoal dependencies, and the links associated with them will be suppressed because they are qualitative goals which are less important for developing a precise process specification and will be not modeled in the *ConGolog* model. Because we want to focus on the activities in the processing of orders, the goal

dependency `Accurate(Orders)` between `Customer` and `OrderProcessor` will also be suppressed. We will not model how the customer can make sure that his order is accurate. After this, we obtain the simplified SR model of Figure 6.3.

6.3.2 Operationalizing Dependencies

Now, the dependencies between actors need to be operationalized. In the SR model of Figure 6.3, some of the dependencies are not related specifically to nodes inside actors. Before we discuss the operationalization, we have to specify from which nodes inside the depender the dependencies start and at which nodes inside the dependee the dependencies terminate. When we do this, we obtain the SR model in Figure 6.4.

In the SR diagram of Figure 6.3, the stock clerk plays two roles `ShipmentProcessor` and `StockInformant`. These two roles depend on the stock clerk to provide information about the stock and the on-hold stock. In the SR diagram of Figure 6.4, we introduce another role `UpdateStockProcessor` to be played by the `StockClerk`, whose job is to maintain and update the stock and on-hold stock information as necessary. The top task inside this role `UpdateStockProcessor` is modeled as the internal task node `UpdateStockInfo`. We make the dependencies between the `ShipmentProcessor` and `StockInformant` roles and the `StockClerk` agent end at the `UpdateStockInfo` node in the new `UpdateStockProcessor`.

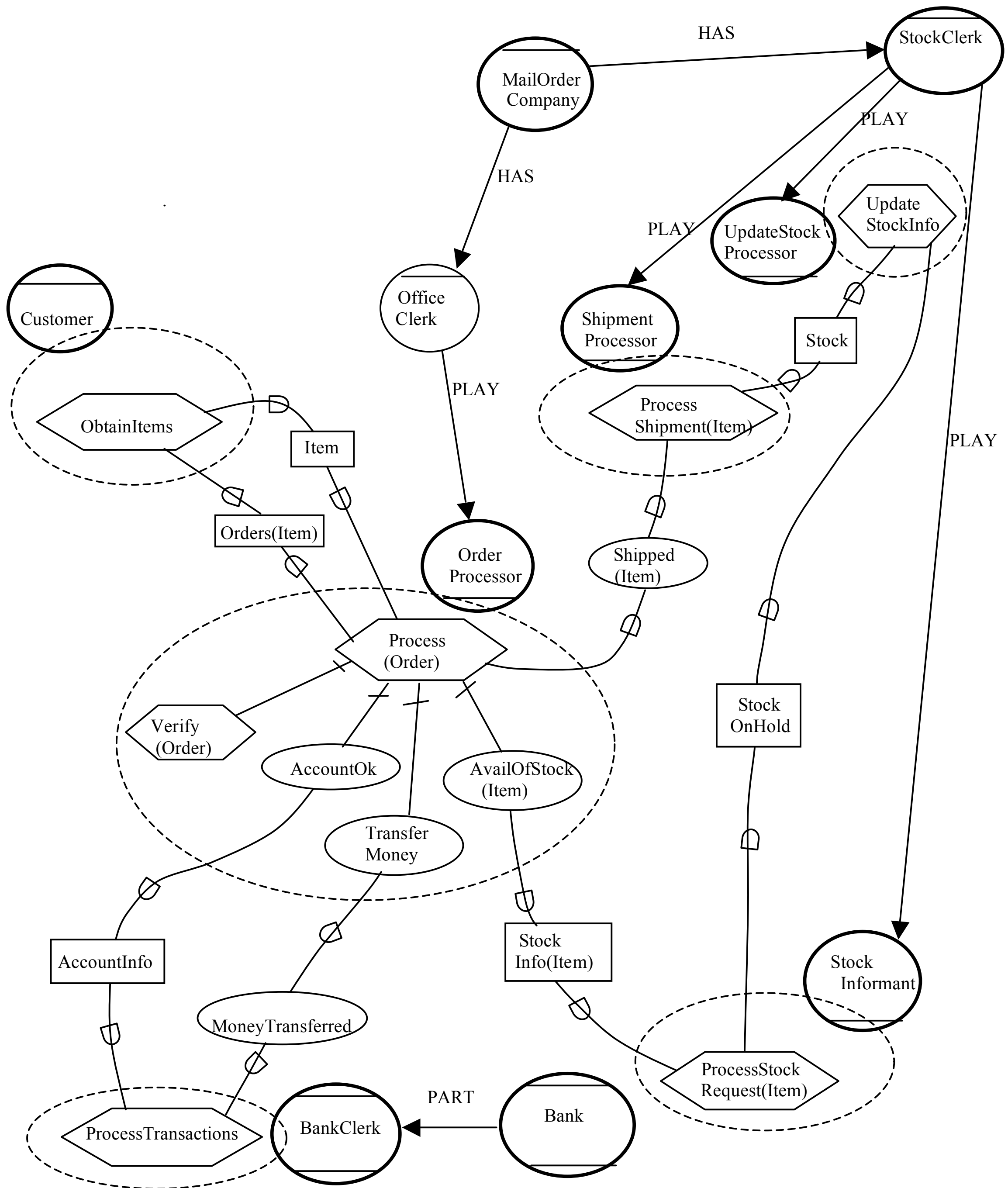


Figure 6.4 The third version of the SR model for the mail-order process.

In Figure 6.3, the Customer agent's behavior is not specified. We introduce a top task ObtainItems inside Customer into the SR diagram of Figure 6.4, which describes the customer's activities to obtain items from the mail-order company. All the dependencies between Customer and the role OrderProcessor are made to originate/terminate at this new task node. Similarly, in the SR diagram of Figure 6.3, the activities of the BankClerk position are also not shown. We introduce a top task ProcessTransactions inside BankClerk into the SR diagram of figure 6.4. This task specifies that the BankClerk is responsible for processing transactions requested by the bank's clients. The dependencies between BankClerk and OrderProcessor are made to terminate at this new node.

Next, we start operationalizing the dependencies in the SR diagram of Figure 6.4. Some of the resource/task/goal dependencies are between different agents or roles played by different agents. These dependencies can be operationalized as we did for the examples of chapter 5. Other dependencies are between roles played by the same agent. For example, the stock clerk plays both the ShipmentProcessor and UpdateStockProcessor roles, and there is a resource dependency between them. We will discuss in detail how we operationalize such dependencies.

The dependencies between the three roles ShipmentProcessor, UpdateStockProcessor, and StockInformant played by the StockClerk agent are shown in Figure 6.5.

When we operationalize these dependencies, we obtain the SR diagram in Figure 6.6.

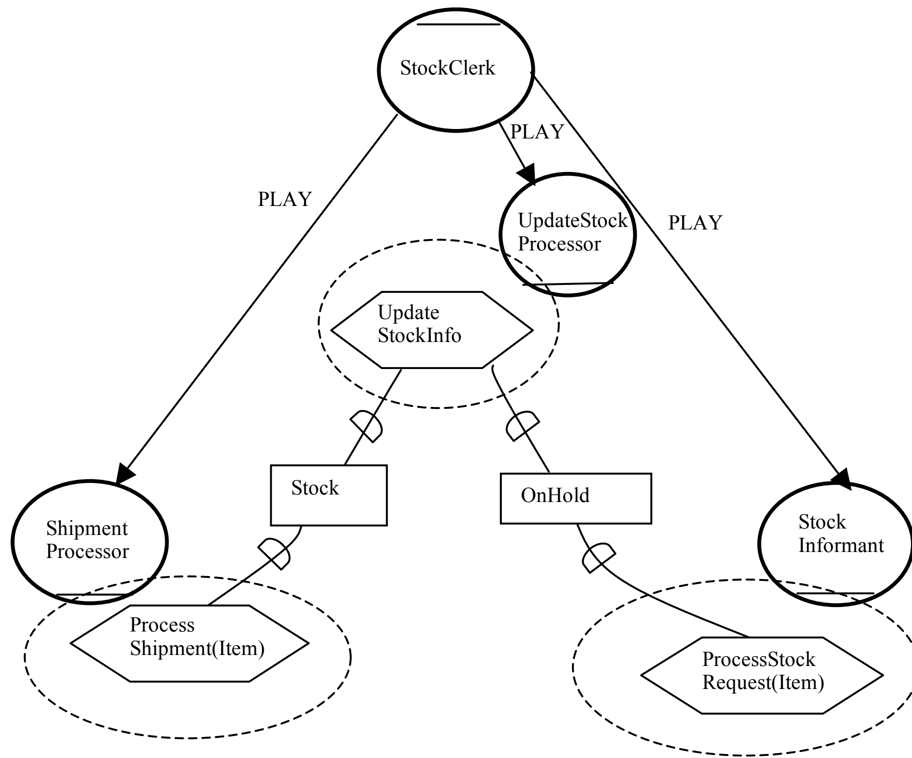


Figure 6.5 Dependencies between roles played by `StockClerk` agent.

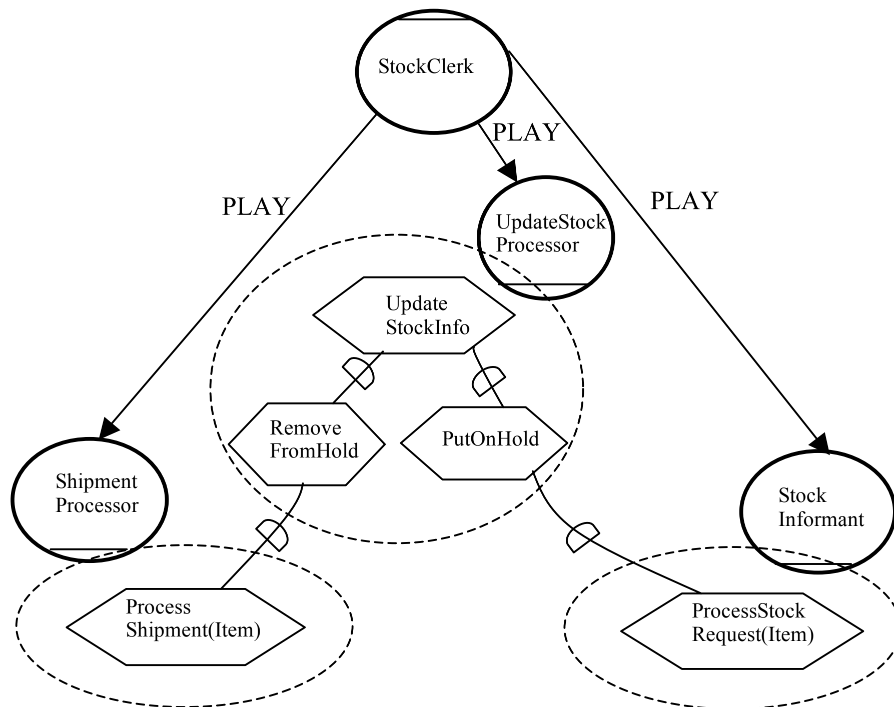


Figure 6.6 SR diagram for the dependencies of Figure 6.5 after operationalization.

Let us go over this new SR model. To obtain the dependum in the resource dependency `Stock`, the `ShipmentProcessor` doesn't need to make a request to the `UpdateStockProcessor` because these two roles are played by a single agent, `StockClerk`, and he does not need to make a request to himself. The stock clerk just updates the stock information when he has shipped the ordered item. But the `UpdateStockProcessor` has to perform a task to remove the item from the on-hold stock, and this modeled as a subtask `RemoveFromHold` of the task `UpdateStockInfo`.

Similarly, to obtain the dependum in the resource dependency `OnHoldStock`, the `StockInformant` does not need to make a request to the `UpdateStockProcessor` because `StockClerk` plays both roles. The `StockClerk` just updates the stock information himself when the `StockInformant` has accepted the stock request for the ordered item. But the `UpdateStockProcessor` has to perform a task to remove the item from the real stock and put it in the on-hold stock, and this modeled as a subtask `PutOnHold` of `UpdateStockInfo`.

So let us summarize how the operationalization of dependencies between roles played by the same agent can be done. Generally, we can operationalize such dependencies simply by adding a task into the dependee to supply the dependum. This is because we assume that the agent knows when the dependum has to be supplied between two roles played by the same agent. The depender role does not need to make a request to the dependee role to supply the dependum and the dependee role does not need to confirm the dependum has been supplied. The agent will know what is the state of the system when the appropriate actions are performed by its roles.

Note that, in some cases, the mail-order company may want to keep a record or trace of the whole process at a detailed level, and then it might be necessary for the stock clerk agent to make a request to himself to ask for stock information, perhaps by filling a form. The stock clerk may also have to confirm the stock allocation by signing the form. Then, the role `StockInformant` would have to perform a subtask to request the on-hold stock information and the role `UpdateStockProcessor` would have to perform a subtask to confirm that stock has been put on hold for the order.

Next, we operationalize the dependencies between the position `BankClerk` and the role `OrderProcessor`. This is done in the same way as in the examples of chapter 5, i.e., the dependencies between agents and roles in the SR model of Figure 5.4. The depender and dependee associated with a dependency here will both participate in having the dependum supplied. The depender may have to make a request to the dependee, and the dependee may have to wait for the request from the depender, and then the dependee has to perform some task to supply the dependum, while the depender waits for the dependum to be supplied. Figure 6.7 shows the SR diagram after operationalizing the dependencies between `BankClerk` and `OrderProcessor`.

To supply the resource dependency `AccountInfo` between `OrderProcessor` and `BankClerk` in the SR diagram of Figure 6.4, `OrderProcessor` first requests `BankClerk` to check whether the customer has sufficient money to pay for his order; this is modeled as a subtask `RequestDebit` of the goal node `AccountOk`. Note that `RequestDebit` does not mean requesting the bank to debit the customer's account, but just requesting the bank to check whether the customer's account has enough money to pay for the order. (We use the same name as in [Bissener97] to represent the action) `BankClerk` replies to such a request whenever it is received; this is modeled as a task `ReplyDebit` with the link annotation `*whenever(requestedDebit);` `ReplyDebit` is a subtask of `ProcessTransactions`. Then, `OrderProcessor`

waits for the answer to the debit request; this is modeled as a subtask WaitForDebitAnswer of AccountOk; the two subtasks are performed in sequence.

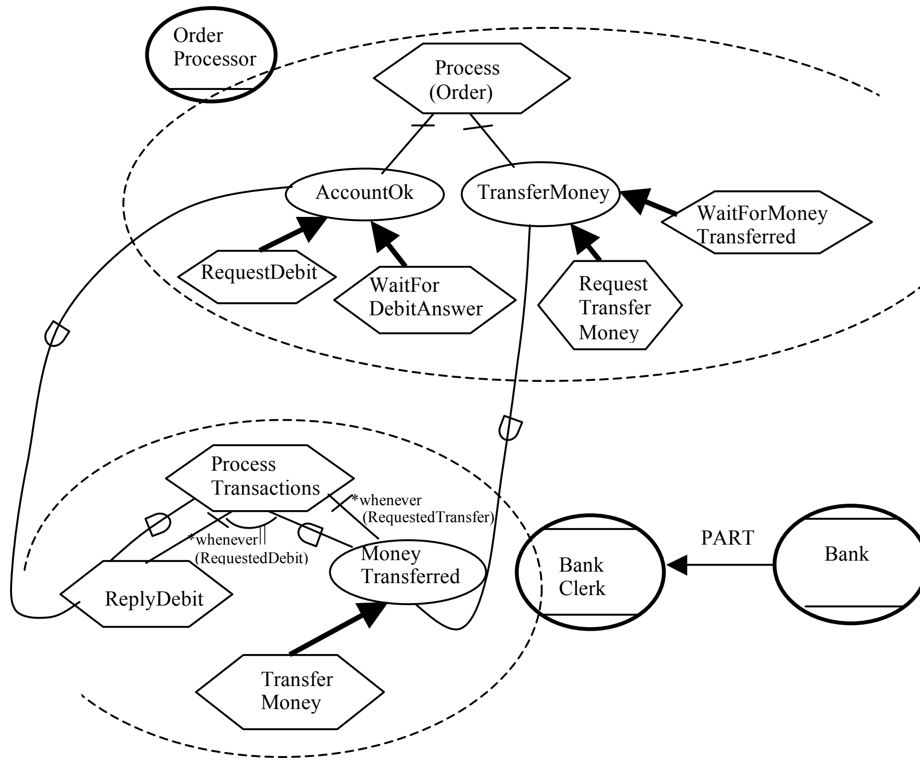


Figure 6.7 SR diagram for dependencies between BankClerk and OrderProcessor after operationalization.

To supply the goal dependency MoneyTransferred between OrderProcessor and BankClerk in the SR diagram of Figure 6.4, first OrderProcessor requests BankClerk to transfer money from the customer's account into the company's account; this is modeled as a task RequestTransferMoney. Then, BankClerk performs a task TransferMoney, which is its means to achieve the goal dependum MoneyTransferred, which must be achieved whenever the request received; this is modeled as an internal goal MoneyTransferred with a goal-decomposition link to the task TransferMoney. Finally, OrderProcessor has to wait for the money to

be transferred, and this is modeled as task WaitForMoneyTransferred with a goal-decomposition link to the goal TransferMoney.

Next, we operationalize the dependencies between the Customer and the OrderProcessor. Figure 6.8 shows the SR diagram of the dependencies between Customer and OrderProcessor after the operationalization.

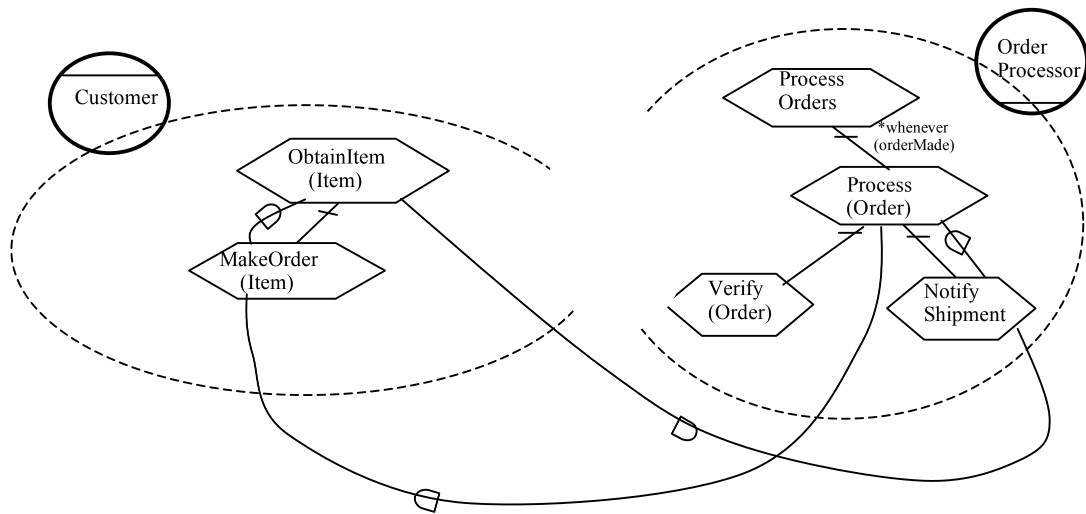


Figure 6.8 SR diagram for the dependencies between Customer and OrderProcessor after operationalization.

To supply the resource dependency Orders, Customer has to make a request for the order to OrderProcessor, so we introduce an internal task inside the Customer agent MakeOrder(Item). OrderProcessor has to process an order when he gets such a request. We added a top task ProcessOrders into OrderProcessor, which means OrderProcessor will process multiple orders, and a link with an interrupt annotation *whenever(orderMade) between the tasks ProcessOrders and Process(Order) to model how the subtask is triggered when an order is made. To supply the resource dependency Item(Order), OrderProcessor has to perform a

task NotifyShipment to notify Customer that the item has been shipped. We assume that it is not necessary for Customer to wait for or confirm having received the notification. We also assume that the task NotifyShippment will be performed only when the mail-order company has shipped the ordered item to the Customer.

Finally, we operationalize the dependencies between OrderProcessor, StockInformant and ShipmentProcessor in the SR diagram of Figure 6.4. See Figure 6.9 for the SR diagram after operationalizing these dependencies.

In Figure 6.9, to supply the resource dependency StockInfo between OrderProcessor and StockInformant, OrderProcessor has to make a request for an ordered item to StockInformant; this is modeled as a subtask RequestStock of the task AvailOfStock inside OrderProcessor. StockInformant replies to such stock requests whenever they are received; this is modeled as a task ReplyStockRequest inside StockInformant with a *whenever(requestedStock) annotation. OrderProcessor has to wait for the reply from StockInformant for the stock request; this is modeled as a task WaitForStockRequestAnswer inside OrderProcessor.

To supply the goal dependency Shipped(Item) between OrderProcessor and ShipmentProcessor, OrderProcessor has to make a request to ShipmentProcessor to ship the ordered item to the customer; this is modeled as a task MakeInvoice inside OrderProcessor. The goal Shipped(Item) is moved into ShipmentProcessor as a subgoal of the task ProcessShipment(Order). An internal task inside ShipmentProcessor is added to provide a means to achieve the internal goal Shipped(Item), the task ShipOrder.

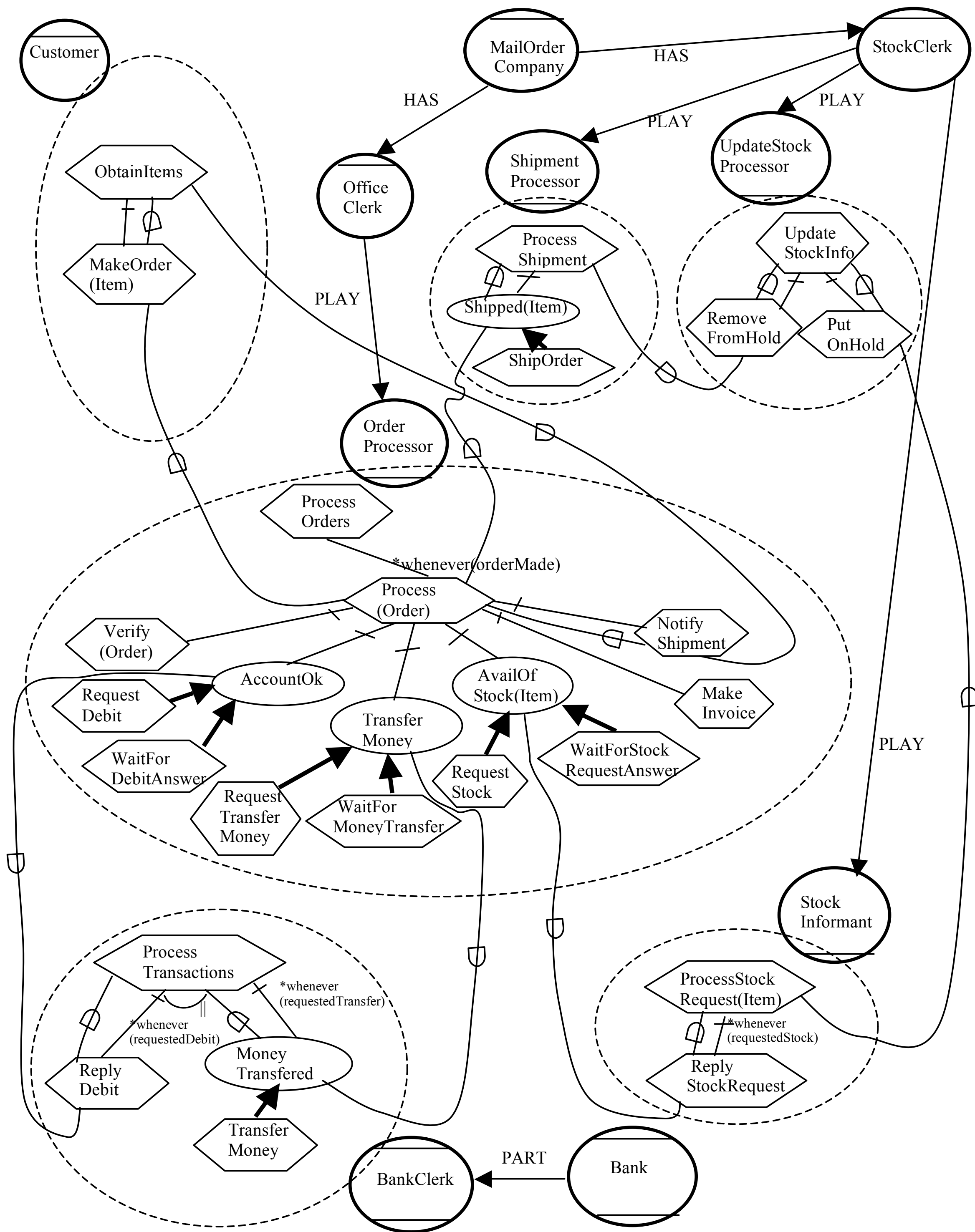
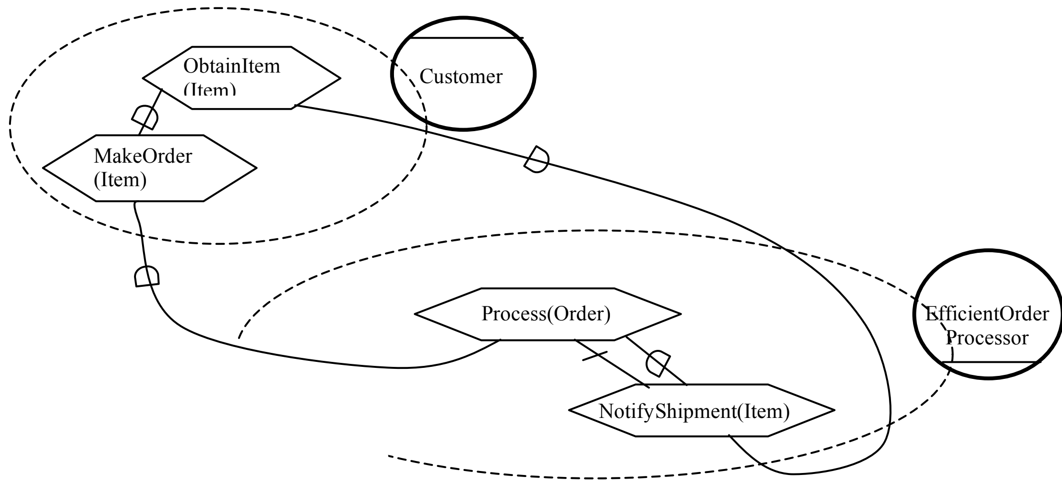


Figure 6.9 The SR model for the mail-order process after operationalizing all dependencies.

6.3.3 Relativizing the Goals that Cannot Always Be Achieved

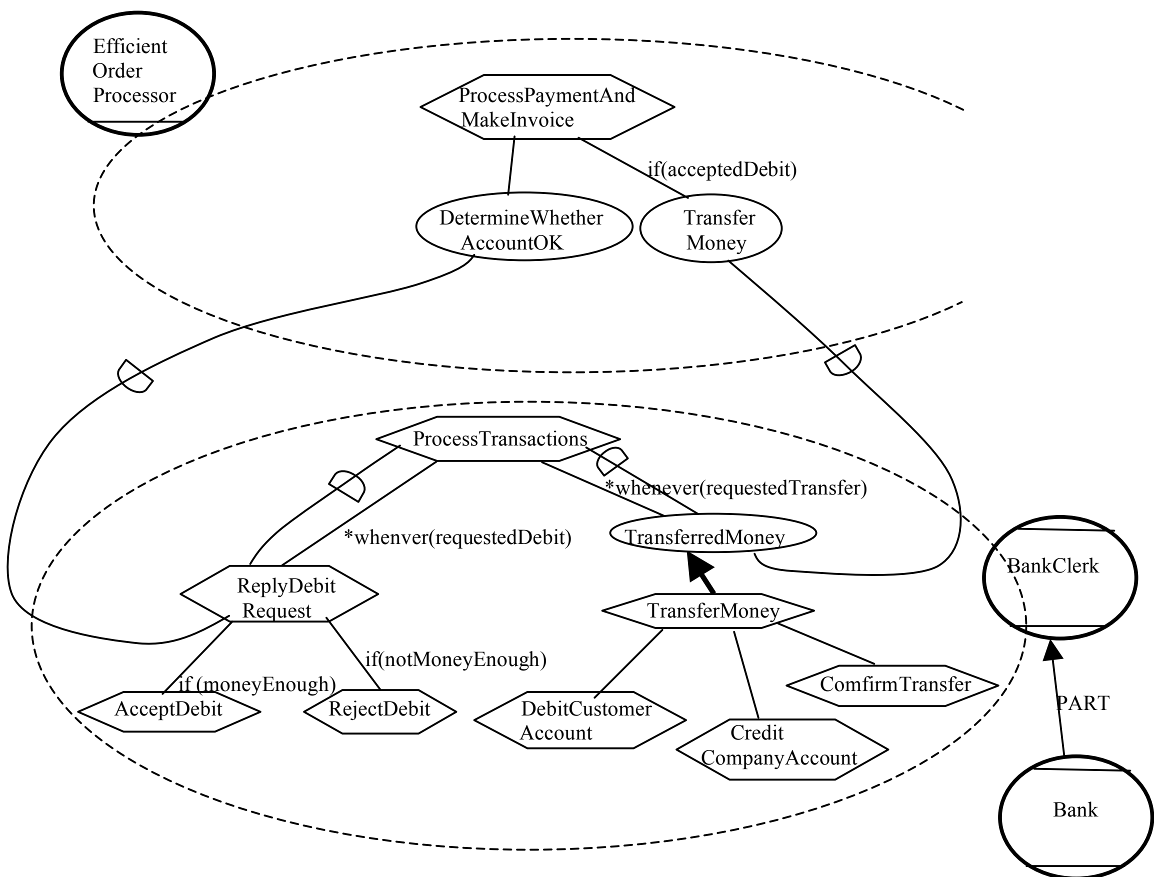
In this step, goals that cannot always be achieved by actors are refined into weaker goals that can always be achieved. The decompositions related to these goals are modified as appropriate. In the SR diagram of Figure 6.9, the goal `AvailOfStock` cannot always be achieved because the company may not have stock for an ordered item. So we have to relativize this goal into `TriedFindAvailOfStock`, which means that `OrderProcessor` tries to find available stock for an order. Also the goal `AccountOk` cannot always be achieved because the customer may not have enough money to pay for his order. So we have to relativize this goal into `DetermineWhetherAccountok`. The steps in the process that follow these goals also have to be changed to depend on the outcome. The SR diagram incorporating these changes appears in Figure 6.10 (d).

6.3.4 Filling out Process Details Using Decomposition and Annotations

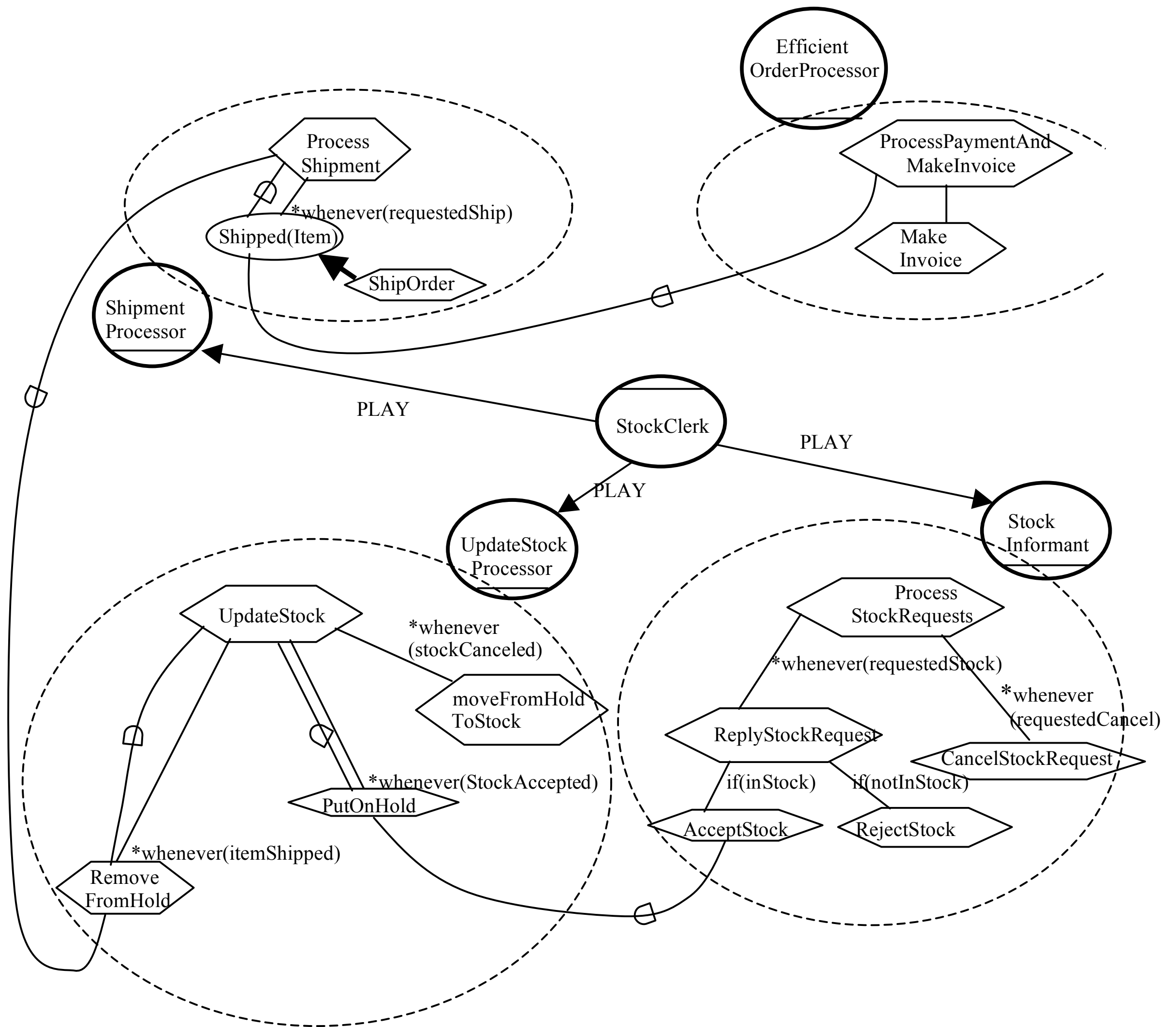


6.10 (a) The annotated SR diagram for the agent `Customer` agent.

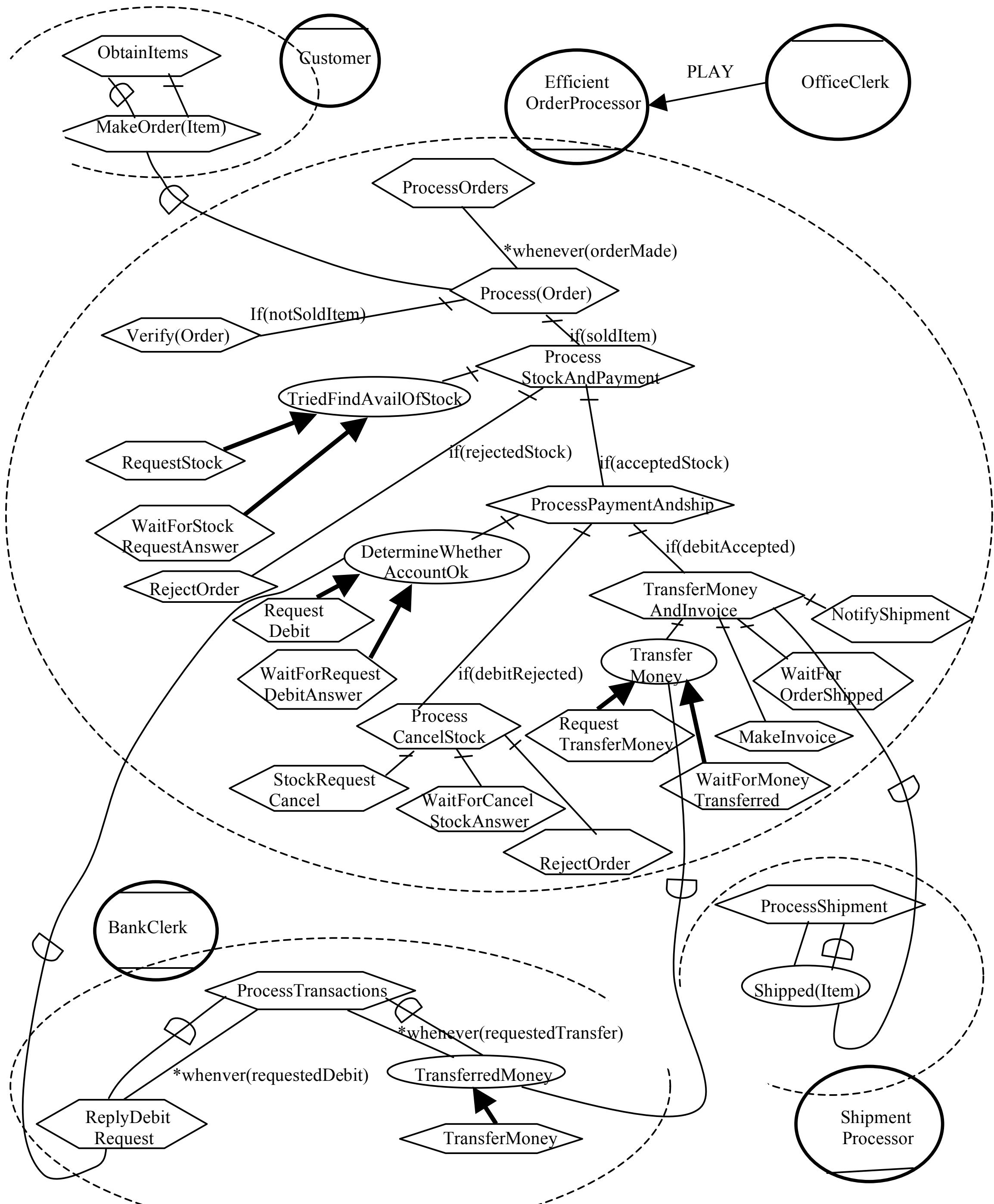
The SR diagram of Figure 6.9 doesn't show how an order is processed in detail. For example, when an ordered item is put on hold, and later we find out from the bank that the customer does not have enough money to pay for the order, then the stock clerk has to cancel the hold on the ordered item and return it to free stock. We want to detail all circumstances that can occur when processing an order. The annotated SR diagrams are shown in Figure 6.10 (a), (b), (c), and (d).



6.10 (b) The annotated SR diagram for positions `Bank` and `BankClerk.`



6.10 (c) The annotated SR diagram for agent `StockClerk` with its three roles.



6.10(d) The annotated *i** SR diagram for the agent `OfficeClerk`

Let us explain how these annotated SR diagrams capture the details of the mail-order business process. We start with the `OfficeClerk` agent, whose behavior is described in the annotated SR diagram in Figure 6.10(d). We say that the `OfficeClerk` agent plays one role, `EfficientOrderProcessor` (renamed from `OrderProcessor` because we want orders to be processed efficiently). This role is to accept orders from customers and process them; this is modeled as a top task `ProcessOrders`. Whenever an order is made, the top task performs a subtask `Process(Order)` to process this specific order. So the top task `ProcessOrders` is decomposed into a subtask `Process(Order)` with a link annotation `*whenever(orderMade)`.

The task `Process(order)` processes a specific order that has been made. This process goes as follows. If the ordered item is not of a type that is sold, the order processor alarms the customer that the ordered item is not of a type that is sold and rejects the order; this is modeled as a subtask `Verify(Order)` with a link annotation `if(notSoldItem)`. Here we simplify the process of verifying orders as that of alarming the customer and rejecting the order. If the ordered item is of a type that is sold, then the order processor continues to process the order; this is modeled as a subtask `ProcessStockAndPayment` with a link annotation `if(SoldItem)`.

The subtask `ProcessStockAndPayment` proceeds as follows. First, the order processor queries whether there is enough stock for this order; this is modeled as a subgoal `TriedFindAvailOfStock`. Then, if there is not enough stock for the order, the order processor rejects it. This is modeled as a subtask `RejectOrder` with a link annotation `if(rejectedStock)`. If there is enough stock for the order, then the order processor continues by processing payment and shipping the order. This is modeled as a subtask `ProcessPaymentAndShip` with a link annotation `if(acceptedStock)`. The subgoal `TriedFindAvailStock` can be decomposed into two subtasks: a subtask `RequestStock` of requesting the stock information for the ordered item and a

subtask WaitForStockRequestAnswer of waiting for the reply from the stock informant. A default sequence annotation ";" is applied to this group of goal-decomposition links.

The subtask ProcessPaymentAndShip can be decomposed into the following sub-processes. First, the order processor requests the bank clerk to check the customer's account and see whether the customer has enough credit to pay for the order. This is modeled as a goal DetermineWhetherAccountOk. This goal is decomposed into two subtasks: a subtask RequestDebit of querying whether the debit is possible and a subtask WaitForDebitRquestAnswer of waiting for the reply from the bank. A default sequence annotation ";" is applied to this group of goal-decomposition links. If the debit request is rejected, then the order processor has to request the stock informant to cancel the stock request for the ordered item, and reject the order. This is modeled as a subtask ProcessCancelStock with a link annotation if(debitRejected). If the debit request is accepted (modeled as a link annotation if(debitAccepted)), then the order processor continues with a subtask TransferMoneyAndInvoice.

The subtask ProcessCancelStock is further decomposed into the following subprocesses. First, the order processor has to request the stock informant to cancel the stock request for the ordered item; this is modeled as a subtask StockRequestCancel. Then the order processor has to wait for a confirmation that the stock request has been canceled; this is modeled as a subtask WaitForCancelStockAnswer. Finally, the order processor has to reject the order; this is modeled as a subtask RejectOrder.

The subtask TransferMoneyAndInvoice is decomposed into the following subprocesses. First, the order processor requests the bank clerk to transfer money from the customer's account to the company's account; this is modeled as a goal

TransferMoney. This goal is decomposed into a subtask RequestTransferMoney and a subtask WaitForMoneyTransferred as described in the SR diagram of Figure 6.9. The default sequence annotation ";" is applied to this group of goal-decomposition links. Then, the order processor makes an invoice for the order; this is modeled as a subtask MakeInvoice. Then, the order processor waits for the order being shipped; this is modeled as a subtask WaitForOrderShipped. Finally, the order processor notifies the customer that the ordered item has been shipped; this is modeled as a task NotifyShipment. The default sequence annotation ";" is applied to the group of decomposition links.

Next, let us look at the annotated SR diagram for Bank and its BankClerk position, which appears in Figure 6.10(b). The bank clerk is mainly responsible for processing transactions involving debits and credits to accounts. This is modeled as a top task ProcessTransactions. This top task can be decomposed into the following. Whenever the bank clerk receives a request for checking account information, he will perform a task to reply to the request. This is modeled as a subtask ReplyDebitRequest with a link annotation *whenever(requestedDebit). Whenever the bank clerk receives a request for transferring money, he will achieve the subgoal TransferredMoney by performing a task TransferMoney. A link annotation *whenever(requestedTransfer) is attached to the link between the subgoal and its super-task. The task TransferMoney is the only means to achieve this subgoal.

The task ReplyDebitRequest can be further decomposed into the following subprocesses. If the customer's account has enough credit to pay for the order, the bank clerk accepts the debit request. This is modeled as a subtask AcceptDebit with a link annotation if(moneyEnough). If the customer's account does not have enough credit

to pay, then the bank clerk rejects the debit request. This is modeled as a subtask `RejectDebit` with a link annotation `if(notMoneyEnough)`.

The task `TransferMoney` is decomposed as follows. First, the bank clerk debits the customer's account; this is modeled as a subtask `DebitCustomerAccount`. Then, he credits the company's account; this is modeled as a subtask `CreditCompanyAccount`. Finally, he notifies the order processor that the payment for the order has been transferred; this is modeled as a subtask `ConfirmTransfer`. These subtasks are performed in sequence.

Next, let us explain the annotated SR diagram for the `StockClerk` with its three roles `UpdateStockProcessor`, `StockInformant`, and `ShipmentProcessor`. It appears in Figure 6.10(c). `StockInformant` only processes requests from the order processor to obtain stock for an order and to cancel a stock request for an order. This is modeled as a top task `ProcessStockRequesets`. This top task can be decomposed into two subtasks: a subtask of replying to a stock request whenever the order processor makes one, modeled as a subtask `ReplyStockRequest` with a link annotation `*whenever(requestedStock)`, and a subtask of replying to a request for canceling a stock request for an order, modeled as a subtask `CancelStockRequest` with a link annotation `*whenever(requestedCancel)`.

The `ShipmentProcessor` only processes requests from the order processor to ship an invoiced order. This is modeled as a top task `ProcessShipment`. This top task can be decomposed into one subgoal `Shipped(Item)` to supply the goal dependency between the order processor and the shipment processor `Shipped(Item)`. A link annotation `*whenever(requestedShip)` is attached to the link between this subgoal and its super-task. The subgoal can be achieved by a task `ShipOrder`, attached with a means-ends link.

The UpdateStockProcessor role maintains the free stock and on-hold stock information; this is modeled as a top task UpdateStock. Whenever an item is shipped by the shipment processor, the update stock processor will remove the item from the on-hold stock; this is modeled as a subtask RemoveFromHold with a link annotation *whenever(itemShipped). Whenever the stock informant accepts an order, the update stock processor will remove the item from the free stock and put it into the on-hold stock; this is modeled as a subtask PutOnHold with a link annotation *whenever(OrderAccepted). Whenever the stock informant accepts an request for canceling reserved stock for an order, the update stock processor will remove the item from the on-hold stock and put it back to the free stock; this is modeled as a subtask moveOnHoldToStock with a link annotation *whenever(stockCanceled). As we mentioned earlier, there is no need for communication actions between the roles UpdateStockProcessor and ShipmentProcessor and StockInformant because these roles are played by the same agent.

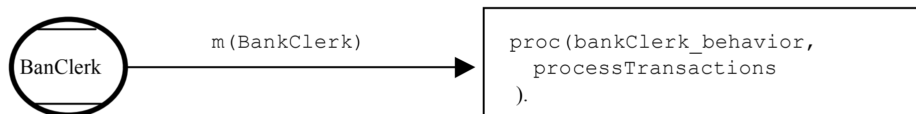
Finally, let us explain the annotated SR diagram for the Customer agent, which is shown in Figure 6.10(a). The Customer agent has a top task OtainItems(Item) which is decomposed into a task MakeOrder(Item), which means that the customer obtain the item from the mail-order company by making an order for the item.

6.4 Developing the Initial *ConGolog* Model

After the annotated SR diagrams of the mail-order business process has been specified, the initial *ConGolog* model can be developed by mapping elements of the SR diagrams into corresponding entities in the *ConGolog* model according to the mapping rules we specified in chapter 4. How to map roles, agents, tasks, goals, task decompositions, and goal decompositions into elements of a *ConGolog* model is explained in chapter 4 and some examples of how to map these components were shown in section 5.4. So we will

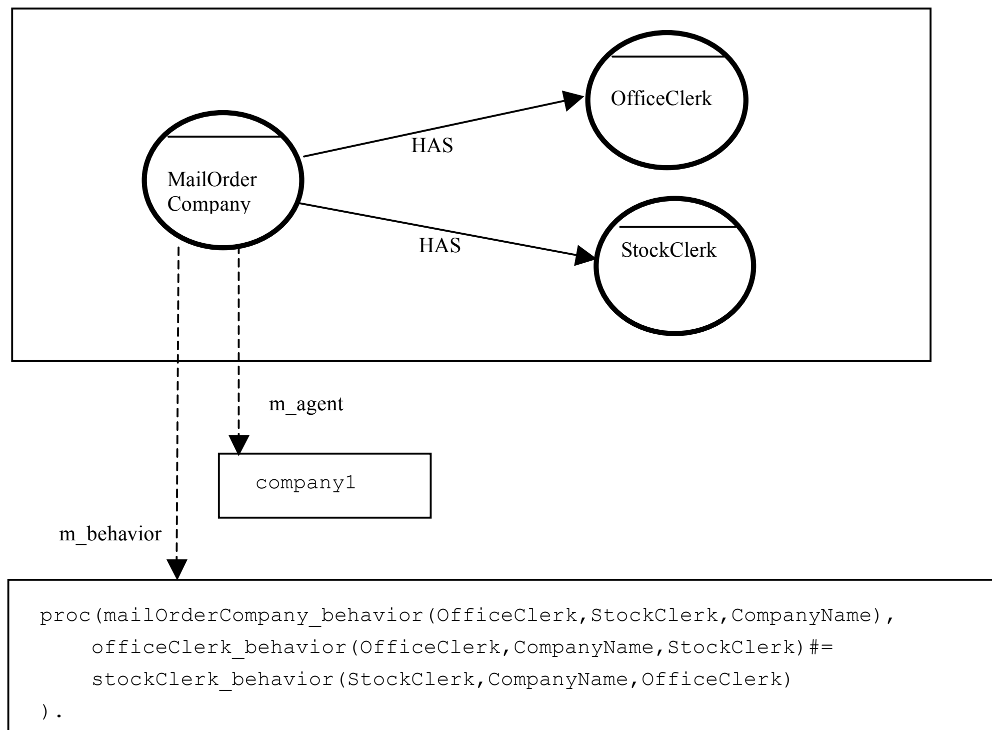
not explain this again here as we map the annotated SR diagrams of Figure 6.10 into a *ConGolog* model. There are some types of components in the annotated SR diagrams of Figure 6.10 whose mapping has not been explained before. For example, how does one map position nodes such as `Bank` and `BankClerk`? How does one map the links that specify that an agent such as `StockClerk` plays three different roles, such as `StockInformant`, `UpdateStockProcessor`, and `ShipmentProcessor`? We will now explain these cases in detail.

First, let us discuss how a position node can be mapped. We take the `BankClerk` position as an example. This node is mapped into a *ConGolog* procedure `bankClerk_behavior` which specifies the behavior this position. The mapping is as follows:



The task `ProcessTransaction` is the top-level node in this position. As we can see, positions are mapped just like roles.

Next, let us explain how to map an agent that has several sub-agents. We take the `MailOrderCompany` as an example. `MailOrderCompany` has both `StockClerk` and `OfficeClerk` agents working within its order processing business. The mapping is as follows:



The `MailOrderCompany` agent is mapped into a *ConGolog* procedure `mailOrderCompany_behavior` and an agent name `company1`. The procedure `mailOrderCompany_behavior` invokes the *ConGolog* procedure `officeClerk_behavior` which is mapped from the agent `OfficeClerk` and the *ConGolog* procedure `stockClerk_behavior` which is mapped from the agent `StockClerk`, and the behavior of these agents is executed concurrently. As we will see below, `OfficeClerk` agent will be mapped into a *ConGolog* procedure `officeClerk_behavior` and an office clerk `clerk1`. The `StockClerk` agent will be mapped into a *ConGolog* procedure `stockClerk_behavior` and an stock clerk `clerk2`. The relationship between the two sub-agents `OfficeClerk` and `StockClerk` and their super agent `MailOrderCompany` is reflected in the invocation of the procedures `officeClerk_behavior` and `stockClerk_behavior` inside the procedure `mailOrderCompany_behavior`. The behaviors of the sub-agents run concurrently in the super-agent.

Next, let us discuss the case where an agent plays one or more roles. For instance, the `OfficeClerk` agent plays the `EfficientOrderProcessor` role. This relationship is reflected in the mapping. The `OfficeClerk` agent is mapped into the *ConGolog* procedure `officeClerk_behavior`, specifying its behavior, and this procedure invokes the *ConGolog* procedure `efficientOrderProcessor`, specifying the behavior of the `EfficientOrderProcessor` role that the agent plays.

```
proc(officeClerk_behavior(OfficeClerk,CompanyName,StockClerk),
     efficientOrderProcessor(OfficeClerk,CompanyName,StockClerk)
).
```

The `StockClerk` agent plays three roles: `StockInformant`, `UpdateStockProcessor`, and `ShipmentProcessor`. So the `StockClerk` agent is mapped into the *ConGolog* procedure `stockClerk_behavior`, which invokes the *ConGolog* procedure `stockInformant`, specifying the behavior of the `StockInformant` role, the *ConGolog* procedure `updateStockProcessor`, specifying the behavior of the `UpdateStockProcessor` role, and the *ConGolog* procedure `shipmentProcessor`, specifying the behavior of the `ShipmentProcessor` role. These behaviors are executed concurrently inside their agent.

```
proc(stockClerk_behavior(StockClerk,CompanyName,OfficeClerk),
     stockInformant(StockClerk,CompanyName,OfficeClerk)#=
     updateStockProcessor(StockClerk,CompanyName)#=
     shipmentProcessor(StockClerk,CompanyName)
).
```

6.4.1 Developing the Initial *ConGolog* Model for `StockClerk`

The initial *ConGolog* model for the `StockClerk` agent is shown below. The `StockClerk` agent plays three roles `UpdateStockProcessor`, `StockInformant`, and `ShipmentProcessor`, and this is reflected in the *ConGolog* model as explained earlier. It can be checked that the *ConGolog* model corresponds to the annotated SR diagram of Figure 6.10(c) as required by the mapping rules.

```
/*behavior of the StockClerk agent*/
proc(stockClerk_behavior(StockClerk,CompanyName,OfficeClerk),
     stockInformant(StockClerk,CompanyName,OfficeClerk)#=
     updateStockProcessor(StockClerk,CompanyName)#=
     shipmentProcessor(StockClerk,CompanyName)
).

/* behavior of shipmentProcessor role */
proc(shipmentProcessor(StockClerk,Company),
     processShipment(StockClerk,Company)
).

proc(processShipment(StockClerk,Company),
     ==>([orderID,itemID,custID],
          and(val(orderItem(orderID),itemID),
           and(val(orderCustomer(orderID),custID),
            and(val(orderCompanyName(orderID),Company),
             and(invoiceMade(orderID),
              not(orderShipped(orderID))
         )))),
         achieve_ItemShipped(StockClerk,Company,custID,orderID,itemID)
    )
).

proc(achieve_ItemShipped(StockClerkName,CompanyName,Customer,OrderID,ItemID),
    [
     shipOrder(StockClerkName,CompanyName,Customer,OrderID,ItemID),
     orderShipped(OrderID)?
    ]
).

/* behavior of UpdateStockProcessor role */
proc(updateStockProcessor(StockClerk,Company),
     updateStock(StockClerk,Company)
).

proc(updateStock(StockClerkName,CompanyName),
     ==>([orderID,item],
         and(val(orderItem(orderID),item),
          and(val(orderCompanyName(orderID),CompanyName),
           and(stockRequestAccepted(orderID),
            not(onHoldPut(orderID))
         ))),
         putOnHold(StockClerkName,CompanyName,item,orderID)
     )
     #=
     ==>([orderID,item],
         and(val(orderItem(orderID),item),
          and(val(orderCompanyName(orderID),CompanyName),
```

```
            and(orderShipped(orderID),
             not(itemRmvFromHoldForshipment(orderID))
          ))),
          rmvFromHoldForShipment(StockClerkName,CompanyName,item,orderID)
      )
      #=
      ==>([orderID,item],
          and(val(orderItem(orderID),item),
           and(stockcancelled(orderID),
            and(val(orderCompanyName(orderID),CompanyName),
             not(stockRtndToInventory(orderID))
          ))),
          moveOnHoldBackToStock(StockClerkName,CompanyName,item,orderID)
     )
).

/* behavior of StockInformant role */
proc(stockInformant(StockClerkName,CompanyName,OfficeClerk),
      processStockRequest(StockClerkName,CompanyName,OfficeClerk)
).


proc(processStockRequest(StockClerkName,CompanyName,OfficeClerk),
      ==>([orderID,itemID],
           and(requestedStock(itemID,orderID),
            not(stockRequestAnswered(orderID))
           ),
           replyStockRequest(StockClerkName,CompanyName,OfficeClerk,orderID,itemID)
      )
      #=
      ==>([orderID,itemID],
           and(stockRequestCancelled(orderID,itemID),
            not(stockRtndToInventory(orderID))
           ),
           cancelStockRequestProcess(StockClerkName,CompanyName,OfficeClerk,
                                                        orderID,itemID)
      )
).

proc(replyStockRequest(StockClerkName,CompanyName,OfficeClerk,OrderID,ItemID),
      if(some(n,and(val(inStock(ItemID),n),n > 0)),
         acceptRequestStock(StockClerkName,CompanyName,OfficeClerk,ItemID,OrderID),
          rejectStockRequest(StockClerkName,CompanyName,OfficeClerk,ItemID,OrderID)
       )
).

proc(cancelStockRequestProcess(StockClerkName,CompanyName,OfficeClerk,
                                                   OrderID,ItemID),
     [
        confirmCancelStock(StockClerkName,CompanyName,OfficeClerk,OrderID,ItemID),
         stockRtndToInventory(OrderID)?
     ]
).

proc(acceptRequestStock(StockClerkName,CompanyName,OfficeClerk,ItemID,OrderID),
     [ acceptStockRequest(StockClerkName,CompanyName,OfficeClerk,ItemID,OrderID),
       onHoldPut(OrderID)?
     ]
).
```

The initial *ConGolog* models for the `OfficeClerk` role, `BankClerk` position, and `Customer` agent contain no new features and are shown in Appendix B-4, B-5, and B-6 respectively.

6.4.2 Specifying the Domain Dynamics

We must produce the *ConGolog* domain dynamics specification for the mail-order business application as we did for the meeting scheduling application in chapter 5. Primitive actions and fluents are introduced to model aspects of the domain. Precondition axioms, successor state axioms, and initial state axioms are also given to specify their dynamics. See Appendix B-7 for a list of all actions and fluents in the *ConGolog* model for the mail-order business process. Here we will give some examples to illustrate the *ConGolog* model.

- **Primitive Actions**

The primitive actions in the mail-order domain include:

`mkOrder(Customer1,Item1,CardNo1,Company1)`, i.e., `customer1` makes an order for `Item1` to `Company1` and the order is to be charged to `CardNo1`,

`transferMoneyForOrder(OfficeClerk,Company,Customer,Order,CardNo,Amt)`, i.e., `OfficeClerk` asks the bank clerk to transfer `Amt` of money from the `Customer`’s account `CardNo` to the `Company`’s account to pay for the `Order` made by `Customer`.

- **Predicate Fluents**

The primitive predicate fluents include:

orderMade(OrderID), which represents the fact that an order OrderID has been made to the company (the attributes of the order are modeled by functional fluents, e.g., OrderCustomer(OrderID) denotes the order's customer),

transferMoneyAccepted(OrderID), which represents the fact what the bank clerk has transferred the money for the order OrderID.

- **Functional Fluents**

The functional fluents include:

price(Item), which represents the price for the Item,

inStock(Item), which represents the quantity of Item that are currently in stock,

onHold(Item), which represents the quantity of Item that are currently on-hold for some orders.

acctBalance(CardNo), which represents the current balance of account CardNo.

The defined fluents include:

stockRequestAnswered(Order), which becomes true when the stock informant has accepted the stock request or rejected the stock request for Order,

debitRequestAnswered(Order), which becomes true when the bank clerk has confirmed or rejected the debit request for Order.

- **Precondition Axioms**

The precondition axioms include:

```
poss(putOnHold(Item,_),S):- holds(val(inStock(Item),N),S),
N > 0,
```

which means that only when the stock for `Item` is greater than zero, can `Item` be put on hold for some order, otherwise the action `putOnHold` cannot be performed.

- **Successor State Axioms**

The successor state axioms include:

```
holds(stockRequestAccepted(OrderID),do(A,S)):-
      A = acceptStockRequest(OrderID);
      holds(stockRequestAccepted(OrderID),S),
```

The axiom means that the stock request for the order `OrderID` has been accepted in situation `do(A,S)` if and only if the action `A` is the stock informant's accepting the stock request or if the stock request had already been accepted in situation `S`.

See Appendix B-7 for a complete list of the action precondition and successor state axioms specified in the *ConGolog* model for the mail-order business domain. .

6.5 Validating the *ConGolog* Model by Simulation

In the next step, the *ConGolog* model is evaluated through simulation and its shortcomings are identified. For simulation, one must first specify a system instance and the initial state of the simulation.

6.5.1 Specifying a System Instance

To specify an instance of the overall system, one first defines a main procedure, which corresponds to the whole system's behavior:

```
proc(main,
      customer(cust1,item4,1111,company1)
              /* this can be adjusted according to the situation */
      #>
      mailOrderCompany_behavior(officeClerk2,stockClerk2,company2)
      #=
      mailOrderCompany_behavior(officeClerk1,stockClerk1,company1)
      #=
      bank_behavior
 ).
```

This main procedure specifies the actors involved in the mail-order business process and their behavior by invoking the corresponding actor procedures and assigning specific individuals to the role, position, and agent parameters. In the system instance specified above, there is one customer agent `cust1` whose account number is `1111`, two mail-order company agents `company1` and `company2`, and the bank position. The bank position is not assigned to an individual because we don't care who this bank is. "#=" means that the behaviors of the actors are performed concurrently without any priority. "#>" means that the behavior of the actor on the left side has higher priority to perform its actions. This is used here to ensure that the customer `cust1` makes an order first, and then the mail-order companies, the bank, and their sub-actors get to execute and process this order.

Then, the behavior of the actors is specified. The behavior of the customer agent is specified by the following procedure:

```
proc(customer(CustID,Item,CardNo,Company),
      obtainItem(CustID,Item,CardNo,Company)
).
```

The customer agent performs one task: to obtain `Item` from `Company`; the customer's name is `CustID` and its account is `CardNo`.

The behavior of the mail-order company agent is specified as shown in the previous section:

```
proc(mailOrderCompany_behavior(OfficeClerk,StockClerk,CompanyName),
     officeClerk_behavior(OfficeClerk,CompanyName,StockClerk)#=
     stockClerk_behavior(StockClerk,CompanyName,OfficeClerk)
).
```

That is, the mail-order company has two sub-agents: the office clerk and the stock clerk. So the procedures corresponding to the behaviors of the office clerk and stock clerk are invoked concurrently inside the procedure corresponding to the mail-order company. The names of the instances of the company, office clerk, and stock clerk are assigned to the `CompanyName`, `OfficeClerk`, and `StockClerk` parameters by the main procedure.

The bank and the bank clerk position procedures have their behaviors specified as follows:

```
proc(bank_behavior,
    bankClerk_behavior

).

proc(bankClerk_behavior,
    processTransactions

).
```

The bank position has one sub-position, the bank clerk, who works for it. The bank clerk is responsible for processing the transactions for checking customers' credit and transferring payment from customers' accounts to company’s accounts. No specific individual is assigned to these positions.

The office clerk behavior is specified as shown earlier:

```
proc(officeClerk_behavior(OfficeClerk,CompanyName,StockClerk),
     efficientOrderProcessor(OfficeClerk,CompanyName,StockClerk)
).
```

That is, it plays the role of an efficient order processor.

The stock clerk is also specified as shown earlier:

```
proc(stockClerk_behavior(StockClerk,CompanyName,OfficeClerk),
     stockInformant(StockClerk,CompanyName,OfficeClerk)#=
     updateStockProcessor(StockClerk,CompanyName)#=
     shipmentProcessor(StockClerk,CompanyName)
).
```

That is, it plays three roles: stock informant, update stock processor, and shipment processor. The stock clerk performs the behaviors for these roles concurrently.

6.5.2 Simulation Examples

The complete *ConGolog* model for our mail-order process example appears in Appendix B-7. We will now go over some simulation examples. Appendix B-1 contains the complete simulation trace for our examples. The initial state of the system for all our examples is as follows:

```
holds(val(creditLimit,-10),_).        /* The credit limits of all account is –10. */

holds(val(inStock(item1),10),s0).
holds(val(inStock(item2),0),s0).
holds(val(inStock(item3),3),s0).
  /* Initially, the stock for item1 is 10, for item2 is 0, and for item3  is 3.*/

holds(val(acctBalance(1111),100),s0).
holds(val(acctBalance(2222),20),s0).
holds(val(acctBalance(3333),0),s0).
  /* Initially, the account balance for the card number 1111 is 100, */
  /* for the card number 2222 is 20, and for the card number 3333 is 0. */

holds(val(acctBalance(c1),0),s0).
holds(val(acctBalance(c2),0),s0).
  /* Initially, the account balance for companies' accounts c1 and c2 are both 0. */
```

```
holds(val(price(item1),10),s0).
holds(val(price(item2),20),s0).
holds(val(price(item3),30),s0).
  /* The price for item1 is 10, for item2 is 20, and for item3 is 30. */

non_fluent(isSoldItem(_)).
isSoldItem(item1).
isSoldItem(item2).
isSoldItem(item3).
 /* item1, item2, and item3 are sold items, and other items are not sold. */
```

Now, let us look at our simulation examples.

Example 1: A customer makes an order for `item4` which is not a type of item sold:

```
/*the customer CUST1 make an order for item4 which is not a type of item sold */
 proc(main,
      customer(cust1,item4,1111,company1) #>
      mailOrderCompany_behavior(officeClerk1,stockClerk1,company1)#=
      mailOrderCompany_behavior(officeClerk2,stockClerk2,company2)#=
      bank_behavior
 ).
```

The sequence of actions performed is as follows:

```
mkOrder(cust1,item4,2222,company1)
  /* cust1 orders a non-sold item4 from company1 and his credit card number is 1111 */

alarmCustomer(officeClerk1,company1,cust1,1,item4))
  /* officeClerk1 in company1 alarms cust1 that item4 is not a sold type.

do(rejectOrder(officeClerk1,company1,cust1,1,item4)
  /* officeClerk1 in company1 rejects cust1's order for item4. */
```

The trace result shows that if the ordered item is not of a sold type, then the office clerk rejects the customer's order.

Example 2: A customer makes an order for an item `item2` that is of a sold type to `company1`, but `item2` is out of stock:

```
proc(main,
  customer(cust1,item2,1111,company1) #>
  mailOrderCompany_behavior(officeClerk1,stockClerk1,company1)#=
  mailOrderCompany_behavior(officeClerk2,stockClerk2,company2)#=
  bank_behavior
 ).
```

The trace of actions performed in the simulation is:

`mkOrder(cust1,item2,2222,company1)`
/* `cust1` makes an order to `company1` for `item2`, and his credit card number is `1111`. */

`requestStock(officeClerk1,company1,stockClerk1,item2,1)`
/* `officeClerk1` in `company1` requests `stockClerk1` to provide stock for ordered `item2.*/`

`rejectStockRequest(stockClerk1,company1,officeClerk1,item2,1)`
/* `stockClerk1` in tells `officeClerk1` that there is no stock for the ordered `item2`. */

`rejectOrder(officeClerk1,company1,cust1,1,item2)`
/* `officeClerk1` in `company1` rejects the order from `cust1` for the ordered `item2`. */

The trace shows that if an ordered item is out of stock, then the office clerk rejects the customer's order for this item.

Example 3: A customer makes an order for an item `item3` that is in stock, but the customer does not have enough money to pay for this order:

```
proc(main,
   customer(cust3,item3,3333,company1) #>
   mailOrderCompany_behavior(officeClerk1,stockClerk1,company1)#=
   mailOrderCompany_behavior(officeClerk2,stockClerk2,company2)#=
   bankClerk
).
```

The trace of actions performed is:

`mkOrder(cust3,item3,3333,company1)`
/* `cust3` makes an order for `item3` from `company1` and his credit card number is `3333`. */

`requestStock(officeClerk1,company1,stockClerk1,item3,1)`
/* `officeClerk1` in `company1` requests `stockClerk1` to provide */
/* the stock for `item3` for order no.1 */

acceptStockRequest(stockClerk1,company1,officeClerk1,item3,1)
/* stockClerk1 in company1 accepts the stock request for item3 of order no.1. */

putOnHold(stockClerk1,company1,item3,1)
/* stockClerk1 in company1 puts one of the item3 into the on-hold stock. */

requestDebit(officeClerk1,company1,1,cust3,3333,30)
/* officeClerk1 in company1 requests the bank to check whether cust3's */
/* credit account 3333 has enough money to pay the amount of 30 for order no.1. */

rejectDebit(1,company1,cust3,3333,30)
/* The bank rejects the request to check the possibility of debiting 30 */
/* from cust3's card number 3333 for order no.1 */

cancelStockRequest(officeClerk1,company1,stockClerk1,item3,1)
/* officeClerk1 in company1 asks stockClerk1 to */
/* cancel the reserved stock for item3 for order no.1. */

confirmCancelStock(stockClerk1,company1,officeClerk1,1,item3)
/* stockClerk1 confirm canceling the reserved stock for item3 to officeClerk1 */

moveOnHoldBackToStock(stockClerk1,company1,item3,1)
/* stockClerk1 in company1 moves an item3 from the on-hold stock back to real stock. */

rejectOrder(officeClerk1,company1,cust3,1,item3)
/* officeClerk1 in company1 rejects order no.1 made by cust3 for item3. */

The trace shows that when the ordered item is in stock, but the customer does not have enough money to pay for the ordered item, then the office clerk gets the stock clerk to cancel the reservation of stock for the ordered item and rejects the customer's order.

Example 4: A customer makes an order for a sold item that is in stock and he has enough money to pay for this order:

```
proc(main,
   customer(cust3,item1,3333,company1) #>
   mailOrderCompany_behavior(officeClerk1,stockClerk1,company1)#=
   mailOrderCompany_behavior(officeClerk2,stockClerk2,company2)#=
   bank_behavior
).
```

The trace is:

mkOrder(cust3,item1,3333,company1)
/* cust3 makes an order for item3 from company1 and his credit card number is 3333. */

```
requestStock(officeClerk1,company1,stockClerk1,item1,1)
  /* officeClerk1 requests stockClerk1 to provide stock for item1 for order no.1. */

acceptStockRequest(stockClerk1,company1,officeClerk1,item1,1)
  /* stockClerk1 accepts officeClerk1's the stock request for item1 for order no.1. */

putOnHold(stockClerk1,company1,item1,1)
  /* stockClerk1 in company1 puts an item1 into the on-hold stock for order no.1. */

requestDebit(officeClerk1,company1,1,cust3,3333,10)
  /* officeClerk1 in company1  requests the bank to check cust3's account 3333 */
  /* to see whether cust3 has enough money to be debited 10 for order no.1. */

acceptDebit(1,company1,cust3,3333,10))
  /* The bank tells company1 that cust3's account 3333 can be debit 10. */

transferMoneyForOrder(officeClerk1,company1,cust3,1,3333,10)
  /* officeClerk1 in company1 requests the bank to transfer an amount of 10  */
  /* from cust3's account 3333 into company1's account . */

debitAcct(3333,10)  /*  The bank debits 10 from the account 3333. */

creditAcct(c1,10)   /*  The bank credits 10 into the account c1. */

confirmTransferMoney(2,cust3,3333,company1,10)
  /* The bank confirms that an amount of 10 was transferred from the account 3333 */
  /* into the company1's  account. */

mkInvoice(officeClerk1,company1,stockClerk1,item1,1)
  /* officeClerk1 makes an invoice for item1 in order no.1 and gives it stockClerk1  */

shipOrder(stockClerk1,company1,cust3,1,item1)
  /* stockClerk1 ships  item1 for order no. 1 to  cust3. */

rmvFromHoldForShipment(stockClerk1,company1,item1,1)
  /* stockClerk1 in company1 removes an item3 from the on-hold stock  for shipment. */

notifyShipment(officeClerk1,company1,cust3,item1,1)
  /* officeClerk1 notifies cust3  that item1 was shipped to him for order no.1. */
```

The trace shows that if the customer has enough money to pay for the ordered item and if the ordered item is in stock, then the process will be completed successfully. The stock clerk will ship the ordered item and the office clerk will notify the customer that the item has been shipped.

Example 5: Our *ConGolog* model also supports multiple orders made by various customers to different mail-order companies. In the following system instance, there are three customers: `cust1`, `cust2`, and `cust3`, and two mail-order companies: `company1` and `company2`. Four orders are made, as specified by the main procedure below:

```
proc(main,
   customer(cust1,item2,1111,company1) #=
   customer(cust1,item1,1111,company1) #=
   customer(cust3,item3,3333,company2) #=
   customer(cust2,item2,2222,company2) #>
   mailOrderCompany_behavior(officeClerk2,stockClerk2,company2)#=
   mailOrderCompany_behavior(officeClerk1,stockClerk1,company1)#=
   bank_behavior
).
```

The four orders are:

Order no. 1: `cust1` makes an order for `item2` from `company1` and his card number is `1111`

Order no. 2: `cust1` makes an order for `item1` from `company1` and his card number is `1111`.

Order no. 3: `cust3` makes an order for `item3` from `company2` and his card number `3333`.

Order no. 4: `cust2` makes an order for `item2` from `company` 2 and his card number is `2222`.

The simulation trace is long and appears in Appendix B-2. It shows the following:

For order no. 4, `officeClerk2` requests `StockClerk2` to provide stock for `item2`, but `stockClerk2` rejects the stock request because `item2` is out of stock in `company2`. Then `officeClerk2` rejects this order made by `cust2` without proceeding into processing payment.

For order no. 3, officeClerk2 requests StockClerk2 to provide stock for item3, and stockClerk2 accepts the stock request and puts the reserved stock for item3 on hold. Then, officeClerk2 requests the bank to check whether cust3 has enough money to pay the order in the account 3333. The bank finds out that cust3 does not have enough money to pay for the order and notifies officeClerk2. Then, officeClerk2 requests stockClerk2 to cancel the reserved stock of item3. stockClerk2 confirms his cancellation and removes the reserved stock for item3 from the on-hold stock back to real stock. Then, officeClerk2 rejects the order no. 3 made by cust3.

For order no. 2, officeClerk1 first requests stock for item1 from stockClerk1. Then, stockClerk1 accepts the stock request and puts an item1 on hold. Then, officeClerk1 requests the bank to check cust1's account 1111 and the bank confirms that cust1 has enough money to pay for the ordered item1. Then, officeClerk1 asks the bank to transfer payment for the ordered item1 from cust1's account into company1's account. Then, the bank debits cust1's account, and credits the same amount of money into company1's account. Then, officeClerk1 makes an invoice for the ordered item1 and stockClerk1 ships the ordered item1 to cust1. Finally, officeClerk1 notifies cust1 that the ordered item1 has been shipped.

For order no. 1, the process is similar to the one for processing order no. 4. Because the stock for item2 has run out, officeClerk1 rejects the order without proceeding into processing payments.

6.6 Refining the *i** and *ConGolog* Models Based on Validation Results

The example simulation traces presented show that our *ConGolog* model of the mail-order business behaves as expected. Here, we want to briefly show how the model can be modified when one finds that some aspects of the specification are not as expected or as desired. We discuss how the model could be changed to allow new supplies of items to arrive as the system operates. Also, we discuss how alternatives for the process could be modeled with our methodology.

6.6.1 Modifying the *ConGolog* Model and Corresponding Parts of the *i** Model — An Example

In our model of the mail-order business process, the stock for items never increases and getting new supplies is not modeled. Suppose that we want to allow for this. Then, we could model the reception of supplies in our *ConGolog* model as an exogenous action `supply(Item, Quantity)`. We really don't care who will supply the new stock. So we leave out the supplier. This modification of the *ConGolog* model is specified as follows.

We specify the precondition axiom for this exogenous action:

```
poss(supply(Item,_),S):-holds(isSoldItem(Item),S).
```

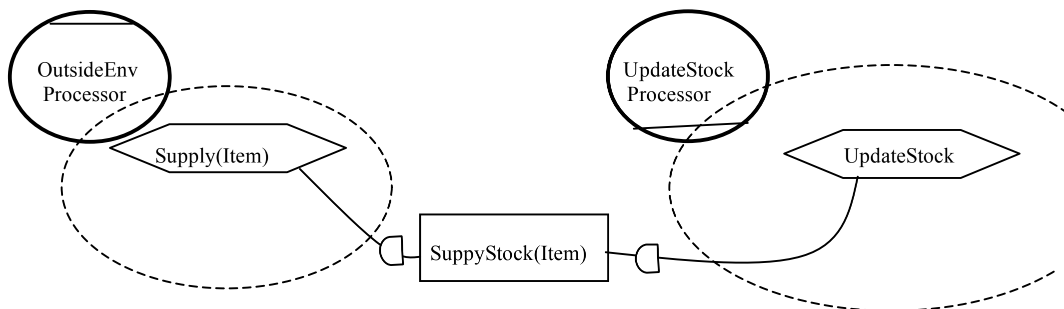
i.e., only sold items can be supplied.

We modify the successor state axioms for the fluents that are affected by the exogenous action:

```
holds(val(inStock(Item),N),do(A,S)):-
   (A = rmvFromHold(_,_,Item,_),
    holds(val(inStock(Item),M),S), N is M + 1);
   (A = supply(Item,Q),
    holds(val(inStock(Item),M),S), N is M + Q);
   (A = putOnHold(_,_,Item,_),
    holds(val(inStock(Item),M),S), N is M - 1);
   (holds(val(inStock(Item),N),S), ground(Item),
    A \= rmvFromHold(_,_,Item,_), A \= supply(Item,Q),
    A \=putOnHold(_,_,Item,_)).
```

i.e., the stock for `Item` will increase by `Q` when the exogenous action `supply(Item,Q)` is performed (shown by the bold line above).

We also have to modify the corresponding elements in the annotated SR diagram. We introduce an outside environment agent `OutSideEnv` who will performs the exogenous action `supply(Item,Quantity)`. The fluent modeling the stock of items will be affected and the update stock processor maintains the stock. We view this relationship between the outside environment and the update stock processor as a resource dependency `SupplyStock(Item)`, which is modeled as follows.



Once the modifications have been done, we can validate the modified model by simulation.

Example 6: There are two orders for `item2` which is out of stock:

Order no. 1: `cust2` makes an order for `item2` to `company1` and his card number is `2222`.

Order no. 2: `cust1` makes an order for `item2` to `company1` and his card number is `1111`.

We specify the system instance with the following main procedure:

```
proc(main,
   customer(cust2,item2,2222,company1) #=
   customer(cust1,item2,1111,company1) #>
   bank_behavior#>
   mailOrderCompany_behavior(officeClerk2,stockClerk2,company2)#=
   mailOrderCompany_behavior(officeClerk1,stockClerk1,company1)
).
```

The simulation trace for this example is long and appears in Appendix B-3. The exogenous action `supply(item2,6)` is generated during the execution. In the simulation trace, before the exogenous action `supply(item2,6)` is performed, the stock for `item2` is `0`. So `officeClerk1` rejects `cust1`'s order for `item2` in order no. 2 because there is no stock when it tries to obtain it. After the exogenous actions occurs, the stock for `item2` increases to `6`. So when OfficeClerk1 later requests stock for `cust2`'s order for `item2` the request will be accepted and the order can be shipped. In the simulation, the process of handling orders is the same as before except the stock being supplied by the exogenous action.

6.6.2 The Process Alternatives for the Example

In the above *ConGolog* model, we suppose that the processing of orders proceeds in a certain way. We mentioned at the beginning of chapter 6 that there is another option for the process. After the customer has made an order for an item, the office clerk could request the bank to debit the customer for the order first before checking whether the company has stock for it. If the customer cannot pay, then the office clerk rejects the order without proceeding to request stock for the order. On the other hand, if the debit

goes through, but later it is found that there is no stock, then the company must get the bank to credit the customer back for his payment.

Let us call the alternative process described above alternative 2, and the one that we have modeled alternative 1. If we compare the two alternatives, we can make the following observations:

- These two alternatives don't differ significantly in terms of processing orders. Both alternatives can successfully complete the whole process of processing orders.
- Alternative 1 grants higher priority to processing stock for the order than to processing payment for it. This is reasonable because stock availability can be determined within the company. The shortcoming is that if stock is reserved for an order, and later the company finds out that the customer has no enough money to pay for it, then the reserved stock has to be returned to free stock. Moreover orders that come in during the interval cannot be allocated the reserved stock and have to wait. If the item involved is expensive, high profit, and in short supply, then a waiting customer might cancel his order because of the waiting time. The profit of the company may be affected. On the other hand, this alternative minimizes bank transactions. This may lead to lower transaction fees for the company.
- Alternative 2 grants higher priority to processing payments than to processing stock requests. This can be good for the company, because the company wants to make sure that the customer has enough money to pay for the order before it puts efforts into processing the stock and shipping the ordered items. If the customer does not have money to pay for the order, the company doesn't want to spend time on the order and hold stock for it especially for expensive items. On the other hand, this alternative leads to more bank transactions. If the company finds out there is no stock for the ordered item, then it has to have the bank to refund payment to the customer's account. This could lead to higher bank fees for the company and a higher error rate.

- For the customer, when he makes sure that he has enough money to pay the order, it is better if he is allocated stock earlier because then he cannot lose it to another customer. So he would prefer alternative 1.
- We would analyze the above two alternatives for processing orders in terms of the interests of each actor using *i** notions, such as contribution to softgoals, workability, believability, etc.
- In our *i** and *ConGolog* models, we can model the two alternatives together as well as separately. In *i** modeling alternatives together is the normal way to proceed (as we saw in the early examples of chapter 5). The alternatives are represented as different means for achieving goals or as different alternative task decompositions. In *ConGolog*, we can use the nondeterministic constructs provided by *ConGolog* to specify the alternative processes in one model.

We leave modeling these two alternatives together for future work. Also we could assign the `StockInformant` role into a computerized inventory management system agent. We could modify the selected process to include this computerized component easily. This would be an interesting modification to study too.

7 Discussion

Our methodology supports the combined use of the *i** and *ConGolog* frameworks. We have evaluated the methodology in light of two case studies: a meeting scheduling application and a mail-order business process. In this chapter, we evaluate our approach in light of the case studies. We also discuss various issues involved in mapping *i** and *ConGolog*.

7.1 An Evaluation of the Methodology

Our preliminary work suggests the following advantages to our methodology:

- *Using i* is helpful for capturing intentional goals and the rationales behind the selected process, and analyzing actor vulnerabilities.*

The SD model describes the dependency relationships between the actors involved in the process, and helps in identifying stakeholders, analyzing opportunities and vulnerabilities, and recognizing patterns of relationships, such as various mechanisms for mitigating vulnerability. The SD model shows external (but nevertheless intentional) relationships among actors, while hiding the intentional constructs within each actor. The SD model can be useful in understanding organizational and systems configurations as they exist, or as proposed new configurations.

The SR model provides a way of modeling stakeholder interests, and how they might be met, and the stakeholder's evaluation of various alternatives with respect to his interests. Task-decomposition links provide a hierarchical description of intentional elements that make up a *routine*. The means-ends links in the SR model provide understanding about why an actor would engage in some tasks, pursue a goal, need a resource, or want a softgoal. From the softgoals, one can tell why one alternative may be chosen over others.

- *Using ConGolog is helpful for modeling complex processes involving loops, concurrency, multiple agents etc., producing formal specifications, and validating them by simulation and verification.*

ConGolog is based on a logical formalism, the situation calculus. It is very expressive and fully formal. It is well adapted to the late-requirements-engineering and early-design stages of system development, when detailed alternative process designs have to be specified and need to be compared. The *ConGolog* framework can be used to model complex processes involving loops, nondeterminism, concurrency and multiple-agents. Because of its logical foundations, *ConGolog* can accommodate incompletely specified models, either in the sense that the initial state of the system is not completely specified, or in the sense that the processes involved are nondeterministic and may evolve in any number of ways. These features are especially useful when one models business process and open-ended real world situations. A process simulation tool can be used for process model validation. The framework also supports verification.

- *Both graphical/informal and non-graphical/formal notations are used, which supports a progressive specification process and helps in communicating with the clients.*

The *i** SD model uses a graphical notation involving actor nodes and dependency relationships to represent the intentional relationships between actors. The *i** SR model uses a graphical notation involving task/goal/resource/softgoal internal nodes and mean-ends and task-decomposition links, to represent the intentional behavior inside the actors and the rationale behind their activities. *i** provides analysis methods for early-phase RE using notions such as ability, workability, viability, believability, etc., which help the analyst understand the process, how actors' and stakeholders' goals can be met, and how alternatives can be chosen.

The *ConGolog* framework has a fully logical semantic based on the situation calculus. It supports precise modeling of the actions performed by agents, when the actions can be

performed, what the result of performing the actions is, how the actions are composed in the process, and how the whole process proceeds under given initial conditions. Process evolution can be traced using a simulation tool that can be used for validating the process specification. Verification can also be performed.

The annotated SR diagram notation that we have developed is both graphical and intuitive and allows a precise and formal specification of processes.

- *Several process alternatives can be studied and compared; simulation, which is supported by the ConGolog framework, will help the modeler and client choose suitable alternatives.*

Using the *i** model, the modeler can analyze actor vulnerability based on the dependency network and choose alternatives for the process based on this and on contribution to softgoals. Different alternatives selected by *i** analysis can be mapped into *ConGolog* models. Simulation can be performed on the models for different system instances and initial states. The results can help the modeler and client choose a suitable process alternative.

Disadvantages of our methodology identified in our study include:

- *The methodology does not currently have tool support except for simulation.*

This hinders the analyst in applying the methodology. But work is in progress to address this. Yu and his colleagues are working on a support tool for developing an *i** model from the initial system requirements and performing analysis. There is also a graphical viewer tool for displaying simulations of *ConGolog* models. These tools need to be integrated and extended in future work. Problems with traceability may arise too. We discuss this in chapter 8.

- *It is not clear how to go from requirements analysis to the design phase*.

ConGolog can be used to produce a preliminary design for the whole process of the system that focuses on the main alternative and suppresses unnecessary details. But how to obtain a whole system design specification from the resulting requirements specification is not addressed by our methodology. This could cause problems for traceability. We discuss this in chapter 8.

7.2 Issues in Mapping *i** to *ConGolog*

There are several issues that should be explored further with respect to mapping *i** SR models into *ConGolog* models and the definition of mapping rules:

- How complete must the mapping be? Must all SR diagram nodes and links be mapped? Must all *ConGolog* procedures, actions, and fluents be mapped into? It may be the case that some goals or tasks in the SR diagram are not important to modeling and specifying the core processes of interest. Here we allow the modeler to suppress unimportant nodes and associated links in the annotated *i** SR diagram before defining the mapping. But this could be resolved in another way. From a practical point of view, a possible answer to the question is that the mapping must be complete enough to allow analysis through simulation or verification.

- How should parameters in procedures and goals be handled? In *i** diagrams, they are often absent, while in *ConGolog*, they are always listed explicitly. Perhaps we can think of them as present in *i** diagrams, but kept hidden unless explicitly made visible (a tool could easily support that).

- Goals have both a declarative and procedural interpretation. The associated procedure specifies a selected set of means for achieving the goal (usually not complete), and the procedure must achieve the goal to terminate. Is this treatment satisfactory?

- The diagram notation does not support well the distinction between a generic system and a system instance (one used in particular simulation experiment). How do we extend it to capture this?

- We haven't distinguished between design goals and execution-time goals. But the distinction can be fuzzy. Should we distinguish, and if so, how?

- Must the incompleteness of *i** diagrams be captured in the *ConGolog* model resulting from the mapping? *i** diagrams are not assumed to be complete. There may be ways of achieving goals or performing tasks that are not represented. In *ConGolog* models, however, all of the alternative ways to accomplish a task/goal are assumed to be specified (since tasks/goals are specified by procedures). Here, we suppose that a "closure assumption" is made when going from the SR diagram to the annotated SR diagram. One could also write open *ConGolog* specifications, for example by mapping a goal `g` into a procedure as follows:

```
 proc(achieve_Goal_g,
          [task1 $ task2 $ (pi( a , a)@, g?]
)
```

  This says that one can achieve goal `g` by doing `task1`, or by doing `task2`, or by doing zero or more actions after which `g` is true (`@` is the nondeterministic iteration operator).

- Can non-annotated SR models be mapped to *ConGolog* models? We have not pursued this, but we think it is possible. This would involve imposing much weaker constraints in mapping elements of the SR models to elements of the *ConGolog* models. For example, when there is a decomposition link between a subtask node and a super-task node, we could only require that there be some execution of the super-

task in the *ConGolog* model that involves executing the subtask. This can be specified in the *ConGolog* semantics as follows:

$$\exists\ s,s',s_1,s_2\ \ (\mathrm{Do}(m(\text{super-task}),\ s,\ s')\ \wedge$$
$$\mathrm{Do}((m(\text{subtask})\ \|\ pi(a,a)@),\ s_1,\ s_2)\ \wedge$$
$$s <= s_1\ \wedge\ s_2 <= s').$$

Here $\mathrm{Do}(\sigma,\ s_1,\ s_2)$ means that there is an execution of process σ that starts in situation s_1 and terminates in situation s_2. We use `pi(a,a)@` to allows other concurrent activities to be performed during the interval $[s_1,s_2]$. `m(subtask)` is the result of the mapping for the subtask and `m(super-task)`is the result of the mapping for the super-task.

The mapping rules can be viewed as giving a formal semantics to annotated SR diagrams by mapping this notation into *ConGolog*, a language which already has one. We believe that this semantics is largely consistent with the somewhat abstract (based on the notion of an actor having a routine) axiomatic semantic for *i** developed in [YU95B]. As such, it could perhaps be viewed as a formal semantics for SR diagrams more generally. But as mentioned above, one point where the two semantics diverge is with respect to completeness: in *i**, task/goal decompositions are generally not assumed to be complete, but in *ConGolog* and in our mapping rules they are assumed to be. Should we try to accommodate incompleteness? Should we distinguish between a set of task/goal decompositions and its completion? More study of these questions is required.

8 Conclusion

This thesis has developed a methodology for the combined use of the *i** and *ConGolog* frameworks for requirements engineering. The methodology allows the requirements engineer to exploit the complementary features of the two frameworks to develop better models of the application of interest and produce requirements specifications that fulfill the client's goals.

8.1 Contributions

We can summarize the main contributions of the thesis as the follows:

- *A methodology for the combined use of the i* model and the ConGolog framework for requirements engineering has been developed.*

The methodology was presented in chapter 4 and tested in two case studies in chapters 5 and 6. The methodology involves using the *i** framework to perform early-phase RE, that is, model and analyze intentional relationships between actors, the rationale behind their activities, vulnerabilities and opportunities for actors, and compare different alternatives for the process. It also involves using an intermediate notation, annotated SR diagrams, to specify processes precisely so that they can be mapped into the *ConGolog* framework. Finally, the methodology involves using the *ConGolog* model of the process to validate the specification by performing simulation experiments. This shows whether the process proceeds as the modeler expected. The methodology also allows for modifying the *i** and *ConGolog* models based on the clients' opinion and the simulation results.

- *To support the methodology, a set of annotations was introduced into the i* SR diagram notation to allow more detailed information about processes to be represented.*

Link annotations are used to specify under what conditions a task/goal should be performed, and whether it should be performed repeatedly. Composition annotations are used to specify whether the subtasks/subgoals of a decomposition should be performed concurrently, sequentially, concurrently with different priorities, or whether they are alternatives. The annotations allow the analyst to specify the details of how a process should proceed and help him and the client clarify their system specification choice. A formal semantics for these annotations is defined in chapter 4 through a mapping into *ConGolog*.

- *We have explained how annotated i* SR diagrams can be developed. These bridge the gap between the i* and ConGolog models.*

This involves among others things the operationalization of dependencies to clarify the communication behavior of the different actors to achieve the goal/perform the task/provide the resource. Decomposition links are also annotated using the defined link notations and composition annotations. This produces a precise specification of the processes involved in the system. The modeler is required to define a mapping from the components of his annotated SR diagram into the components of a *ConGolog* model which respects some mapping rules.

- *A set of mapping rules is defined to help ensure consistency between the i* and ConGolog models.*

The *mapping rules* constrain the modeler to map elements of the annotated SR diagram into appropriate entities in the *ConGolog* model and ensure that the models are consistent. This allows us to trace corresponding elements in the two models when changes are made.

Every element in the annotated SR diagram must be mapped into an appropriate element of the *ConGolog* model. If some part of the annotated SR diagram needs to be changed, then the corresponding part of the *ConGolog* model can be updated too, and vice versa. The mapping rules provide a kind of formal semantic for annotated SR diagrams by reducing them to *ConGolog*, which already has a formal semantics.

- *Two case studies are performed to show how the methodology is applied.*

The meeting scheduling application involves a computerized scheduling system in an organization. We apply the methodology to this case study and show how the initiator, the participants, and the meeting scheduler's interests are addressed, how alternatives can be chosen, and how the process is specified and validated by *ConGolog* simulation. The mail-order business case study is more of a business process modeling exercise. We compare alternatives and discuss how different choices affect the actors. We show how alternative processes for handling an order can be simulated in the *ConGolog* model. We also discuss how one can model agents that play different roles to fulfill their responsibilities; we show how the dependencies between these roles can be operationalized.

8.2 Comparison to Related Work

As mentioned earlier, there are various agent-oriented or goal-oriented requirements engineering frameworks in existence that are related to ours. One is *ALBERT-II* [DuBois95], a formal framework designed for specifying distributed real-time systems. *ALBERT-II* is based on temporal logic. Agents' states and behavior are specified through constraints expressed in a logic-based notation. Which aspects of agents' states or actions are known/visible to other agents is also specified formally through "cooperation constraints". Typical patterns of constraints are identified to support the analyst in requirements elaboration. In [YDDM97] and [Bissener97], the combined use of *i** and

ALBERT-II for requirements engineering is investigated. The approach proposed in these papers is very different from ours. In Bissener's approach [Bissener97], there are no detailed steps to be followed in developing the *ALBER*T-II model from the *i** SR model. There is no attempt to develop an intermediate notation to enable a direct mapping from *i** to the *ALBERT-II* formal framework. No process specification annotations are introduced to elaborate the original *i** SR diagram into one that specifies the process in detail. There is no discussion of operationalizing dependencies in order to specify how agents interact each other to have these dependencies supplied. There are no explicit constraints for mapping the elements of the *i** SR diagram into corresponding elements in *ALBERT-II*. The process models do not specifically show how the whole system proceeds step by step. Also there is no executable model can be simulated.

On the other hand, ALBERT–II does specify the differences between what agents know through cooperation constraints. Action perception and state perception constraints specify what agents know about others. Action information and state information constraints specify what agents show to others. Local constraints, such as operational constraints and declarative constraints, specify how agents perform their actions and what their effects are, and how agents perform a complex process by decomposing it into atomic actions. But *ALBERT-II* does not have a rich procedural process specification language. It also does not have a support tool for simulation. *ConGolog* provides these, but does not support modeling what different agents know (unless one uses the extended version of *ConGolog* defined in [SL01]).

Our work is also related to the *KAOS* framework [DVF93], which focuses on the formal modeling of functional and non-functional requirements. *KAOS* has a formal specification language based on temporal logic. There is also an elaboration method to help the modeler refine the system goals into more operational components that can be

assigned to agents. As for *ALBERT-II*, there is no rich procedural process specification language and simulation tool for validating the model. The framework addresses issues in requirements acquisition, i.e., goal-directed, scenario-directed, and viewpoint-directed strategies, and the reuse of requirements specifications. We are not aware of attempts to use *KAOS* in combination with *i** or another early-phase RE framework.

8.3 Future Work

Future work is necessary to fully realize the benefits of our approach. This involves work in the following areas:

- *Support tool*

The work in this thesis should be followed by efforts to develop a computerized tool to help the modeler complete the steps of the methodology. Once a user-friendly tool has been developed for applying this methodology in requirements analysis, broader use of the methodology could be achieved.

Some existing tools could be used for this. A support tool called OME was built for developing *i** models; it is discussed in [OME00]. The tool supports graphical editing to help the modeler build SD and SR models. There is also some support for analysis. There is also a graphical viewer for displaying the simulations in *ConGolog* [LKMY99]. The tool shows the trace of actions performed during the process and the change in the world state when an action is performed.

We suggest that a computerized graphical tool be developed based on these components. The computerized graphical tool would instruct the modeler to apply the methodology step by step with on-line help. First, the tool would ask the modeler to identify the roles, positions, and agents in the system, and the dependency relationships between them. From this, the SD model would be built. After this, the tool would ask the modeler to fill

out every role, position, and agent to specify their goals, tasks, and softgoals, the decompositions of the goals/tasks, and the contribution links to the softgoals. From this, the SR model would be built. Different alternatives would be included in the SR model and the tool would help the analyst and clients clarify what their real needs for the desired system are. Then, based on the initial SR model, the computerized tool would allow the modeler to suppress softgoals and related links, as well as tasks/goals and dependencies that do not need to be modeled in *ConGolog*. In this step, a second version of SR model would be built. Then, the tool would ask the modeler to operationalize every dependency, producing a third version of the SR model. After this, the tool would ask the modeler to go over every role, position, and agent node to clarify its top task. Then, the tool would ask the modeler to go over every task/goal in every actor (role/position/agent) from the top level to decompose it, and put composition annotations on groups of decomposition links and link annotations on single decomposition link if appropriate. The result of this step is the annotated SR diagram. Then, the tool would have the modeler to go over every entity in the annotated SR diagram and map it into the corresponding entities of a *ConGolog* model according to the mapping rules. Much of the process code generation would be done automatically. The *ConGolog* domain specification could be written in the high-level *Golog* Domain Language (GDL) [LRLLS97] and automatically translated into Prolog code. Then, the tool would help the modeler specify the initial state for the system and simulate the process. The above steps would be repeated when some modification has to be done in the *i** or *ConGolog* model.

- *Extending the methodology to the design phase.*

If the analyst decides that the requirements specification obtained by applying the methodology is satisfactory, then he would move to the next phase of obtaining a design specification. The annotated SR diagram could be used as a starting point to develop an architectural design and detailed design using *UML*, *BON* or some agent-oriented design notation. UML might be appropriate based on the work of Odell *et al.* on agent-UML [OPB2000]. Suppressed information would be considered in the design to select among

alternatives. The design would also refer to the *ConGolog* model and simulation traces to make sure that the system design captures the details of the process. The implementation of the system would be based on this design specification. Some aspects of this task could be automated.

- *Verification of ConGolog process specifications*

Some work has already been done on verification methods and tools for the *ConGolog* framework [LS99]. It would be good to integrate them in our methodology.

- *Refining the Mapping Rules*

As we discussed in section 7.3, issues remain with respect to the mapping rules, such as whether every element in the *i** SR diagram should be mapped into an element of the *ConGolog* model, whether softgoals should be relativized into hard goals which can be satisfied by performing some tasks/goals, and whether all alternatives should be compared.

- *Extending the methodology to formally model agents mental states*

We would like to refine the methodology to better model agent's metal states — what agents know and want. For this, we will use an extended version of the *ConGolog* framework [SL01] [SSL98] [LS99] [LLR99] that explicitly represents agents' knowledge and goals (using modal operators) and their dynamics, i.e., how they are affected by communication actions (e.g., inform, request, cancel-request, etc.) and perception actions.

- *Testing the methodology in more realistic case studies/projects.*

Bibliography

[Ben85] J. Benett. A Knowledge-Based System for Acquiring the Conceptual Structure of a Diagnostic Expert System, *Journal of Automated Reasoning*, Vol.1, 1985 pp. 49-74.

[Bissener97] M. Bissener. *A Proposal For A Requirements Engineering Method Dealing with Organizational, Non-Functional and Functional Requirements*, Ph.D. Thesis, Computer Science Department, University of Namur, 1997.

[Boe81] B.W. Boehm. *Software Engineering Economics*, Englewood Cliffs, NJ: Prentice-Hall, 1981.

[BMR92] A. Borgida, J. Mylopoulos, and R. Reiter. On the Frame Problem in Procedure Specifications, *IEEE Transactions on Software Engineering*, 21(10), Oct., pp.785-798 1995.

[BD95] F. Brazier, B. Dunin-Keplicz, N. R. Jennings, and J. Treur. Formal Specification of Multi-Agent Systems: A Real-World Case. In *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS-95)*, pages 25-32, San Francisco, CA, June 1995, Springer-Verlag.

[Bubenko80] J.A. Bubenko. Information Modeling in the Context of System Development, *Proc. IFIP*, 1980, pp. 395-411.

[Bubenko95] J.A. Bubenko. Challenges in Requirements Engineering, *Proc. 2nd IEEE Int. Symposium on Requirements Engineering*, York, England, March 1995, pp. 160-165.

[Burmeister96] B. Burmeister. Models and Methodologies for Agent-Oriented Analysis and Design, in Klaus Fischer, editor, *Working Notes of the KI'96 workshop on agent-oriented programming and distributed systems*, 1996, *KFKI Document D-96-06*.

[CK00] J. Castro, M. Kolp and J. Mylopoulos. A Requirements-Driven Development Methodology, to appear in *Proc of the 13th International Conference on Advanced Information Systems Engineering CAiSE 01*, Interlaken, Switzerland, June 4-8, 2001.

[CRFS98] F. Chabot, J.-F. Raskin, L. Férier, and P.-Y. Schobbens. The Formal Semantics of *Albert-II, Technical report* (draft), 1998.

[DVF93] A. Dardenne, A.Van Lamsweerde, and S. Fickas. Goal-Directed Requirements Acquisitions, *Science of Computer Programming*, 20, pp.3 -50, 1993

[DDMV98] R. Darimont, E. Delor, P. Massonet, and A. van Lamsweerde. GRAIL/*KAOS*: An Environment for Goal-Driven Requirements Engineering*, IEEE, Proceedings of the 20th International Conference on Software Engineering*, Kyoto, April 1998, Vol. 2, pp. 58-62.

[DV96] R. Darimont and A. van Lamsweerde. Formal Refinement Patterns for Goal-Driven Requirements Elaboration, *Proc. FSE'4 - Fourth ACM SIGSOFT Symp. on the Foundations of Software Engineering*, San Francisco, October 1996, pp.179-190.

[Davis90] A.M. Davis. *Software Requirements: Analysis and specification*, Prentice-hall, Englewood Cliffs, New Jersey, 1990.

[Davis95] A. M. Davis. *201 Principles of Software Development*, McGraw-Hill Inc., 1995.

[DLL97] G. De Giacomo, Y. Lespérance, and H. J. Levesque. Reasoning About Concurrent Execution, Prioritized Interrupts and Exogenous Actions in the Situation

Calculus, in *Proceedings of the Fifteenth International Joining Conference on Artificial Intelligence*, pp. 1221-1226, Nagoya, Japan, August 1997.

[DLL00] G. De Giacomo, Y. Lespérance, and H.J. Levesque. *ConGolog*, a Concurrent Programming Language Based on the Situation Calculus, *Artificial Intelligence*, 121, pp. 109 –169, 2000.

[Deloach99] S. A. DeLoach. Multiagent Systems Engineering: a Methodology and Language for Designing Agent Systems. *Proceedings of Agent Oriented Information Systems '99 (AOIS'99)*, pp. 45-57. Seattle WA, May 1999.

[DW00] S. A. DeLoach and M. Wooldridge. Developing Multiagent Systems with Agent Tool, in *Proceedings of The Seventh International Workshop on Agent Theories, Architectures, and Language*s, Boston, Massachusetts, July 2000.

[Dubois98] E. Dubois. *ALBERT*: a Formal Language and Its Supporting Tools for Requirements Engineering in *Proceedings Fundamental Aspects of Software Engineering (ETAPS/FASE'98) LNCS* 1382, Springer-Verlag, Lisboa, 1998.

[DUDU94] E. Dubois, Ph. Du Bois, F. Dubru, and M. Petit. Agent-Oriented Requirements Engineering - a Case Study Using the *ALBERT* Language. In *Proceedings of the 4th International Working Conference on Dynamic Modeling and Information Systems*, 1994.

[DUDZ95] E. Dubois, Ph. Du Bois, and J.-M. Zeippen. *A Formal Requirements Engineering Method for Real-Time, Concurrent, and Distributed Systems*, Computer Science Department, University of Namur, January 1995.

[DUHLPR86] E. Dubois, J. Hagelstein, E. Lahou, F. Ponsaert, and A. Rifaut. A Knowledge Representation Language for Requirements Engineering, *Proc. IEEE,* 74(10), Oct. 1986, pp. 1431-1444.

[DUP94] E. Dubois and M. Petit. The Formal Requirements Engineering of Manufacturing Systems In S.M. Deen (ed.), *Proc. of the Second International Working Conference on Cooperative Knowledge Based Systems - CKBS'94*, Keele (UK), June 14-17, 1994, pp. 67-82

[DUYP98] E. Dubois, E. Yu, and M. Petit. From Early to Late Formal Requirements: a Process-Control Case Study. *In Proc. of IWSSD9*, Isobe, Japan, April 1998.

[DuBois95] Ph. Du Bois. *The ALBERT-II Language — On the Design and the Use of a Formal Specification Language for Requirements Analysis,* Ph.D. thesis, Dept. of Computer Science, University of Namur, Namur, Belgium, 1995.

[DuBois97] Ph. DuBois. *The ALBERT II Reference Manual* - Version 2.0, Computer Science Department, University of Namur, March 1997.

[GH93] J.V. Guttag and J.J. Horning. *LARCH: Languages and Tools for Formal Specification,* Springer-Verlag, 1993.

[HHJ98] P. Haumer, P. Heymans, M. Jarke, and K. Pohl. Bridging the Gap Between Past and Future in RE: A Scenario-based Approach, to appear *in Proc. of the Fourth IEEE International Symposium on Requirements Engineering (RE'99)*, Limerick, Ireland, 1999

[HD98] P. Heymans and E. Dubois. Scenario-based Techniques Supporting the Elaboration and Validation of Formal Requirements, to appear in the *Requirements Engineering Journal*, September 1998, Springer-verlag.

[IG98] C.A.Iglesias, M. Garijo, and J. Gonzalez. A Survey of Agent-Oriented Methodologies, In: Müller, J.P., Singh, M.P., Rao, A.S., (Eds.): *Intelligent Agents V. Agents Theories, Architectures, and Languages. Lecture Notes in Computer Science*, Vol. 1555. Springer-Verlag, Berlin Heidelberg (1998) 4.

[IEEE83] Institute of Electrical and Electronics Engineers*, IEEE Standard Glossary of Software Engineering Terminology. ANSI/IEEE Standard 729-1983*, New York, 1983.

[JP84] M. Jarke and K. Pohl. Requirements Engineering in 2001: (virtually) Managing a Changing Reality, *Joint Special Issue on Software Engineering beyond 2001*, *IEE Software Engineering Journal 9*, 5 (1994).

[JSW98] N. Jennings, K. Sycara, and M. Wooldridge. A Roadmap of Agent Research and Development, *Journal of Autonomous Agents and Multi-Agent Systems*, 1:275--306, 1998.

[JW98] N.R. Jennings and M. Wooldrige (Eds.). *Agent Technology: Foundations, Applications, and Markers*, Springer-Verlag, Berlin, 1998.

[JW00] N. R. Jennings and M Wooldridge. *Agent-Oriented Software Engineering in Handbook of Agent Technology* (ed. J. Bradshaw), AAAI/MIT Press (to appear), 2000.

[KG97] D. Kinny and M. Georgeff. Modeling and Design of Multi-agent Systems, In J. P. Muller, M. Wooldridge, and N. R. Jennings, editors, *Intelligent Agents III* (LNAI Volume 1193), pp.1-20. Springer-Verlag: Berlin, Germany, 1997.

[KGR96] D. Kinny, M. Georgeff, and A. Rao. A methodology and Modeling technique for systems of BDI agents, in W. Van De Velde and J. W. Perram, editors, *agents breaking away: proceedings of the seventh European Workshop on modeling autonomous agents in a multi-agent world*, (LNAI Volume 1038), pp. 56-71. Springer-Verlag: Berlin, Germany, 1996.

[Leach2000] R. J. Leach. *Introduction to Software Engineering*, CRC Press, Boca Raton, Florida, 2000.

[LKMY99] Y. Lespérance, T.G. Kelley, J. Mylopoulos, and E. Yu. Modeling Dynamic Domains with *ConGolog*, in *Advanced Information Systems Engineering, 11th International Conference, CAiSE-99, Proceedings*, pp. 365-380, Heidelberg, Germany, June 1999, LNCS vol. 1626, Springer-Verlag, Berlin.

[LLR99] Y. Lespérance, H.J. Levesque, and R. Reiter. A Situation Calculus Approach to Modelling and Programming Agents, *Foundations of Rational Agency*, pp. 275-299. M. Wooldridge and A.Rao (eds.), Kluwer Academic Publishers, Printed in the Netherlands, 1999.

[LLRU97] Y. Lespérance, H.J. Levesque, and S. Ruman. An Experiment in Using *Golog* to Build a Personal Banking Assistant, in *Intelligent Agent systems: Theoretical and Practical Issues*, Cavedon, L.,Rao, A., and Wobcke, W.,(Eds.), LNAI volume 1209, 27-43, Springer-Verlag, 1997.

[LLLS00] Y. Lespérance, H.J. Levesque, F. Lin, and R.B. Scherl. Ability and Knowing How in the Situation Calculus, *Studia Logica*, 66(1), 165-186, October 2000.

[LS99] Y. Lespérance and S. Shapiro. On Agent-Oriented Requirements Engineering, Position paper for the *Agent-Oriented Information Systems Workshop (AOIS'99)*, Heidelberg, Germany, June 1999, available at http://www.cs.yorku.ca/~lesperan/

[LTJ98] Y.Lespérance, K.Tam, and M.Jenkin. Reactivity in a Logic-Based Robot Programming Framework, *Cognitive Robotics*, papers for the 1998 *AAAI Fall Symposium*, *Technical Report FS-98-02*, AAAI Press, pp. 98-105, Orlando, FL, USA, October 1998.

[LRLLS97] H. J. Levesque, R. Reiter, Y. Lespérance, F. Lin, and R. B. Scherl. *GOLOG*: A Logic Programming Language for Dynamic Domains, *Journal of Logic Programming*, 31(59-84), 1997.

[ML97] Ph. Massonet and A. van Lamsweerde. Analogical Reuse of Requirements Frameworks, *Proceedings RE'97 - Third International Conference on Requirements Engineering*, Washington DC, IEEE, January 1997, pp. 26-37.

[Meyer91] B. Meyer. Design by contract, in D. Mandrioli and B. Meyer, editors, *Advances in Object-Oriented Software Engineering*, pages 1—50, Prentice Hall, Englewood Cliffs, N.J., 1991.

[MH79] J. McCarthy and P. Hayes. Some Philosophical Problems Form the Stand Point of Artificial Intelligence, in B. Meltzer and D. Michie, editors, *Machine intelligence*, Volume 4, pp. 463-502. Edinburgh University Press, Edinburgh, UK, 1979.

[MMT91] M. Merritt, F. Modugno, and M. Tuttle. Time Constrained Automata, in *Concur'91: 2nd Intl Conf. on Concurrency Theor*y. LNCS 527, Springer-Verlag, 1991.

[MYBJK91] J. Mylopoulos, A. Borgida, M. Jarke, and M. Koubarakis. Telos: Representing Knowledge about Information Systems, *ACM Trans. Info. Sys*., 8 (4), 1991.

[MYCY99] J. Mylopoulos, L. Chung, and E. Yu. From Object-Oriented to Goal-Oriented Requirements Analysis, *Communications of the ACM*, pp. 31-37, Jan. 1999.

[OPB2000] J. Odell, H.V.D. Parunak, and B. Bauer. Extending UML for Agents, in *Proceedings of the Agent-Oriented Information Systems, Workshop at the 17th National conference on Artificial Intelligence*, 2000.

[OME00] OME Online Document, http://www.cs.toronto.edu/km/ome/documentation.html, University of Toronto, April 2001.

[Reiter91] R. Reiter. The frame problem in the situation calculus: a simple solution (sometimes) and a completeness result for goal regression, in Vladimire Lifschitz, ecitro*,* *Artificial Intelligence and Mathematical Theory of Computations*, Papers in Honor of John McCarthy, pp. 359-380, Academic Press, San Diego, CA, 1991.

[REQ97] *Requirements Targeting Software and Systems Engineering: International Workshop RTSE '97*, Bernried, Germany, October 12 -14, 1997.

[ROY70] W. Royce. Managing the Development of Large Software Systems, In *IEEE WESCON*, pp. 1-9, August 1970. Reprinted in *Ninth IEEE International Conference on*

Software Engineering, Washington D.C.: Computer Society Press of the Institute of Electrical and Electronics Engineers, 1987, pp. 328-38.

[RUJB99] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*, Addison-Wesley, 1999.

[SL01] S. Shapiro and Y. Lespérance. Modeling Multiagent Systems with the Cognitive Agents Specification Language - A Feature Interaction Resolution Application, To appear in Castelfranchi, C. and Lespérance, Y., editors, *Intelligent Agents Volume VII - Proceedings of the 2000 Workshop on Agent Theories, Architectures, and Languages (ATAL-2000)*, LNAI, Springer-Verlag, Berlin, 2001.

[SLL97] S. Shapiro, Y. Lespérance, and H.J. Levesque. Specifying Communicative Multi-Agent Systems with *ConGolog*, in *Working Notes of the AAAI Fall 1997 Symposium on Communicative Action in Humans and Machines*, pp. 75-82, Cambridge, MA, AAAI Press, November 1997.

[SLL98] S. Shapiro, Y. Lespérance, and H.J. Levesque. Specifying Communicative Multi-Agent Systems with *ConGolog*, in *Agents and Multi-Agent Systems – Formalisms, Methodologies, and Applications*, W. Wobcke, M. Pagenucco, and C. Zhang, eds., 1-14, LNAI, Springer-Verlag, Berlin, 1998.

[Spivey92] J. Spivey. *The Z Notation: A Reference Manual*, Prentice Hall, 1992.

[Tam98] K. Tam. *Experiments in High-Level Robot Control Using ConGolog - Reactivity, Failure Handling, and Knowledge-Based Search,* M.Sc. Thesis, Dept. of Computer Science, York University, 1998.

[UML98] *Unified Modeling Language Specification*, Object-Management Group, 1998.

[VanL91] A. van Lamsweerde. Learning Machine Learning, in: *Introducing a Logic Based Approach to Artificial Intelligenc*e, A. Thayse (Ed.), Vol. 3, Wiley, 1991, 263-356.

[VanL98] A. van Lamsweerde. Divergent Views in Goal-Driven Requirements Engineering, *Proceedings of the ACM SIGSOFT Workshop on Viewpoints in Software Development*, San Francisco, pp. 252-256, Oct. 1996.

[VanL00] A. van Lamsweerde. Requirements Engineering in the Year 00: A Research Perspective, *Proc. 22nd International Conference on Software Engineering*, Limerick, ACM Press, June 2000.

[VLDD91] A. van Lamsweerde, A. Dardenne, and F. Dubisy. *KAOS* Knowledge Representations as Initial Support for Formal Specification Processes, *Report RR-91-8*, Unite d'Informatique, University of Louvain, 1991.

[VLDDD91] A. van Lamsweerde, A. Dardenne, B. Delcourt, and F. Dubisy. The *KAOS* Project: Knowledge Acquisition in Automated Specification of Software, *Proceedings AAAI Spring Symposium Series, Track: "Design of Composite Systems",* Stanford University, 59-62, March 1991.

[VLDM95] A. van Lamsweerde, R. Darimont, and P. Massonet. Goal-directed elaboration of requirements for a meeting scheduler: Problems and lessons learnt. In *Second IEEE International Symposium on Requirements Engineering, IEEE CS Press*, March 1995.

[VLW98] A. van Lamsweerde and L. Willemet. Inferring Declarative Requirements Specifications from Operational Scenarios*, IEEE Transactions on Software Engineering, Special Issue on Scenario Management,* December 1998.

[WN95] K. Walden and J.-M. Nerson. *Seamless Object-Oriented Software Architecture*, Prentice-Hall, 1995.

[Web84] *Webster's Ninth new collegiate dictionary*, Springfield, Mass.: G. and C. Merriam, 1984.

[Wooldridge92] M. Wooldridge. *The Logical Modeling of Computational Multi-Agent Systems*, Ph.D. thesis, Department of Computation, UMIST, Manchester, UK. (Also available as Technical Report MMU-- DOC--94--01, Department of Computing, Manchester Metropolitan University, Chester St., Manchester, UK), 1992.

[Wooldridge97] M. Wooldridge. *Agent-based software engineering*, IEEE Proc. Software Engineering, 144: pp. 25-37, 1997.

[Wooldridge98] M. Wooldridge. Agents and software engineering, In *AI*IA Notizie XI(3)*, pages 31-37, September 1998.

[WD00] M. Wooldridge and S. A. DeLoach. An Overview of the Multiagent Systems Engineering Methodology, *The First International Workshop on Agent-Oriented Software Engineering (AOSE-2000)*, Limerick, Ireland, June 10, 2000.

[WJ95] M. Wooldridge, and N. Jennings. Intelligent Agents: Theory and Practice, *Knowledge Engineering Review*, 10(2): 115-152, 1995

[WJ96] M. Wooldridge and N.R. Jennings. Agent Theories, Architectures and Languages: A Survey, in Wooldridge and Jennings (ed*.), Intelligent Agent*, *Springer-Verlag*, 1-22. 1996.

[WJK99] M. Wooldridge, N. R. Jennings, and D. Kinny. A Methodology for Agent-Oriented Analysis and Design, In *Proc. 3rd Int. Conf. on Autonomous Agents (Agents '99)*, pages 69--76. Seattle, WA, 1999.

[WJK00] M Wooldridge, N. Jennings, and D. Kinny. The Gaia Methodology for Agent-Oriented Analysis and Design*, Journal of Autonomous Agents and Multi-Agent Systems*, 3 (3), 2000.

[Yu93] E. Yu. Modeling Organizations for Information Systems Requirements Engineering*, Proc. 1st IEEE International Symposium on Requirements Engineering*, San Diego, California, USA. pp. 34-41, January 1993.

[Yu95A] E. Yu. Models for Supporting the Redesign of Organizational Work*, Proceedings, Con. on Organizational Computing Systems (COOCS'95)*, Milpitas, California, USA, pp. 225-136, August 13-16, 1995.

[Yu95B] E. Yu. *Modelling Strategic Relationships for Process Reengineering*, Ph.D. Thesis, Department of Computer Science, University of Toronto, 1995.

[Yu97] E. Yu. Towards Modelling and Reasoning Support for Early-Phase Requirements Engineering, *Proceedings of the 3rd IEEE Int. Symp. on Requirements Engineering (RE'97)*, Washington D.C., USA. pp. 226-235, Jan. 6-8, 1997.

[YDDM95] E. Yu, Ph. Du Bois, E. Dubois, and J. Mylopoulos. From Organizational Models to System Requirements -- A "Cooperating Agents" Approach, *Proc. 3rd International Conference on Cooperative Information Systems--CoopIS-95*, Vienna, Austria, pp. 194-204, May 9-12, 1995.

[YDDM97] E. Yu, Ph. Du Bois, E. Dubois, and J. Mylopoulos. From Organization Models to System Requirements – A "Cooperating Agents" Approach, in *Cooperative Information System: Trends and Direction*s, M.P. Papazoglou and G. Schlageter, eds., 293 –312, Academic Press, 1997.

[Yu2000] E. Yu and L. Liu, Modeling Trust in the *i** Strategic Actors Framework, *Proceedings of the 3rd Workshop on Deception, Fraud and Trust in Agent Societies*, Barcelona, Catalonia, Spain (at Agents2000), June 3-4, 2000.

[YM93] E. Yu and J. Mylopoulos. An Actor Dependency Model of Organizational Work --With Application to Business Process Reengineering, *Proc. Conference on Organizational Computing Systems*, Milpitas, Calif., USA, Simon Kaplan, ed., ACM Press, pp. 258-268, Nov. 1-4, 1993.

[YM94] E. Yu and J. Mylopoulos. Using Goals, Rules, and Methods To Support Reasoning in Business Process Reengineering, *Proceeding of the 27th Annual Hawaii International Conference on Systems Sciences*, Hawaii, Vol. 4, pp. 234-243, Jan. 1994.

[YM94A] E. Yu and J. Mylopoulos. Understanding "Why" in Software Process Modeling, Analysis, and Design, *Proceedings of 16th International Conference on Software Engineering*, Sorrento, Italy, pp. 159-168, May 16-24, 1994.

[YM94B] E.Yu and J. Mylopoulos. Understanding "Why" in Requirements Engineering-with an Example, *Workshop on System Requirements: Analysis, Management and Exploitation*, Schloß Dagstuhl, Saarland, Germany, October 4-7, 1994.

[YM94C] E. Yu and J. Mylopoulos. Towards Modeling Strategic Actor Relationships for Information Systems Development--With Examples from Business Processing Reengineering*, Proceedings of the 4th Workshop on Information Technologies and systems*, Vancouver, B.C., Canada, pp. 21-28, December 17-18, 1994.

[YM94D] E. Yu and J. Mylopoulos. From E-R to "A-R" –Modelling Strategic Actor Relationships for Business Process Reengineering, in Entity-Relationship Approach (ER'94) –Business Modelling and Re-Engineering, (*Proc. 13th Int. Conference on the Entity-Relationship Approach*, Manchester, U.K., December 1994) Springer-Verlag, LNCS-889, pp. 548-565.

[YM97] E. Yu and J. Mylopoulos. Modeling Organizational Issues for Enterprise Integration, *Proceedings of International Conference on Enterprise Integration of Modeling Technology*, Turin, Italy, October 28-30, 1997.

[YML96] E. Yu, J. Mylopoulos, and Y. Lespérance. AI Models for Business Process Reengineering, *IEEE Expert: Intelligent Systems and Their Applications*, August 1996, pp. 16-23.

[ZDD98] J.-M. Zeippen, E. Dubois, Ph. Du Bois. Supporting the Analyst when Reasoning on Requirements Specifications for Real-Time and Distributed Systems, in the *Proceedings of the First IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC '98)*, Kyoto (Japan), April 20-22, 1998, pp. 215-219.

Appendix A:
Modeling the Meeting Scheduling Process.

A-1 The *ConGolog* Model for `Participant`

```
proc(participant_behavior(Participant,MS),
  tryArrangeMeetingsAndmaintainSchedule(Participant,MS)
).

proc(tryArrangeMeetingsAndMaintainSchedule(Participant,MS),
  tryArrangeMeetings(MS,Participant)
  #=
  ==>([reqID,date,xlist],
        and(val(reqParticipant(reqID),Participant),
        and(val(reqDate(reqID),date),
        and(participantDateoccupied(Participant,date),
        and(val(participantDateInfo(Participant),xlist),
        not(occupyAcknowledged(Participant,date))
      )))),
     occupyDate(Participant,date)
  )
).

proc(tryArrangeMeetings(MS,Participant),
   findAgreeableDateUsingScheduler(MS,Participant)
).

proc(findAgreeableDateUsingScheduler(MS,Participant),
   ==>([reqID,xlist],
      and(obtainReqRcvd(reqID),
          and(not(obtainReqProc(reqID)),
              and(val(reqParticipant(reqID),Participant),
                  val(availableDates(Participant),xlist)
      ))),
       sendAvailDates(Parlticipant,MS,reqID,XliSt)
   )
   #=
  tryAgreeToDate(Participant,MS)
).

proc(tryAgreeToDate(Participant,MS),
   ==>([reqID,date,tlist],
        and(agreementReqRcvd(reqID),
            and(not(agreementReqProc(reqID)),
```

```
                and(val(participantDateInfo(Participant),tlist),
                    and(val(reqDate(reqID),date),
                        val(reqParticipant(reqID),Participant)
                )))),
  replyAgreement(Participant,MS,reqID,date,tlist)
  )
  #=
  ==>((reqID,date,tlist],
       and(cancelReqRcvd(reqID),
           and(val(reqParticipant(reqID),Participant),
               and(not(cancelReqProc(reqID)),
                   and(val(participantDateInfo(Participant),tlist),
                val(reqDate(reqID),date)
       )))),
      cancelAgreementOnDate(Participant,MS,reqID,date)
     )
  ).

  proc(cancelAgreementOnDate(Participant,MS,ReqID,Date),
     [
        rmvDateFromSchedule(Participant,Date),
        acceptCancel(Participant,MS,ReqID,Date)
     ]
  ).

proc(replyAgreement(Participant,MS,ReqID,Date,Datelist),
     [ if( dateIsFree(Date,Datelist),
           rejectAgreement(Participant,MS,ReqID,Date),
           acceptAgreementOnDate(Participant,MS,ReqID,Date)
        )
     ]
).

proc(acceptAgreementOnDate(Participant,MS,ReqID,Date),
   [ addDateToSchedule(Participant,Date),
     acceptAgreement(Participant,MS,ReqID,Date)
   ]
).

proc(occupyDate(Participant,Date),
   [
      addDateToSchedule(Participant,Date),
      acknowledgeoccupy(Participant,Date)
    ]
 ).
```

A-2 Successor State Axioms for Actions

```
holds(allMergedlistSet(SchedulerID),do(A,S)):-
     (A = setAllMergedlist(_,SchedulerID,_));
     holds(allMergedlistSet(SchedulerID),S).

holds(letedSchedulerSked(SchedulerID),do(A,S)):-
     (A = requestSchedulemeeting(_,_,_),
      holds(val(schedulerCtr,SchedulerID),S));
      holds(letedSchedulerSked(SchedulerID),S).

holds(requestedEnterDateRange(SchedulerID),do(A,S)):-
      A = requestEnterDateRange(_,_,SchedulerID);
```

```
      holds(requestedEnterDateRange(SchedulerID),S).

holds(enteredDateRange(SchedulerID,Tlist),do(A,S)):-
      A = enterDateRange(_,_,SchedulerID,Tlist);
      holds(enteredDateRange(SchedulerID,Tlist),S).

holds(dateRangeEntered(SchedulerID),do(A,S)):-
      A = enterDateRange(_,_,SchedulerID,-);
      holds(dateRangeEntered(SchedulerID),S).

holds(waitingForAgreeAns(SchedulerID,Participant,Date),do(A,S)):-
     A = requestAgreement(_,Participant,SchedulerID,Date);
     (holds(waitingForAgreeAns(SchedulerID,Participant,Date),S),
     A\=acceptAgreement(Participant,_,_,Date),
     A\=rejectAgreement(Participant,_,_,Date)).

holds(waitingForCancelAns(SchedulerID,Participant,Date),do(A,S)):-
     A = cancelAgreement(-,Participant,SchedulerID,Date);
     (holds(waitingForCancelAns(SchedulerID,Participant,Date),S),            A\=
     acceptCancel(Participant,_,_,Date)).
holds(acceptedCancel(SchedulerID,Participant,Date),do(A,S)):-
      (A=acceptCancel(Participant,_,_,Date),
      holds(waitingForCancelAns(SchedulerID,Participant,Date),S));
      holds(acceptedCancel(SchedulerID,Participant,Date),S).

holds(submittedAgreement(SchedulerID,Participant,Date),do(A,S)):-
      A = requestAgreement(_,Participant,SchedulerID,Date);
      holds(submittedAgreement(SchedulerID,Participant,Date),S).

holds(agreementAccepted(SchedulerID,Participant,Date),do(A,S)):-
      (A = acceptAgreement(Participant,MS,_,Date),
      holds(waitingForAgreeAns(SchedulerID,Participant,Date),S));
      holds(agreementACcepted(SchedulerID,Participant,Date),S).

holds(agreementRejected(SchedulerID,Participant,Date),do(A,S)):-
      (A = rejectAgreement(Participant,_,_,Date),
      holds(waitingForAgreeAns(SchedulerID,Participant,Date),S));
      holds(agreementRejected(SchedulerID,Participant,Date),S).

holds(waitingSendAns(SchedulerID,Participant),do(A,S)):-
      A = obtainAvailDates(_,Participant,SchedulerID);
      (holds(waitingSendAns(SchedulerID,Participant),S),
      A \= sendAvailDates(Participant,_,_,_)).

holds(submittedCancel(SchedulerID,Participant,Date),do(A,S)):-
      A = cancelAgreement(_,Participant,SchedulerID,Date);
      holds(submittedCancel(SchedulerID,Participant,Date),S).

holds(submittedObtain(SchedulerID,Participant),do(A,S)):-
      A = obtainAvailDates(_,Participant,SchedulerID);
      holds(submittedobtain(SchedulerID,Participant),S).

holds(agreementReqRcvd(ReqID),do(A,S)):-
      (A = requestAgreement(_,_,_,_),
      holds(val(reqCtr,ReqID),S));
      holds(agreementReqRcvd(ReqID),S).

holds(agreementReqProc(ReqID),do(A,S)):-
      A = acceptAgreement(_,_,ReqID,_);
      A = rejectAgreement(_,_,ReqID,_);
      holds(agreementReqProc(ReqID),S).

holds(cancelReqRcvd(ReqID),do(A,S)):-
     (A = cancelAgreement(_,_,_,_),
      holds(val(reqCtr,ReqID),S));
      holds(cancelReqRcvd(ReqID),S).
```

```
holds(cancelReqProc(ReqID),do(A,S)):-
      A = acceptCancel(_,_,ReqID,_);
      holds(cancelReqProc(ReqID),S).

holds(obtainReqRcvd(ReqID),do(A,S)):-
     (A = obtainAvailDates(_,_,_), holds(val(reqCtr,ReqID),S));
      holds(obtainReqRcvd(ReqID),S).

holds(obtainReqProc(ReqID),do(A,S)):-
      A = sendAvailDates(_,_,ReqID,_);
      holds(obtainReqProc(ReqID),S).

holds(succeSsNotified(SchedulerID,Participant,Date),do(A,S)):-
      A = notifySuccess(_,_,SchedulerID,Participant,Date);
      holds(successNotified(SchedulerID,Participant,Date),S).

holds(failNotified(SchedulerID,Peoplelist,Dlist),do(A,S)):-
      (A = notifyFail(-,-,SchedulerID,Peoplelist),
      holds(and(val(skedTlist(SchedulerID),Dlist),
                val(skedPeoplelist(SchedulerID),Peoplelist)),S));
      holds(failNotified(SchedulerID,Peoplelist,Dlist),S).

holds(agreementNotified(SchedulerID,Participant,Date),do(A,S)):-
      A = notifyAgreement(_,Participant,SchedulerID,Date);
      holds(agreementNotified(SchedulerID,Participant,Date),S).

holds(participantDateOccupied(Participant,Date),do(A,S)):-
      A = occupyDateFromParticipant(Participant,Date);
      holds(participantDateOccupied(Participant,Date),S).

holds(occupyAcknowledged(Participant,Date),do(A,S)):-
     A= acknowledgeoccupy(Participant,Date);
     holds(occupyAcknowledged(Participant,Date),S).

holds(oneSubmittedCancel(SchedulerID,Peoplelist,Date),S):-
     member(P,Peoplelist),holds(submittedCancel(SchedulerID,P,Date),S).

/* function fluent*/
holds(val(skedPeoplelist(ID),Peoplelist),do(A,S)):-
      (A = requestScheduleMeeting(_,_,Peoplelist),
      holds(val(schedulerCtr,ID),S));
      holds(val(skedPeoplelist(ID),Peoplelist),S).

holds(val(skedTlist(ID),Tlist),do(A,S)):-
      A = enterDateRange(_,_,ID,Tlist);
      holds(val(skedTlist(ID),Tlist),S).

holds(val(reqParticipant(ID),Participant),do(A,S)):-
     (A = requestAgreement(_,Participant,_,_), holds(val(reqCtr,ID),S));
     (A = cancelAgreement(-,Participant,-,-), holds(val(reqCtr,ID),S));
     (A = obtainAvailDates(_,Participant,_), holds(val(reqCtr,ID),S));
     (A = occupyDateFromParticipant(Participant,-),holds(val(reqCtr,ID),S));
      holds(val(reqParticipant(ID),Participant),S).

holds(val(reqDate(ID),Date),do(A,S)):-
      (A = requestAgreement(_,_,_,Date), holds(val(reqCtr,ID),S));
      (A = cancelAgreement(_,_,_,Date), holds(val(reqCtr,ID),S));
      (A = occupyDateFromParticipant(_,Date),holds(val(reqCtr,ID),S));
      holds(val(reqDate(ID),Date),S).
```

```
holds(val(reqSchedulerID(ID),SchedulerID),do(A,S)):-
       (A = requestAgreement(-,-,SchedulerID,_), holds(val(reqCtr,ID),S));
       (A = cancelAgreement(-,-,SchedulerID,-), holds(val(reqCtr,ID),S));
       (A = obtainAvailDates(_,_,SchedulerID),holds(val(reqCtr,ID),S));
       holds(val(reqSchedulerID(ID),SchedulerID),S).

holds(val(participantDateInfo(Participant),TliSt),do(A,S)):-
      (A = addDateToSchedule(Participant,Date),
       holds(val(participantDateInfo(Participant),Mlist),S),
       merg(Date,MliSt,Tlist));
      (A = rmvDateFromSchedule(Participant,Date),
       holds(val(participantDateInfo(Participant),Mlist),S),
       delete(Date,Mlist,Tlist));
      (holds(val(participantDateInfo(Participant),Tlist),S),
       A\= rmvDateFromSchedule(Participant,-),
       A\= addDateToSchedule(Participant,_)).

holds(val(allmergedlist(SchedulerID),Dlist),do(A,S)):-
       (A = setAllMergedlist(_,SchedulerID,Dlist));
       (holds(val(allmergedlist(SchedulerID),Dlist),S),
        A\= setAllMergedlist(_,SchedulerID,_)).

holds(val(schedulerCtr,N),do(A,S)):-
       (A = requestScheduleMeeting(_,_,_), holds(val(schedulerCtr,M),S), N is M 1);
       (holds(val(schedulerCtr,N),S), A \= requestScheduleMeeting(_,_,_)).

 holds(val(reqCtr,N),do(A,S)):-
       (A = requestAgreement(_,_,_,_), holds(val(reqCtr,M),S), N is M + 1);
       (A = cancelAgreement(_,_,_,_), holds(val(reqCtr,M),S), N is M + 1);
       (A = obtainAvailDates(_,_,_), holds(val(reqCtr,M),S), N is M + 1);
       (A = occupyDateFromParticipant(_,_),holds(val(reqCtr,M),S),N is M + 1);
       (holds(val(reqCtr,N)IS),
        A\= requestAgreement(_,_,_,_), A \= cancelAgreement(_,_,_,_),
        A\= obtainAvailDates(_,_,_), A \= OCCUPyDateFromParticipant(_,_)

 holds(val(availableDates(Participant),Tlist),S):-
       holds(val(feblist,mlist),S),
       holds(val(participantDateInfo(Participant),NliSt),S),
       deletelist(Nlist,Mlist,Tlist).

 holds(sentAvailDates(SchedulerID,Participant,Tlist),do(A,S)):-
       (A=sendAvailDates(Participant,-,_,Tlist),
       holds(waitingSendAns(SchedulerID,Participant),S));
       holds(sentAvailDates(SchedulerID,Participant,TliSt),S).

 /* Defined Fluents */

 holds(agreementAnswered(SchedulerID,Participant,Date),S):-
       holds(agreementAccepted(SchedulerID,Participant,Date),S);
       holds(agreementRejected(SchedulerID,Participant,Date),S).
```

```
holds(oneNotRequestAnswered(SchedulerID,Peoplelist,Date),S):-
        member(P,Peoplelist),
        \+holds(agreernentAnswered(SchedulerID,P,Date),S).

holds(oneDateRequestAnswered(SchedulerID,Peoplelist,Date),S):-
        holds((member(P,Peoplelist)-->agreementAnswered(SchedulerID,P,Date)),S).

holds(oneObtainSubmitted(SchedulerID,Peoplelist),S):-
        member(Participant,Peoplelist),
        holds(submittedobtain(SchedulerID,Participant),S).

holds(oneRejected(SchedulerID,Peoplelist,Date),S):-
        member(Participant,Peoplelist),
       holds(agreementRejected(SchedulerID,Participant,Date),S).

holds(allRejected(SchedulerID,Peoplelist,Dlist),S):-
      holds(member(Date,Dlist)-->oneRejected(SchedulerID,Peoplelist,Date),S).

holds(oneNotifySuccess(SchedulerID,Peoplelist,Dlist),S):-
       member(Date,Dlist),
      holds(successNotified(SchedulerID,Peoplelist,Date),S).

holds(oneAnsReq(SchedulerID,Peoplelist,Dlist),S):-
      member(Participant,Peoplelist),
      member(Date,Dlist),
      holds(submittedAgreement(SchedulerID,Participant,Date),S).

holds(someNotSendAvailDates(SchedulerID,Peoplelist),S):-
       member(Participant,Peoplelist),
     \+holds(sentAvailDates(SchedulerID,Participant,_),S).

holds(allAccepted(schedulerID,Peoplelist,Date),S):-
      holds(member(P,Peoplelist)-->agreementAccepted(SchedulerID,P,Date),S).

holds(waitForAllParticipantSendAvailDates(SchedulerID,Peoplelist),S):-
      holds(not(someNotSendAvailDates(SchedulerID,Peoplelist)),S).

holds(someDateNotTryAndNoAgreement(SchedulerID,Peoplelist,Xlist),S):-
      holds(some(date,and(member(date,Xlist),
              not(oneDateRequestAnswered(SchedulerID,Peoplelist,date)))),S),
      holds(not(some(date,and(member(date,xlist),
              allAccepted(SchedulerID,Peoplelist,date)))),S).

holds(meetingFail(SchedulerID,Peoplelist,Date),S):-
      member(Participant,Peoplelist),
      \+holds(agreementACcepted(SchedulerID,Participant,Date),S).

holds(meetingBeScheduledIfPossible(SchedulerID),do(A,S)):-
      A = notifyFail(_,_,SchedulerID,-);A = notifySuccess(_,_,SchedulerID,_,_);
      holds(meetingBeScheduledIfPossible(SchedulerID),S).
```

```
holds(waitForSchedulingResultFromScheduler(Init,MS,Peoplelist,Datelist),S):-
holds(some(date,and(member(date,Datelist),succeSsNotified(_,Peoplelist,date))),S);
      holds(failNotified(_,Peoplelist,Datelist),S).

holds(waitForSchedulerRequeStDateRange(Init,MS,Peoplelist,Datelist),S):-
      holds(some(CschedulerIDI,and(val(skedPeoplelist(schedulerID),Peoplelist),
      and(requestedEnterDateRange(schedulerID),
      not(dateRangeEntered(schedulerID))
      ))),S).

holds(meetingBeenScheduledIfPossible(Peoplelist,Datelist),S):-
     holds(some(date,and(member(date,Datelist),successNotified(-
                                              ,Peoplelist,date))),S);
      holds(failNotified(_,Peoplelist,Datelist),S).

holds(agreeableDateForMeeting(SchedulerID,Participant),S):-
      holds(some(date,agreementNotified(SchedulerID,Participant,date)),
      holds(some(datelist,failNotified(SchedulerID,_,datelist)),S).

  holds(waitForAllAnswerRequest(SchedulerID,Peoplelist,Date),S):-
      holds(oneDateRequestAnswered(SchedulerID,Peoplelist,Date),S).

  holds(findAvailDateSlot(SchedulerID),S):-
      holds(allmergedlistSet(SchedulerID),S).

  holds(dateIsFree(Date,Datelist),S):-
      holds(member(Date,Datelist),S).

  holds(val(interSection(Tllist,T21ist),T31ist),do(A,S)):-
      (A=setIntersection(Tllist,T21ist),intersectionlist(Tllist,T21ist,T31ist));
      holds(val(interSection(Tllist,T21ist),T31ist),S).

  holds(added(Participant,Date),do(A,S)):-
       A=addDateToParticipant(Participant,Date);
       holds(added(Participant,Date),S).

  holds(interSectionlist(TlLIST,T2LIST,T3LIST),S):-
      intersectionlist(TlLIST,T2LIST,T3LIST),holds(true=true,S).
```

A-3 Actions and Fluents

(1) Primitive Actions

```
requestScheduleMeeting(Init,MS,People)
requestEnterDateRange(MS,Init,SchedulerID)
enterDateRange(Init,MS,SchedulerID,Tlist)
obtainAvailDates(MS,Participant,SchedulerID)
sendAvailDates(Participant,MS,ReqID,Tlist)
```

```
requestAgreement(MS,Participant,SchedulerID,Date)
acceptAgreement(Participant,MS,ReqID,Date)
rejectAgreement(Participant,MS,ReqID,Date)
cancelAgreement(MS,Participant,SchedulerID,Date)
acceptCancel(Participant,MS,ReqID,Date)
notifyAgreement(MS,Participant,SchedulerID,Date)
notifySuccess(MS,Init,SchedulerID,Peoplelist,Date)
notifyFail(MS,Participant,SchedulerID,Peoplelist)
setAllMergedlist(MS,schedulerID,Dlist)
addDateToSchedule(Participant,Date)
rmvDateFromSchedule(Participant,Date)
```

(2) Exogenous Actions

```
occupyDateFromParticipant(Participant,Date)
```

(3) Predicate Fluents

```
letedSchedulerSked(SchedulerID)
requestedEnterDateRange(SchedulerID,Datelist)
enteredDateRange(SchedulerID,Datelist)
submittedObtain(SchedulerID,Participant,Date)
sentAvailDates(SchedulerID,Participant,AvailDates)
submittedAgreement(SchedulerID,Participant,Date)
agreementAccepted(SchedulerID,Participant,Date)
agreementRejected(SchedulerID,Participant,Date)
waitingForAgreeAns(SchedulerID,Participant,Date)
submittedCancel(SchedulerID,Participant,Date)
cancelAccepted(SchedulerID,Participant,Date)
agreementReqRcvd(ReqID,Participant,Date)
cancelReqRcvd(ReqID,Participant,Date)
obtainReqRcvd(ReqID,Participant,Date)
agreementReqProc(ReqID,Participant,Date)
cancelReqProc(ReqID,Participant,Date)
obtainReqProc(ReqID,Participant,Date)
AgreementNotified(SchedulerID,Participant,Date)
successNotified(SchedulerID,Peoplelist,Date)
failNotified(SchedulerID,Peoplelist,Dlist)
ParticipantDateOccupied(Participant,Date)
```

(4) Functional Fluents

```
allmergedlist(SchedulerID)
participantDateInfo(Participant)
skedTlist(SchedulerID)
skedPeoplelist(SchedulerID)
reqParticipant(ReqID)
reqDate(ReqID)
```

```
reqDlist(ReqID)
reqSchedulerID(ReqID)
schedulerCtr
reqCtr
```

A-4 Obtaining Simulation Traces under Unix

```
tiger 41 % ConGolog meetingscheduling.pl
% compiling file /cs/home/fac1/lesperan/cogrobo/ConGolog98/congolog.pl
Disabled further Prolog informational messages.

WARNING: The GDL compiler has not yet been hooked up to this version
         of the system.

ConGolog Interpreter and GDL compiler
-------------------------------------
Loaded model from file /cs/home/grad2/xiyun/thesis/meetingscheduling.pl
Use the 'viewer.' goal to launch the viewer.

Quintus Prolog Release 3.2 (Sun 4, SunOS 5.5.1)
Copyright (C) 1994, Quintus Corporation.  All rights reserved.
301 East Evelyn Ave, Mountain View, California U.S.A. (415) 254-2800
Licensed to York Univerity, Canada

| ?- run.

$$$ >>>> startInterrupts in do([ ],s0)
$$$ >>>> requestScheduleMeeting(ini1,ms1,[paige,yves]) in do([ startInterrupts ],s0)
$$$ >>>> requestEnterDateRange(ms1,ini1,1) in do([
requestScheduleMeeting(ini1,ms1,[paige,yves]) startInterrupts ],s0)
$$$ >>>> enterDateRange(ini1,ms1,1,[12,14]) in do([ requestEnterDateRange(ms1,ini1,1)
requestScheduleMeeting(ini1,ms1,[paige,yves]) startInterrupts ],s0)
$$$ >>>> obtainAvailDates(ms1,paige,1) in do([ enterDateRange(ini1,ms1,1,[12,14])
requestEnterDateRange(ms1,ini1,1) requestScheduleMeeting(ini1,ms1,[paige,yves]) ...],s0)
$$$ >>>> obtainAvailDates(ms1,yves,1) in do([ obtainAvailDates(ms1,paige,1)
enterDateRange(ini1,ms1,1,[12,14]) requestEnterDateRange(ms1,ini1,1) ...],s0)
$$$ >>>>
sendAvailDates(paige,ms1,1,[1,2,3,4,5,6,7,8,9,10,13,15,16,17,18,19,20,21,22,23,24,25,26,2
7,28,29]) in do([ obtainAvailDates(ms1,yves,1) obtainAvailDates(ms1,paige,1)
enterDateRange(ini1,ms1,1,[12,14]) ...],s0)
$$$ >>>>
sendAvailDates(yves,ms1,2,[1,2,3,4,5,6,7,8,9,11,13,14,15,16,17,18,19,20,21,22,23,24,25,26
,27,28,29]) in do([
```

```
sendAvailDates(paige,ms1,1,[1,2,3,4,5,6,7,8,9,10,13,15,16,17,18,19,20,21,22,23,24,25,26,2
7,28,29]) obtainAvailDates(ms1,yves,1) obtainAvailDates(ms1,paige,1) ...],s0)
$$$ Exog occupyDateFromParticipant(paige,15) in do([
sendAvailDates(yves,ms1,2,[1,2,3,4,5,6,7,8,9,11,13,14,15,16,17,18,19,20,21,22,23,24,25,26
,27,28,29])
sendAvailDates(paige,ms1,1,[1,2,3,4,5,6,7,8,9,10,13,15,16,17,18,19,20,21,22,23,24,25,26,2
7,28,29]) obtainAvailDates(ms1,yves,1) ...],s0)
$$$ >>>> addDateToSchedule(paige,15) in do([ occupyDateFromParticipant(paige,15)
sendAvailDates(yves,ms1,2,[1,2,3,4,5,6,7,8,9,11,13,14,15,16,17,18,19,20,21,22,23,24,25,26
,27,28,29])
sendAvailDates(paige,ms1,1,[1,2,3,4,5,6,7,8,9,10,13,15,16,17,18,19,20,21,22,23,24,25,26,2
7,28,29]) ...],s0)
$$$ >>>> acknowledgeOccupy(paige,15) in do([ addDateToSchedule(paige,15)
occupyDateFromParticipant(paige,15)
sendAvailDates(yves,ms1,2,[1,2,3,4,5,6,7,8,9,11,13,14,15,16,17,18,19,20,21,22,23,24,25,26
,27,28,29]) ...],s0)
$$$ >>>> setAllMergedlist(ms1,1,[]) in do([ acknowledgeOccupy(paige,15)
addDateToSchedule(paige,15) occupyDateFromParticipant(paige,15) ...],s0)
$$$ >>>> notifyFail(ms1,ini1,1,[paige,yves]) in do([ setAllMergedlist(ms1,1,[])
acknowledgeOccupy(paige,15) addDateToSchedule(paige,15) ...],s0)
$$$ >>>> notifyFail(ms1,paige,1,[paige,yves]) in do([ notifyFail(ms1,ini1,1,[paige,yves])
setAllMergedlist(ms1,1,[]) acknowledgeOccupy(paige,15) ...],s0)
$$$ >>>> notifyFail(ms1,yves,1,[paige,yves]) in do([ notifyFail(ms1,paige,1,[paige,yves])
notifyFail(ms1,ini1,1,[paige,yves]) setAllMergedlist(ms1,1,[]) ...],s0)
$$$ >>>> stopInterrupts in do([ notifyFail(ms1,yves,1,[paige,yves])
notifyFail(ms1,paige,1,[paige,yves]) notifyFail(ms1,ini1,1,[paige,yves]) ...],s0)

Final situation:
do(stopInterrupts,do(notifyFail(ms1,yves,1,[paige,yves]),do(notifyFail(ms1,paige,1,[paige
,yves]),do(notifyFail(ms1,ini1,1,[paige,yves]),do(setAllMergedlist(ms1,1,[]),do(acknowled
geOccupy(paige,15),do(addDateToSchedule(paige,15),do(occupyDateFromParticipant(paige,15),
do(sendAvailDates(yves,ms1,2,[1,2,3,4,5,6,7,8,9,11,13,14,15,16,17,18,19,20,21,22,23,24,25
,26,27,28,29]),do(sendAvailDates(paige,ms1,1,[1,2,3,4,5,6,7,8,9,10,13,15,16,17,18,19,20,2
1,22,23,24,25,26,27,28,29]),do(obtainAvailDates(ms1,yves,1),do(obtainAvailDates(ms1,paige
,1),do(enterDateRange(ini1,ms1,1,[12,14]),do(requestEnterDateRange(ms1,ini1,1),do(request
ScheduleMeeting(ini1,ms1,[paige,yves]),do(startInterrupts,s0))))))))))))))))

yes
| ?-
```

A-5 The Simulation Trace for Example 3 in Section 5. 5

Two meetings are to be scheduled: Meeting No.1: one of the dates of Feb. 12, 14, 15, 16, and 17 for `paige` and `yves`. Meeting No. 2: one of the dates of Feb. 12, 14, and 15 for `paige` and `yves`.

The sequence of actions performed are as follows:

```
requestScheduleMeeting(ini1,ms1,[paige,yves])

requestEnterDateRange(ms1,ini1,1)

enterDateRange(ini1,ms1,1,[12,14,15,16,17])

obtainAvailDates(ms1,paige,1),

obtainAvailDates(ms1,yves,1),

sendAvailDates(paige,ms1,1,[1,2,3,4,5,6,7,8,9,10,13,15,16,17,18,19,20,21,22,23,24,25,26,2
7,28,29]),

sendAvailDates(yves,ms1,2,[1,2,3,4,5,6,7,8,9,11,13,14,15,16,17,18,19,20,21,22,23,24,25,26
,27,28,29]),

occupyDateFromParticipant(paige,15),

addDateToSchedule(paige,15),

acknowledgeOccupy(paige,15),

setAllMergedlist(ms1,1,[15,16,17]),

requestAgreement(ms1,paige,1,15),

requestAgreement(ms1,yves,1,15),

rejectAgreement(paige,ms1,4,15),do(

addDateToSchedule(yves,15),

acceptAgreement(yves,ms1,5,15),

cancelAgreement(ms1,yves,1,15),

requestAgreement(ms1,paige,1,16),

requestAgreement(ms1,yves,1,16),

addDateToSchedule(yves,16),

acceptAgreement(yves,ms1,8,16),do),do(
```

```
rmvDateFromSchedule(yves,15),

acceptCancel(yves,ms1,6,15)

addDateToSchedule(paige,16),

acceptAgreement(paige,ms1,7,16),

notifySuccess(ms1,ini1,1,[paige,yves],16),

notifyAgreement(ms1,paige,1,16),

notifyAgreement(ms1,yves,1,16),

requestScheduleMeeting(ini1,ms1,[paige,yves]),

requestEnterDateRange(ms1,ini1,2),

enterDateRange(ini1,ms1,2,[12,14,15]),

obtainAvailDates(ms1,paige,2),

obtainAvailDates(ms1,yves,2),

sendAvailDates(paige,ms1,9,[1,2,3,4,5,6,7,8,9,10,13,17,18,19,20,21,22,23,24,25,26,27,28,2

9]),

sendAvailDates(yves,ms1,10,[1,2,3,4,5,6,7,8,9,11,13,14,15,17,18,19,20,21,22,23,24,25,26,2

7,28,29]),

setAllMergedlist(ms1,2,[]),

notifyFail(ms1,ini1,2,[paige,yves]),

notifyFail(ms1,paige,2,[paige,yves]),

notifyFail(ms1,yves,2,[paige,yves]),
```

A-6 The Whole *ConGolog* Model for the Meeting Scheduling Process

```
/* Declarations for Primitive and Exogenous Actions  */

primAct(occupyDateFromParticipant(_,_)).  /*Exogenous action*/
primAct(requestEnterDateRange(_,_,_)).
primAct(enterDateRange(_,_,_,_)).
primAct(requestScheduleMeeting(_,_,_)).
primAct(obtainAvailDates(_,_,_)).
primAct(sendAvailDates(_,_,_,_)).
primAct(setAllMergedlist(_,_,_)).
primAct(requestAgreement(_,_,_,_)).
primAct(acceptAgreement(_,_,_,_)).
primAct(rejectAgreement(_,_,_,_)).
primAct(cancelAgreement(_,_,_,_)).
primAct(acceptCancel(_,_,_,_)).
primAct(notifyAgreement(_,_,_,_)).
primAct(notifySuccess(_,_,_,_,_)).
primAct(notifyFail(_,_,_,_)).
primAct(acknowledgeOccupy(_,_)).
primAct(addDateToSchedule(_,_)).
primAct(rmvDateFromSchedule(_,_)).

/* Precondition Axioms for Primitive and Exogenous Actions  */

poss(addDateToSchedule(_,_),_).
poss(rmvDateFromSchedule(_,_),_).
poss(acknowledgeOccupy(_,_),_).
poss(occupyDateFromParticipant(Participant,Date),S):-
     holds(not(participantDateOccupied(Participant,Date)),S),
     holds(not(submittedAgreement(_,Participant,Date)),S).
poss(requestScheduleMeeting(_,_,_),_).
poss(requestEnterDateRange(_,_,_),_).
poss(enterDateRange(_,_,_,_),_).
poss(obtainAvailDates(_,_,_),_).
poss(sendAvailDates(_,_,_,_),_).
poss(setAllMergedlist(_,_,_),_).
poss(requestAgreement(_,_,_,_),_).
poss(acceptAgreement(_,_,ReqID,_),S):- holds(agreementReqRcvd(ReqID),S).
poss(rejectAgreement(_,_,ReqID,_),S):- holds(agreementReqRcvd(ReqID),S).
poss(cancelAgreement(_,_,_,_),_).
```

```
poss(acceptCancel(_,_,_,_),_).
poss(notifyAgreement(_,_,_,_),_).
poss(notifySuccess(_,_,_,_,_),_).
poss(notifyFail(_,_,_,_),_).

/* Successor State Axioms for actions*/

holds(allMergedlistSet(SchedulerID),do(A,S)):-
     (A = setAllMergedlist(_,SchedulerID,_));
     holds(allMergedlistSet(SchedulerID),S).

holds(letedSchedulerSked(SchedulerID),do(A,S)):-
     (A = requestScheduleMeeting(_,_,_),
      holds(val(schedulerCtr,SchedulerID),S));
      holds(letedSchedulerSked(SchedulerID),S).

holds(requestedEnterDateRange(SchedulerID),do(A,S)):-
      A = requestEnterDateRange(_,_,SchedulerID);
      holds(requestedEnterDateRange(SchedulerID),S).

holds(enteredDateRange(SchedulerID,Tlist),do(A,S)):-
      A = enterDateRange(_,_,SchedulerID,Tlist);
      holds(enteredDateRange(SchedulerID,Tlist),S).

holds(dateRangeEntered(SchedulerID),do(A,S)):-
      A = enterDateRange(_,_,SchedulerID,_);
      holds(dateRangeEntered(SchedulerID),S).
      holds(waitingForAgreeAns(SchedulerID,Participant,Date),do(A,S)):-
      A = requestAgreement(_,Participant,SchedulerID,Date);
     (holds(waitingForAgreeAns(SchedulerID,Participant,Date),S),
      A\=acceptAgreement(Participant,_,_,Date),
      A\=rejectAgreement(Participant,_,_,Date)).

holds(waitingForCancelAns(SchedulerID,Participant,Date),do(A,S)):-
      A = cancelAgreement(_,Participant,SchedulerID,Date);
     (holds(waitingForCancelAns(SchedulerID,Participant,Date),S),
      A\= acceptCancel(Participant,_,_,Date)).

holds(acceptedCancel(SchedulerID,Participant,Date),do(A,S)):-
     (A=acceptCancel(Participant,_,_,Date),
      holds(waitingForCancelAns(SchedulerID,Participant,Date),S));
      holds(acceptedCancel(SchedulerID,Participant,Date),S).
```

```
holds(submittedAgreement(SchedulerID,Participant,Date),do(A,S)):-
      A = requestAgreement(_,Participant,SchedulerID,Date);
      holds(submittedAgreement(SchedulerID,Participant,Date),S).

holds(agreementAccepted(SchedulerID,Participant,Date),do(A,S)):-
     (A = acceptAgreement(Participant,MS,_,Date),
      holds(waitingForAgreeAns(SchedulerID,Participant,Date),S));
      holds(agreementAccepted(SchedulerID,Participant,Date),S).

holds(agreementRejected(SchedulerID,Participant,Date),do(A,S)):-
     (A = rejectAgreement(Participant,_,_,Date),
      holds(waitingForAgreeAns(SchedulerID,Participant,Date),S));
      holds(agreementRejected(SchedulerID,Participant,Date),S).

holds(waitingSendAns(SchedulerID,Participant),do(A,S)):-
      A = obtainAvailDates(_,Participant,SchedulerID);
     (holds(waitingSendAns(SchedulerID,Participant),S),
      A \= sendAvailDates(Participant,_,_,_)).

holds(submittedCancel(SchedulerID,Participant,Date),do(A,S)):-
      A = cancelAgreement(_,Participant,SchedulerID,Date);
      holds(submittedCancel(SchedulerID,Participant,Date),S).

holds(oneSubmittedCancel(SchedulerID,Peoplelist,Date),S):-
      member(P,Peoplelist),holds(submittedCancel(SchedulerID,P,Date),S).

holds(submittedObtain(SchedulerID,Participant),do(A,S)):-
      A = obtainAvailDates(_,Participant,SchedulerID);
      holds(submittedObtain(SchedulerID,Participant),S).

holds(agreementReqRcvd(ReqID),do(A,S)):-
     (A = requestAgreement(_,_,_,_), holds(val(reqCtr,ReqID),S));
      holds(agreementReqRcvd(ReqID),S).

holds(agreementReqProc(ReqID),do(A,S)):-
      A = acceptAgreement(_,_,ReqID,_);
      A = rejectAgreement(_,_,ReqID,_);
      holds(agreementReqProc(ReqID),S).

holds(cancelReqRcvd(ReqID),do(A,S)):-
     (A = cancelAgreement(_,_,_,_), holds(val(reqCtr,ReqID),S));
      holds(cancelReqRcvd(ReqID),S).
```

```
holds(cancelReqProc(ReqID),do(A,S)):-
      A = acceptCancel(_,_,ReqID,_);
      holds(cancelReqProc(ReqID),S).

holds(obtainReqRcvd(ReqID),do(A,S)):-
     (A = obtainAvailDates(_,_,_), holds(val(reqCtr,ReqID),S));
      holds(obtainReqRcvd(ReqID),S).

holds(obtainReqProc(ReqID),do(A,S)):-
      A = sendAvailDates(_,_,ReqID,_);
      holds(obtainReqProc(ReqID),S).

holds(successNotified(SchedulerID,Participant,Date),do(A,S)):-
      A = notifySuccess(_,_,SchedulerID,Participant,Date);
      holds(successNotified(SchedulerID,Participant,Date),S).

holds(failNotified(SchedulerID,Peoplelist,Dlist),do(A,S)):-
     (A = notifyFail(_,_,SchedulerID,Peoplelist),
      holds(and(val(skedTlist(SchedulerID),Dlist),
      val(skedPeoplelist(SchedulerID),Peoplelist)),S));
      holds(failNotified(SchedulerID,Peoplelist,Dlist),S).

holds(agreementNotified(SchedulerID,Participant,Date),do(A,S)):-
      A = notifyAgreement(_,Participant,SchedulerID,Date);
      holds(agreementNotified(SchedulerID,Participant,Date),S).

holds(participantDateOccupied(Participant,Date),do(A,S)):-
      A = occupyDateFromParticipant(Participant,Date);
      holds(participantDateOccupied(Participant,Date),S).

holds(occupyAcknowledged(Participant,Date),do(A,S)):-
      A= acknowledgeOccupy(Participant,Date);
      holds(occupyAcknowledged(Participant,Date),S).

holds(val(skedPeoplelist(ID),Peoplelist),do(A,S)):-
     (A = requestScheduleMeeting(_,_,Peoplelist),
      holds(val(schedulerCtr,ID),S));
      holds(val(skedPeoplelist(ID),Peoplelist),S).

holds(val(skedTlist(ID),Tlist),do(A,S)):-
      A = enterDateRange(_,_,ID,Tlist);
      holds(val(skedTlist(ID),Tlist),S).
```

```
holds(val(reqParticipant(ID),Participant),do(A,S)):-
     (A = requestAgreement(_,Participant,_,_), holds(val(reqCtr,ID),S));
     (A = cancelAgreement(_,Participant,_,_), holds(val(reqCtr,ID),S));
     (A = obtainAvailDates(_,Participant,_), holds(val(reqCtr,ID),S));
     (A = occupyDateFromParticipant(Participant,_),holds(val(reqCtr,ID),S));
      holds(val(reqParticipant(ID),Participant),S).

holds(val(reqDate(ID),Date),do(A,S)):-
     (A = requestAgreement(_,_,_,Date), holds(val(reqCtr,ID),S));
     (A = cancelAgreement(_,_,_,Date), holds(val(reqCtr,ID),S));
     (A = occupyDateFromParticipant(_,Date),holds(val(reqCtr,ID),S));
      holds(val(reqDate(ID),Date),S).

holds(val(reqSchedulerID(ID),SchedulerID),do(A,S)):-
     (A = requestAgreement(_,_,SchedulerID,_), holds(val(reqCtr,ID),S));
     (A = cancelAgreement(_,_,SchedulerID,_), holds(val(reqCtr,ID),S));
     (A = obtainAvailDates(_,_,SchedulerID),holds(val(reqCtr,ID),S));
      holds(val(reqSchedulerID(ID),SchedulerID),S).

holds(val(participantDateInfo(Participant),Tlist),do(A,S)):-
     (A = addDateToSchedule(Participant,Date),
       holds(val(participantDateInfo(Participant),Mlist),S),
       merg(Date,Mlist,Tlist));
     (A = rmvDateFromSchedule(Participant,Date),
      holds(val(participantDateInfo(Participant),Mlist),S),
      delete(Date,Mlist,Tlist));
     (holds(val(participantDateInfo(Participant),Tlist),S),
      A\= rmvDateFromSchedule(Participant,_),
      A\= addDateToSchedule(Participant,_)).

holds(val(allmergedlist(SchedulerID),Dlist),do(A,S)):-
     (A = setAllMergedlist(_,SchedulerID,Dlist));
     (holds(val(allmergedlist(SchedulerID),Dlist),S),
      A\= setAllMergedlist(_,SchedulerID,_)).

holds(val(schedulerCtr,N),do(A,S)):-
     (A = requestScheduleMeeting(_,_,_),
      holds(val(schedulerCtr,M),S), N is M + 1);
     (holds(val(schedulerCtr,N),S), A \= requestScheduleMeeting(_,_,_)).

holds(val(reqCtr,N),do(A,S)):-
     (A = requestAgreement(_,_,_,_), holds(val(reqCtr,M),S), N is M + 1);
     (A = cancelAgreement(_,_,_,_), holds(val(reqCtr,M),S), N is M + 1);
```

```
      (A = obtainAvailDates(_,_,_), holds(val(reqCtr,M),S), N is M + 1);
      (A = occupyDateFromParticipant(_,_),holds(val(reqCtr,M),S),N is M + 1);
      (holds(val(reqCtr,N),S),
       A\= requestAgreement(_,_,_,_), A \= cancelAgreement(_,_,_,_),
       A\= obtainAvailDates(_,_,_), A \= occupyDateFromParticipant(_,_)).

holds(val(availableDates(Participant),Tlist),S):-
       holds(val(feblist,Mlist),S),
       holds(val(participantDateInfo(Participant),Nlist),S),
       deletelist(Nlist,Mlist,Tlist).

holds(val(feblist,Tlist),do(_,S)):-holds(val(feblist,Tlist),S).

holds(sentAvailDates(SchedulerID,Participant,Tlist),do(A,S)):-
      (A=sendAvailDates(Participant,_,_,Tlist),
       holds(waitingSendAns(SchedulerID,Participant),S));
       holds(sentAvailDates(SchedulerID,Participant,Tlist),S).

holds(agreementAnswered(SchedulerID,Participant,Date),S):-
       holds(agreementAccepted(SchedulerID,Participant,Date),S);
       holds(agreementRejected(SchedulerID,Participant,Date),S).

holds(oneNotRequestAnswered(SchedulerID,Peoplelist,Date),S):-
       member(P,Peoplelist),
      \+holds(agreementAnswered(SchedulerID,P,Date),S).

holds(oneDateRequestAnswered(SchedulerID,Peoplelist,Date),S):-
       holds((member(P,Peoplelist)-->agreementAnswered(SchedulerID,P,Date)),S).

holds(oneObtainSubmitted(SchedulerID,Peoplelist),S):-
       member(Participant,Peoplelist),
      holds(submittedObtain(SchedulerID,Participant),S).

holds(oneRejected(SchedulerID,Peoplelist,Date),S):-
       member(Participant,Peoplelist),
       holds(agreementRejected(SchedulerID,Participant,Date),S).

holds(allRejected(SchedulerID,Peoplelist,Dlist),S):-
      holds(member(Date,Dlist)-->oneRejected(SchedulerID,Peoplelist,Date),S).

holds(oneNotifySuccess(SchedulerID,Peoplelist,Dlist),S):-
       member(Date,Dlist),
       holds(successNotified(SchedulerID,Peoplelist,Date),S).
```

```
holds(oneAnsReq(SchedulerID,Peoplelist,Dlist),S):-
      member(Participant,Peoplelist),
      member(Date,Dlist),
      holds(submittedAgreement(SchedulerID,Participant,Date),S).

holds(someNotSendAvailDates(SchedulerID,Peoplelist),S):-
      member(Participant,Peoplelist),
      \+holds(sentAvailDates(SchedulerID,Participant,_),S).

holds(allAccepted(SchedulerID,Peoplelist,Date),S):-
      holds(member(P,Peoplelist)-->agreementAccepted(SchedulerID,P,Date),S).

holds(waitForAllParticipantSendAvailDates(SchedulerID,Peoplelist),S):-
      holds(not(someNotSendAvailDates(SchedulerID,Peoplelist)),S).

holds(someDateNotTryAndNoAgreement(SchedulerID,Peoplelist,Xlist),S):-
      holds(some(date,and(member(date,Xlist),
      not(oneDateRequestAnswered(SchedulerID,Peoplelist,date)))),S),
      holds(not(some(date,and(member(date,Xlist),
                      allAccepted(SchedulerID,Peoplelist,date)))),S).

holds(meetingFail(SchedulerID,Peoplelist,Date),S):-
      member(Participant,Peoplelist),
      \+holds(agreementAccepted(SchedulerID,Participant,Date),S).

holds(meetingBeScheduledIfPossible(SchedulerID),do(A,S)):-
      A = notifyFail(_,_,SchedulerID,_);
      A = notifySuccess(_,_,SchedulerID,_,_);
      holds(meetingBeScheduledIfPossible(SchedulerID),S).

holds(waitForSchedulingResultFromScheduler(Init,MS,Peoplelist,Datelist),S):-
      holds(some(date,and(member(date,Datelist),
                        successNotified(_,Peoplelist,date))),S);
      holds(failNotified(_,Peoplelist,Datelist),S).

holds(waitForSchedulerRequestDateRange(Init,MS,Peoplelist,Datelist),S):-
      holds(some([schedulerID],and(val(skedPeoplelist(schedulerID),Peoplelist),
            and(requestedEnterDateRange(schedulerID),
               not(dateRangeEntered(schedulerID))))),S).

holds(meetingBeenScheduledIfPossible(Peoplelist,Datelist),S):-
      holds(some(date,and(member(date,Datelist),
```

```
                        successNotified(_,Peoplelist,date))),S);
      holds(failNotified(_,Peoplelist,Datelist),S).

holds(agreeableDateForMeeting(SchedulerID,Participant),S):-
      holds(some(date,agreementNotified(SchedulerID,Participant,date)),S);
      holds(some(datelist,failNotified(SchedulerID,_,datelist)),S).

holds(waitForAllAnswerRequest(SchedulerID,Peoplelist,Date),S):-
      holds(oneDateRequestAnswered(SchedulerID,Peoplelist,Date),S).

holds(findAvailDateSlot(SchedulerID),S):-holds(allMergedlistSet(SchedulerID),S).

holds(dateIsFree(Date,Datelist),S):-holds(member(Date,Datelist),S).

holds(val(interSection(T1list,T2list),T3list),do(A,S)):-
     (A=setIntersection(T1list,T2list),intersectionlist(T1list,T2list,T3list));
      holds(val(interSection(T1list,T2list),T3list),S).

holds(added(Participant,Date),do(A,S)):-
      A=addDateToParticipant(Participant,Date);
      holds(added(Participant,Date),S).

holds(interSectionlist(T1LIST,T2LIST,T3LIST),S):-
      intersectionlist(T1LIST,T2LIST,T3LIST),holds(true=true,S).

/* Initial State Axioms */
holds(val(participantDateInfo(paige),[11,12,14]),s0).
holds(val(participantDateInfo(yves),[10,12]),s0).
holds(val(feblist,[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,2
7,28,29]),s0).
holds(val(schedulerCtr,1),s0).
holds(val(reqCtr,1),s0).
holds(val(allmergedlist(_),[]),s0).

/* Density delaration forExogenous actions */
exoDensity(100).          /* uncomment to get exogenous actions*/
exoAct(occupyDateFromParticipant(paige,15),47,60).

/* Tracing Controls */
/* tracingProg. */
/* tracingTest. */
tracingExec.
```

```
/* prolog funtions */
less(X,Y):-X<Y.
eQual(X,Y):-X=Y.
unEqual(X,Y):-X<Y;X>Y.
merglist([],L2,L2).
merglist(L1,[],L1).
merglist([X|L1],[X|L2],[X|L]):-merglist(L1,L2,L).
merglist([X|L1],[Y|L2],[X|L]):-less(X,Y),merglist(L1,[Y|L2],L).
merglist([X|L1],[Y|L2],[Y|L]):-less(Y,X),merglist([X|L1],L2,L).
merg([],Y,[Y]).
merg(X,[],[X]).
merg(X,[X|L2],[X|L2]).
merg(X,[Y|L2],[X|L]):-less(X,Y),merg(Y,[Y|L2],L).
merg(X,[Y|L2],[Y|L]):-less(Y,X),merg(X,L2,L).
delete([],L2,[L2]).
delete(_,[],[]).
delete(X,[X|L2],L2).
delete(X,[Y|L2],[Y|L]):-unEqual(X,Y),delete(X,L2,L).
deletelist([],L2,L2).
deletelist(_,[],[]).
deletelist([X|L1],L2,L):-delete(X,L2,L3),deletelist(L1,L3,L).
intersection([],_,[]).
intersection(_,[],[]).
intersection(X,[X|_],X).
intersection(X,[Y|L2],L):-unEqual(X,Y),intersection(X,L2,L).
intersectionlist([],_,[]).
intersectionlist(_,[],[]).
intersectionlist([X|L1],[X|L2],[X|L]):-intersectionlist(L1,L2,L).
intersectionlist([X|L1],L2,[X|L]):-
member(X,L2),delete(X,L2,L3),intersectionlist(L1,L3,L).
intersectionlist([X|L1],L2,L):-
not(member(X,L2)),delete(X,L2,L3),intersectionlist(L1,L3,L).

/* Process model for actors */

/* Main process */
proc(main,[
   initiator_behavior(ini1,ms1)#=
   meetingScheduler_behavior(ms1,ini1)#=
   participant_behavior(paige,ms1)!#=
   participant_behavior(yves,ms1)!
   ]
).
```

```
/* The initiator process model */
proc(initiator_behavior(Init,MS),
  [   tryOrganizeMeeting(Init,MS,[paige,yves],[12,14,15,16,17])
  /*  tryOrganizeMeeting(Init,MS,[paige,yves],[12,14,15])*/
  ]
).

proc(tryOrganizeMeeting(Init,MS,Peoplelist,Datelist),
  achieve_meetingBeenScheduledIfPossible(Init,MS,Peoplelist,Datelist)
).

proc(achieve_meetingBeenScheduledIfPossible(Init,MS,Peoplelist,Datelist),
  [  letSchedulerScheduleMeeting(Init,MS,Peoplelist,Datelist),
     meetingBeenScheduledIfPossible(Peoplelist,Datelist)?
  ]
).

proc(letSchedulerScheduleMeeting(Init,MS,Peoplelist,Datelist),
  [ requestScheduleMeeting(Init,MS,Peoplelist),
    waitForSchedulerRequestDateRange(Init,MS,Peoplelist,Datelist)?,
    enterDateRangeToScheduler(Init,MS,Peoplelist,Datelist),
    waitForSchedulingResultFromScheduler(Init,MS,Peoplelist,Datelist)?
  ]
).

proc(enterDateRangeToScheduler(Init,MS,Peoplelist,Datelist),
  pi([schedulerID],[
       and(val(skedPeoplelist(schedulerID),Peoplelist),
           and(requestedEnterDateRange(schedulerID),
               not(dateRangeEntered(schedulerID))
       ))?,
       enterDateRange(Init,MS,schedulerID,Datelist)
  ])
).

/* The process model for meeting scheduler */
proc(meetingScheduler_behavior(MS,Init),
  scheduleMeetings(MS,Init)
).

proc(scheduleMeetings(MS,Init),
  ==>([schedulerID,peoplelist],
```

```
       and(letedSchedulerSked(schedulerID),
          and(val(skedPeoplelist(schedulerID),peoplelist),
              and(not(requestedEnterDateRange(schedulerID)),
                  not(meetingBeScheduledIfPossible(schedulerID))
       ))),
       achieve_MeetingBeScheduledIfPossible(MS,Init,schedulerID,peoplelist)
     )
).
proc(achieve_MeetingBeScheduledIfPossible(MS,Init,SchedulerID,Peoplelist),
  [ tryScheduleMeeting(MS,Init,SchedulerID,Peoplelist),
    meetingBeScheduledIfPossible(SchedulerID)?
  ]
).

proc(tryScheduleMeeting(MS,Init,SchedulerID,Peoplelist),
  [   requestEnterDateRange(MS,Init,SchedulerID),
      some(datelist,enteredDateRange(SchedulerID,datelist))?,
      for(participant,Peoplelist,[],
          obtainAvailDates(MS,participant,SchedulerID),
          true=true
      ),
      waitForAllParticipantSendAvailDates(SchedulerID,Peoplelist)?,
      achieve_findAvailDateSlot(MS,SchedulerID,Peoplelist),
      pi(xlist,[
         val(allmergedlist(SchedulerID),xlist)?,
         tryObtainAgreement(MS,Init,SchedulerID,Peoplelist,xlist)
      ])
  ]
).

proc(tryObtainAgreement(MS,Init,SchedulerID,Peoplelist,Xlist),
  [    while(and(someDateNotTryAndNoAgreement(SchedulerID,Peoplelist,Xlist),
                not(Xlist=[])),
       tryARemainedDates(MS,Init,SchedulerID,Peoplelist,Xlist)
     ),
     if( or(allRejected(SchedulerID,Peoplelist,Xlist),Xlist=[]),
         notifyFailScheduleMeeting(MS,Init,SchedulerID,Peoplelist)
     )
  ]
).

proc(tryARemainedDates(MS,Init,SchedulerID,Peoplelist,Xlist),
  pi([date], [
```

```
        and(member(date,Xlist),
            not(oneDateRequestAnswered(SchedulerID,Peoplelist,date))
        )?,
        tryTheDate(MS,Init,SchedulerID,Peoplelist,date)
  ])
).

proc(tryTheDate(MS,Init,SchedulerID,Peoplelist,Date),
 [   for(participant,Peoplelist,[],
         requestAgreement(MS,participant,SchedulerID,Date),
         true=true
     ),
     waitForAllAnswerRequest(SchedulerID,Peoplelist,Date)?,
     if(oneRejected(SchedulerID,Peoplelist,Date),
          cancelAgreementForTheDate(MS,SchedulerID,Peoplelist,Date)
     ),
     if(allAccepted(SchedulerID,Peoplelist,Date),
          notifySuccessOnDate(MS,Init,SchedulerID,Peoplelist,Date)
     )
  ]
).

proc(achieve_findAvailDateSlot(MS,SchedulerID,Peoplelist),
  [ mergeAllAvailDates(MS,SchedulerID,Peoplelist),
    findAvailDateSlot(SchedulerID)?
   ]
).

proc(mergeAllAvailDates(MS,SchedulerID,Peoplelist),
  pi([datelist],[
       enteredDateRange(SchedulerID,datelist)?,
       mergeAll(MS,SchedulerID,Peoplelist,datelist)
  ])
).

/*procedure to merge all the available dates from participants of a meeting*/
proc(mergeAll(MS,SCHEDULERID,PEOPLELIST,TLIST),
  if(PEOPLELIST=[],
       setAllMergedlist(MS,SCHEDULERID,TLIST),
       [  pi([f,r],[
             PEOPLELIST=[f|r]?,
             pi( [availdate,templist],
                 [sentAvailDates(SCHEDULERID,f,availdate)?,
```

```
                   interSectionlist(availdate,TLIST,templist)?,
                   mergeAll(MS,SCHEDULERID,r,templist)
              ])
           ])
        ]
   )
).

proc(cancelAgreementForTheDate(MS,SchedulerID,Peoplelist,Date),
  for(participant,Peoplelist,[],
      requestCancel(MS,participant,SchedulerID,Date),
      true
  )
).

proc(requestCancel(MS,Participant,SchedulerID,Date),
  if(agreementAccepted(SchedulerID,Participant,Date),
       cancelAgreement(MS,Participant,SchedulerID,Date)
  )
).

proc(notifySuccessOnDate(MS,Init,SchedulerID,Peoplelist,Date),
  [
     notifySuccess(MS,Init,SchedulerID,Peoplelist,Date),
     for(participant,Peoplelist,[],
         notifyAgreement(MS,participant,SchedulerID,Date),
         true=true
     )
  ]
).

proc(notifyFailScheduleMeeting(MS,Init,SchedulerID,Peoplelist),
  [   notifyFail(MS,Init,SchedulerID,Peoplelist),
      for(participant,Peoplelist,[],
         notifyFail(MS,participant,SchedulerID,Peoplelist),true=true
      )
  ]
).

/* Process model for the Participant */
proc(participant_behavior(Participant,MS),
 tryArrangeMeetingsAndMaintainSchedule(Participant,MS)
).
```

```
proc(tryArrangeMeetingsAndMaintainSchedule(Participant,MS),
  tryArrangeMeetings(MS,Participant)
  #=
  ==>([reqID,date,xlist],
       and(val(reqParticipant(reqID),Participant),
         and(val(reqDate(reqID),date),
           and(participantDateOccupied(Participant,date),
             and(val(participantDateInfo(Participant),xlist),
              not(occupyAcknowledged(Participant,date))
        )))),
        occupyDate(Participant,date)
   )
).

proc(tryArrangeMeetings(MS,Participant),
  findAgreeableDateUsingScheduler(MS,Participant)
).

proc(findAgreeableDateUsingScheduler(MS,Participant),
  /*if request for sending available dates*/
  ==>([reqID,xlist],
     and(obtainReqRcvd(reqID),
       and(not(obtainReqProc(reqID)),
           and(val(reqParticipant(reqID),Participant),
               val(availableDates(Participant),xlist)
     ))),
     sendAvailDates(Participant,MS,reqID,xlist)
 )
 #=
  tryAgreeToDate(Participant,MS)
).

proc(tryAgreeToDate(Participant,MS),
  ==>([reqID,date,tlist],
       and(agreementReqRcvd(reqID),
        and(not(agreementReqProc(reqID)),
          and(val(participantDateInfo(Participant),tlist),
            and(val(reqDate(reqID),date),
                val(reqParticipant(reqID),Participant)
       )))),
       replyAgreement(Participant,MS,reqID,date,tlist)
  )
```

```
    /* if request for cancel a meeting on a date*/
    #=
    ==>([reqID,date,tlist],
       and(cancelReqRcvd(reqID),
         and(val(reqParticipant(reqID),Participant),
           and(not(cancelReqProc(reqID)),
             and(val(participantDateInfo(Participant),tlist),
                val(reqDate(reqID),date)
       )))),
       cancelAgreementOnDate(Participant,MS,reqID,date)
     )
).

proc(cancelAgreementOnDate(Participant,MS,ReqID,Date),
  [  rmvDateFromSchedule(Participant,Date),
     acceptCancel(Participant,MS,ReqID,Date)
  ]
).

proc(replyAgreement(Participant,MS,ReqID,Date,Datelist),
  /*if request for agreement a meeting*/
  [  if( dateIsFree(Date,Datelist),
         rejectAgreement(Participant,MS,ReqID,Date),
         acceptAgreementOnDate(Participant,MS,ReqID,Date)
      )
   ]
).

proc(acceptAgreementOnDate(Participant,MS,ReqID,Date),
  [ addDateToSchedule(Participant,Date),
    acceptAgreement(Participant,MS,ReqID,Date)
  ]
).

proc(occupyDate(Participant,Date),
 /*pick up the exogeous action, occupy any date from the participant's date*/
  [
    addDateToSchedule(Participant,Date),
    acknowledgeOccupy(Participant,Date)
  ]
).
```

A-7 The Initial *ConGolog* Model `MeetingScheduler`

```
proc(meetingScheduler_behavior(MS,Init),
  scheduleMeetings(MS,Init)
).

proc(scheduleMeetings(MS,Init),
  ==>([schedulerID,peoplelist],
       requestedScheduleAMeeting(Init,MS,peoplelist,schedulerID),
       achieve_MeetingBeScheduledIfPossible(MS,Init,schedulerID,peoplelist)
     )
).

proc(achieve_MeetingBeScheduledIfPossible(MS,Init,SchedulerID,Peoplelist),
  [
     tryScheduleAMeeting(MS,Init,SchedulerID,Peoplelist),
     meetingBeScheduledIfPossible(SchedulerID)?
  ]
).

proc(tryScheduleMeeting(MS,Init,SchedulerID,Peoplelist),
  [requestEnterDateRange(MS,Init,SchedulerID),
     some(datelist,enteredDateRange(SchedulerID,datelist))?,
    for(participant,Peoplelist,[],
          obtainAvailDates(MS,participant,SchedulerID),
           true=true),
     waitForAllParticipantSendAvailDates(SchedulerID,Peoplelist)?,
     achieve_findAvailDateSlot(MS,SchedulerID,Peoplelist),
     pi(xlist,[
        val(allmergedlist(SchedulerID),xlist)?,
        tryObtainAgreement(MS,Init,SchedulerID,Peoplelist,xlist)
     ])
  ]
).
proc(tryObtainAgreement(MS,Init,SchedulerID,Peoplelist,Xlist),
[
    while(and(someDateNotTryAndNoAgreement(SchedulerID,Peoplelist,Xlist),
               not(Xlist=[])),
       tryARemainedDates(MS,Init,SchedulerID,Peoplelist,Xlist)
    ),
    if( or(allRejected(SchedulerID,Peoplelist,Xlist),Xlist=[]),
          notifyFailScheduleMeeting(MS,Init,SchedulerID,Peoplelist)
     )
  ]
).

proc(tryARemainedDates(MS,Init,SchedulerID,Peoplelist,Xlist),
  pi([date], [
       and(member(date,Xlist),
           not(oneDateRequestAnswered(SchedulerID,Peoplelist,date))
        )?,
        tryTheDate(MS,Init,SchedulerID,Peoplelist,date)
  ])
).

proc(tryTheProposedDate(MS,Init,SchedulerID,Peoplelist,Date),
  [
     for(participant,Peoplelist,[],
          requestAgreement(MS,participant,SchedulerID,Date),
         true=true),
     waitForAllAnswerRequest(SchedulerID,Peoplelist,Date)?,
    if(oneRejected(SchedulerID,Peoplelist,Date),
          cancelAgreementForTheDate(MS,SchedulerID,Peoplelist,Date)
     ),
     if(allAccepted(SchedulerID,Peoplelist,Date),
```

```
          notifySuccessOnDate(MS,Init,SchedulerID,Peoplelist,Date)
    )
  ]
).


proc(achieve_findAvailDateSlot(MS,SchedulerID,Peoplelist),
  [
    mergeAllAvailDates(MS,SchedulerID,Peoplelist),
    findAvailDateSlot(SchedulerID)?
  ]).

proc(mergeAllAvailDates(MS,SchedulerID,Peoplelist),
  pi([datelist],[
        enteredDateRange(SchedulerID,datelist)?,
        mergeAll(MS,SchedulerID,Peoplelist,datelist)
  ])
).


/*procedure to merge all the available dates from participants of a meeting*/


proc(mergeAll(MS,SCHEDULERID,PEOPLELIST,TLIST),
   if(PEOPLELIST=[],
        setAllMergedlist(MS,SCHEDULERID,TLIST),
        [  pi([f,r],[
              PEOPLELIST=[f|r]?,
              pi( [availdate,templist],
                  [sentAvailDates(SCHEDULERID,f,availdate)?,
                   interSectionlist(availdate,TLIST,templist)?,
                   mergeAll(MS,SCHEDULERID,r,templist)
             ])
           ])
        ]
   )
).

proc(cancelAgreementForTheDate(MS,SchedulerID,Peoplelist,Date),
  for(participant,Peoplelist,[],
      requestCancelIfNecessary(MS,participant,SchedulerID,Date),
      true)
).

proc(requestCancel(MS,Participant,SchedulerID,Date),
  if(agreementAccepted(SchedulerID,Participant,Date),
       cancelAgreement(MS,Participant,SchedulerID,Date)
  )
).

proc(notifySuccessOnDate(MS,Init,SchedulerID,Peoplelist,Date),
 [
     notifySuccess(MS,Init,SchedulerID,Peoplelist,Date),
     for(participant,Peoplelist,[],
         NotifyAgreementToParticipant(MS,participant,SchedulerID,Date),
         true=true)
  ]).

proc(notifyFailScheduleMeeting(MS,Init,SchedulerID,Peoplelist),
  [
     notifyFail(MS,Init,SchedulerID,Peoplelist),
     for(participant,Peoplelist,[],
         notifyFail(MS,participant,SchedulerID,Peoplelist),true=true)
]).
```

A-8 The Precondition Axioms for Actions

```
poss(requestSchedulerScheduleAMeeting(_,_,_),_)
poss(requestEnterDateRange(_,_,_),_).               /* The action can be performed at any time * /
poss(enterDateRange(_,_,_,_),_).                    /* The action can be performed at any time * /
poss(obtainAvailDates(_,_,_),_).                    /* The action can be performed at any time * /
poss(sendAvailDates(_,_,_,_),_).                    /* The action can be performed at any time * /
poss(setAllMergedlist(_,_,_),_).                    /* The action can be performed at any time * /
poss(requestAgreement(_,_,_,_),_).                  /* The action can be performed at any time * /
poss(rejectAgreement(_,_,ReqID,_),S):- holds(agreementReqRcvd(ReqID),S).
    / * The action can be performed if the Participant received a request from the MS for an agreement to meet on the
    Date*/.
poss(cancelAgreement(_,_,_,_),_).                   / * The action can be performed at any time * /
poss(acceptCancel(_,_,_,_),_).                      / * The action can be performed at any time * /
poss(notifyAgreement(_,_,_,_),_).                   / * The action can be performed at any time * /
poss(notifySuccess(_,_,_,_,_),_).                   / * The action can be performed at any time * /
poss(notifyFail(_,_,_,_),_).                        / * The action can be performed at any time * /
```

Appendix B:
Modeling the Mail Order Business Process

B-1 Obtaining Simulation Traces under Unix

```
tiger 148 % congolog mailorder.pl

% compiling file /cs/home/fac1/lesperan/cogrobo/ConGolog98/congolog.pl
Disabled further Prolog informational messages.

WARNING: The GDL compiler has not yet been hooked up to this version
         of the system.

ConGolog Interpreter and GDL compiler
-------------------------------------

* Singleton variables, clause 20 of poss/2: S
* Approximate line: 144, file: '/cs/home/grad2/xiyun/thesis/mailorder.pl'
* multifile declaration missing for predicate non_fluent/1
Loaded model from file /cs/home/grad2/xiyun/thesis/mailorder.pl
Use the 'viewer.' goal to launch the viewer.

Quintus Prolog Release 3.2 (Sun 4, SunOS 5.5.1)
Copyright (C) 1994, Quintus Corporation.  All rights reserved.
301 East Evelyn Ave, Mountain View, California U.S.A. (415) 254-2800
Licensed to York Univerity, Canada

| ?- run.

$$$ >>>> startInterrupts in do([ ],s0)
$$$ >>>> mkOrder(cust1,item4,1111,company1) in do([ startInterrupts ],s0)
$$$ >>>> alarmCustomer(officeClerk1,company1,cust1,1,item4) in do([
mkOrder(cust1,item4,1111,company1) startInterrupts ],s0)
$$$ >>>> rejectOrder(officeClerk1,company1,cust1,1,item4) in do([
alarmCustomer(officeClerk1,company1,cust1,1,item4) mkOrder(cust1,item4,1111,company1)
startInterrupts ],s0)
$$$ >>>> stopInterrupts in do([ rejectOrder(officeClerk1,company1,cust1,1,item4)
 alarmCustomer(officeClerk1,company1,cust1,1,item4) mkOrder(cust1,item4,1111,com
pany1) ...],s0)

Final situation:
do(stopInterrupts,do(rejectOrder(officeClerk1,company1,cust1,1,item4),do(alarmCu
stomer(officeClerk1,company1,cust1,1,item4),do(mkOrder(cust1,item4,1111,company1
),do(startInterrupts,s0)))))

yes
| ?-
```

B-2 The Simulation Trace for Example 5 in Section 6.5

The sequence of actions performed is as follows:

- mkOrder(cust1,item2,1111,company1): order No. 1: cust1 makes an order for item2 to company1 and his card number is 1111.

- mkOrder(cust1,item1,1111,company1): order No. 2: cust1 makes an order for item1 to company1 and his card number is 1111.

- mkOrder(cust3,item3,3333,company2): order No. 3: cust3 makes an order for item3 to company2 and his card number is 3333.

- mkOrder(cust2,item2,2222,company2): order No. 4: cust2 makes an order for item2 to company2 and his card number is 2222.

- requestStock(officeClerk2,company2,stockClerk2,item2,4): officeClerk2 in company2 requests the stockClerk2 to provide the stock for item2 of order No. 4.

- rejectStockRequest(stockClerk2,company2,officeClerk2,item2,4): stockClerk2 in company2 rejects the request for the stock for item2 of order No.4 from the officeClerk2.

- rejectOrder(officeClerk2,company2,cust2,4,item2): officeClerk2 rejects the order made by cust2 for item2.

- requestStock(officeClerk2,company2,stockClerk2,item3,3): officeClerk2 in company2 requests the stock for item3 of order No. 3 to stockClerk2.

- acceptStockRequest(stockClerk2,company2,officeClerk2,item3,3): stockClerk2 in company2 accepts officeClerks's request for the stock for item3 of order No. 3 from .

- putOnHold(stockClerk2,company2,item3,3) : stockClerk2 in company2 puts an item3 into on-hold stock for order No.3.

- requestDebit(officeClerk2,company2,3,cust3,3333,30): officeClerk2 requests the bank clerk to check cust3's account 3333 for debiting 30.

- rejectDebit(1,company2,cust3,3333,30) : The bank clerk tells company2 that cust3 cannot pay 30 credits for order No. 1.

- cancelStockRequest(officeClerk2,company2,stockClerk2,item3,3): officeClerk2 requests stockClerk2 to cancel the reserved stock for item3 of order No.3.

- confirmCancelStock(stockClerk2,company2,officeClerk2,3,item3): stockClerk2 confirms officeClerk2 about canceling the reserved stock for item3 of order No. 3.

- moveOnHoldBackToStock(stockClerk2,company2,item3,3): stockClerk2 move an item3 from the on-hold stock for order No. 3 back to real stock.

- rejectOrder(officeClerk2,company2,cust3,3,item3): officeClerk2 rejects the order No. 3 for item3 made by cust3.

- requestStock(officeClerk1,company1,stockClerk1,item1,2): officeClerk1 in company1 requests stockClerk1 to provide the stock for item1 of order No.2.
- acceptStockRequest(stockClerk1,company1,officeClerk1,item1,2): stockClerk1 accepts officeClerk1's request for item1 of order No. 2.
- putOnHold(stockClerk1,company1,item1,2): stockClerk1 puts an item1 for order No. 2 into the on-hold stock.
- requestDebit(officeClerk1,company1,2,cust1,1111,10): officeClerk1 requests the bank to check cust1's account 1111 for debiting 10 from this account for order No. 2.
- acceptDebit(2,company1,cust1,1111,10): The bank clerk tells company1 that cust1 has enough money pay 10 credits.
- transferMoneyForOrder(officeClerk1,company1,cust1,2,1111,10): officeClerk1 requests the bank transfer 10 credits from cust1's account 1111 to company1's account for order No. 2.
- debitAcct(1111,10): The bank clerk debits 10 credits from the account 1111.
- creditAcct(c1,10): The bank clerk credits 10 into company1's account c1.
- confirmTransferMoney(3,cust1,1111,company1,10): The bank clerk confirm s company1 that credits 10 has been transferred from cust1's account to company1's account.
- mkInvoice(officeClerk1,company1,stockClerk1,item1,2): officeClerk1 tells stockClerk1 he invoiced the order No. 2 for item1.
- shipOrder(stockClerk1,company1,cust1,2,item1): stockClerk1 ships item1 of order No. 2 to cust1.
- RmvFromHoldForShipment(stockClerk1,company1,item1,2): stockClerk1 remove the shipped item from on hold stock.
- notifyShipment(officeClerk1,company1,cust1,item1,2): officeClerk1 notifies cust1 that item1 for order No. 2 shipped.
- requestStock(officeClerk1,company1,stockClerk1,item2,1): officeClerk1 request stockClerk1 to provide the stock for item2 of order No. 1.
- rejectStockRequest(stockClerk1,company1,officeClerk1,item2,1): stockClerk1 rejects the officeClerk1' request for stock for item2 of order No 1.
- rejectOrder(officeClerk1,company1,cust1,1,item2): officeClerk1 rejects cust1's order for item2 for order No. 1.

B-3 The Simulation Trace for Example 6 in Section 6.6

The sequence of actions performed is as follows:

- mkOrder(cust2,item2,2222,company1): cust2 makes an order for item2 to company1 and his card number is 2222.
- mkOrder(cust1,item2,1111,company1):cust1 makes an order for item2 to the company1 and his card number is 1111.
- requestStock(officeClerk1,company1,stockClerk1,item2,2): officeClerk1 requests stock for item2 for order No.2.
- rejectStockRequest(stockClerk1,company1,officeClerk1,item2,2): stockClerk1 rejects stock request for item2 for order No.2
- supply(item2,6): 6 item2 is supplied to the stock.
- supply(item2,6): 6 item2 is supplied to the stock.
- rejectOrder(officeClerk1,company1,cust1,2,item2): officeClerk1 rejects cust1's order No.2 for item2 .
- requestStock(officeClerk1,company1,stockClerk1,item2,1): officeClerk1 requests stock for item2 for order No.1.
- acceptStockRequest(stockClerk1,company1,officeClerk1,item2,1):stockClerk1 accepts the stock request from stockClerk1 for item2 for order No. 1.
- putOnHold(stockClerk1,company1,item2,1): stockClerk1 puts item2 on hold for order No.1.
- supply(item2,6): 6 item2 is supplied to the stock.
- requestDebit(officeClerk1,company1,1,cust2,2222,20): officeClerk1 requests the bank to check where it can debit 20 from the cust2's account 2222.
- acceptDebit(1,company1,cust2,2222,20): cust2's account 2222 has enough money to pay 20 debits.
- transferMoneyForOrder(officeClerk1,company1,cust2,1,2222,20): officeClker1 asks the bank transfers money 20 credits from cust2's account 2222 to coompany1' account.
- debitAcct(2222,20): Account 2222 is debited 20.
- creditAcct(c1,20): Account c1 is credited 20.
- confirmTransferMoney(2,cust2,2222,company1,20): the bank confirms company1 that money was transferred from cust2's account 2222 to company1 's account.
- mkInvoice(officeClerk1,company1,stockClerk1,item2,1): officeClerk1 makes an invoice for item2 for order No.1.
- shipOrder(stockClerk1,company1,cust2,1,item2),do(supply(item2,6): stockClerk1 ships the ordered item2 for order No.1 to cust2.
- RmvFromHoldForShipment(stockClerk1,company1,item2,1): stockClerk1 remove the shipped item from on hold stock.
- notifyShipment(officeClerk1,company1,cust2,item2,1): officeClerk1 notifies cust2 that the ordered item2 for order No. 1 was shipped.

B-4 The *ConGolog* Model for `OfficeClerk`

```
proc(officeClerk_behavior(OfficeClerk,CompanyName,StockClerk),
   efficientOrderProcessor(OfficeClerk,CompanyName,StockClerk)
).

proc(efficientOrderProcessor(OfficeClerk,Company,StockClerk),
  processOrders(OfficeClerk,Company,StockClerk)).

proc(processOrders(OfficeClerkName,CompanyName,StockClerk),
   ==>([orderID,custID,itemID],
       and(orderMade(orderID),
           and(val(orderCustomer(orderID),custID),
            and(val(orderItem(orderID),itemID),
              and(val(orderCompanyName(orderID),CompanyName),
                and(not(orderRejected(orderID)),
                 not(requestedStock(itemID,orderID))
            ))))),
       process(OfficeClerkName,CompanyName,StockClerk,custID,orderID,itemID)
   )
).

proc(process(OfficeClerkName,CompanyName,StockClerk,Customer,OrderID,ItemID),
  if(not(isSoldItem(ItemID)),
    verifyOrder(OfficeClerkName,CompanyName,Customer,OrderID,ItemID),
    processStockAndPayment(OfficeClerkName,CompanyName,StockClerk,
                                                 Customer,OrderID,ItemID)
  )
).

proc(verifyOrder(OfficeClerkName,CompanyName,Customer,OrderID,ItemID),
  [ alarmCustomer(OfficeClerkName,CompanyName,Customer,OrderID,ItemID),
    rejectOrder(OfficeClerkName,CompanyName,Customer,OrderID,ItemID)
  ]).

proc(processStockAndPayment(OfficeClerkName,CompanyName,StockClerk,Customer,
                                                          OrderID,ItemID),
[  achieve_AvailOfStock(OfficeClerkName,CompanyName,StockClerk,OrderID,ItemID),
   if(stockRequestRejected(OrderID),
      rejectOrder(OfficeClerkName,CompanyName,Customer,OrderID,ItemID),
      processPayment(OfficeClerkName,CompanyName,StockClerk,Customer,OrderID,ItemID)
   )
]).

proc(achieve_AvailOfStock(OfficeClerkName,CompanyName,StockClerk,OrderID,ItemID),
 [ requestStock(OfficeClerkName,CompanyName,StockClerk,ItemID,OrderID),
   stockRequestAnswered(OrderID)?
 ]
).

proc(processPayment(OfficeClerkName,CompanyName,StockClerk,Customer,OrderID,ItemID),
  pi([cardNo,amt],[
     and(val(orderCustomer(OrderID),Customer),
      and(val(orderCardNo(OrderID),cardNo),
       and(val(orderCompanyName(OrderID),CompanyName),
        and(val(orderItem(OrderID),ItemID),
            val(price(ItemID),amt)
     ))))?,
    achieve_DetermineWhetherAccountOk(OfficeClerkName,CompanyName,OrderID,
                                                Customer,cardNo,amt),
    if(debitReqAccepted(OrderID),
    transferMoneyAndInvoice(OfficeClerkName,CompanyName,
                       StockClerk,OrderID,ItemID,Customer,cardNo,amt),
    processCancel(OfficeClerkName,CompanyName,StockClerk,Customer,OrderID,ItemID)
```

```
        )
      ])
    ).

    proc(achieve_DetermineWhetherAccountOk(OfficeClerkName,CompanyName,
                                            OrderID,Customer,CardNo,Amt),
    [ requestDebit(OfficeClerkName,CompanyName,OrderID,Customer,CardNo,Amt),
      debitReqAnswered(OrderID)?
    ]
    ).

    proc(processCancel(OfficeClerkName,CompanyName,StockClerk,Customer,OrderID,ItemID),
      [cancelStockRequest(OfficeClerkName,CompanyName,StockClerk,ItemID,OrderID),
       stockRtndToInventory(OrderID)?,
       rejectOrder(OfficeClerkName,CompanyName,Customer,OrderID,ItemID)
      ]
    ).

    proc(transferMoneyAndInvoice(OfficeClerkName,CompanyName,StockClerk,OrderID,
                                 ItemID,CustomerID,CardNo,Amount),
      [achieve_TransferMoney(OfficeClerkName,CompanyName,CustomerID,
                                               OrderID,CardNo,Amount),
       mkInvoice(OfficeClerkName,CompanyName,StockClerk,ItemID,OrderID),
       orderShipped(OrderID)?,
       notifyShipment(OfficeClerkName,CompanyName,CustomerID,ItemID,OrderID)
      ]
    ).

    proc(achieve_TransferMoney(OfficeClerkName,CompanyName,CustomerID,
                                                OrderID,CardNo,Amount),
     [ transferMoneyForOrder(OfficeClerkName,CompanyName,CustomerID,
                                              OrderID,CardNo,Amount),
       transferMoneyAccepted(OrderID)?
     ]
    ).
```

B-5 The *ConGolog* Model for `BankClerk`

```
proc(bankClerk,
    processTransactions
).

proc(processTransactions,
   ==>([transID],
         and(debitReqTransRcvd(transID),
            not(debitReqTransProc(transID))),
          pi([cust,cardNo,companyName,amt],[
             and(val(transCustomer(transID),cust),
              and(val(transCardNo(transID),cardNo),
                and(val(transCompanyName(transID),companyName),
                 val(transAmount(transID),amt)
             )))?,
             replyDebitRequest(transID,companyName,cust,cardNo,amt)
         ])
      )
  #=
  ==>([transID],
        and(transferMoneyTransRcvd(transID),
          not(transferMoneyTransProc(transID))),

       pi([cust,cardNo,companyName,amt],[
```

```
          and(val(transCustomer(transID),cust),
              and(val(transCardNo(transID),cardNo),
                 and(val(transCompanyName(transID),companyName),
                   val(transAmount(transID),amt)
           )))?,
          achieve_TransferredMoney(transID,cust,cardNo,companyName,amt)
        ])
     )
).


proc(replyDebitRequest(TransID,CompanyName,Cust,CardNo,Amt),
            if(some(n,(some(m,and(and(val(acctBalance(CardNo),n),
                                    val(creditLimit,m)),n - Amt >= m)))),
             acceptDebit(TransID,CompanyName,Cust,CardNo,Amt),
             rejectDebit(TransID,CompanyName,Cust,CardNo,Amt)
          )
).

proc(achieve_TransferredMoney(TransID,Cust,CardNo,CompanyName,Amt),
   [
    transferMoney(TransID,Cust,CardNo,CompanyName,Amt),
    transferMoneyTransProc(TransID)?
   ]
).

proc(transferMoney(TransID,Cust,CardNo,CompanyName,Amt),
  [
     debitAcct(CardNo,Amt),
     creditCompanyAccount(CompanyName,Amt),
     confirmTransferMoney(TransID,Cust,CardNo,CompanyName,Amt)
  ]
).

proc(creditCompanyAccount(CompanyName,Amt),
   pi([companyAccount], [
        val(companyAccountNo(CompanyName),companyAccount)?,
        creditAcct(companyAccount,Amt)

   ])
).
```

B-6 The *ConGolog* Model for `Customer`

```
proc(customer(CustID,ItemID,CardNo,CompanyName),
     obtainItem(CustID,ItemID,CardNo,CompanyName)
).

proc(obtainItem(CustID,ItemID,CardNo,CompanyName),
     mkOrder(CustID,ItemID,CardNo,CompanyName)
).
```

B-7 The Whole *ConGolog* Model for the Mail-Order Business Process

/* Primitive Action Used in the *ConGolog* Model */

```
/*
      requestStock(OfficeClerk,Company,StockClerk,Item,OrderID)
          /* OfficeClerk in Company requests stock for Item for OrderID to StockClerk */

      acceptStockRequest(StockClerk,Company,OfficeClerk,Item,OrderID)
          /* StockClerk in Company accepts the stock request for Item for OrderID from OfficeClerk */

      rejectStockRequest(StockClerk,Company,OfficeClerk,Item,OrderID)
          /* StockClerk in Company rejects OfficeClerk's stock request for Item for OrderID*/

      cancelStockRequest(OfficeClerk,Company,StockClerk,Item,OrderID)
          /* OfficeClerk in Company cancel his request for stock for Item for OrderID to StockClerk */

      confirmCancelStock(StockClerk,Company,OfficeClerk,OrderID,Item),
          /* StockClerk confirms canceling the stock for Item for OrderID to the OfficeClerk */

      putOnHold(StockClerk,Company,Item,OrderID)
          /* StockClerk puts the ordered Item for OrderID on hold */

      rmvFromHoldForShipment(StockClerk,Company,Item,OrderID)
          /* StockClerk removes Item for OderID from on hold stock for shipment */

      moveOnHoldBackToStock(StockClerk,Company,Item,OrderID)
          /* StockClerk moves Item reserved for OderID from on hold stock back to free stock*/

      requestDebit(OfficeClerk,Company,OrderID,Customer,CardNo,Amount)
          /* OfficeClerk requests a bank to check whether Customer's account CardNo has enough money Amount
                                                                                   to pay for OrderID*/

      confirmDebit(TransactionID,Customer,CardNo,Amount)
          /* The bank confirms that Customer has enough money to pay the Amount*/

      rejectDebit(TransactionID,Company,Customer,CardNo,Amount)
          /* The bank rejects debits Amount of money from Customer's account CardNo*/

      transferMoneyForOrder(OfficeClerk,Company,Customer,Order,CardNo,Amount)
          /* OfficeClerk asks the bank to transfer Amount of money from Customer's CardNo for Order */

      confirmTransferMoney(TransactionID,CustomerID,CardNo,Company,Amount)
          /* The bank confirms to transfer Amount of money from Customer's CardNo for Order */

      debitAcct(CardNo,Amount)
          /* The bank debits the Amount of money from CardNo */

      creditAcct(CardNo,Amount)
          /* The bank credits the Amount of money into CardNo*/

      mkInvoice(OfficeClerk,Company,StockClerk,Item,OrderID)
          /* OfficeClerk notifies StockClerk that he made a invoice for Item for the OrderID */

      shipOrder(StockClerk,Company,Customer,OrderID,Item)
          /* StockClerk ships the Item for OrderID to Customer*/

      notifyShipment(OfficeClerk,Company,Customer,ItemID,OrderID)
          /* OfficeClerk notifies Customer the Item for OrderID shipped*/

      rejectOrder(OfficeClerk,Company,Customer,Order,Item)
          /* OfficeClerk rejects Customer's Order reqest for Item */
```

```
alarmCustomer(OfficeClerk,Company,Customer,OrderID,Item)
      /* OfficeClerk alarms Customer that Item for OrderID is not sold type */

mkOrder(CustomerID,Item,CardNo,Company )
      /* Customer makes an order for Item to Company and send his CardNo */
```

/* Exogenous Actions Used in the *ConGolog* Model */

```
supply(Item,Quantity
```

/* Predicate Fluents Defined in the *ConGolog* Model */

```
orderMade(OrderID)
requestedStock(Item,OrderID)
stockRequestAccepted(OrderID)
stockRequestRejected(OrderID)
stockRequestCancelled(OrderID,Item)
stockRtndToInventory(OrderID)
waitingForDebitAns(OrderID)
debitRequestSubmitted(OrderID)
debitRequestAccepted(OrderID)
debitRequestRejected(OrderID)
submittedTransferMoney(OrderID)
debitTransRcvd(TransactionID)
debitTransProc(TransactionID)
transferMoneyTransRcvd(TransactionID)
transferMoneyTransProc(TransactionID)
invoiceMade(OrderID)
orderShipped(OrderID)
orderRejected(OrderID)
Functional Fluents:
inStock(ItemType)
onHold(ItemType)
acctBalance(AcctNo)
orderCustomer(OrderID)
orderItem(OrderID)
orderCardNo(OrderID)
orderCompanyName(OrderID)
transCustomer(TransactionID)
transCardNo(TransactionID)
transAmount(TransactionID)
price(Item)
orderCtr
transCtr
*/
```

/* The Original ConGolog Model */

```
/* declaration for the primitive actions */

primAct(requestStock(_,_,_,_,_)).
primAct(acceptStockRequest(_,_,_,_,_)).
primAct(rejectStockRequest(_,_,_,_,_)).
primAct(cancelStockRequest(_,_,_,_,_)).
primAct(putOnHold(_,_,_,_)).
primAct(moveOnHoldBackToStock(_,_,_,_)).
PrimAct(rmvFromHoldForShipment(_,_,_,_)).
primAct(requestDebit(_,_,_,_,_,_)).
primAct(submitDebit(_,_,_,_)).
primAct(acceptDebit(_,_,_,_,_)).
primAct(confirmDebit(_,_,_,_)).
primAct(rejectDebit(_,_,_,_,_)).
primAct(transferMoneyForOrder(_,_,_,_,_,_)).
```

```
primAct(confirmTransferMoney(_,_,_,_,_)).
primAct(debitAcct(_,_)).
primAct(creditAcct(_,_)).
primAct(mkInvoice(_,_,_,_,_)).
primAct(shipOrder(_,_,_,_,_)).
primAct(notifyShiment(_,_,_,_,_)).
primAct(rejectOrder(_,_,_,_,_)).
primAct(alarmCustomer(_,_,_,_,_)).
primAct(mkOrder(_,_,_,_)).
primAct(confirmCancelStock(_,_,_,_,_)).


/* declaration for the exogenous actions */
primAct(supply(_,_)).


/* precondition Axioms for primitive actions */
poss(alarmCustomer(_,_,_,_,_),_).
poss(requestStock(_,_,_,_,_),_).
poss(confirmCancelStock(_,_,_,_,_),_).
poss(acceptStockRequest(_,_,_,_,OrderID),S):- holds(requestedStock(_,OrderID),S).
poss(rejectStockRequest(_,_,_,_,OrderID),S):- holds(requestedStock(_,OrderID),S).
poss(cancelStockRequest(_,_,_,_,OrderID),S):- holds(stockRequestAccepted(OrderID),S).
poss(putOnHold(_,_,Item,_),S):- holds(val(inStock(Item),N),S), N > 0.
poss(moveOnHoldBackToStock(_,_,Item,_),S):- holds(and(val(onHold(Item),N),N>0),S).
poss(rmvFromHoldForShipment(_,_,Item,_),S):- holds(and(val(onHold(Item),N),N>0),S).
poss(submitDebit(_,_,_,Amt),_):- Amt > 0.
poss(requestDebit(_,_,_,_,_,Amt),_):- Amt > 0.
poss(acceptDebit(TransID,_,_,_,_),S):- holds(debitReqTransRcvd(TransID),S).
poss(confirmDebit(TransID,_,_,_),S):- holds(debitTransRcvd(TransID),S).
poss(rejectDebit(TransID,_,_,_,_),S):- holds(debitReqTransRcvd(TransID),S).
poss(transferMoneyForOrder(_,_,_,_,_,Amt),_):- Amt > 0.
poss(confirmTransferMoney(TransID,_,_,_,_),S):- holds(transferMoneyTransRcvd(TransID),S).
poss(debitAcct(_,Amt),_):- Amt > 0.
poss(creditAcct(_,Amt),_):- Amt > 0.
poss(mkInvoice(_,_,_,_,_),_).
poss(shipOrder(_,_,_,OrderID,ItemID),S):-
     holds(and(val(orderItem(OrderID),ItemID),and(val(onHold(ItemID),N), N >  0)),S).
poss(notifyShipment(_,_,_,_,_),S).
poss(rejectOrder(_,_,_,_,_),_).
poss(mkOrder(_,_,_,_),_).


/* precondition Axioms for exogenous actions */
poss(supply(_,_),_).

/* Successor State Axioms for all actions */

holds(orderMade(OrderID),do(A,S)):-
     (A = mkOrder(_,_,_,_),
      holds(val(orderCtr,OrderID),S));
      holds(orderMade(OrderID),S).

holds(requestedStock(Item,OrderID),do(A,S)):-
      A = requestStock(_,_,_,Item,OrderID);
      holds(requestedStock(Item,OrderID),S).

holds(stockRequestAccepted(OrderID),do(A,S)):-
      A = acceptStockRequest(_,_,_,_,OrderID);
      holds(stockRequestAccepted(OrderID),S).

holds(onHoldPut(OrderID),do(A,S)):-
      A=putOnHold(_,_,_,OrderID);
      holds(onHoldPut(OrderID),S).

holds(stockRequestRejected(OrderID),do(A,S)):-
      A = rejectStockRequest(_,_,_,_,OrderID);
```

```
      holds(stockRequestRejected(OrderID),S).

holds(stockRequestCancelled(OrderID,Item),do(A,S)):-
      A = cancelStockRequest(_,_,_,Item,OrderID);
      holds(stockRequestCancelled(OrderID,Item),S).

holds(stockRtndToInventory(OrderID),do(A,S)):-
      A = moveOnHoldBackToStock(_,_,_,OrderID);
      holds(stockRtndToInventory(OrderID),S).

holds(itemRmvFromHoldForShipment(OrderID),do(A,S)):-
      A = rmvFromHoldForShipment(_,_,_,OrderID);
      holds(itemRmvFromHoldForShipment(OrderID),S).

holds(waitingForDebitAns(OrderID),do(A,S)):-
      A = submitDebit(OrderID,_,_,_);
     (holds(waitingForDebitAns(OrderID),S),
      A \= confirmDebit(_,_,_,_)).

holds(waitingForDebitReqAns(OrderID),do(A,S)):-
      A = requestDebit(_,_,OrderID,_,_,_);
     (holds(waitingForDebitReqAns(OrderID),S),
      A \= acceptDebit(_,_,_,_,_), A \= rejectDebit(_,_,_,_,_)).

holds(waitingForTransferMoneyAns(OrderID),do(A,S)):-
      A = transferMoneyForOrder(_,_,_,OrderID,_,_);
     (holds(waitingForTransferMoneyAns(OrderID),S),
      A \=confirmTransferMoney(_,_,_,_,_)).

holds(debitRequestSubmitted(OrderID),do(A,S)):-
      A = submitDebit(OrderID,_,_,_);
      holds(debitRequestSubmitted(OrderID),S).

holds(debitReqSubmitted(OrderID),do(A,S)):-
      A = requestDebit(_,_,OrderID,_,_,_);
      holds(debitReqSubmitted(OrderID),S).

holds(transferMoneyAccepted(OrderID),do(A,S)):-

     (A = confirmTransferMoney(_,_,_,_,_),
      holds(waitingForTransferMoneyAns(OrderID),S));
      holds(transferMoneyAccepted(OrderID),S).

holds(debitRequestAccepted(OrderID),do(A,S)):-
     (A = confirmDebit(_,_,_,_), holds(waitingForDebitAns(OrderID),S));
      holds(debitRequestAccepted(OrderID),S).

holds(debitReqAccepted(OrderID),do(A,S)):-
     (A = acceptDebit(_,_,_,_,_),holds(waitingForDebitReqAns(OrderID),S));
      holds(debitReqAccepted(OrderID),S).

holds(debitReqRejected(OrderID),do(A,S)):-
     (A = rejectDebit(_,_,_,_,_),holds(waitingForDebitReqAns(OrderID),S));
      holds(debitReqRejected(OrderID),S).

holds(debitRequestRejected(OrderID),do(A,S)):-
     (A = rejectDebit(_,_,_,_,_),holds(waitingForDebitAns(OrderID),S));
      holds(debitRequestRejected(OrderID),S).

holds(stockcancelled(OrderID),do(A,S)):-
     (A=confirmCancelStock(_,_,_,OrderID,_));
      holds(stockcancelled(OrderID),S).

holds(submittedTransferMoney(OrderID),do(A,S)):-
      A = transferMoneyForOrder(_,_,_,OrderID,_,_);
      holds(submittedTransferMoney(OrderID),S).
```

```
holds(debitReqTransRcvd(TransID),do(A,S)):-
     (A = requestDebit(_,_,_,_,_,_), holds(val(transCtr,TransID),S));
      holds(debitReqTransRcvd(TransID),S).

holds(debitTransRcvd(TransID),do(A,S)):-
     (A = submitDebit(_,_,_,_), holds(val(transCtr,TransID),S));
      holds(debitTransRcvd(TransID),S).

holds(debitTransProc(TransID),do(A,S)):-
      A = confirmDebit(TransID,_,_,_);
      A = rejectDebit(TransID,_,_,_,_);
      holds(debitTransProc(TransID),S).

holds(debitReqTransProc(TransID),do(A,S)):-
      A = acceptDebit(TransID,_,_,_,_);
      A = rejectDebit(TransID,_,_,_,_);
      holds(debitReqTransProc(TransID),S).

holds(transferMoneyTransRcvd(TransID),do(A,S)):-
     (A = transferMoneyForOrder(_,_,_,_,_,_), holds(val(transCtr,TransID),S));
      holds(transferMoneyTransRcvd(TransID),S).

holds(transferMoneyTransProc(TransID),do(A,S)):-
      A = confirmTransferMoney(TransID,_,_,_,_);
      holds(transferMoneyTransProc(TransID),S).

holds(invoiceMade(OrderID),do(A,S)):-
      A = mkInvoice(_,_,_,_,OrderID);
      holds(invoiceMade(OrderID),S).

holds(orderShipped(OrderID),do(A,S)):-
      A = shipOrder(_,_,_,OrderID,_);
      holds(orderShipped(OrderID),S).

holds(orderRejected(OrderID),do(A,S)):-
      A = rejectOrder(_,_,_,OrderID,_);
      holds(orderRejected(OrderID),S).

holds(shipmentNotified(CustomerID,ItemID,OrderID),do(A,S)):-
      A = notifyShipment(_,_,CustomerID,ItemID,OrderID);
      holds(shipmentNotified(CustomerID,ItemID,OrderID),S).

holds(val(inStock(Item),N),do(A,S)):-    /* assumes Item is ground */
     (A = moveOnHoldBackToStock(_,_,Item,_), holds(val(inStock(Item),M),S), N is M + 1);
     (A = supply(Item,Q), holds(val(inStock(Item),M),S), N is M + Q);
     (A = putOnHold(_,_,Item,_), holds(val(inStock(Item),M),S), N is M - 1);
     (holds(val(inStock(Item),N),S), ground(Item),
      A \= moveOnHoldBackToStock(_,_,Item,_),
      A \= supply(Item,Q), A \= putOnHold(_,_,Item,_)).

holds(val(onHold(Item),N),do(A,S)):-     /* assumes Item is ground */
     (A = moveOnHoldBackToStock(_,_,Item,_), holds(val(onHold(Item),M),S), N is M - 1);
     (A = rmvFromHoldForShipment(_,_,Item,_),holds(val(onHold(Item),M),S), N is M - 1);
     (A = putOnHold(_,_,Item,_), holds(val(onHold(Item),M),S), N is M + 1);
     (holds(val(onHold(Item),N),S), ground(Item),
      A \= rmvFromHoldForShipment(_,_,Item,_)
      A \= moveOnHoldBackToStock(_,_,Item,_), A \= putOnHold(_,_,Item,_)).

holds(val(acctBalance(CardNo),N),do(A,S)):-
     (A = creditAcct(CardNo,Amt),
      holds(val(acctBalance(CardNo),M),S), N is M + Amt);
     (A = debitAcct(CardNo,Amt),
      holds(val(acctBalance(CardNo),M),S), N is M - Amt);
     (holds(val(acctBalance(CardNo),N),S),
      A \= creditAcct(CardNo,_), A \= debitAcct(CardNo,_)).

holds(val(orderCustomer(ID),Cust),do(A,S)):-
```

```
     (A = mkOrder(Cust,_,_,_), holds(val(orderCtr,ID),S));
      holds(val(orderCustomer(ID),Cust),S).

holds(val(orderItem(ID),Item),do(A,S)):-
     (A = mkOrder(_,Item,_,_), holds(val(orderCtr,ID),S));
      holds(val(orderItem(ID),Item),S).

holds(val(orderCardNo(ID),CardNo),do(A,S)):-
     (A = mkOrder(_,_,CardNo,_), holds(val(orderCtr,ID),S));
      holds(val(orderCardNo(ID),CardNo),S).

holds(val(orderCompanyName(ID),CardNo),do(A,S)):-
     (A = mkOrder(_,_,_,CardNo), holds(val(orderCtr,ID),S));
      holds(val(orderCompanyName(ID),CardNo),S).

holds(val(transCustomer(ID),Cust),do(A,S)):-
     (A = requestDebit(_,_,_,Cust,_,_), holds(val(transCtr,ID),S));
     (A = submitDebit(_,Cust,_,_), holds(val(transCtr,ID),S));
     (A = transferMoneyForOrder(_,_,Cust,_,_,_), holds(val(transCtr,ID),S));
      holds(val(transCustomer(ID),Cust),S).

holds(val(transCardNo(ID),CardNo),do(A,S)):-
     (A = requestDebit(_,_,_,_,CardNo,_), holds(val(transCtr,ID),S));
     (A = submitDebit(_,_,CardNo,_), holds(val(transCtr,ID),S));
     (A = transferMoneyForOrder(_,_,_,_,CardNo,_), holds(val(transCtr,ID),S));
      holds(val(transCardNo(ID),CardNo),S).

holds(val(transCompanyName(ID),CompanyName),do(A,S)):-
     (A = requestDebit(_,CompanyName,_,_,_,_),
      holds(val(transCtr,ID),S));
     (A = transferMoneyForOrder(_,CompanyName,_,_,_,_),
      holds(val(transCtr,ID),S));
      holds(val(transCompanyName(ID),CompanyName),S).

holds(val(transAmount(ID),Amt),do(A,S)):-
     (A = requestDebit(_,_,_,_,_,Amt), holds(val(transCtr,ID),S));
     (A = submitDebit(_,_,_,Amt), holds(val(transCtr,ID),S));
     (A = transferMoneyForOrder(_,_,_,_,_,Amt), holds(val(transCtr,ID),S));
      holds(val(transAmount(ID),Amt),S).

holds(val(price(Item),Amt),do(_,S)):-
      holds(val(price(Item),Amt),S).

holds(val(orderCtr,N),do(A,S)):-
     (A = mkOrder(_,_,_,_), holds(val(orderCtr,M),S), N is M + 1);
     (holds(val(orderCtr,N),S), A \= mkOrder(_,_,_,_)).

holds(val(transCtr,N),do(A,S)):-
     (A = requestDebit(_,_,_,_,_,_),holds(val(transCtr,M),S), N is M + 1);
     (A = submitDebit(_,_,_,_), holds(val(transCtr,M),S), N is M + 1);
     (A = transferMoneyForOrder(_,_,_,_,_,_),
      holds(val(transCtr,M),S), N is M + 1);
     (holds(val(transCtr,N),S),  A \= requestDebit(_,_,_,_,_,_),
      A \= submitDebit(_,_,_,_), A \= transferMoneyForOrder(_,_,_,_,_,_)).

/* Defined Fluents */
holds(stockRequestAnswered(OrderID),S):-
      holds(onHoldPut(OrderID),S); holds(stockRequestRejected(OrderID),S).

holds(debitRequestAnswered(OrderID),S):-
      holds(debitRequestAccepted(OrderID),S);
      holds(debitRequestRejected(OrderID),S).

holds(debitReqAnswered(OrderID),S):-
      holds(debitReqAccepted(OrderID),S);
      holds(debitReqRejected(OrderID),S).
```

```
/* Initial State */

holds(val(creditLimit,-10),_).
holds(val(inStock(item1),10),s0).
holds(val(inStock(item2),0),s0).
holds(val(inStock(item3),3),s0).
holds(val(onHold(_),0),s0).
holds(val(acctBalance(1111),100),s0).
holds(val(acctBalance(2222),20),s0).
holds(val(acctBalance(3333),0),s0).
holds(val(companyAccountNo(company1),c1),_).
holds(val(acctBalance(c1),0),s0).
holds(val(companyAccountNo(company2),c2),_).
holds(val(acctBalance(c2),0),s0).
holds(val(price(item1),10),s0).
holds(val(price(item2),20),s0).
holds(val(price(item3),30),s0).
holds(val(orderCtr,1),s0).
holds(val(transCtr,1),s0).

/*non fulents*/
non_fluent(isSoldItem(_)).
isSoldItem(item1).
isSoldItem(item2).
isSoldItem(item3).

/* Exogenous actions, comments the following block to disable the exogenous action */

/* exoDensity(100).*/
/*exoAct(supply(item1,6),6,6). */
/*exoAct(supply(item2,6),5,5).*/
/*exoAct(supply(item3,6),7,7). */

/* tracingTest. */
tracingExec.

/* procedure definitions */

/* Main procedure to instantiated the whole mail order process */
proc(main,
     customer(cust1,item4,1111,company1) #>
                    /* this can be adjusted to a real situation */
     mailOrderCompany_behavior(officeClerk2,stockClerk2,company2)#=
     mailOrderCompany_behavior(officeClerk1,stockClerk1,company1)#=
     bank_behavior
).

/* customer's behavior in the mail order process */

proc(customer(CustID,ItemID,CardNo,CompanyName),
     obtainItem(CustID,ItemID,CardNo,CompanyName)
).

proc(obtainItem(CustID,ItemID,CardNo,CompanyName),
     mkOrder(CustID,ItemID,CardNo,CompanyName)
).

/* mail order company's behavior in the mail order process */
proc(mailOrderCompany_behavior(OfficeClerk,StockClerk,CompanyName),
     officeClerk_behavior(OfficeClerk,CompanyName,StockClerk)#=
     stockClerk_behavior(StockClerk,CompanyName,OfficeClerk)
).

/* office clerk's behavior in the mail order process */

proc(officeClerk_behavior(OfficeClerk,CompanyName,StockClerk),
     efficientOrderProcessor(OfficeClerk,CompanyName,StockClerk)
```

```
).

/* stock clerk's behavior in the mail order process */

proc(stockClerk_behavior(StockClerk,CompanyName,OfficeClerk),
     stockInformant(StockClerk,CompanyName,OfficeClerk)#=
     updateStockProcessor(StockClerk,CompanyName)#=
     shipmentProcessor(StockClerk,CompanyName)
 ).

/* bank's behavior in the mail order process */

proc(bank_behavior,
     bankClerk_behavior
).

/* bank clerk's behavior in the mail order process */

proc(bankClerk_behavior,
    processTransactions
).


/* ConGolog process model for the EffientOrderProcessor role */

proc(efficientOrderProcessor(OfficeClerk,Company,StockClerk),
     processOrders(OfficeClerk,Company,StockClerk)).

proc(processOrders(OfficeClerkName,CompanyName,StockClerk),
   ==>([orderID,custID,itemID],
        and(orderMade(orderID),
         and(val(orderCustomer(orderID),custID),
          and(val(orderItem(orderID),itemID),
           and(val(orderCompanyName(orderID),CompanyName),
            and(not(orderRejected(orderID)),
             not(requestedStock(itemID,orderID))
      ))))),
      process(OfficeClerkName,CompanyName,StockClerk,custID,orderID,itemID)
   )
).

proc(process(OfficeClerkName,CompanyName,StockClerk,Customer,OrderID,ItemID),
     if(not(isSoldItem(ItemID)),
        verifyOrder(OfficeClerkName,CompanyName,Customer,OrderID,ItemID),

processStockAndPayment(OfficeClerkName,CompanyName,StockClerk,Customer,OrderID,ItemID)
     )
).

proc(verifyOrder(OfficeClerkName,CompanyName,Customer,OrderID,ItemID),
     [ alarmCustomer(OfficeClerkName,CompanyName,Customer,OrderID,ItemID),
       rejectOrder(OfficeClerkName,CompanyName,Customer,OrderID,ItemID)
    ]
).

proc(processStockAndPayment(OfficeClerkName,CompanyName,StockClerk,Customer,OrderID,ItemI
D),
     [
       achieve_AvailOfStock(OfficeClerkName,CompanyName,StockClerk,OrderID,ItemID),
       if(stockRequestRejected(OrderID),
          rejectOrder(OfficeClerkName,CompanyName,Customer,OrderID,ItemID),
          processPayment(OfficeClerkName,CompanyName,StockClerk,Customer,OrderID,ItemID)
       )
     ]
).

proc(achieve_AvailOfStock(OfficeClerkName,CompanyName,StockClerk,OrderID,ItemID),
```

```
     [
       requestStock(OfficeClerkName,CompanyName,StockClerk,ItemID,OrderID),
       stockRequestAnswered(OrderID)?
     ]
).

proc(processPayment(OfficeClerkName,CompanyName,StockClerk,Customer,OrderID,ItemID),
     pi([cardNo,amt],[
         and(val(orderCustomer(OrderID),Customer),
          and(val(orderCardNo(OrderID),cardNo),
           and(val(orderCompanyName(OrderID),CompanyName),
            and(val(orderItem(OrderID),ItemID),
             val(price(ItemID),amt)
         ))))?,
         achieve_AccountOkOrNot(OfficeClerkName,CompanyName,OrderID,Customer,cardNo,amt),
         if(debitReqAccepted(OrderID),
            transferMoneyAndInvoice(OfficeClerkName,CompanyName,StockClerk,
                                        OrderID,ItemID,Customer,cardNo,amt),
            processCancel(OfficeClerkName,CompanyName,StockClerk,Customer,OrderID,ItemID)
         )
     ])
).

proc(achieve_AccountOkOrNot(OfficeClerkName,CompanyName,OrderID,Customer,CardNo,Amt),
     [
       requestDebit(OfficeClerkName,CompanyName,OrderID,Customer,CardNo,Amt),
       debitReqAnswered(OrderID)?
     ]
).

proc(processCancel(OfficeClerkName,CompanyName,StockClerk,Customer,OrderID,ItemID),
     [
       cancelStockRequest(OfficeClerkName,CompanyName,StockClerk,ItemID,OrderID),
       stockRtndToInventory(OrderID)?,
       rejectOrder(OfficeClerkName,CompanyName,Customer,OrderID,ItemID)
     ]
).

proc(transferMoneyAndInvoice(OfficeClerkName,CompanyName,StockClerk,OrderID,ItemID,
                             CustomerID,CardNo,Amount),
    [ achieve_TransferMoney(OfficeClerkName,CompanyName,CustomerID,
                            OrderID,CardNo,Amount),
      mkInvoice(OfficeClerkName,CompanyName,StockClerk,ItemID,OrderID),
      orderShipped(OrderID)?,
      notifyShipment(OfficeClerkName,CompanyName,CustomerID,ItemID,OrderID)
   ]
).

proc(achieve_TransferMoney(OfficeClerkName,CompanyName,CustomerID,OrderID,CardNo,Amount),
     [
       transferMoneyForOrder(OfficeClerkName,CompanyName,CustomerID,
                             OrderID,CardNo,Amount),
       transferMoneyAccepted(OrderID)?
     ]
).

/* process ConGolog model for the update-stock processor role */

proc(updateStockProcessor(StockClerk,Company),
     updateStock(StockClerk,Company)
).

proc(updateStock(StockClerkName,CompanyName),
     ==>([orderID,item],
          and(val(orderItem(orderID),item),
           and(val(orderCompanyName(orderID),CompanyName),
            and(stockRequestAccepted(orderID),
```

```
            not(onHoldPut(orderID))
          ))),
          putOnHold(StockClerkName,CompanyName,item,orderID)
     )
     #=
     ==>([orderID,item],
          and(val(orderItem(orderID),item),
           and(ordershipped(orderID),
            and(val(orderCompanyName(orderID),CompanyName),
             not(itemRmvFromHoldForShipment(orderID))
          ))),
          rmvFromHoldForShipment(StockName,CompanyName,item,orderID)
     )
     #=
     ==>([orderID,item],
          and(val(orderItem(orderID),item),
           and(stockcancelled(orderID),
            and(val(orderCompanyName(orderID),CompanyName),
             not(stockRtndToInventory(orderID))
          ))),
          moveOnHoldBackToStock(StockClerkName,CompanyName,item,orderID)
     )
).

/* process ConGolog model for the stock informant role */

proc(stockInformant(StockClerkName,CompanyName,OfficeClerk),
     processStockRequest(StockClerkName,CompanyName,OfficeClerk)
).

proc(processStockRequest(StockClerkName,CompanyName,OfficeClerk),
     ==>([orderID,itemID],
          and(requestedStock(itemID,orderID),
           not(stockRequestAnswered(orderID))
          ),
          replyStockRequest(StockClerkName,CompanyName,OfficeClerk,orderID,itemID)
     )
     #=
     ==>([orderID,itemID],
          and(stockRequestCancelled(orderID,itemID),
           not(stockRtndToInventory(orderID))
          ),
          cancelStockRequestProcess(StockClerkName,CompanyName,OfficeClerk,
                                    orderID,itemID)
     )
).

proc(replyStockRequest(StockClerkName,CompanyName,OfficeClerk,OrderID,ItemID),
     if(some(n,and(val(inStock(ItemID),n),n > 0)),
        acceptRequestStock(StockClerkName,CompanyName,OfficeClerk,ItemID,OrderID),
        rejectStockRequest(StockClerkName,CompanyName,OfficeClerk,ItemID,OrderID)
     )
).

proc(cancelStockRequestProcess(StockClerkName,CompanyName,OfficeClerk,OrderID,ItemID),
     [
       confirmCancelStock(StockClerkName,CompanyName,OfficeClerk,OrderID,ItemID),
       stockRtndToInventory(OrderID)?
     ]
).

proc(acceptRequestStock(StockClerkName,CompanyName,OfficeClerk,ItemID,OrderID),
     [
       acceptStockRequest(StockClerkName,CompanyName,OfficeClerk,ItemID,OrderID),
       onHoldPut(OrderID)?
     ]
).
```

```
/* process ConGolog model for the shipment processor role */

proc(shipmentProcessor(StockClerk,Company),
     processShipment(StockClerk,Company)).

proc(processShipment(StockClerk,Company),
     ==>([orderID,itemID,custID],
          and(val(orderItem(orderID),itemID),
           and(val(orderCustomer(orderID),custID),
            and(val(orderCompanyName(orderID),Company),
             and(invoiceMade(orderID),
               not(orderShipped(orderID))
          )))),
          achieve_ItemShipped(StockClerk,Company,custID,orderID,itemID)
     )
).

proc(achieve_ItemShipped(StockClerkName,CompanyName,Customer,OrderID,ItemID),
     [
       shipOrder(StockClerkName,CompanyName,Customer,OrderID,ItemID),
       orderShipped(OrderID)?
     ]).

/* process ConGolog model for the activities inside the bank clerk position */
proc(processTransactions,
     ==>([transID],
          and(debitReqTransRcvd(transID),
           not(debitReqTransProc(transID))
          ),
          pi([cust,cardNo,companyName,amt],[
              and(val(transCustomer(transID),cust),
               and(val(transCardNo(transID),cardNo),
                and(val(transCompanyName(transID),companyName),
                 val(transAmount(transID),amt)
              )))?,
              replyDebitRequest(transID,companyName,cust,cardNo,amt)
          ])
     )
     #=
     ==>([transID],
          and(transferMoneyTransRcvd(transID),
           not(transferMoneyTransProc(transID))
          ),
          pi([cust,cardNo,companyName,amt],[
              and(val(transCustomer(transID),cust),
               and(val(transCardNo(transID),cardNo),
                and(val(transCompanyName(transID),companyName),
                 val(transAmount(transID),amt)
              )))?,
              achieve_TransferredMoney(transID,cust,cardNo,companyName,amt)
          ])
      )
).

proc(replyDebitRequest(TransID,CompanyName,Cust,CardNo,Amt),
     if(some(n,(some(m,and(and(val(acctBalance(CardNo),n),val(creditLimit,m)),
                                                            n - Amt >= m)))),
        acceptDebit(TransID,CompanyName,Cust,CardNo,Amt),
        rejectDebit(TransID,CompanyName,Cust,CardNo,Amt)
     )
).

proc(achieve_TransferredMoney(TransID,Cust,CardNo,CompanyName,Amt),
     [
       transferMoney(TransID,Cust,CardNo,CompanyName,Amt),
       transferMoneyTransProc(TransID)?
```

```
     ]
).

proc(transferMoney(TransID,Cust,CardNo,CompanyName,Amt),
     [
       debitAcct(CardNo,Amt),
       creditCompanyAccount(CompanyName,Amt),
       confirmTransferMoney(TransID,Cust,CardNo,CompanyName,Amt)
     ]
).

proc(creditCompanyAccount(CompanyName,Amt),
     pi([companyAccount], [
        val(companyAccountNo(CompanyName),companyAccount)?,
        creditAcct(companyAccount,Amt)
     ])
).
```