Modeling Mental States in Requirements Engineering – An Agent-Oriented Framework Based on *i** and CASL

Alexei Lapouchnian

A thesis submitted to the Faculty of Graduate Studies in
partial fulfillment of the requirements
for the degree of

Master of Science

Supervisor: Yves Lespérance

Graduate Programme in Computer Science

York University

Toronto, Canada

July, 2004

Abstract

As the problems that software systems are used to solve grow in size and complexity, it becomes harder and harder to analyze these problems and come up with system requirements specifications using informal requirements engineering approaches. Recently, Goal-Oriented Requirements Engineering (GORE), where stakeholder goals are identified, analyzed/decomposed and then assigned to software components or actors in the environment, and Agent-Oriented Software Engineering (AOSE), where goals are objectives that agents strive to achieve, have been gaining popularity. Their reliance on goals makes GORE and AOSE a good match. A number of goal-oriented approaches include formal components that allow for rigorous analysis of system properties. However, they do not support reasoning about the goals and knowledge of agents. This thesis presents an agent-oriented requirements engineering approach that combines informal *i** models with formal specifications written in the CASL language. CASL's support for agent goals and knowledge allows for formal analysis of agent interactions, goal decompositions, and epistemic feasibility of agent plans. Intentional Annotated Strategic Rationale (iASR) diagrams based on the SR diagrams of *i** are proposed in this thesis, together with the mapping rules for creating the corresponding formal CASL specifications. A methodology for the combined use of *i** and CASL is proposed and applied to a meeting scheduling process specification.

Acknowledgments

I would like to thank my supervisor, Professor Yves Lespérance, for introducing me to this research area and for his advice, guidance, careful editing, and patience during the preparation of this thesis.

I am grateful to my parents, my brother, and the rest of my family for their continuous support and encouragement during my graduate studies at York.

Contents

1 Introduction

In this thesis, we propose an agent-oriented requirements engineering methodology that supports formal modeling of the agents' mental states (goals and knowledge). This chapter provides an overview of the problem area (Section 1.1), describes the specific problem that we deal with in this work (Section 1.2), sketches our approach for solving this problem (Section 1.3), and outlines the rest of the thesis (Section 1.4).

1.1 Overview of the Area

Modern software systems are becoming increasingly complex, with lots of subsystems and interactions. In today's world, software applications are not confined to just one computer or even a local area network. More and more software systems are becoming internet-aware and distributed. The recent popularity of electronic commerce, web services, and peer-to-peer applications confirms the need for software engineering methods for constructing applications that are open, distributed, and adaptable to change. While technologies like remote procedure calls are still adequate for certain types of software systems, more and more researchers and practitioners are looking at agent technology as a basis for distributed applications.

Agents are active, social, and adaptable software system entities situated in some environment and capable of autonomous execution of actions in order to achieve their set objectives [Wooldridge, 1997]. Most problems are too complex to be solved by just one agent — one must create a multiagent system (MAS) with several agents having to work together to achieve their objectives and ultimately deliver the desired application. Multiple agents in a multiagent system represent multiple perspectives, multiple threads of control, or competing interests [Jennings, 1999]. Therefore, adopting the agent-oriented approach to software engineering means that the problem is decomposed into multiple, autonomous, interacting agents, each with a particular objective. Multiagent systems are a great match for open, complex systems in many areas such as process control, manufacturing, information management, electronic commerce, etc. [Jennings and Wooldridge, 1998] discusses domains that are suitable for the application of agent technology.

Agents in a multiagent system frequently represent individuals, companies, or parts of companies. This means that there is an "underlying organizational context" [Jennings, 1999] in multiagent systems. Therefore, one of the most important features of agents is their social ability. Like humans, they need to coordinate their activities, cooperate, negotiate, request help from others, etc. Unlike in object-oriented or component-based systems, interactions in multiagent systems occur through high-level speech act-based [Searle, 1969] agent communication languages, so interactions are mostly performed not at the syntactic level, but at the knowledge level, in terms of goal delegation, etc. [Jennings, 1999]. Speech act-based agent communication allows for the specification and execution of generic communication protocols and supports reasoning about the effects of communication without knowing the exact content of the messages, just the types of the messages (e.g., request, inform). For example, requests by other agents make the requested agent acquire new goals, while the messages informing the agent of something modify its knowledge base. Since agents are aware of the purpose of their interactions, not all of them need to be fixed at design time. This supports flexible runtime behaviour with dynamic goal delegation and team formation.

Requirements engineering (RE) is the area of software engineering that deals with the discovery and specification of the objectives for the system under development. This is an extremely important activity since the measure of success of software systems is the degree to which they satisfy their requirements. Many of the existing software engineering methods have concentrated on the design of the system and devoted relatively little attention to requirements engineering. As well, most RE approaches have assumed that the initial formulation of the requirements was given. However, the activities that lead to the formulation of the initial requirements for the system-to-be were mostly ignored. Recently, *goal-oriented requirements engineering* [Dardenne *et al.*, 1993] has become prominent. Goal-oriented RE approaches (e.g., KAOS [Dardenne *et al.*, 1993] and Tropos [Castro *et al.*, 2002]) devote a lot of attention to understanding the environment of the system-to-be (the organizational context) and the rationale (the "why" [Yu and Mylopoulos, 1994]) for the system. This is usually referred to as the *early phase* of requirements engineering. The goals of the stakeholders (the individuals or organizations that influence the system and/or are influenced by it) are analyzed, refined and later assigned to the components that are part of the system and the agents that are in its environment. These approaches are requirements-driven as opposed to being driven by the design or implementation factors. Their reliance on goals makes goal-oriented requirements engineering methods and agent-oriented software engineering a great match. Agent-oriented analysis is central to requirements engineering since the assignment of responsibilities for goals and constraints among components in the software-to-be and agents in the environment is the main outcome of the RE process [van Lamsweerde, 2000]. Therefore, it is natural to use a goal-oriented requirements engineering approach when developing multiagent systems. The goals of the stakeholders are refined and then assigned to the agents, thus turning into the objectives of agents in a multiagent system.

The early phase of requirements engineering — the domain analysis — is usually done informally, possibly with the help of informal diagrammatic notations. *i** [Yu, 1995] is one such notation. It supports the modeling of stakeholders (actors), their goals, the intentional dependencies that exist in the organization, as well as the reasoning each actor goes through while attempting to achieve its goals. As problems grow in size and complexity, the need for formal analysis during the early RE phase increases. However, formal support for the early-phase requirements engineering has been relatively sparse. There are several approaches that augment the *i** framework with tools for formal analysis. For example, Formal Tropos [Fuxman *et al.*, 2001b] adds model checking support for *i**'s Strategic Dependency diagrams, while another approach [Wang, 2001] uses the *ConGolog* [De Giacomo *et al.*, 2000] agent programming language to animate and verify *i** models. Unfortunately, in the above methods, the goals of the agents are abstracted out of the formal specifications. This is done due to the fact that the formal components of these approaches (the NuSMV [Cimatti *et al.*, 1999] input language and ConGolog respectively) do not support reasoning about agent goals. Reasoning about agent knowledge is not supported either. However, we note that agent communication is very important in multiagent systems since these systems are developed as social structures. Most of the interactions among agents involve knowledge exchange and goal delegation. Thus, complementing informal modeling techniques such as *i** with formal analysis of agent goals and knowledge is very useful for the design of multiagent systems. Moreover, the adoption of goal-oriented requirements engineering makes formal analysis of agent goals very useful at the RE stage of the software engineering process.

1.2 The Problem

Goals are the most appropriate abstraction for specifying the needs of the stakeholders in the environment of the system under development and are the natural means for identifying the purpose of agents in multiagent systems. During the requirements engineering phase of software development, the goals of stakeholders are identified, refined, and assigned to either the actors in the environment, thus becoming the assumptions that the designers make about the environment, or to the agents that are part of the system, therefore becoming the system requirements.

When goal-oriented RE is used, it is easy to make the transition from the requirements to the high-level system specifications. For example, goal-oriented RE approaches assign goals or tasks to the system's agents. In multiagent systems, these requirements become the objectives that the agents strive to achieve. During requirements analysis, it is important to determine how the stakeholders as well as the agents in the system-to-be will collaborate to achieve the system's objectives. In multiagent systems, agents delegate some of their responsibilities to other agents that are better equipped for achieving them. On the other hand, the same agents may be asked to help others with their goals. Strategic relationships among agents (represented by intentional dependencies in *i** models) thus become high-level patterns of inter-agent communication.

With the *i** modeling framework, designers identify system stakeholders, determine their goals and analyze how responsibilities can be assigned to system components and how inter-agent dependencies can be configured so that the system's objectives are met. In the above context, while it is possible to informally analyze small systems, formal analysis is needed for any realistically-sized system to determine whether such distributed requirements imposed on each agent in the system are correctly decomposed from the stakeholder goals, consistent and, if properly met, achieve the system's objectives.

The aim of this work is to devise an agent-oriented requirements engineering approach with a formal component that supports reasoning about agents' goals (and knowledge), thus allowing for rigorous formal analysis of the requirements expressed as the objectives of the agents in multiagent systems. While some goal-oriented RE approaches (e.g., KAOS) support the formal analysis of goals, their temporal logic-based formalizations are not agent-oriented, i.e., which agent has a goal is not formalized. Also, most RE approaches do not support the formal analysis of agent knowledge.

1.3 The Approach

In this thesis, we are proposing an agent-oriented requirements engineering approach that combines the *i** modeling framework [Yu, 1995] and the Cognitive Agents Specification

Language [Shapiro and Lespérance, 2001]. CASL is a language with formal semantics and thus can be used to complement the informal diagrammatic notation of *i**.

The *i** approach has two types of models: Strategic Dependency (SD) and Strategic Rationale (SR). SD diagrams are used to model actors (stakeholders) that are in the environment of the system-to-be or are part of the system itself along with the intentional dependencies (goal, task, etc. delegation) among these actors. These diagrams model the external relationships among the actors. On the other hand, Strategic Rationale (SR) diagrams are used to model the reasoning each actor goes through while attempting to achieve its goals. Thus, SR models concentrate more on the internals of the actors.

With our *i**-CASL-based approach, a CASL model can be used both as a requirements analysis tool and as a formal high-level specification for a multiagent system that satisfies the requirements because it is possible to produce a high-level, formal model of the MAS right from the *i** diagrams developed during the requirements analysis performed with our proposed method. This model can be formally analyzed using the CASLve [Shapiro *et al.*, 2002] tool or other tools and the results can be fed back into the requirements model. After requirements analysis is finished, the CASL model becomes the starting point for the design of the system. CASL's support for reasoning about knowledge also opens new possibilities for the formal analysis of issues such as privacy and allows for more precise specification of agent interactions.

We extend the approach presented in [Wang, 2001] that proposed the use of the ConGolog language to provide formal semantics as well as verification and animation tools for the SR diagrams of *i**. In our framework, we use CASL to provide formal semantics for the *i** diagrams. Since CASL supports specification of agent goals and knowledge, formal reasoning about agent goals, goal delegation, agent knowledge, and inter-agent communication is now possible. In this thesis, we define a mapping from *i** diagrams into corresponding CASL models that can be formally analyzed.

*i** models have limited facilities for specifying agent processes. Rather, the *i** framework concentrates on the social and intentional aspects of the system-to-be and its environment. CASL specifications have two parts: a procedural part that is used to specify the behaviour of the agents in a multiagent system and a declarative part where the changes to the mental states of the agents are specified. It is natural to pair up CASL specifications with SR-level *i** diagrams since they are a much closer match for CASL than SD diagrams. SD diagrams are simply too high-level for CASL to be useful in providing a formal semantics for them. In fact, not unlike ConGolog in the framework described in [Wang, 2001], CASL requires a greater level of detail and precision than even SR diagrams can offer.

An *i** model is associated with the corresponding CASL specifications through a mapping. We introduce extensions to the notation of SR diagrams to allow the modeler to disambiguate and add details to the models so they can be mapped into CASL. We call these Intentional Annotated SR (iASR) diagrams. We use the idea of composition and link annotations [Wang, 2001] to add (along with *applicability condition* annotations) the necessary details to the SR diagrams. We also suggest a number of rules for the use of means-ends decompositions (see Section 4.3.6 for details) in iASR diagrams to simplify and streamline the mapping to CASL. Each element of iASR diagrams is then mapped into CASL for formal analysis with the help of the CASL verification environment (CASLve) [Shapiro *et al.*, 2002] or other CASL-based tools. Unlike the approach in [Wang, 2001], agent goals are not removed from the agent specifications, but are added to the agents' mental states, thus allowing the agents to reason about their objectives. Information exchanges among agents are also formalized as mental state changes in CASL specifications. The intentional dependencies that are present in the *i** models will be modeled in CASL as interactions among agents with the help of such actions as `request` and `inform`. These interactions bring change into the agents' mental states allowing them to receive new information or accept responsibility for achieving certain goals for their fellow agents. We also introduce the notion of self-acquired goals (Section 4.3.8), which can be used by the designer to assign goals to agents.

We develop a methodology for the combined use of *i** and CASL for requirements engineering. This methodology is rooted in Tropos and Wang's methodology for the combined use of *i** and ConGolog and includes a number of suggested modifications to the *i** modeling framework.

1.4 Outline of the Thesis

Chapter 2 provides the background material for the thesis. In Chapter 3, we review related approaches and compare them to our own. There, we talk about the related areas of computer science and software engineering — agents, agent-oriented software engineering, requirements engineering, as well as describe the specific methodologies, frameworks, and languages that we use in our approach — Tropos, *i**, and the Cognitive Agents Specification Language. Chapter 4 discusses our formalism and notation, and the mapping process from iASR diagrams into CASL models, while Chapter 5 describes our goal-oriented requirements engineering methodology that combines the *i** modeling framework with CASL. A case study, which illustrates our methodology on the process of scheduling meetings in an organization, is presented in Chapter 6. We conclude in Chapter 7 by outlining the contributions of the thesis and identifying avenues for future work.

2 Foundations

In this chapter, we introduce the background material for the thesis. Here, we talk about specific techniques and approaches that we used in this research. Section 2.1 introduces the *i** modeling framework, while in Section 2.2 we talk about the Cognitive Agents Specification Language (CASL) as well as the epistemic feasibility of agent plans.

2.1 The *i** Modeling Framework

*i** [Yu, 1995] is an agent-oriented modeling framework that can be used for requirements engineering, business process reengineering, organizational impact analysis, and software process modeling. Since we are most interested in the application of the framework to modeling systems' requirements, our description of *i** is geared towards requirements engineering. The framework has two main components: the Strategic Dependency (SD) model and the Strategic Rationale (SR) model.

Since *i** supports the modeling activities that take place before the system requirements are formulated, it can be used for both the early and late phases of the requirements engineering process. During the early requirements phase, the *i** framework is used to model the environment of the system-to-be. It facilitates the analysis of the domain by allowing the modeler to diagrammatically represent the stakeholders of the system, their objectives, and their relationships. The analyst can therefore visualize the current processes in the organization and examine the rationale behind these processes. The *i** models developed at this stage help in understanding why a new system is needed. During the late requirements phase, the *i** models are used to propose the new system configurations and the new processes and evaluate them based on how well they meet the functional and non-functional (qualitative) needs of the users.

In the following sections, we briefly present the elements of the *i** modeling framework: actors, intentional dependencies, as well as Strategic Dependency and Strategic Rationale models.

2.1.1 Actors

*i** centers on the notion of *intentional actor* and *intentional dependency*. The actors are described in their organizational setting and have attributes such as goals, abilities, beliefs, and commitments. In *i** models, an actor depends on other actors for the achievement of its goals, the execution of tasks, and the supply of resources, which it cannot achieve, execute, and obtain by itself, or not as cheaply, efficiently, etc. Therefore, each actor can use various opportunities to achieve more by depending on other actors. At the same time, the actor becomes vulnerable if the actors it depends upon do not deliver. Actors are seen as being *strategic* in the sense that they are concerned with the achievement of their objectives and strive to find a balance between their opportunities and vulnerabilities. It is easy to see that most human organizations (as well as software systems) can be modeled using the ideas of actors and intentional dependencies.

While actors in *i** are any units to which "intentional dependencies can be ascribed" [Yu, 1995], when *i** is used for requirements engineering, the actors represent the system's stakeholders ("people or organizations who will be affected by the system and who have a direct or indirect influence on the system requirements" [Kotonya and Sommerville, 1998]) as well as the agents of the system-to-be. The goals of the stakeholders thus become the objectives of the actors representing them. In the early requirements phase, when the organizational context for the system-to-be is analyzed, the dependencies in the models represent the current state of affairs in the organization, the way the stakeholders' objectives are presently met (without the help of the new system). In the late requirements phase, they represent the desired new state of the organization including the new system.

It is useful to subdivide actors into agents, roles, and positions. *Agents* are concrete physical actors, systems or humans. A *role* is an "abstract characterization of the behaviour of a social actor within some specialized context, domain, or endeavour" [Alencar *et al.*, 2000]. A *position* is a set of *socially recognized* roles typically played by one agent. For example, the same person/agent in a company can play the roles of a webmaster and a database administrator with different sets of dependencies associated with each role. The introduction of roles and positions facilitates the analysis of dependencies by grouping the related ones together.

2.1.2 Intentional Dependencies

When a pair of agents participate in an intentional dependency, the depending agent is called the *depender*, the depended agent is called the *dependee*, and the subject of the dependency (the goal, the task, the resource, or the softgoal) is called the *dependum*. Dependencies between actors are identified as *intentional* if they appear as a result of agents pursuing their goals.

There are four types of dependencies in *i**. In a *goal dependency*, the depender depends on the dependee for the achievement of one of its goals (for bringing about a certain state in the world). It is up to the dependee to decide how to achieve that goal. In a *task dependency* the depender depends on the dependee for the execution of a certain task. Unlike a goal dependency, here the dependee must execute exactly the task that the depender requested. In a *resource dependency*, the dependee is expected to provide a resource for the depender. The resource could be some physical resource (e.g., money or steel) or information. In a *softgoal dependency*, the depender depends on the dependee to "perform some task that meets the softgoal" [Yu, 1995]. The notion of softgoals (quality goals) is related to the notion of non-functional requirements [Chung *et al.*, 2000]. Softgoals are the goals that do not have a clear-cut satisfaction condition. For softgoals one needs to find solutions that are just "good enough". Examples of softgoals are security, efficiency, speed, customer satisfaction, etc. In *i**, the dependee usually identifies alternative ways for achieving (to a certain degree) the softgoals. The selection of the right alternative is generally made by the depender.

The model distinguishes among three degrees of dependency. For the depender, the strength of the dependency corresponds to the level of its vulnerability to the failure of the dependency. The stronger the dependency, the more vulnerable the depender becomes if the dependee cannot provide the dependum. On the other hand, a stronger dependency means that the dependee will make more effort in providing the dependum (the assumption here is that the dependee is cooperative or there exists a reciprocal dependency). In an *open dependency*, the dependee's failure to achieve the goal, perform the task, or furnish the resource will to a certain degree affect the depender, but without serious consequences. From the point of view of the dependee, an open dependency is a declaration that it can provide the dependum for some depender. In a *committed dependency*, the depender will be quite seriously affected by the failure of the dependee to provide the dependum — some chosen course of action would fail as a result. There can be other potential courses of action for the depender, but, in any case, the depender would incur losses if the dependum is not provided. This means that the depender is concerned about the dependee having a viable way to supply the dependum. The dependee will also try to make sure that it has a viable way to do that. The strongest dependency is called *critical*. Here, if the dependee does not provide the dependum, all courses of action for the achievement of an associated goal, which are known to the depender, would fail. In this case, the depender is concerned not only with the viability of that dependency, but also with the viability of the dependee's dependencies, and so on [Yu, 1995].

As we noted above, in all four dependencies, the depender gains the ability to achieve the goal/softgoal, execute the required task, or acquire the needed resource, but at the same time becomes vulnerable should the dependee fail to deliver. We must remind the reader here that since the agents are not objects, one cannot guarantee that the dependee will always provide the dependum even if it is perfectly capable of doing so: it may simply choose not do it based on its current situation. Therefore, it is important for dependers to try to mitigate the vulnerability that stems from their dependencies. One way to do it is to create an *enforceable* dependency [Yu, 1995]. A dependency is enforceable if there is a reciprocal dependency. This means that providing the dependum is in the dependee's interests since if it fails to supply the dependum, the depender can make one of the dependee's own goals fail. For example, a car owner depends on a garage for the quality of his/her car repairs, while the garage depends on the car owner for continued business. [Yu, 1995] identifies several ways of reducing the depender's vulnerability. The way to get *assurance* is to make sure that the dependency is reciprocal or otherwise is the interests of the dependee. Insurance reduces the depender's vulnerability by reducing the degree of dependence on a particular dependee. For example, one way to do this is to have several dependees for the same dependum.

2.1.3 Strategic Dependency Models

A Strategic Dependency (SD) model is a "network of dependency relationships among actors" [Yu, 1995]. This model represents the actors and their intentional dependencies graphically allowing for easier understanding and analysis of the organization and its processes. The SD model captures the intentionality of the processes, what is important to its participants, while abstracting over all other details. The diagrammatic notation allows for representing agents, positions, and roles, all four types of intentional dependencies, as well as their strength. The model allows for the analysis of the direct or indirect dependencies of each actor and exploration of the opportunities and vulnerabilities of actors (analysis of chains of dependencies emanating from actor nodes is helpful for vulnerability analysis). Figure 3.1 presents the SD diagram notation. The diagram has two actors: Agent1 and Role1. Role1 depends on Agent1 for the achievement of the goal

`Goal1`. On the other hand, Agent1 needs Role1 to provide it with `Resource1` and to execute `Task1`.

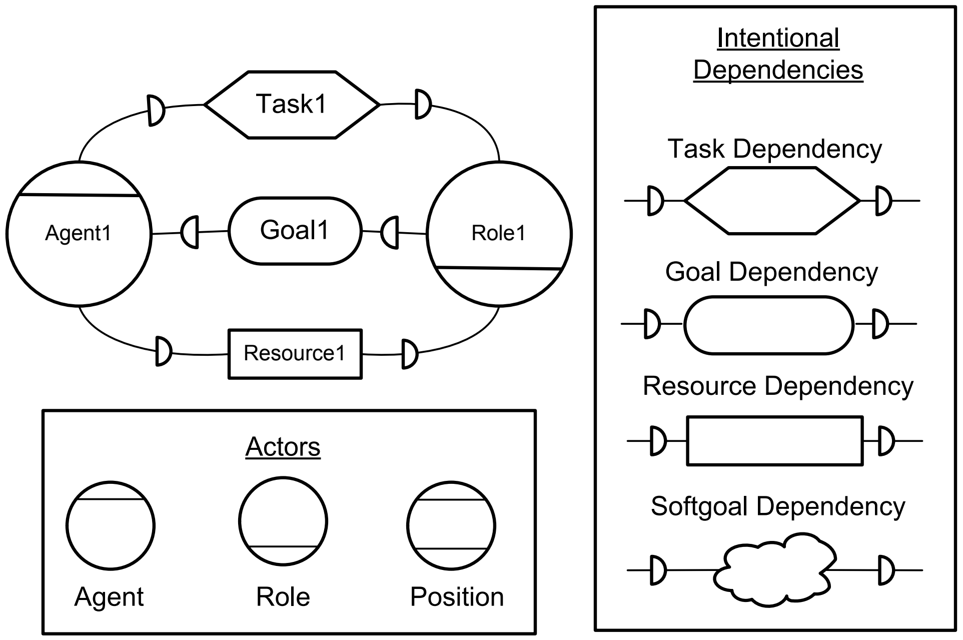


Figure 3.1. The SD diagram notation.

SD models can be used to model the existing processes in the organization (the early requirements phase) before the new system is introduced. In this case, the actors are the stakeholders of the system-to-be. SD models help in the analysis of stakeholder goals, their current dependencies, and the rationale for the current processes in the organization. The model is also useful in the identification of the organization's need for a new system.

On the other hand, SD diagrams can be used during the late requirements analysis phase to create a high-level model of the system and its organizational environment. This model is used to understand how the network of intentional dependencies can be reorganized with the introduction of the system as one or more actors in the diagram. Opportunity and vulnerability analysis can be performed from the point of view of each actor. One can also analyze the potential system-and-environment configurations and see which ones are the best with respect to the achievement of stakeholder goals and softgoals.

2.1.4 Strategic Rationale Models

Strategic Rationale models are used to explore the rationale behind the processes in systems and organizations. In SR models, the rationale behind process configurations can be explicitly described, in terms of process elements and relationships among them [Yu, 1995]. The model provides a lower-level abstraction to represent the intentional aspects of organizations and systems: while the SD model only looked at the external relationships among actors, the SR model provides the capability to analyze in great detail the internal processes within the actors that are part of the system and of its environment. The model allows for deeper understanding of what each actor's needs are and how these needs are met; it also enables the analyst to assess possible alternatives in the definition of the processes to better address the concerns of the actors.

SR models have four types of nodes, one for each dependency type. These nodes are: *goal*, *task*, *resource*, and *softgoal*. Goals are the states of the world that the actors would like to achieve. The goal node itself does not specify how the goal is to be achieved, therefore alternatives can be considered. Task nodes specify "particular ways of achieving something" [Yu, 1995]. Resources are physical or informational entities that can be provided and/or required by actors. Softgoals are similar to goals in that they are the conditions in the world that the actor would like to achieve; the difference is that these conditions are qualitative, without a clear-cut satisfaction condition. Softgoals in SR diagrams are used as restrictions or selection criteria for choosing among alternatives. Figure 3.2 illustrates the main elements of the SR modeling notation. It is the SR diagram for the Meeting Initiator actor, who is part of the meeting scheduling process fully described in Chapter 6.

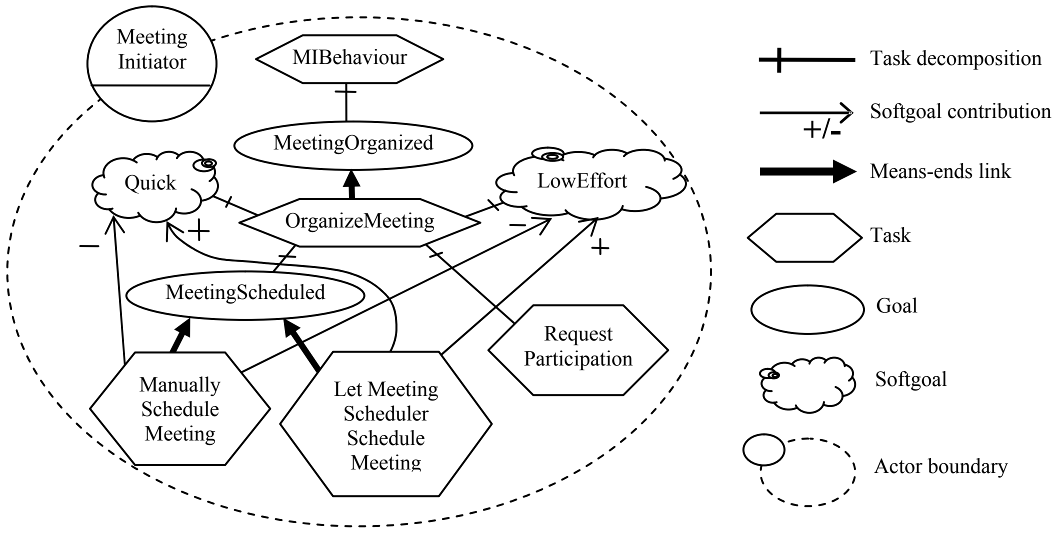


Figure 3.2. The SR diagram notation.

The above-described nodes are related by two types of links: *task decomposition* links, and *means-ends* links. Means-ends links specify alternative ways to achieve goals, perform tasks, etc. Means-ends links connect the end, which is usually a goal, but can also be a task, a resource, or a softgoal, with a means of achieving it. The means is usually a task since "the notion of task embodies *how* to do something" [Yu, 1995]. For example, in the model in Figure 3.2, there are two ways to achieve the goal `MeetingOrganized`: the tasks `ManuallyScheduleMeeting` and `LetMeetingSchedulerScheduleMeeting`. Task decomposition links connect a task with its components. These components could be simpler tasks, subgoals, resources needed for the task, or softgoals. In Figure 3.2, the task `OrganizeMeeting` is decomposed into the goal `MeetingOrganized` and the task `RequestParticipation`. It is possible to link a task/goal node with an intentional dependency going to another actor to represent its delegation to that actor. The SR model is strategic in that its elements are included only if they are considered important enough to affect the achievement of some goal [Yu, 1995]. The presence of softgoal nodes with positive or negative contribution links connecting them with other nodes in the SR diagram allows the analyst to evaluate alternative process configurations and select the best one. For example, in Figure 3.2 the task

`ManuallyScheduleMeeting` contributes negatively to the softgoals `Quick` and `LowEffort`, while the task `LetMeetingSchedulerScheduleMeeting` contributes positively to both softgoals.

The central concept in the SR model is the notion of a *routine* — an interconnected collection of process elements serving some purpose for an agent. An actor has the *ability* to do something if it has a routine for it. A routine may delegate tasks, goals, etc. to other actors. A routine is said to be *workable* if the actor believes (at "process design time") that it can successfully carry it out (at "run-time") [Yu, 1995]. The notions of ability and workability are important for analyzing whether the actor has processes for accomplishing its goals and whether these processes are going to work.

2.2 The Cognitive Agents Specification Language

The Cognitive Agents Specification Language (CASL) [Shapiro and Lespérance, 2001] is a formal specification language that combines theories of action [Reiter, 1991] and mental states [Scherl and Levesque, 1993] expressed in the situation calculus [McCarthy and Hayes, 1969] with ConGolog [De Giacomo *et al.*, 2000], a concurrent, non-deterministic agent-oriented programming language with a formal semantics. CASL uses special predicates [Shapiro and Lespérance, 2001] to formally express the agents' knowledge and goals; communicative actions (e.g., `inform`) are used to describe inter-agent communication and ConGolog is then employed to specify the behaviour of agents. This combination produces a very expressive language that supports high-level reasoning about the agents' mental states. The logical foundations of CASL allow it to be used to specify and analyze a wide variety of multiagent systems. For example, it can support non-deterministic systems and systems with incompletely specified initial state. In addition, different agents can be specified at different levels of abstraction. For instance, some agent specifications will make use of both goals and knowledge, while others will use only knowledge (or goals), and some may use neither. CASL specifications consist of two parts: the model of the domain and its dynamics and the specification of the agents' behaviour.

2.2.1 Modeling the Domain

In order to be able to reason about the processes executing in a certain domain, that domain must be formally specified — what predicates describe the domain, what primitive actions are available to the agents, what the preconditions and effects of these actions are, what is known about the initial state of the system. CASL represents a dynamic domain declaratively using an action theory [Reiter, 1991] that is formulated in situation calculus [McCarthy and Hayes, 1969]. The situation calculus is a predicate calculus language for describing dynamic domains. The key notion in the situation calculus is that of a *situation*. A situation is a state of the domain that results from a particular sequence of actions. The situation calculus assumes that the world starts in a certain state (initial situation). In the model of a domain, there is a set of initial situations corresponding to the ways agents think the world might be like initially. The constant S_0 specifies the domain's actual initial situation. Initial situations do not have predecessors meaning that no actions already have been performed. Actions performed by various agents (and only these actions) change the current situation. The term *do(a,s)* represents the unique situation that results from performing the action *a* in situation *s*. Situations therefore can be organized in trees. The situations are the nodes of the tree with the roots being the initial situation; the actions are the edges in those trees.

Fluents are the predicates and functions that change from situation to situation. They have a situation as their last argument. For example, *Near(a,b,s)* might be the fluent that is used to state that some object *a* is near some object *b* in situation *s*. *Primitive actions* are executed by the agents in the domain. Precondition axioms for actions specify under which conditions the primitive actions are executable. The predicate *Poss(a,s)* denotes the executability of action *a* in situation *s*. For example, the formula below states that the action *pickup(x)* is possible when the robot arm is not holding anything and its is next to *x* and *x* is not heavy [De Giacomo *et al.*, 2000]:

$$Poss(pickup(x), s) \equiv \forall y. \neg Holding(y, s) \wedge NextTo(x, s) \wedge \neg Heavy(x)$$

To find legal executions of programs, the interpreter must reason about the preconditions and effects of actions. Primitive actions have effects on fluents. It is also necessary to provide *frame* axioms to describe which fluents are not affected by the actions. The number of these axioms is potentially very large. In the worst case, one would need a frame axiom for every combination of primitive action and fluent. This is known as the *frame problem*. CASL uses the solution to this problem proposed in [Reiter, 1991]. In this approach, *successor state axioms*, which encode effects of primitive actions on fluents, are used. Each successor state axiom defines which primitive actions have what effects on some particular fluent. For example, below is a possible successor state axiom for *Holding(x,s)*. It says that for the agent to be holding some object *x* in the situation following the execution of the action *a* in situation *s*, either *a* must have been the action of picking up *x*, or the agent had been holding the object *x* in situation *s* before *a* occurred, and *a* was not the action of dropping *x*:

$$Holding(x, do(a, s)) \equiv a = pickup(x) \vee (Holding(x, s) \wedge a \neq drop(x))$$

The relation $s \preceq s'$ over situations holds if *s'* results from performing an executable (possibly empty) sequence of primitive actions in *s*. In this approach, a domain is specified by a theory containing axioms of these types:

- Axioms that describe the initial state of the domain and the initial mental states of all of the agents.
- Action precondition axioms, one for each primitive action *a*, defining *Poss(a,s)*.

- Successor state axioms, one for each fluent $F(x_1,\ldots,x_n,s)$, which characterize the conditions under which $F(x_1,\ldots,x_n,do(a,s))$ holds in terms of what holds in situation s.
- Unique name axioms for primitive actions.
- Domain-independent foundational axioms (these include the unique name axioms for situations and the induction axiom).

2.2.2 Modeling Agents' Mental States

CASL models two characteristics of agents' mental states, goals and knowledge. The formal representation for both goals and knowledge is based on a possible worlds semantics incorporated in the situation calculus, where situations are used as possible worlds [Moore, 1985; Scherl and Levesque, 1993]. What agents know is modeled through an accessibility relation *K(agt,s',s)*. It holds if the situation *s'* is compatible with what the agent *agt* knows in situation *s*, i.e., in situation *s*, the agent thinks that it might be in the situation *s'*. In this case, the situation *s'* is called *K*-accessible. The *K* accessibility relation is used to define what the agents know. An agent knows some formula φ if φ is true in all *K*-accessible situations:

$$\mathbf{Know}(agt,\phi,s)\overset{def}{=}\forall s'(K(agt,s',s)\supset\phi[s'])$$

Here, $\phi[s]$ is the formula ϕ with all free occurrences of the special constant *now* substituted with s. For example, **Know***(agt,HaveKey(Door1,now),s)* stands for $\forall s'.(K(agt,s',s) \supset HaveKey(Door1,s'))$. Often, when there is no possibility of confusion, the *now* constant is suppressed (e.g., **Know***(agt, HaveKey(Door1),s)*). Constraints on the *K* relation ensure that what is known is true and agents have positive and negative introspection, i.e., they always know whether or not they know something. If the agent knows either φ or its negation, it knows whether it holds or not:

$$\mathbf{KWhether}(agt,\phi,s) \overset{def}{=} \mathbf{Know}(agt,\phi,s) \vee \mathbf{Know}(agt,\neg\phi,s)$$

Also, the abbreviation **KRef***(agt,θ,s)* is used for $\exists t.\mathbf{Know}(agt,t=\theta,s)$. This means that the agent knows the value of *θ*. Additional communicative actions, `informWhether` and `informRef`, are used to inform agents about the truth value of a formula or the value of a function respectively.

The *K* relation on initial situations is specified through the initial state axioms. For non-initial state axioms the successor state axiom for the *K* relation shows how agent knowledge changes from situation to situation. Here, we assume that the action `inform` is the only knowledge-producing action (the modifications needed to support other knowledge producing actions are straightforward), and use the following successor state axiom:

$$\begin{aligned} &K(agent,s'',do(a,s)) \equiv \exists s'\ (K(agent,s',s) \wedge s'' = do(a,s') \wedge Poss(a,s') \\ &\qquad \wedge\ \forall informer,\varphi\ (a=inform(informer,agent,\varphi) \supset \varphi[s']))) \end{aligned}$$

It says that the situation *s''* is *K*-accessible from the situation *do(a,s)* under the following conditions: if the action *a* is not an `inform` action, then the situation preceding *s''*, *s'*, must be *K*-accessible from *s*, it must be the case that executing the action *a* in *s'* takes you to *s''*, and that action must be executable in the situation *s'*. If, on the other hand, the action *a* is the `inform` action, then in addition to the above conditions, *φ* must hold in *s'*, i.e., when an agent informs another that *φ*, *φ* must be true. Here, *φ* is a formula encoded as a term as in [De Giacomo *et al.*, 2000]. This successor state axiom ensures that the agents are aware of the execution of every action that occurs (and the fact that those actions were executable). In case of the `inform` actions, the recipient is sure that the content of the message holds. This axiom only handles knowledge expansion with the `inform` action. It is easy to modify the successor state axiom to handle other types of communicative actions or belief revision. For example, [Lespérance, 2002] shows the version that supports informing about the value of a function (using the communicative action `informRef`), [Shapiro and Lespérance, 2001] modifies the axiom to handle encrypted messages, while [Shapiro *et al.*, 2000] provides an account of belief revision compatible with CASL. Perception actions can also be modeled.

Another accessibility relation on situations is used to model agents' goals. The relation *W(agt,s',s)* holds if the situation *s'* is compatible with what the agent wants in *s*. Here, the agent may want something that does not currently hold. Agents may even want things that they know are impossible to attain, but the goals of agents must be consistent with what they know. Thus, the goals of agents are defined to be the formulae that are true in all *W*-accessible situations that have a *K*-accessible situation in the past [Shapiro and Lespérance, 2001]. The formula ψ below has two free variables, *now* and *then*. *then* refers to the future situation where the goal has been achieved, while *now* is the current situation along the path to *then*. So, goals are formulae that are true in the *W*-accessible paths that start from a *K*-accessible situation:

$$\mathbf{Goal}(agt,\psi,s) \overset{def}{=}$$
$$\forall now, then.(K(agt,now,s) \wedge W(agt,then,s) \wedge now \ \underline{\mathbf{p}} \ then \supset \psi[now,then])$$

Here, $\psi[s',s'']$ is the formula ψ with s' and s'' substituted for *now* and *then* respectively. The successor state axiom for *W* is similar to the one for *K* and describes what affects the agents' goals. In particular, `request` actions coming from other agents make the agent acquire new goals provided that the goals are consistent with existing goals of the agent. We use the following successor state axiom for *W*:

$$W(agt,then,do(a,s)) \equiv (W(agt,then,s) \wedge$$
$$\forall requester,\psi,now.(a = request(requester,agt,\psi) \wedge K(agt,now,s) \wedge$$
$$now \ \underline{\mathbf{p}} \ then \wedge \neg\mathbf{Goal}(agt,\neg\psi,s) \supset \psi[do(a,now),then]))$$

The axiom says that the situation *then* is *W*-accessible from *do(a,s)* iff it is *W*-accessible from *s* and if the action *a* is a *request* action requesting that ψ hold, and *now* is the current situation along the path defined by *then*, and the agent does not have the goal that $\neg\psi$ in *s*, then ψ holds at *do(a,now),then)*. This formalization does not support goal revision. However, the communicative action `cancelRequest` can be used by the requester to cancel a previously made request (we omit the modifications to the successor state axiom for *W*). Note that if the agent already has the goal that $\neg\psi$, it will not adopt the goal that ψ to avoid inconsistency.

In our approach, we mostly use achievement goals that specify the desired states of the world that one must eventually reach. These goals have one situation variable *now*. For such goals, we use the definition that says that eventually φ holds (note that *now* is replaced by *s'*) on the path defined by (*now*,*then*):

$$\textbf{Eventually}(\varphi(now),now,then) \overset{def}{=} \exists s'.now \preceq s' \preceq then \wedge \varphi[now/s'],$$

Note that in CASL to simplify modeling, it is assumed that every agent knows about all the actions performed by all the agents. If this is not desired, one can use encrypted messages [Shapiro and Lespérance, 2001] or use communication actions that do not make the complete message explicit (`informWhether` and `informRef` as in [Lespérance, 2002]).

As mentioned earlier, the framework requires that the content of the `inform` messages be true. Thus, after receiving a message, the recipient knows that the sender knew that the content of the message was true. This prevents agents from sending false information. Also, when executing the `request` action, the sender must not have goals that conflict with the request. Alternatively, one could require that the requesting agent have the goal that it is asking another agent to adopt. These constraints are expressed through the precondition axioms for the `inform` and `request` actions:

$$Poss(inform(informer, recepient, \phi), s) \equiv \mathbf{Know}(informer, \phi, s)$$
$$Poss(request(requester, recepient, \psi), s) \equiv \neg\mathbf{Goal}(requester, \neg\psi, s)$$

2.2.3 Specifying Agent Behaviour

The ConGolog agent programming language [De Giacomo *et al.*, 2000] is used to specify the behaviour of the agents in this framework. ConGolog is used here as a specification language even though it has been implemented. Here, we take a ConGolog program to consist of a sequence of procedure declarations and a complex action (in [De Giacomo *et al.*, 2000], nested procedures are handled). Complex actions can be constructed out of the following constructs:

a,	primitive action
φ?,	wait for condition
$\delta_1;\delta_2$,	sequence
$\delta_1 \vert \delta_2$,	nondeterministic choice of actions
δ^*,	nondeterministic iteration
$\pi v.\delta$	nondeterministic choice of argument
if φ **then** δ_1 **else** δ_2 **endIf**,	conditional
for $x \in \Sigma$ **do** δ **endFor**,	for loop
while φ **do** δ **endWhile**,	while loop
$\delta_1 \vert\vert \delta_2$,	concurrency with same priority
$\delta_1 >> \delta_2$,	concurrency with δ_1 at higher priority
$\langle x : \phi \rightarrow \delta \rangle$,	interrupt
$\beta(p)$,	procedure call.

In the above table, *a* is a primitive action; φ is a situation calculus formula with the situation argument of its fluents suppressed; δ, δ_1, and δ_2 represent complex actions; Σ is a set, *x* is a list of variables; β is a procedure name, while *p* represents the actual parameters of the procedure. Procedures are defined as **proc** β(y) δ **endProc**, where β is the name of the procedure, y represents its formal parameters, and δ is the body of the procedure, a complex action [Shapiro *et al.*, 1998].

One of the most expressive facilities of ConGolog is the interrupt construct. The interrupt fires whenever there is a binding of variables in x that makes the trigger condition φ true. The body of the interrupt, δ, is then executed. Once the execution of the body is complete, the interrupt is ready to fire again. Interrupts allow for modeling of concurrent and nondeterministic systems, reactive controllers, etc. One can easily write programs that can stop their normal activities and proceed to handling high-priority tasks when the need arises.

The semantics of ConGolog is defined in terms of *transitions* [De Giacomo *et al.*, 2000]. Transition semantics defines single steps of computation (as opposed to defining complete computations) that can be either primitive actions or tests of whether certain things hold in situations. The semantics is defined using two special predicates, $Trans(\delta,s,\delta',s')$, which means that program δ in situation s may legally execute one step of computation, ending in situation s′ with program δ' remaining, and $Final(\delta,s)$, which means that program δ may legally terminate in situation s. These predicates are characterized by axioms that define possible computations and legal terminations for all ConGolog language constructs (e.g., primitive action, concurrency operator, etc.). The overall semantics of a ConGolog program is defined by the *Do* relation, which is defined in terms of *Trans* and *Final* (see [De Giacomo *et al.*, 2000] for details). $Do(\delta,s,s')$ holds iff s' is a legal termination situation of process δ started in situation s.

2.2.4 Epistemic Feasibility

In [Lespérance, 2002] the notion of *epistemic feasibility* of CASL programs is introduced and formalized. CASL, like many other multiagent specification frameworks, does not provide a good way for ensuring that plans of the agents in the MAS are epistemically feasible, that is that the agents have enough knowledge to be able to successfully execute their plans. It is noted in [Lespérance, 2002] that in CASL, the behaviour of the system is specified as a set of concurrent processes that may or may not refer to the agents' mental states during their execution. Thus, it may be the case that an agent chosen to execute a certain process does not have enough knowledge to do it. For example (adapted from [Lespérance, 2002]), suppose there is a safe and a robot capable of dialling the safe's combination. The designer can write a program that makes the robot dial the safe's combination (whatever it is), for example, `dial(Robot1,combination(Safe1),safe1).` While this program is physically executable, it is not epistemically feasible if the robot does not know the combination. Such programs are useful if we just want to identify some possible execution traces of the system, but they do now take into consideration how the mental state of the agents affects their actions.

In CASL, like in many other specification frameworks, systems are specified from the third-person point of view. While this could have some advantages (e.g., some systems are best specified objectively), it is quite easy to write specifications that are not executable. To remedy this, [Lespérance, 2002] provides ways to guarantee that specifications are epistemically feasible for the agents in the system. The paper defines subjective plan execution (with or without lookahead), which ensures that the plan can be executed by the agent based on its own knowledge.

How can an agent know that it can execute an action? Before being able to execute an action, the agent must, for one thing, know that its preconditions are true. These kinds of requirements are known as "knowledge prerequisites of action" [Moore, 1985]. While the modeler could specify these preconditions manually, as suggested in [Lespérance, 2002], it would be better if these preconditions fell out automatically when the modeler specified that the process was to be executed subjectively by some specific agent.

2.2.4.1 Blind Subjective Execution of Processes

A new construct, **Subj***(agt, δ)*, is proposed in [Lespérance, 2002]. This means the process *δ* is to be subjectively executed by the agent *agt*, that is, in terms of its knowledge. **Subj** is formally defined in terms of transitions. The standard ConGolog axioms for *Trans* and *Final* are appropriately modified (the details are presented in [Lespérance, 2002]). With subjective execution, only if the agent knows that it can make a transition can the system make that transition. Also, if the transition involves a primitive action, this action must be executed by the agent itself. A subjectively executed program can terminate only if the executing agent knows that it may do so. So, during subjective execution all the fluents, tests, and action preconditions are evaluated with respect to the agents' knowledge state as opposed to being evaluated against the global world state.

For single agent programs without nondeterministic operators, such as π, *, |, and ||, **Subj** is an adequate formalization of epistemic feasibility since it is sufficient for the agent to know which transition to perform at each step and when it can legally terminate. On the other hand, if the program contains nondeterminism, there are possibly many transitions that are allowed at any step and the agent must choose which one to execute. Nondeterministic programs enclosed in the **Subj** construct are executed blindly with the executing agent choosing the next transaction arbitrarily. Therefore, it can end up making rather bad choices. For example, suppose that the agent knows that both actions *a* and *b* are executable and the program is **Subj***(agt, (a; False?)| b)*. Since the agent performs no lookahead, it may very well choose to execute the action *a* and be left with *False?*, which can never become true, so the agent is stuck. Therefore, to show that a nondeterministic program is subjectively (and blindly) executable by an agent, one must show that all the paths are subjectively executable and lead to successful termination. That is, if there is a path in a nondeterministic program that is subjectively executable by an agent, it is not enough to guarantee epistemic feasibility. One must make sure that all the paths are subjectively executable. In [Lespérance, 2002] this is captured by the predicate

AllDo(δ,s), which is true if all the possible executions of the program δ starting in situation s successfully terminate. That is why the new predicate is called **AllDo** — all paths lead to a successful termination of the program.

Epistemic feasibility for single-agent programs that are blindly executed as described above is formalized by the predicate **KnowHowSubj**(agt,δ,s) defined as:

$$\textbf{KnowHowSubj}(agt,\delta,s) \overset{def}{=} \textbf{AllDo}(\textbf{Subj}(agt,\delta),s)$$

This means that every subjective execution starting in s successfully terminates. If we want to generalize this to the system consisting of two agents executing in parallel, we can define the epistemic feasibility as:

$$\textbf{KnowHowSubj}(agt_1, \delta_1, agt_2, \delta_2, s) \overset{def}{=} \textbf{AllDo}(\textbf{Subj}(agt_1, \delta_1)||\textbf{Subj}(agt_2, \delta_2), s)$$

This guarantees that no matter how the two programs are executed and how they are interleaved, the execution will terminate successfully. This can be generalized further. If δ is a multiagent process with all the agents subjectively executing their programs in a blind manner (their programs are all inside the **Subj** operator), then the epistemic feasibility is defined as:

$$\textbf{KnowHowSubj}(\delta,s) \overset{def}{=} \textbf{AllDo}(\delta,s)$$

2.2.4.2 Deliberate Execution of Processes

[Lespérance, 2002] also proposes the notion of *deliberate execution* for the smart agents that can deliberate and perform lookahead. The **Delib**(agt,δ) operator is used for such programs. It is again formalized using the transition semantics of ConGolog (full details can be found in [Lespérance, 2002]). The epistemic feasibility for deliberately-executed single-agent (deterministic or nondeterministic) programs is defined as

KnowHowDelib(*agt,δ,s*). The essence of epistemic feasibility in a deliberate execution is that the system can make a transition only if the executing agent knows that it can make this transition and, in addition, it knows how to deliberately execute the remainder of program δ until the legal final situation. Therefore, before selecting an action to execute the agent must ensure that it knows how to complete the execution of the program. Since one can depend on the agent for the smart selection of its transitions, there is no need to require that all the paths through the program lead to the successful termination — it is enough to show that at least one such path exists. There is no definition of **Delib** for general multiagent processes yet. Scaling **Delib** up to the case of multiple agents is hard since the choice of actions of each agent will depend on other agents' deliberations.

3 Related Work

In this chapter, we briefly introduce Software Engineering in Section 3.1 and Requirements Engineering in Section 3.2. We talk about software agents, multiagent systems, and Agent-Oriented Software Engineering in Section 3.3, introduce a number of agent-oriented development methodologies in Section 3.4, discuss several important goal oriented requirements engineering approaches in Section 3.5, and describe two *i**-based methodologies, on which our approach is based, in Section 3.6.

3.1 Software Engineering

Creation of reliable, maintainable, etc. software systems that satisfy customer requirements is a hard task, which is steadily becoming more and more difficult. Software systems are constantly growing in size and complexity. In many cases, small teams of developers can no longer deliver new product versions on schedule because of the ever increasing number of features. As the number of developers grows, the management of the development team as well as the systematic and efficient communication and document sharing among its members becomes essential. Software development organizations now routinely use third party libraries, components and services in their projects. They also experience pressure from their customers to deliver customized and easily maintainable systems quickly and cheaply. Software Engineering studies the methods and techniques that help develop reliable, efficient, maintainable, evolvable, etc. software systems on time and on budget, thus alleviating the above-mentioned difficulties.

There are many definitions of what Software Engineering is. The definition given in [IEEE, 1990] states the following:

> Software Engineering is the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software.

Software Engineering studies the following activities that are part of the software development process: requirements engineering, design, implementation, testing, deployment, and maintenance. In this thesis, we concentrate on requirements engineering and high-level design.

3.2 Requirements engineering

The main measure of the success of software systems is the degree to which it meets its purpose. Requirements engineering is the process of discovering the purpose of software systems [Nuseibeh and Easterbrook, 2000]. Requirements engineering is defined by [Zave, 1997] as: "the branch of software engineering concerned with the real-world goals for, functions of, and constraints on software systems. It is also concerned with the relationship of these factors to precise specifications of software behaviour, and their evolution over time and across software families". The core RE activities are as follows ([Nuseibeh and Easterbrook, 2000]):

- eliciting requirements,
- modeling and analyzing requirements,
- communicating requirements,
- agreeing to requirements,
- evolving requirements.

In this thesis, we are concerned with the fist two activities of RE, namely the requirements elicitation, as well as the requirements modeling and analysis.

Requirements elicitation deals with analyzing the environment of the system-to-be, identifying the system's stakeholders — people or organizations that influence the system-to-be and are affected by it — and recognizing their needs.

Much of the RE research in the last decade has been concentrated in the area of *goal-oriented* requirements engineering. Goals are the objectives of the stakeholders, thus eliciting stakeholder goals makes the requirements engineer concentrate on the problem, rather then on finding solutions to the problem. Goal-oriented approaches like KAOS [Dardenne *et al.*, 1993] and Tropos [Castro *et al.*, 2002] provide means to systematically refine stakeholder goals to the point where they can be assigned for achievement to either the system (or one of its components) or the environment of the system-to-be.

One of the main difficulties of requirements elicitation and modeling is the imprecise, informal nature of requirements. Requirements for a system exist in a certain social context and therefore during requirements elicitation a high degree of communication, negotiation, and other interaction skills is needed. For example, various stakeholders have different, possibly (and usually) conflicting points of view on what the system-to-be should deliver. Thus, getting stakeholders to agree on requirements is an important step of the process and requires a good social and communication skills. Requirements engineers should also be very careful while gathering the requirements: the same words and phrases used by different stakeholders may mean different things for them. One way to overcome this problem is to construct a common ontology to be used by all the stakeholders. Another way is to model the environment formally. One approach for carefully describing the environment using ground terms, designations and definitions is presented in [Jackson, 1997]. A brief overview of the requirements elicitation techniques is presented in [Nuseibeh and Easterbrook, 2000].

Requirements modeling is the activity of "building abstract descriptions of the requirements that are amenable to interpretation" [Nuseibeh and Easterbrook, 2000].

Modeling facilitates in requirements elicitation by guiding it and helping the requirements engineer look at the domain systematically. Domain models allow for requirements reuse within the domain. The presence of inconsistencies in the models is indicative of conflicting and/or infeasible requirements. While informal models are analyzed by humans, formal models of system requirements allow for precise analysis by both the software tools and humans. As systems grow in size and complexity, formal analysis of requirements becomes more and more important. System requirements specifications that are expressed formally allow the designer to rigorously verify that the software system satisfies its requirements.

Requirements engineering is generally viewed as a process containing two phases. The *early requirements phase* concentrates on the analysis and modeling of the environment for the system-to-be, the organizational context, the stakeholders, their objectives and their relationships. A good analysis of the domain is extremely important for the success of the system. Understanding the needs and motivations of stakeholders and analyzing their complex social relationships helps in coming up with the correct requirements for the system-to-be. Domain models are a great way to organize the knowledge acquired during the domain analysis. Such models are reference points for the system requirements and can be used to support the evolution of requirements that stems from changes in the organizational context of the system.

The *late requirements phase* is concerned with modeling the system together with its environment. The system is embedded into the organization; the boundaries of the system and its environment are identified and adjusted, if needed; the system requirements and assumptions about the environment are identified. The boundary between the system and its environment is initially not well defined. The analyst will try to determine the best configuration of the system and its environment to reliably achieve the goals of the stakeholders. Putting too much functionality into the system could make it, for example, too complex and hard to maintain and evolve, while making too many assumptions about the environment may be unrealistic.

It is generally acknowledged that uncaught errors in requirements cost tens of times more to fix at the later stages of the software development process than if they had been caught at the requirements stage. Also, a lot of times failures of software systems have to do not with poor design or programming errors, but with the failure to properly capture the needs of the stakeholders. Thus, spending time to carefully figure out what the system is supposed to do is extremely important. On the other hand, requirements are quite volatile and supporting the evolution of the system requirements is also crucial. Creating traceability links between requirements models and design-level models helps when requirements change either during development or after the deployment of the system.

3.3 Agents, Multiagent Systems and Agent-Oriented Software Engineering

Agent-Oriented software engineering is a relatively new approach to software construction. With the advent of the internet, electronic commerce, web services, and peer-to-peer networks more and more software systems are becoming distributed, with complex communication patterns. These systems are frequently built out of components that were developed and may be operated by different companies. In this context, many researchers and practitioners find that object-oriented and even component-oriented approaches offer abstractions that are too low-level to concisely and conveniently describe such systems.

In agent-oriented software engineering, the problems are decomposed into autonomous, interacting components called agents, each of which has a goal or a set of goals to achieve. The system's objectives are achieved as these components *act together* to achieve their individual goals. Therefore, *agent interaction* is needed for the agents to achieve their individual goals and for the system to achieve its objectives. Most frequently, agents in multiagent systems represent and act on behalf of individuals, companies, departments within companies, etc. Therefore, as it is noted in [Jennings, 1999], when agents interact, there is usually some "underlying organizational context" that characterizes the relationship between the agents in the system. Multiagent systems often mimic human organizations and social relationships by involving agent coalitions and teams and by distinguishing among bosses, peers, etc. This makes multiagent systems a great match for goal-oriented requirements engineering approaches such as those based on the *i** modeling framework. The *i** models of the future system and its organizational context contain the stakeholders, the system agents, and their intentional dependencies and can be easily mapped into an analogous multiagent system with the stakeholders modeled by software agents and intentional dependencies replaced by similarly patterned agent interactions. Multiagent systems have the essential ability to adapt to the changes in their underlying social context and thus promise big savings for supporting the system evolution.

There is no universally accepted definition of software agent. One of the most popular is presented in [Wooldridge, 1997]:

> *an agent is 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*

Agents are thought of as self-contained software or (rarely) hardware systems that encapsulate some state information and are able to communicate with each other. Wooldridge and Jennings ([Wooldridge and Jennings, 1995]) define weak and strong notions of agency. Weak agency is the basic, minimal characterization of agents. Weak agents have the following properties:

- *autonomy*: agents are able to pursue their objectives without direct guidance from humans or other systems and have control over their actions and state;
- *social ability*: agents are capable of interacting with other agents using some kind of agent communication language;
- *reactivity*: agents are capable of reacting in a timely manner to changes in their environment;
- *pro-activeness*: agents exhibit goal-directed behaviour.

The above properties define the essence of being an agent. Systems not exhibiting the above properties most likely are not agents. Strong agency involves characterizing the states of the agents using mentalistic notions such as knowledge, belief, intention (e.g., BDI formalisms of [Rao and Georgeff, 1995]). Therefore, strong agents have more artificial intelligence/knowledge-based technology built into them and to a certain extent are specified formally. Describing agents in terms of their mental states allows for higher-level specifications. When matched with the appropriate agent interaction language (e.g., FIPA ACL [FIPA, 2001] or KQML [Finin *et al.*, 1997]), these specifications support reasoning about the effects of inter-agent communication without knowing the exact content of the messages (knowing the type of the message is enough to predict the type of effect it has on the mental state of the agent). This allows for more thorough analysis of multiagent systems and for prediction of changes in the agents' behaviour in response to change in the agents' environment. Equipped with appropriate reasoning tools, agents can determine the best course of action under changing circumstances, thus displaying autonomy and reactivity in the environment. They can change partners, form teams, request assistance, switch to achieving lower priority goals while higher priority ones cannot be effectively achieved, etc. Moreover, in open and dynamic systems that are well suited for agent technology, agent interactions cannot be fully specified at design time since multiagent system configurations are very fluid. Using AI agent technologies allows designers to create highly flexible and dynamic systems by deferring much of the decision making until runtime.

Agent research argues that agent orientation is the next logical step for software engineering. Jennings ([Jennings, 1999]) shows that agents are a good way of partitioning the complex problems and that agent-oriented systems are much better equipped for dealing with complex organizational relationships that exist in today's software than systems built using other paradigms, most notably the object-oriented one. Agents, unlike objects, have a much greater degree of control over which actions they execute. While objects (or, more generally, software *components*) are passive and need some sort of invocation to become active, agents "encapsulate behaviour activation (action choice)" [Jennings, 1999]. Any object can call any publicly accessible method of another object and the called object cannot refuse the invocation. This may not be appropriate for competitive environments. On the other hand, an agent can refuse to execute an action for one reason or another. This way action invocation becomes a "process of mutual consent" [Jennings, 1999]. Another advantage of agent-oriented approach is that agent interactions occur at a higher level than interactions among components and objects. Agents usually employ a speech act-based agent communication language that allows them to communicate at a semantic level as opposed to the purely syntactic-level communication employed by objects.

Agent orientation, while a very promising approach, also has some important problems [Jennings, 1999]. For example, agents heavily loaded with AI may be hard to implement efficiently and require substantial expertise in artificial intelligence. Designers must balance reactive and proactive behaviour, which is hard even for small systems (e.g., [Lapouchnian and Lespérance, 2002]). Since agents interact in flexible ways, quite frequently the pattern of interactions as well as their timing cannot be predicted. Also, generally the behaviour of the multiagent system is difficult to understand only in terms of the behaviour of its individual agents. The system's overall *emergent behaviour* can be hard to predict.

3.4 Agent-Oriented Development Methodologies

The growth of interest in agents and multiagent systems in the 1990's generated a need for software engineering methodologies designed specifically for agent-based systems. A number of agent-oriented software development methodologies were reviewed in [Iglesias *et al.*, 1998] and in [Wooldridge and Ciancarini, 2001]. Iglesias et al. note that many researchers avoid developing agent-oriented methodologies from scratch. Instead, most agent-oriented methodologies extend existing methodologies to include aspects relevant to agents. The majority of agent-oriented methodologies are based on two types of existing methodologies [Iglesias *et al.*, 1998]: object-oriented (OO) and knowledge engineering (KE).

Extending existing object-oriented approaches to support agents appears to be quite natural since there are a lot of similarities between agents and objects. Shoham [Shoham, 1993] noted that the agents can be considered as active objects (objects with a mental state that act on their own volition). Message passing, inheritance, and aggregation are used in both agent-oriented and object-oriented analysis paradigms [Iglesias *et al.*, 1998]. The popularity of object-oriented analysis and design methods and the abundance of tools for object-oriented development are other factors in the selection of object-oriented methodologies as a base for the development of new agent-oriented ones.

One example of the above approach is the methodology of Kinny et al. [Kinny *et al.*, 1996] developed at the Australian AI Institute. This approach is mainly based on object-oriented methodologies with the addition of some agent concepts. It provides *internal* and *external* models, which define a multiagent system specification. The external model is concerned with agents and relationships among them. On the other hand, the internal model describes the internals of the agents, their beliefs, desires, and intentions. The external model is divided into an *interaction* model and an *agent* model, which is in turn subdivided into an *agent class* model and an *agent instance* model. They define classes of agents and their relationships via inheritance, aggregation, and instantiation. The methodology in can be briefly described as follows:

- relevant roles are identified in the problem domain;
- the responsibilities of the roles, and the services required and provided by it are then identified;
- the goals associated with each role are identified;
- achievement plans are chosen for each goal, along with the context conditions that determine when the plan is appropriate;
- the belief structure of the system (the information required for each plan and goal) is mapped out.

The resulting model is closely related to the Procedural Reasoning System (PRS) [Ingrand *et al.*, 1992] agent architecture and thus could be easily implemented using PRS. The PRS, developed at the Stanford Research Institute, is "the first agent-based architecture based on the belief-desire-intention paradigm" [Wooldridge and Ciancarini, 2001]. The PRS was used in Australia in some of the most complex multiagent systems ever built (including an air traffic control system and an air force simulation system).

Another example of a methodology based on object-oriented concepts is *Gaia* [Wooldridge *et al.*, 2000]. This methodology aims at allowing analysts to systematically go from a requirements statement to a sufficiently detailed, directly implementable design [Wooldridge and Ciancarini, 2001]. Gaia borrows some notation as well as terminology from object-oriented analysis methods, but instead of applying them naively to agent-oriented analysis, it introduces a new set of concepts for modeling complex agent-oriented systems. Gaia encourages designers to think of building agent-based systems as a process of *organizational design* [Wooldridge and Ciancarini, 2001]. When applying Gaia, the analyst moves from more abstract concepts (e.g., roles, permissions, responsibilities, etc.) to progressively more concrete concepts (e.g., agent types and services). Abstract entities are used during conceptual analysis of the system. They do not usually have direct realizations within the system, unlike the concrete concepts, which are used during the design of the system and are typically realized in the system at runtime. Each step brings the specification closer to the final design, reducing the number of possible systems that could be implemented to satisfy the requirements specification.

The analysis stage in Gaia is aimed at understanding the system and its structure (no implementation details are introduced at this point). This understanding is captured by the system's *organization* [Wooldridge and Ciancarini, 2001]. An organization in Gaia is viewed as a collection of roles that have relationships with each other and take part in interactions. Roles are defined by their responsibilities (divided into liveness and safety properties), activities (private internal actions of a role), permissions (resources available to a role to realize its responsibilities), and protocols (ways a role can interact with other roles).

Gaia is somewhat similar to the *i** modeling framework, which is used in this thesis, since in both *i** and Gaia, organizational aspects are given a very prominent role in the analysis and development of the system-to-be. Similarly, organizations in Gaia are described using roles and their relationships, while *i** uses the concepts of actor (actors can be agents, positions, and roles, see Section 2.1.1) and intentional dependencies between actors. These dependencies, when operationalized, become patterns of interaction among the actors in the system.

There are also a number of approaches for agent-oriented software engineering that extend knowledge engineering methodologies. As noted in [Iglesias *et al.*, 1998], knowledge engineering methodologies can provide a good basis for modeling multiagent systems since agents have cognitive characteristics. By extending these methodologies, a rich experience in knowledge engineering as well as ontology libraries and problem solving methods can be reused for agent-oriented software engineering. The agent-oriented methodologies based on knowledge engineering approaches concentrate mainly on modeling knowledge acquisition and use.

Several agent-oriented methodologies extend the CommonKADS approach [Schreiber *et al.*, 1994], which can be considered a European standard in knowledge modeling [Iglesias *et al.*, 1998]. Glaser [Glaser, 1996] proposed one such extension. The approach defines the following models to capture the properties of multiagent systems:

- The *Agent model* is the main model and is used to define the agent architecture and knowledge, which is classified as social, cooperative, control, cognitive, or reactive.
- The *Expertise model* is used to describe the cognitive and reactive capabilities of agents. It models three types of knowledge: task knowledge (task decomposition knowledge described in the task model), problem solving knowledge (problem solving methods and strategies for their selection), and reactive knowledge (procedures to react to various events).
- The *Task model* describes the decomposition of tasks and assigns tasks to the agents of the system or the user.
- The *Cooperation model* describes the cooperation among the agents.
- The *System model* defines the organizational aspects of the MAS.
- The *Design model* collects all the above models to operationalize them. It also describes non-functional requirements for the system.

Other agent-oriented development methodologies include DESIRE [Brazier *et al.*, 1995], which is used for specifying composite systems and has an extensive tool support, and Cassiopeia [Collinot *et al.*, 1996], which, unlike the Gaia and many other approaches, is bottom-up in nature. A lot of research has been recently focused on adapting the Universal Modeling Language (UML) [Booch *et al.*, 1999] notation to the specification of agent-oriented systems. One of the outcomes of this research is the AgentUML notation [Odell *et al.*, 2000] (it is *not* a methodology). It is being used in some agent-oriented methodologies (e.g., Tropos [Castro *et al.*, 2002]).

3.5 Goal-Oriented Requirements Engineering Methodologies

3.5.1 KAOS

The KAOS (Knowledge Acquisition in autOmated Specification) approach [Dardenne *et al.*, 1993] is a goal-oriented requirements engineering framework supporting formal modeling and analysis of both functional and non-functional requirements. It can be used for elicitation, specification, and analysis of system goals, scenarios, and responsibility assignments. It was one of the first requirements engineering approaches that put emphasis on "high-level system goals as opposed to their operationalization into constraints" [Dardenne *et al.*, 1993]. Thus, in KAOS, one starts with the system-level goals and organizational objectives and subsequently refines them to produce low-level constraints that can be guaranteed by agents through appropriate actions.

KAOS provides a *conceptual meta-model* and requirements models are acquired as instances of this meta-model. The meta-model is represented as a graph with nodes capturing the abstractions such as goals, actions, agents (in KAOS, agents are objects that are processors for some actions; they have a choice on their behaviour and could be humans, physical devices, or programs [Dardenne *et al.*, 1993]), etc. and edges capturing the semantic links between these abstractions. A requirements acquisition process is then a particular way of traversing the meta-model graph to acquire the instances of abstractions and their relationships. A way to traverse the meta-model graph is called an *acquisition strategy*.

KAOS specifications consist of a number of complementary models. The goal model is the basis for the KAOS methodology. Goals are statements of what the composite system should achieve. Goals in KAOS are classified according to the temporal behaviour patterns required by the goal. The goals can be *achieve* goals, *cease* goals, *maintain* goals, and *avoid* goals. The KAOS goal model is a graph that shows the refinement of the high-level goals of the combined system into lower-level goals suitable for assignment to agents in the system or in the environment. Goals are refined into subgoals through AND or OR-decompositions. An AND-decomposition is used if all of the subgoals must be achieved in order to achieve the parent goal. An OR-decomposition is used if the parent goal can be achieved by achieving alternative subgoals. Examining the goal graph top-down, one can see *how* the system goals are realized, while examining the graph bottom-up, one sees *why* particular subgoals are to be achieved. Goal decomposition in KAOS is done centrally and from the designer's point of view. The *i** framework [Yu, 1995], on the other hand, allows for the subjective analysis of high-level goals from the point of view of actors that are part of the system-to-be or its environment. Another difference between KAOS and *i**-based methods (and thus between KAOS and our approach) is that the KAOS process starts at the late requirements phase by analyzing the goals of the *combined* system, while the *i**-based frameworks support both the early and the late requirements phases and allow the modeling of the environment of the system-to-be as well as the system under development separately.

The KAOS agent responsibility model shows the assignment of responsibilities for achieving goals to agents. In this model, agents can be the components of the system-to-be, existing software components, hardware devices, and humans. Goals assigned to agents are deemed sufficiently fine-grained and are not refined any further. This means that unlike the *i** modeling framework, KAOS only allows the assignment of leaf-level goals to agents. The delegation of subgoals to agents is thus the responsibility of the designer and is done at design time. In *i**, on the other hand, any goal/task can be assigned to an actor, refined within that actor (based on the actor's own reasoning), and lower-level subgoals/subtasks produced during this refinement can be further delegated to other actors or achieved/performed by the actor itself. We feel that this makes the *i** goal decomposition more flexible and more suitable for agent-oriented development. With the use of appropriate goal-supporting formal notation in conjunction with the *i** framework, goal analysis and decomposition can be performed at runtime.

The KAOS approach includes refinement strategies for goal decomposition. For example, [Darimont and van Lamsweerde, 1996] describe a number of tactics for goal refinement including *milestone-driven* (refines achievement goals by introducing some intermediate milestone assertion) and *case-driven* (splits goal achievement into cases). These strategies help the modeler to systematically refine high-level goals into more manageable ones. Moreover, agent-based refinement strategies are described in [Letier and van Lamsweerde, 2002]. In KAOS, goals can only be assigned to agents if they are capable of achieving them. Agent-based refinement tactics are designed to help in dealing with goals that cannot be realized by agents due to the lack of monitorability (the agent cannot monitor the variables that need to be evaluated in the formulation of the goal) or controllability (the agent cannot control the variables that need to be controlled to achieve the goal).

Obstacle analysis is a major component of the KAOS framework. [van Lamsweerde *et al.*, 1995] note that quite frequently initial specifications of goals, refinements, and assumptions are too ideal and are easily violated due to the unexpected behaviour of agents (software, devices, or humans). The need to anticipate exceptional circumstances is at the heart of obstacle analysis. Obstacles are duals of goals — they capture undesirable conditions. An obstacle obstructs some goal: when the obstacle becomes true, the goal may not be achieved [Letier and van Lamsweerde, 2000]. Obstacles can be refined (AND/OR-refinement) into subobstacles. Techniques for obstacle identification and avoidance are also proposed. Obstacle avoidance techniques include goal substitution (coming up with an alternative refinement of goals that does not give rise to the obstacle), agent substitution, obstacle prevention (a new goal, which requires that the obstacle be avoided, is added), and goal deidealization (reformulating the goal so that the obstacle disappears).

3.5.2 Albert II

Albert II [du Bois, 1995a; du Bois *et al.*, 1997] is a formal requirements specification language based on a real-time temporal logic [Chabot *et al.*, 1998]. The name is an acronym for Agent-oriented Language for Building and Eliciting Real-Time requirements. The main purpose of the framework is to model distributed heterogeneous real-time cooperating systems. The development of the language started in 1992. Throughout its development, the language was tested on specifications of non-trivial systems like computer-integrated manufacturing [Dubois and Petit, 1994], process control, and telecommunications systems [Wieringa and Dubois, 1998].

The underlying formal framework of Albert II is based on Albert-CORE [du Bois, 1995a], an object-oriented variant of temporal logic with actions that have been introduced as a means of solving the frame problem. Albert II specifications are centered on describing the behaviour of *agents* found in the environment and the system-to-be. This way large specifications are structured in terms of smaller agent specifications that each guarantee a part of the global behaviour of the system [du Bois, 1995b]. Albert II, like the original Albert [Dubois *et al.*, 1994] language, treats agents as specializations of objects (agents do not have intentions in Albert II). This "object-oriented" [du Bois, 1995b] approach compares favourably to the use of purely logical frameworks that results in large unstructured specifications.

Agents are autonomous entities that can perform *actions*, which may modify the internal *state* (physical state or the state of knowledge about the outside world) of the agent or external agents. Actions may have duration. In this case, the effect of the action on the state takes place at the end of the action. Actions may also be executed concurrently. Conflicting actions, actions that affect the same component of an agent's state, are not allowed. Agents perform their actions based on their obligations that are expressed in terms of *local constraints* or *cooperation constraints*. Local constraints apply to the agent itself while cooperation constraints apply to inter-agent communication in agent societies. Therefore, specifications may be considered at two levels. The agent level specifies the set of possible behaviours of individual agents without regard to the behaviours of other agents in the system. At this level, the modeler specifies which actions the agent is responsible for, the effects of these actions, and the trigger conditions for them. The society level, on the other hand, takes into account the interactions among agents. Constraints on the interactions lead to additional constraints on the behaviours of individual agents. The language supports specification of agent knowledge, which is restricted to action perception and state perception (actions and parts of the states of other agents that are visible to the agent).

Albert II specifications consist of two main components: the graphical component where the vocabulary of the specification is declared and a textual part where temporal logical formulae organized using the above-mentioned templates are used to constrain the admissible behaviours of agents. The graphical declarations for agents define the state components of agents, the actions that the agents perform, and the visibility relationships that link the agents to the other agents. These relationships specify which state components and actions of each agent are not visible from the outside and which are exported to the outside. One can specify to which agent or to which society of agents the information is exported. While these *importation* and *exportation* constraints are static properties of specifications, *perception* and *information* constraints, their dynamic counterparts, are available for more detailed specification of state and action visibility. The graphical notation supports the specification of agent societies. Both *individual* agents and *classes of agents* (i.e., roles) can be declared.

Michael Jackson [Jackson, 1995] proposed to separate requirements into *indicative* — related to the environment of the system-to-be (assumptions about the environment), and *optative* — concerned with the system itself. One of the features of Albert II is that it supports the distinction between optative and indicative requirements. Among other features of the language is its *naturalness*: the language offers the ability to map informal customer requirements statements into formal Albert II statements. The goal is to avoid introducing any new elements in the formal specification, which do not have counterparts in the informal customer statements. This is achieved by supporting operational and procedural styles of requirements specification [Dubois *et al.*, 1998]. The language also helps modelers by providing various templates to guide the elicitation and structuring of requirements.

A number of projects evaluated the use of the Albert II language in combination with the *i** framework and/or the KAOS approach to specify and analyze requirements for cooperating systems. In [Yu *et al.*, 1997], the combined use of *i** and Albert II for requirements engineering in cooperative multiagent systems is proposed and evaluated. *i** is employed for modeling the organizational aspects of the system-to-be. The modeling notation helps in generating and evaluating organizational alternatives, while Albert II is used to produce formal requirements specifications for the system. The authors stress the iterative nature of the requirements engineering process where the detailed elaboration of functional requirements using Albert II may reveal unresolved organizational issues, resulting in further use of *i** to refine the organizational model. A banking system is used as a case study to evaluate the approach.

Another approach, which is described in [Dubois *et al.*, 1998], proposes the use of KAOS, Timed Automata [Merritt *et al.*, 1991], and Albert II together in a requirements engineering framework. In this approach, the authors consider three distinct activities in requirements engineering: the modeling of goals associated with the introduction of the new system into an organization, the modeling of software requirements for the system solving the organizational goals, and the modeling of the internals of the software system. The approach uses three formal languages for each of the three above activities: KAOS is used for reasoning about system goals, Albert II is used to specify the requirements for the software system, while Timed Automata are used for the specification of system internals. The three resultant models are linked at a high level using the *i** modeling framework, which is used to model the organizational issues and rationale behind the introduction of the new system and the particular system configuration. Thus, the *i** model is used to link together the three formal models by creating a high-level intentional model that captures the organizational perspective on the new system. A small process control case study is used to illustrate the approach.

Bissener [Bissener, 1997] proposed a requirements engineering methodology that combined the *i** and Albert II frameworks. He attempts to provide an approach that deals with organizational issues and can effectively model and analyze both functional and non-functional requirements. Here, the *i** modeling framework is used to model and reason about organizational issues and non-functional requirements, while Albert II is used for system specification. The process consists of three main steps: the modeling of the domain and the identification of system objectives, the introduction of the system that achieves the identified objectives, and the specification of the system's internals.

Albert II specifications are completely declarative. On the other hand, the CASL [Shapiro and Lespérance, 2001] specification language used in this thesis, while supporting the declarative specification of agent goals, knowledge, actions and their effects, is process-oriented and is used to specify the behaviour of agents procedurally. The account of knowledge in CASL is more general than in Albert II, where one can only specify action and state perception. Therefore, CASL can be used to model rich agent interactions, to analyze the epistemic feasibility of agent plans [Lespérance, 2002], etc.

3.6 *i**-based Goal-Oriented Development Methodologies

In this section we introduce the Tropos methodology [Castro *et al.*, 2002] and the *i**-ConGolog framework [Wang, 2001], which both form the basis of our own approach.

3.6.1 Tropos

Tropos is a requirements-driven agent-oriented software development methodology. One of the goals of the methodology is to reduce the mismatch between the concepts used to describe the operational environment of information systems and the concepts used to describe the architecture and high-level design of these systems [Castro *et al.*, 2002]. While the environment of the system-under-development is described in terms of stakeholders, responsibilities, objectives, and so on, high-level descriptions of systems typically use the notions of modules, interfaces, objects, etc. The quality of the systems developed using many popular software development methods suffers from this mismatch, which is due to the fact that usually development methodologies are driven by the dominant programming paradigm of the day. For example, the current dominant approach to programming is object orientation and object-oriented analysis and design is the most popular type of software development methods. While using the same concepts to align the design process with the requirements analysis process is a good idea, the authors of the Tropos approach (e.g., [Castro *et al.*, 2002]) stress that the development methodology should be based on concepts from requirement engineering and not on implementation concepts. The reason for this is the growing realization that the requirements analysis phase of software development is crucial to the success of software systems. It is the phase where "technical considerations have to be balanced against social and organizational ones and where the operational environment of the system is modeled" [Castro *et al.*, 2002]. Therefore, the Tropos approach is requirements-driven, that is, based on the concepts from early requirements analysis. The approach adopts the concepts used in the *i** modeling framework [Yu, 1995], such as *actor* and *intentional dependency*, and extensively uses *i** diagrams for modeling.

Below we present an outline of the phases of the Tropos methodology adapted from [Castro *et al.*, 2002]:

1. *Acquisition of Early Requirements*. In this phase, the environment of the system-to-be is analyzed. The stakeholders are identified, as are their goals, and the intentional dependencies among them. The output of this phase is two models:
    a) The SD model that captures the stakeholders, their goals, and their dependencies.
    b) The SR model that gives a better understanding of the processes in the organization.
2. *Definition of Late Requirements*. In this phase, the new system is introduced in the diagrams and possibilities for the reconfiguration of the intentional dependencies are proposed and analyzed. The output of this phase is the revised models:
    a) An actor to represent the new system is included in the original SD diagram with its dependencies.
    b) Means-ends analysis is done on the new system actor and a revised SR diagram is produced. Based on this analysis, the system actor can be decomposed into multiple actors. Both steps in this phase can be iteratively repeated.
3. *Architectural Design*. In this phase, the high-level architecture of the system is produced. The methodology makes use of the NFR framework [Chung *et al.*, 2000] and a repository of organizational architectural styles [Kolp and Mylopoulos, 2001] to select the most suitable (from the organizational point of view) architecture for the system (see Section 5.4.1 for more information). This phase involves the following:

a) Selecting the architectural style based on the desired quality attributes such as security, modularity, etc. At this point, the NFR diagram identifying the rationale for the selected alternative is produced.
b) New system actors, dependencies, sub-actors and sub-dependencies may be introduced if needed. The SD and SR models are revised.
c) Roles and positions are assigned to the agents.

4. *Detailed Design*. In this phase, one uses various types of UML [Rumbaugh *et al.*, 1999] diagrams modified to accommodate Tropos concepts within UML [Mylopoulos *et al.*, 2001b] and possibly AUML [Odell *et al.*, 2000]. This involves:
   a) Producing class diagrams based on the SD and SR models.
   b) Producing sequence and collaboration diagrams depicting inter-agent interactions.
   c) Developing state diagrams describing both inter- and intra-agent dynamics.
5. *Implementation*. In this phase, an appropriate development platform (possibly agent-oriented) is used to produce the implementation of the system.

The approach proposed in this thesis can be easily integrated into the Tropos methodology and offers new modeling, analysis, and verification capabilities to the Tropos framework. In Chapter 5, we present an agent-oriented requirements engineering methodology that is based on Tropos and makes use of the iASR diagrams (see Chapter 4), CASL models, and the CASLve verification tool [Shapiro *et al.*, 2002]. It covers the first three steps of the Tropos approach (early and late requirements and architectural design). The later steps can be performed with agent-oriented or conventional design and implementation methods.

Formal Tropos [Fuxman *et al.*, 2001b] is an addition to the Tropos framework designed to provide a formal analysis tool for the early requirements specification phase. During the early requirements analysis phase the analysts concentrate on modeling the environment of the system-to-be (the domain): the stakeholders, their goals and their relationships. The authors note that most formal approaches are designed to be used later in the software development process and attempts to apply formal techniques at the early requirements stage are often hindered by a concept mismatch between the constructs of the formal approaches and the notions used during the early requirements analysis. Formal Tropos bridges this gap by using the concepts of *i** [Yu, 1995] such as actors, goal, and dependencies, while also adding a KAOS-inspired rich temporal specification language. The Formal Tropos language describes the relevant domain objects and their relationships. The description of objects has two layers: the outer layer defines the structure of instances and their attributes, while the inner layer consists of constraints on the lifetime of the object specified using a linear-time temporal logic. The *i**'s Strategic Dependency (SD) diagrams used during the early requirements analysis can be mapped into the Formal Tropos language. The latter can, then, be mapped into the input language of the NuSMV model checker [Cimatti *et al.*, 1999] for analysis. Thus, early requirements specifications expressed using SD models can be checked for consistency and their properties can be validated.

The Formal Tropos approach is complementary to the method proposed in this thesis. Formal Tropos allows for the formal analysis of early requirements specifications expressed using Strategic Dependency models. Capturing all the properties and only the properties of the domain can be a difficult task. Formal models of the domain along with the appropriate analysis tools can help in identifying inconsistent, missing, etc. properties of the system's environment. Our approach, which combines the *i** modeling framework with the CASL specification language [Shapiro and Lespérance, 2001], is geared more towards late requirements (and possibly high-level design): rather than checking whether the environment is correctly modeled, we use our approach to analyze whether the system together with its environment achieves its goals.

3.6.2 The *i**-ConGolog Approach

An approach to requirements engineering that combines the use of the *i** framework with ConGolog agent programming language [De Giacomo *et al.*, 2000] for requirements analysis is proposed in [Wang, 2001]. ConGolog is a formal process specification and agent programming language and is also a major part of Cognitive Agents Specification Language (CASL) [Shapiro and Lespérance, 2001], where it is used to specify the behaviour of agents. In Wang's approach, the *i** framework is used to model the environment of the system-to-be, analyze the dependencies among the actors in the environment, explore alternative system configurations and the rationale behind agent processes and design choices, while ConGolog is used to formally specify and analyze agent behaviour described informally in *i** [Wang and Lespérance, 2001]. Since fully developed ConGolog specifications are executable, ConGolog system specification can be validated by simulation. Unexpected system behaviour during execution will be indicative of problems with the model such as incomplete or conflicting requirements, etc. The goal of the approach is to devise a method for the analysis and validation of requirements models represented in *i** with ConGolog.

The main difficulty in relating *i** models and ConGolog specifications is that the level of detail and precision provided by *i** is simply not sufficient to derive a corresponding ConGolog specification from an *i** model. ConGolog models suitable for automated analysis need to be complete and precise, while *i** models, intended for informal analysis by requirements engineers are often incomplete and are quite imprecise. For example, goal and task decompositions used in *i**'s Strategic Rationale diagrams (SR) (see Section 2.1.4 for details) do not specify in which order the subgoals/subtasks must be achieved/executed and whether the subtasks or subgoals can be executed/achieved in parallel.

To bridge this gap, a set of *annotations* to the SR diagrams is introduced. These annotations allow the modeler to add the necessary precision to the diagrams. Two types of annotations are introduced: composition annotations and link annotations. Composition annotations are applied to goal/task decompositions in SR diagrams. They specify how the subgoals/subtasks are to achieved/executed — sequentially, in parallel, with different priorities, or non-deterministically. The link annotations, on the other hand, are applied to individual subgoals/subtasks. They specify under what circumstances or how many times they are to be achieved/executed. The resulting diagrams are called the *Annotated SR* (ASR) diagrams. These diagrams are detailed enough to be *mapped* into the corresponding ConGolog models. To support requirements traceability, a mapping between the ASR diagrams and ConGolog models must be defined by the modeler. Using this mapping, it is easy to identify which parts of the *i** model are related to which parts of the ConGolog model and vice versa. In order to "ensure that the mapping respects the semantics of both frameworks" [Wang and Lespérance, 2001] a set of mapping rules is defined, which ensures that the *i** and ConGolog models are closely aligned.

The mapping rules state how each type of node and link of ASR diagrams is to be mapped into ConGolog. In a nutshell, actor nodes are mapped into ConGolog procedures specifying the behaviour of the actor, task nodes are mapped into procedures, and goal nodes are mapped into a pair consisting of a ConGolog fluent (a predicate that changes from situation to situation, see Section 2.2.1 for details) that represents the desired state of affairs for the actor and a procedure that encodes the means for achieving this goal. Leaf-level task nodes are mapped into procedures or primitive actions that represent the task. SR diagram links are also mapped into ConGolog. For example, a portion of an SR diagram with a task node decomposed into a number of subgoals/subtasks is going to be mapped into a ConGolog procedure. The composition and link annotation are reflected by the use of the corresponding ConGolog operators (see [Wang, 2001; Wang and Lespérance, 2001] for details) for composing the ConGolog procedures for the subtasks together. In this way the procedure for accomplishing the parent task captures the meaning of the ASR diagram. Means-ends links are handled similarly — the achievement procedure for the goal being decomposed must involve the constructs specified by the composition and link annotations, which are used in the ASR goal decomposition. We apply the same approach to Intentional ASR diagrams for mapping task and goal decompositions in into CASL specifications in this thesis (see Sections 4.2.3 and 4.3.9 respectively).

Intentional dependencies present in ASR diagrams are not present per se in the ConGolog model. They have to be *operationalized* in ASR diagrams: the modeler must specify the tasks that the depender and the dependee must carry out in order to ensure that the dependency is fulfilled (e.g., requests, communication protocol, etc.). The framework provides no formalization for softgoals, which are used to model non-functional requirements in *i**. Softgoals help in the evaluation and selection of the best process alternatives and then are either approximated by some hard goals, or are abstracted out of the model before the mapping takes place.

Once the ASR diagram is mapped into the corresponding ConGolog model, the ConGolog code can be used to animate the model. Using the ConGolog interpreter, one can run this high-level model of the system on some sample environment/agent parameters and determine if the behaviour of the program corresponds to the expected behaviour of the system-to-be. If discrepancies are found, they can be analyzed and appropriate changes can be made to the ConGolog model and the original ASR diagram. Because of the tight mapping between ASR diagrams and ConGolog models, it is easy to find parts of the ASR diagram that are related to specific parts of the ConGolog program and vice versa.

The approach proposed in this thesis is based on the same idea of mapping appropriately annotated SR diagrams into a formal language for analysis, validation, and verification. The key difference is that in our work, we use a more powerful formal language for the analysis of *i** models. The Cognitive Agents Specification Language is built on top of ConGolog adding support for reasoning about agents' mental states, goals and knowledge. While the *i**-ConGolog approach handles goals purely procedurally, goals and knowledge in our framework are handled declaratively, thus allowing for formal reasoning about them. Therefore, formal analysis of agent goals, goal delegation, agent knowledge, and inter-agent communication is possible in our framework.

4 Modeling and Formalism

In this chapter we will discuss the approach that we are taking to integrate the *i** modeling notation with the CASL agent specification/programming language. First, we describe the Intentional Annotated Strategic Rationale diagrams (Section 4.1), an intermediate notation between *i** and CASL. Then we discuss the mapping to CASL of the basic diagram elements such as tasks, task decompositions, annotations, agents, and roles (Section 4.2). We later introduce the new concepts of goal (Section 4.3), knowledge (Section 4.4), and capability (Section 4.5) along with their mapping into CASL.

4.1 Intentional Annotated Strategic Rationale Diagrams

*i** is a modelling framework that provides a diagrammatic notation for representing the intentional aspects of the system through inter-actor dependencies. *i** diagrams are a tool for modelling the intentional aspects of systems and organizations. It could be used for strategic analysis and the exploration of alternative system models and designs at the requirements engineering stage of the software engineering process. We can represent the motivations and intentions of the stakeholders, give a high-level view of the processes in the system, and represent inter-agent dependencies and (possibly alternative) ways to provide services to other agents. *i** is an intuitive and easily understandable informal notation that can be very beneficial in early requirements engineering and systems modeling.

*i** is a diagrammatic analytical tool and as such allows for incompleteness and impreciseness. On the other hand, CASL is a precise language that, while allowing for non-determinism, concurrency and incomplete information, requires the designer to specify the agents' behaviour with a much greater level of precision. To allow the two approaches to be used together we need an intermediate notation that provide a smooth and semantically correct transition from *i** diagrams to CASL specifications. While SD diagrams are more high-level and are mainly used as a tool for strategic analysis, SR diagrams are much more detailed models of the system. In this sense, SR diagrams are closer to CASL specifications. We therefore will use the SR diagrams as a basis for our intermediate notation. *Annotations* will be added to SR diagrams. Annotated SR diagrams were introduced in [Wang, 2001] for use in the combined *i**-ConGolog approach and we believe the same idea can be successfully applied in our work. Annotations will provide a way for adding the CASL-needed details to the SR diagrams. In order to simplify the mapping process and make the semantics of the ASR diagrams clearer, we will introduce Intentional Annotated SR (iASR) diagrams, which are different from the ASR diagrams of [Wang, 2001] in that they streamline the use of goals in the ASR diagrams, add new link annotations to the set of annotations proposed in [Wang, 2001], and add the support for the new capability nodes. These modifications and extensions are described in Section 4.3. Coupled with methods that provide a mapping from iASR diagram elements (e.g., task nodes) to CASL specifications, the iASR diagrams allow the designer to gradually move from high-level *i** models to more precise and design-oriented CASL specifications. These specifications can then be formally analyzed. CASL programs can also be used in simulations.

4.1.1 Annotations

To add the CASL-required precision to the *i** process specifications we will use SR diagram *annotations*, which are textual constraints on the Strategic Rationale diagrams. The annotations allow the modeler to add details to task and goal decompositions and to specify the applicability conditions for the alternative ways of achieving goals, performing tasks, etc. There are two types of annotations: composition annotations and link annotations. Composition annotations specify how subgoals/subtasks with the same parent goal/task are to be composed together to achieve the parent goal or task. Link annotations are assigned to each subgoal/subtask and describe how this subgoal/subtask is to be performed and under which conditions. There are certain differences in the way goals and tasks are used in iASR diagrams. For example, we require that the link annotations accompanying goal nodes be interrupts or guards. We will go over the restrictions on the goal nodes and the reasons behind those restrictions later in this chapter.

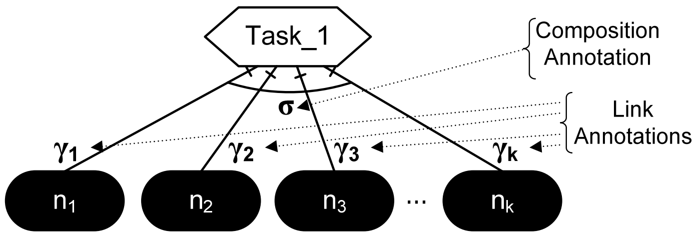


Figure 4.1. The composition and link annotations

Figure 4.1 illustrates the use of composition and link annotations. The task `Task_1` is decomposed into subtasks/subgoals n_1 through n_k (filled ellipses denote nodes that are either goals or tasks) with the link annotations γ_1 through γ_k. The subgoals/subtasks are composed using the composition annotation σ.

4.1.2 Composition Annotations

There are four types of composition annotations: sequence (“;”), concurrency (“||”), prioritized concurrency (“>>”), and alternative (“|”).

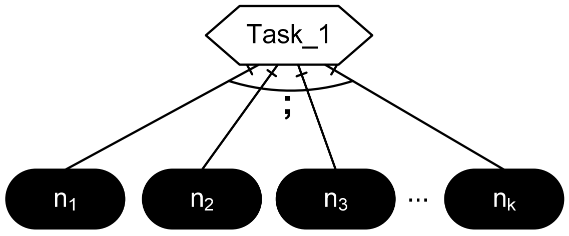


Figure 4.2. The sequence annotation

The sequence annotation (Figure 4.2) is the default composition annotation and can be omitted. Here, the task `Task_1` is decomposed into subgoals/subtasks n_1 through n_k. The sequence annotations means that the subtasks will be performed in sequence from left to right, so in the case of Figure 4.2 the subgoal/subtask n_1 will be performed first, followed by n_2 and so on. The sequence annotation corresponds to the sequence operator in CASL (";").

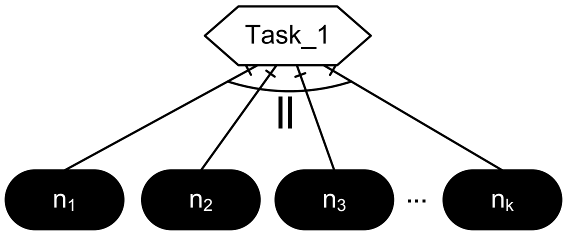


Figure 4.3. The concurrency annotation.

The concurrency annotation "||" (Figure 4.3) specifies that the subtasks/subgoals that the task `Task_1` is decomposed into are to be executed concurrently in order to perform `Task_1`. The concurrency annotation corresponds to the concurrency operator in CASL ("||").

The prioritized concurrency annotation ">>" means that the subgoals/subtasks are to be executed concurrently with priority decreasing from left to right. If we were to replace the concurrency annotation in Figure 4.3 with a prioritized concurrency annotation, then the subtasks/subgoals n_1 through n_k will be executed concurrently with n_1 having the top priority, n_2 having lower priority than n_1, but higher than n_3 and so on. Thus, n_2 is able to execute only if n_1 is blocked or finished executing, n_3 can execute only if n_2 is blocked or finished and so on. The subgoal/subtask n_k has the lowest priority, so it will be executed only when the other subgoals/subtasks are blocked or done executing. The prioritized concurrency composition annotation corresponds to the prioritized concurrency operator in CASL (">>").

The alternative annotation "|" is used to specify alternative ways of accomplishing a task. Any of the subtasks/subgoals that the super-task is decomposed into can be selected to accomplish the super-task. The alternative composition annotation corresponds to the nondeterministic choice operator in CASL ("|").

4.1.3 Link Annotations

Link annotations allow the modeler to present an even more detailed view of the process. A link annotation is applied to a single decomposition link and specifies under which conditions the subtask or subgoals is performed and whether and how it should be repeated. There are six types of link annotations: the *while* loop annotation (`while(condition)`), the *for* loop annotation (`for(variable,valueList)`), the pick annotation (π`(variableList,condition)`) , i.e., non-deterministically pick a value for the variables in `variableList` that satisfies `condition` and do the subtask/subgoal for these bindings, the *if* annotation (`if(condition)`), the interrupt annotation (`whenever(variableList,condition,cancelCondition)`), which adds a cancellation condition to the version used in [Wang, 2001], and guard annotation (`guard(condition)`) that blocks the execution of a subtask until a certain condition is true. Goal nodes can only be linked with the interrupt or guard annotations.

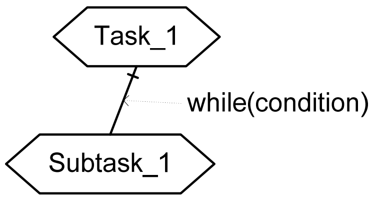


Figure 4.4. The *While* loop link annotation.

The *while* loop link annotation (Figure 4.4) is used to specify that the subtask of the link is meant to be executed repeatedly while `condition` is true in order to accomplish the super-task. The condition is tested before each iteration of the loop and the loop terminates when the condition becomes false. This annotation corresponds to the *while* loop construct in CASL.

Suppose that the *while* loop link annotation is replaced with the *for* loop link annotation in Figure 4.4. The *for* loop link annotation `for(variable,valueList)` is used to specify that `Subtask_1` has to be performed for every value of `valueList` sequentially from left to right. The subtask could be parameterized over `variable` so that `Subtask_1` is performed for every element of `valueList` (e.g., to inform every meeting participant of something) or the loop can be used only to execute the subtask a certain number of times. This annotation corresponds to the *for* loop in CASL.

The pick link annotation `π(variableList,condition)` applied to the task decomposition link in Figure 4.4 specifies that the subtask `Subtask_1` must be executed for some binding of variables from `variableList` that satisfies `condition`. This annotation corresponds to the π (the non-deterministic argument choice operator) in CASL. The *if* link annotation `if(condition)`, if applied to the task decomposition link in Figure 4.4, indicates that `Subtask_1` is to be executed only if `condition` is true. This link annotation corresponds to the conditional operator in CASL.

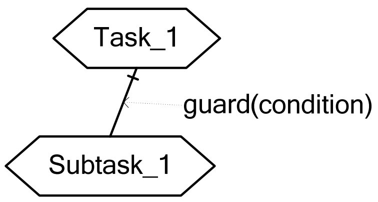


Figure 4.5a. The guard annotation used with a task.

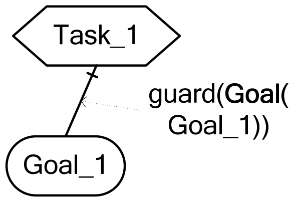


Figure 4.5b. The guard annotation used with a goal.

The guard annotation is similar to an interrupt, which fires just once. There is no guard operator in CASL as defined in [Shapiro and Lespérance, 2001], so we introduce one in Section 4.2.2. Figure 4.5 illustrates the use of the guard annotation with goals and tasks. When used with the task node `Subtask_1` (Figure 4.5a) the guard link annotation makes sure that the task is to be executed only when `condition` becomes true. The difference between the guard annotation used with a task node and the *if* annotation is that in case of the *if* annotation, if `condition` is false, `Subtask_1` is skipped, while in case of the guard annotation, when `condition` is false, the execution is blocked until `condition` becomes true.

The acquisition of goals in CASL is done declaratively (e.g., through the successor state axiom for goals and the `request/commit` actions). To monitor for newly acquired intentional dependency-based goals designers usually (but not always) employ interrupts (see Section 4.3.9.3). On the other hand, self-acquired goals are mostly used with guard annotations (see Section 4.3.8). For example, when used with the goal node `Goal_1` (Figure 4.5b), the guard annotation blocks the execution of the program until the goal is in the mental state of the agent. Thus, the interrupt and guard link annotations are used with both subtasks and subgoals (see Figures 4.5 and 4.6).

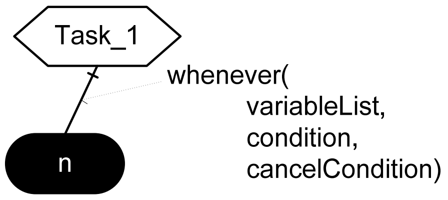


Figure 4.6. The interrupt link annotation.

The interrupt link annotation (Figure 4.6) specifies that the subtask/subgoal `n` is activated whenever `condition` becomes true for some binding of the variables on the list `variableList` provided the cancellation condition `cancelCondition` is false. The cancellation condition used here is a new feature of the interrupt annotations over [Wang, 2001]. When the cancellation condition is true, the interrupt stops firing even though there might be a binding of variables in `variableList` that makes `condition` true.

To illustrate the expressivity of annotations, let us look here (Figures 4.7a and 4.7b) at a simple SR diagram fragment. The diagram is presented with and without the annotations.

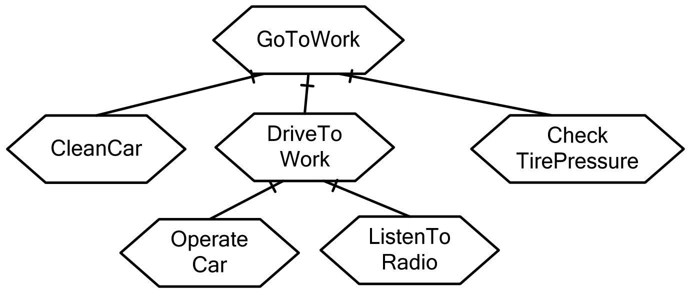


Figure 4.7a. An example diagram without annotations.

The first version of the diagram (Figure 4.7a) is a regular SR diagram. It allows the modeling of agents' internals, but does not provide the level of detail required to map the diagram into CASL. It is not clear from this diagram how the subtasks are related and under what conditions they are executed.

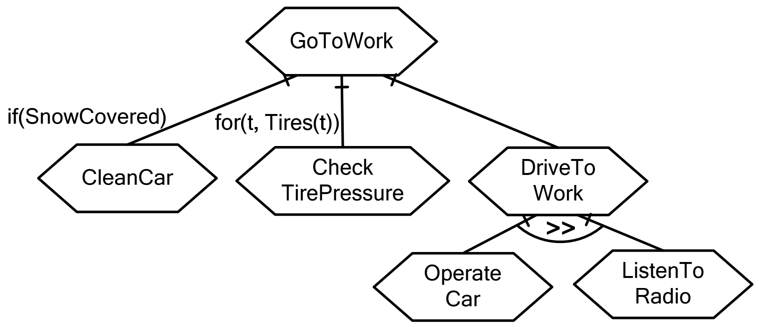


Figure 4.7b. An example diagram with annotations.

With the use of annotations we can present much more information in SR diagrams (Figure 4.7b). The diagram corrects the sequencing of subtasks (the default sequence annotation is used here): tire pressure is checked before the car is driven to work. We can now see that the `CleanCar` task has to be executed only if the car is covered with snow and that the pressure has to be checked for all the tires. Most important, however, is the fact that we now can clarify the relationship between the subtasks `OperateCar` and

`ListedToRadio`, which are composed to achieve the task `DriveToWork`: they are done concurrently and `OperateCar` has a higher priority than `ListenToRadio`. As well, the tasks `CleanCar`, `CheckTirePressure`, and `DriveToWork` are done in sequence, the default composition annotation. This detailed version of the diagram is much easier to map into CASL, but regardless of whether we are mapping iASR diagrams into CASL or not, the annotations provide us with a precision that is quite beneficial for the *i**-based analysis.

4.1.4 Obtaining Intentional Annotated SR Diagrams

In order to get an iASR diagram that is easily mappable to the CASL formalisms, the designer needs to produce a much more precise SR diagram with (in addition to the presence of the required annotations) softgoals operationalized or removed, and inter-agent interactions specified in details. Also, resource dependencies will be represented through goal or task dependencies. A resource dependency either is converted to a goal dependency with the goal being the supply of the required resource or to a task dependency, which executes the task that provides the resource. We will discuss the process of converting a Strategic Rationale diagram into an Intentional Annotated SR diagram thoroughly in this chapter as well as in Chapter 5.

4.2 Mapping ASR Diagrams

iASR diagrams are mapped into CASL models by specifying a mapping **m** for every diagram element. Mapping rules are defined for each class of iASR diagram elements. These are constraints on the mapping process that ensure it is consistent with the semantics of *i** and CASL. We will now describe the mapping rules for link annotations, agents, roles, and tasks. Later in this chapter, we will introduce the new notions of goal, knowledge, and agent capability and provide mapping rules for them.

4.2.1 Mapping Task Nodes

The tasks that are leaf nodes (without any subtasks) of the Intentional Annotated Strategic Rationale diagrams are generally mapped into CASL procedures or primitive actions. These tasks are routines that are simple enough to be represented atomically in iASR diagrams, without further decompositions or analysis.

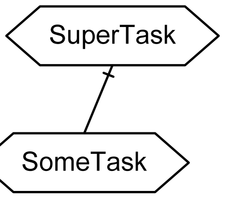


Figure 4.8. The task node `SomeTask` is a leaf node.

Figure 4.8 shows a simple iASR diagram with a task `SomeTask` being a leaf node. The mapping **m** will map this task into either a primitive action or a CASL procedure. Figure 4.9 displays the two possible CASL mappings for `SomeTask`.

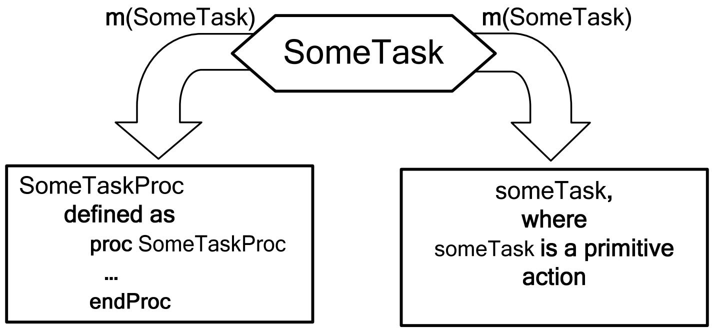


Figure 4.9. The two possible CASL mappings for `someTask`.

4.2.2 Mapping Link Annotations

Link annotations that accompany decomposition links, which connect super-tasks with subtasks, are mapped into the corresponding CASL operators. These operators are then applied to the mapping of the subtasks since link annotations specify under what conditions and in what manner the subtasks are performed.

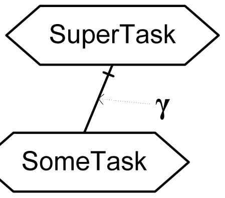


Figure 4.10. The γ link annotation applied to the link connecting `SuperTask` with `SomeTask`.

In Figure 4.10, we have a decomposition link with the link annotation γ connecting `SuperTask` with its subtask called `SomeTask`. As explained above, the CASL mapping for γ will be applied to the mapping of `SomeTask`. Here, **m** is a mapping function from iASR diagram elements into CASL formalisms. **m**(γ) is then the mapping from the link annotation γ into the corresponding CASL operator. It gives us the function to be applied to the mapping of `SomeTask`:

```
m(γ)(m(SomeTask))
```

Conditions that appear in some link annotations (e.g., the *if* annotation) are mapped into CASL formulae.

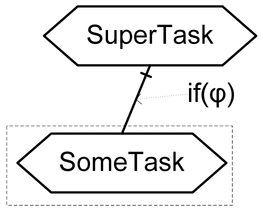


Figure 4.11. An example subtask with an *if* link annotation.

For example, the mapping of the area inside the box on the diagram fragment above (Figure 4.11) will be:

```
m(if(φ))(m(SomeTask))
```

The *if* annotation contains condition φ that has to be mapped into a formula. We therefore expand the mapping for the annotation to get the following:

```
if m(φ) then m(SomeTask) endIf
```

By applying the mapping function **m** to the remaining elements, we would ultimately get a CASL expression.

Suppose that the γ link annotation in Figure 4.10 is `guard(φ)`. For the task node `SomeTask` and the associated link annotation `guard(φ)` we get the following mapping formula:

```
m(guard(φ))(m(SomeTask)),
```

which then, based on the definition of `guard(φ,δ)` (see below) transforms into

```
guard m(φ) do
      m(SomeTask)
endGuard
```

We define the new guard operator **guard**`(φ)` **do** δ **endGuard** as follows:

```
if φ then δ else False? endIf
```

The above conditional operator executes the program δ when the condition φ holds. If the condition is false, the *else* branch of the operator is chosen, but it always blocks since *False* will never become true. Therefore **guard** φ **do** δ **endGuard** makes the program block until the condition φ becomes true and then executes the program δ. This is similar to having an interrupt that fires just once. While the semantics of the guard operator can be expressed using interrupts with the cancellation conditions that make the interrupts fire only once, using the guard operator is more convenient for the analyst.

Now suppose that the γ link annotation in Figure 4.10 is `π(variableList,condition)`. Here, the agent must nondeterministically pick the list of arguments that satisfy `condition` and then execute the procedure for the task node `SomeTask`. The CASL code corresponding to this diagram will then be:

```
π variableList. m(condition)(variableList)?; m(SomeTask)
```

4.2.3 Mapping Task Decompositions and Composition Annotations

Task decompositions are abundant in *i** diagrams and are one of the main instruments in the modeling approach. The subgoals/subtasks that are linked to a super-task with the decomposition links are subcomponents of the super-task and have to be combined to perform this parent task.

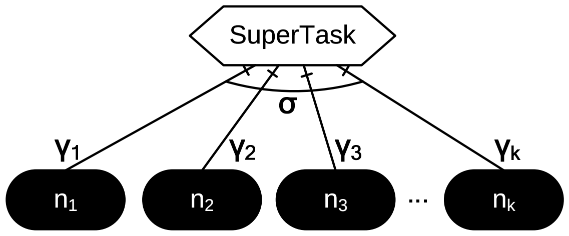


Figure 4.12. A super-task that is decomposed into *k* subtasks/subgoals.

The code for the CASL procedure that the super-task is mapped into has to include the code for all the subgoals/subtasks appropriately combined to perform the super-task. *i** is not strict in requiring that the decomposition of a super-task into subgoals/subtasks be complete — there may be other ways of performing the super-task (other subtask/subgoal decompositions) that are not presented in the SR diagram. In addition, there may be some subtasks that are necessary, but that are not represented. Since CASL, on the other hand, if process-oriented language, it assumes that every way of achieving a goal or performing a task is specified in its models. In our approach we will make a completeness assumption — we assume that all the possible alternative ways of achieving goals and performing tasks that are of interest to the designer are represented in SR diagrams *before* they are converted into iASR diagrams.

In Figure 4.12, we have `SuperTask` being decomposed into k subtasks/subgoals. Each decomposition link connecting `SuperTask` with subtask/subgoal n_i is accompanied by a link annotation γ_i. The composition annotation σ is applied to the subtasks. Complex tasks, such as `SuperTask` of Figure 4.12 will be mapped into CASL procedures according to the following generic rule:

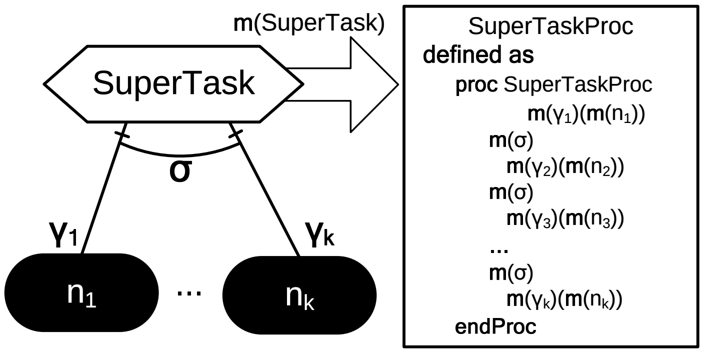


Figure 4.13. The CASL mapping of a complex task decomposition.

The complex task `SuperTask` is mapped into a CASL procedure called `SuperTaskProc`. To obtain the code for that procedure we recursively apply our mapping **m** to the subtasks/subgoals that `SuperTask` is decomposed into as well as to the composition and link annotations that are present in the decomposition. Intuitively, CASL operators matching the composition annotations are used to join the procedure calls (or primitive actions), which correspond to the subtasks/subgoals and are wrapped by the CASL operators corresponding to the appropriate link annotations. If subtasks/subgoals are further decomposed into other subtasks/subgoals, we recursively repeat the mapping process for those subtasks/subgoals thereby moving down the decomposition tree.

We now present a small example illustrating the mapping of complex tasks (Figure 4.14):

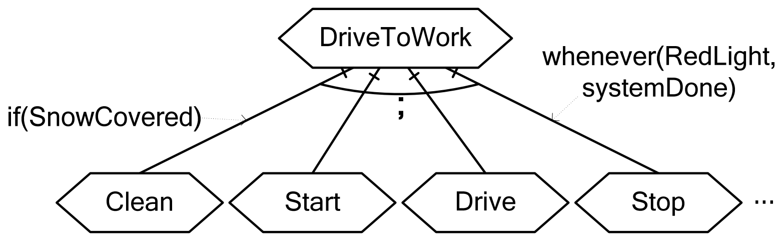


Figure 4.14. An example of a complex task.

The task in the example is `DriveToWork`. It is mapped into the following procedure:

```
proc DriveToWorkProc
      if m(SnowCovered) then m(Clean) endIf;
      m(Start);
      m(Drive);
      <m(RedLight) → m(Stop) until SystemDone>
      ...
endProc
```

`Start`, `Drive` and `Stop` will be mapped into CASL procedures, while `SnowCovered` and `RedLight` will be mapped into fluents or formulae. `SystemDone` is a special condition that becomes true once the system is terminated. The use of this condition in interrupts indicates that these interrupts will not stop firing while the system is running.

4.2.4 Mapping Agents, Roles, and Positions

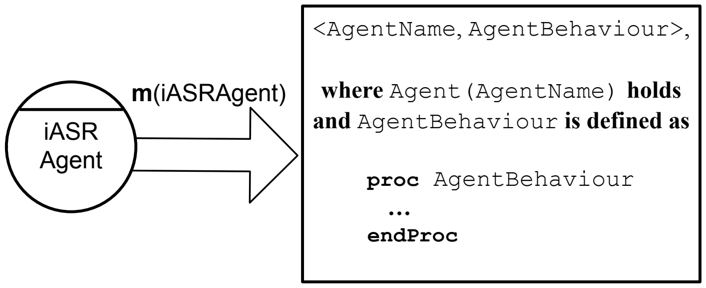
Agent, role and position nodes are used in the iASR diagrams to encapsulate the behaviour(s) associated with the corresponding actors. Agent nodes contain the behaviour specifications of concrete agents in the system and are mapped into a CASL agent name — this name has to be included in the set of agents (we assume that `IsAgent(AgentName)` holds) and a CASL procedure, `AgentBehaviour`, that defines the behaviour of the agent:

```
m(iASRAgent) = <AgentName,AgentBehaviour>
```

The notation below will be used to access the name and the procedure for the agents:

```
m(iASRAgent).name = AgentName
m(iASRAgent).behaviour = AgentBehaviour
```

Figure 4.15 illustrates the mapping:



`<AgentName, AgentBehaviour>`,

where `Agent(AgentName)` **holds and** `AgentBehaviour` **is defined as**

```
proc AgentBehaviour
  ...
endProc
```

Figure 4.15. The mapping of agent nodes.

A role specifies a certain behaviour that could be exhibited by many different agents in the system. We will map an iASR role node into a CASL procedure that models the behaviour associated with that role. In *i**, a role can be played by various agents. Since roles describe the behaviour of many different agents, procedures corresponding to iASR roles will most likely have the agent playing that role as a parameter. On the other hand, agent procedures will have the name of the agent built into them.

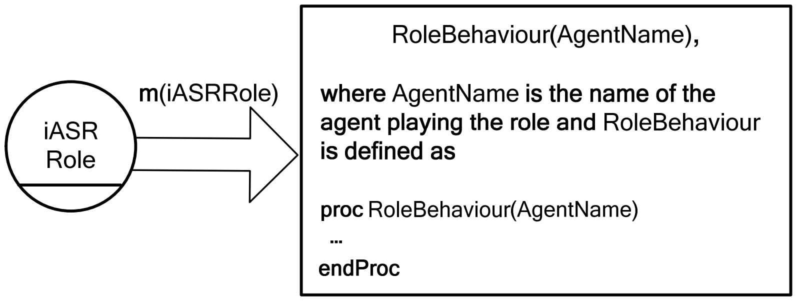

iASR
Role
m(iASRRole)
RoleBehaviour(AgentName),
where AgentName is the name of the agent playing the role and RoleBehaviour is defined as
proc RoleBehaviour(AgentName)
...
endProc

Figure 4.16. The mapping of role nodes.

A position node will be mapped into a CASL procedure that specifies the behaviour associated with the position.

```
m(iASRPosition) = PositionBehaviour(AgentName)
```

Since a position is "a set of socially recognized roles typically played by one agent" [Yu, 1995], it will also feature the agent occupying the position as its parameter. The procedure for a position calls the CASL procedures for roles that are covered by the position. For example, in the code below, procedures for the roles, which are covered by `position1` are combined using prioritized concurrency.

```
proc Position1Proc(Agent)
      Role1Proc(Agent)
      >>
      ...
      >>
      RoleNProc(Agent)
ensProc
```

4.3 Goals

4.3.1 Introducing Goal-Oriented Analysis

Goal-oriented analysis starts with the identification of the goals — the objectives to be achieved — of the stakeholders or of the software system as a whole. Such goals provide rationale for the software system requirements. Various alternative decompositions of these goals are then discovered and analyzed and the possible assignments of responsibility for the subgoals to the agents of the system-to-be are explored.

While goal-oriented analysis has been around for some time (e.g., KAOS [Dardenne *et al.*, 1993]), goals were commonly used only in the earliest stages of the analysis. Frequently, functional goals were operationalized during the late phase of the requirements engineering process and softgoals were removed, operationalized, or *metricized* [Davis, 1993]. We, on the other hand, try to give the modeler the ability to operate with goals at later stages of the RE process.

One of the most important advantages of CASL is that it supports reasoning about agents' goals (as well as knowledge). Therefore, in our combined *i**/CASL method we can push goals down to the simulation and verification stage and use goal-oriented analysis throughout the whole requirements engineering process. Moreover, leaving goals and goal dependencies in the late requirements, verification, and design stages instead of operationalizing them early provides improves the flexibility of the software development process. Leaving goals as part of the late requirements specifications, design, and possibly even implementation means that the new system will be designed with many alternative approaches for achieving those goals in mind. Some of the decisions on the choice of the strategy for achieving these goals can be made at the RE stage, while other decisions will be made during the design phase and the rest will be postponed until runtime. Removing goals early on leads to fragile software systems [Mylopoulos, 1999] with possibly incorrect decisions being hard-coded into the system. Once a goal has been operationalized, the system (at runtime) or the designer (at design time) will cease to recognize the existence of alternative ways of achieving this particular goal (Figure 5.13 illustrates this).

4.3.2 The Use of Goal Decompositions

As previously mentioned, goal decomposition is one of the main aspects of goal-oriented analysis. The *i** approach provides us with means-ends links, which are most commonly used together with goal nodes in the *i**'s SR diagrams, linking task nodes that are means to achieve some goal with that goal node (see Figure 4.17). This way we can represent various design choices or refinements in the system. The sub-tree rooted at `Means1` represents one possibility for achieving `Goal_1`, one design choice, while the sub-tree rooted at `Means2` represents another design choice. Which alternative will be selected depends on many factors. Some of these factors (e.g., the positive or negative contribution of these alternatives to various softgoals) could be taken into consideration at design time while others may be left until runtime. We no longer need to preselect the alternative before moving from *i** diagrams to CASL models. We can model all the alternatives in CASL and capture the semantics of this choice point: the agent will have the goal in its mental state with various procedures for achieving this goal available. Instead of having to make the choice of approach to achieve a goal at the requirements analysis stage, we can now leave the choice to design time or even runtime, if needed. Another advantage of this approach is that it allows us to better document the design decisions that led to the selection of one alternative over the others even if this selection is not made until the verification or design stages. If the goal is abstracted out before runtime, the model that shows appropriate means for achieving this goal and the applicability conditions for those means will help establish the traceability link between the design and the requirements.

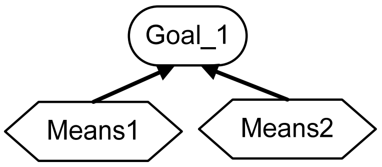


Figure 4.17. Means-ends links connect goals with means to achieve them.

To make our modeling framework more intuitive, and to streamline the way goals are used in Annotated SR diagrams, we introduced Intentional Annotated Strategic Rationale (iASR) diagrams. Let us discuss the features of iASR diagrams.

In *i**, a means-ends link indicates a relationship between an end — which can be a goal to be achieved, a task to be accomplished, a resource to be produced, or a softgoal to be satisficed — and a means of attaining it [Yu, 1995]. Several means to an end model alternative ways to accomplish that end. The ends are usually the goal nodes, since they are used to represent the desired state of affairs in the world for actors and do not specify how the goals are to be achieved, thus allowing for alternatives to be explored. The means are usually expressed in a form of task nodes, since the notion of task embodies how to do something.

We would like to capture the above intuitive use of goal nodes in iASR diagrams. This will streamline both the use of these nodes in the iASR models and the mapping of the goal nodes and goal decompositions into the corresponding CASL code. The iASR diagrams will therefore have the following feature: the means-ends links only connect goal nodes as ends and task/capability nodes as means to achieve these goals (see Section 4.5 for the discussion of capabilities). The alternative composition annotation is applied to the various means, which are specified in the means-ends decompositions.

This restriction provides a clear guidance for the use of goal decompositions in the models and their mapping to CASL. It stresses the fact that goals are a tool to be used in SR diagrams to allow for the exploration of alternatives in system models. While it is quite common for some of the alternative means of achieving goals to be goals themselves (e.g., one of the ways to achieve the goal "become rich" is to achieve the goal "win a lottery"), these means will still have to be tasks nodes in iASR diagrams. The reason for this constrains will be thoroughly examined later in the chapter. It is related to the way the agents acquire goals in CASL and the fact that agents need to use procedural means to monitor for their goals.

The constraint, while restricting the use of means-ends links to goal decomposition only, does not preclude the designer from using other *i** facilities to represent, for example, alternative ways of performing tasks. This can be easily expressed without the use of means-ends links through the use of task decompositions accompanied by the alternative composition link annotation (see Figures 4.18a and 4.18b).

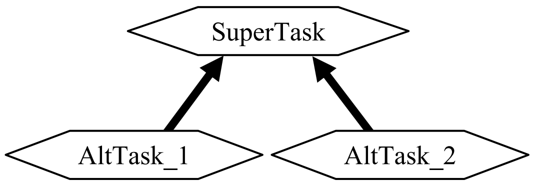


Figure 4.18a. Means-ends links are used to represent alternative ways to execute `SuperTask`.

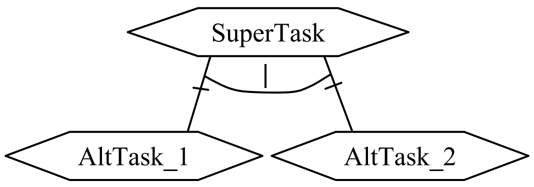


Figure 4.18b. The alternative composition annotation is used to represent alternative ways to execute `SuperTask`.

iASR diagrams can easily support various goal decomposition techniques. Figure 4.19a is an example of an AND goal decomposition in the notation of [Mylopoulos *et al.*, 2001a]. AND-decompositions are marked with an arch in this notation.

Since `Goal_2` and `Goal_3` both must be achieved for `Goal_1` to be achieved, this is in fact a single possibility for achieving `Goal_1`. The diagram in Figure 4.19b satisfies the constraint described above in that the goal node `Goal_1` has the task node `Means_1` as the means of achieving it. The task is further decomposed into several subgoals that must be achieved concurrently. The concurrency annotation is used to represent the AND-decomposition since it is the most unrestrictive way of executing any two programs — it does not specify any ordering of the primitive actions that comprise the two programs. Note that Figure 4.19b omits some details of the decomposition of the task `Means_1` related to another constraint (discussed later) on the use of goal nodes in the iASR diagrams as well as the fact that `Goal_2` and `Goal_3` both have to be acquired by the agent by executing a special `commit` action. The details will be described in the subsequent sections.

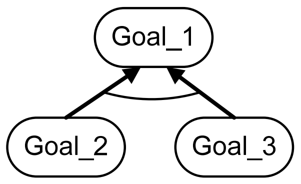


Figure 4.19a. AND-Decomposition of Goals.

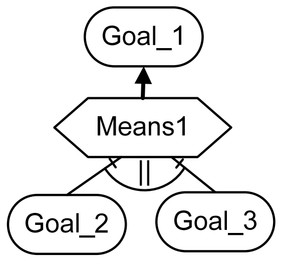


Figure 4.19b. Task decomposition with the concurrent composition annotation is used to model AND-decomposition of goals.

It is important to note that the restriction that the means to achieve goals can only be task nodes is present because of the way goals are handled in CASL. It is not a conceptual restriction, unlike the one that requires that the means to achieve goals be alternatives.

4.3.3 Applicability Conditions

If alternative means of achieving a certain goal exist, the designer is able to specify under which circumstances it makes sense to attempt to execute each alternative. We call these *applicability conditions* and introduce a new annotation `ac(Condition)` to be used with means-ends links to specify these conditions. The presence of the applicability condition (AC) annotation specifies that only when the condition is true the agent may select the associated alternative in attempt to achieve the parent goal. The absence of the condition says that the alternative can always be selected. The applicability condition may, for example, specify the conditions under which the corresponding alternative is effective, efficient, etc.

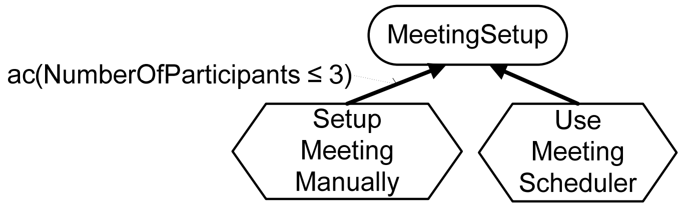


Figure 4.20. The applicability condition annotation.

Figure 4.20 shows that the goal `MeetingSetup` has two means of achieving it: `SetupMeetingManually` and `UseMeetingSchedulter`. We assume that in the first case the organizer of a meeting will manually contact the participants in attempt to setup a meeting, while in the second case the organizer will use a specialized agent to achieve the goal. The figure shows that the first alternative has an applicability constraint, which says that the task `SetupMeetingManually` can be selected as a means for achieving the goal `MeetingSetup` only when the number of the intended participants is less that four. The idea here is that while it makes sense to attempt to organize a meeting of up to 3 people manually (e.g., by calling or emailing them), it is much better to let an automated Meeting Scheduler agent schedule the meeting with the larger number of participants. The absence of an applicability condition for the task `UseMeetingScheduler` means that the Meeting Scheduler agent can be used under any circumstances.

4.3.4 Using Link Annotations in Goal Decompositions

In addition to applicability constraints, means-ends links can be supplemented with the regular link annotations: *while* loops (`while(condition)`), *for* loops (`for(variable, valueList)`) and pick annotations (`π(variableList,condition)`). The *if*, interrupt, and guard annotations are not allowed since they specify under which condition the associated task is to be performed, something that is already covered by the applicability constraints. The link annotations used together with means-ends links provide the details on how the means are to be used in achieving the parent goal.

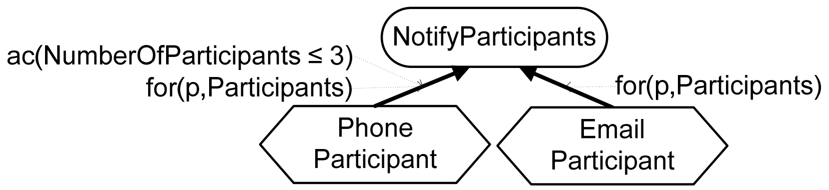


Figure 4.21. The use of link annotations with means-ends links.

Figure 4.21 illustrates the use of link annotations with means-ends links. In the above example, we have the goal `NotifyParticipants` from the Meeting Scheduler domain. Suppose there are two ways to notify the meeting participants of the time of the meeting: by phoning them and by emailing them. The *for* link annotation specifies that the tasks `EmailParticipant` and `PhoneParticipant` are to be executed for every member of the set `Participants`. Note that we use the applicability condition for the first alternative — we assume that if the number of participants is greater than three, it will not be effective to phone each of them, so the first alternative will not be executed.

4.3.5 Explicit Goals in CASL

While other methods (e.g., Formal Tropos [Fuxman, *et al.,* 2001] and the formal notation of KAOS [Dardenne, *et al.*, 1993]) represent goals as assertions, we are able to support explicit goals (e.g., **`Goal`**`(MI,`**`Eventually`**`(MeetingScheduled(mid,now),now,then),s)`). Such explicit goals represent the desirable states of the world for agents and are part of their mental state. These goals allow us, among other things, to model goal acquisition and propagation through speech act-based [Searle, 1969] interaction so that in addition to having some initial goals an agent can communicate and acquire new goals from other agents that are being served or helped by it and delegate some of its goals to agents that are helpful and capable of achieving these goals. Agent interactions is one of the most important components of multiagent systems and the ability to represent interaction protocols accurately using knowledge and goals is crucial for agent-oriented software engineering approaches. Explicit goals allow us to build more “intentionality” into CASL models, thus preserving the *i** idea of strategic, intentional actors. They also allow us to model the conflicts of interests of the system’s stakeholders and help in their resolution. One could say that this layer of explicit goals and goal delegation is an added complexity over the simple method/procedure invocation approach used in [Wang, 2001], but it is this layer that allows us to model systems more naturally and flexibly and to match *i** in representing the strategic and intentional aspects of systems.

The support for explicit goals by CASL also helps us in supporting AND/OR goal analysis [Mylopoulos *et al.*, 2001a] or other types of goal decompositions or goal refinements, which are quite common at the early stages of the requirements engineering process with *i** framework, even at runtime. We expect that most of such goal analysis tasks will be carried out before the *i** diagrams are mapped into CASL so that the CASL models do not contain all of the intermediate goal nodes and all the possible alternatives. Nevertheless, CASL models can include whole chains of goal refinements and goal decomposition if necessary, thus allowing agents to reason about these refinements at runtime. The support for explicit goals also allows us to represent conflicting goals of agents/stakeholders.

4.3.6 Declarative vs. Procedural Specification in CASL

In CASL, agent behaviour (what actions the agent does) is specified procedurally, while goal and knowledge changes (changes to the agent’s mental state) are handled declaratively. This means that while behaviour specification in CASL is fairly similar to mainstream programming languages (the programmer has the usual high-level constructs like loops, procedure calls, etc.), changes to the mental state of the agent can be the effects of perception or commitment actions performed by the agent himself or of communication actions performed by other agents. In CASL, one cannot explicitly ask the agent to change its mental state; it must be done by executing the actions, which have the desired effect on the mental state of the agent. The effects of these actions are specified through the appropriate successor state axioms. In CASL, as it is described in [Shapiro and Lespérance, 2001], only the communication actions such as `inform` (for knowledge acquisition) and `request` (for goal acquisition) have effects on the mental state of the agents. We, on the other hand, would like to get the flexibility of having agents change their mental state on their own. This will support *self-acquired* goals, which we talk about later in the chapter. We, therefore, need a new action to support the acquisition of goals without communications from other agents. We call this action `commit`. This primitive action is thoroughly discussed later in the chapter (Section 4.3.8). In addition, self-acquisition of knowledge is possible with the new action `assume` (Section 4.4.5). See Section 4.6 for the overview of how procedural and declarative components of CASL agents can be synchronized.

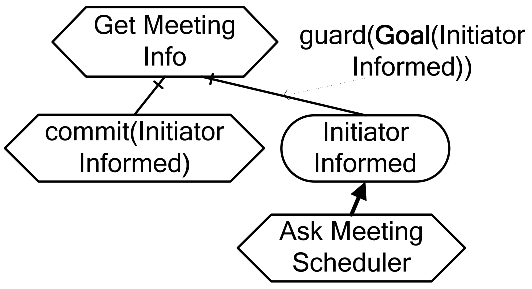


Figure 4.22. An example illustrating the use of the `commit` action in iASR diagrams.

Since goal change in CASL is handled declaratively, but behaviour is specified procedurally, the agents need to monitor for newly acquired goals and to respond accordingly. The agent must recognize that some goal is in its mental state. This is true for the agent *agt* and some goal *φ* when the formula ***Goal(agt,φ,s)*** holds. The natural CASL constructs for this monitoring are the interrupt and the guard. An interrupt fires as soon as some condition (an instance of some goal in this case) is true, executes its body (presumably, the procedure that attempts to achieve the goal), and then goes back into the waiting state. A guard, on the other hand, blocks the execution until the condition (an instance of the goal in the mental state of the agent) is true and then executes some procedure. Figure 4.22 shows how the `commit` action and the guard annotation are used with self-acquired goals.

Therefore, in iASR diagrams *the decomposition link connecting a goal node with its parent task node must be accompanied by an interrupt link annotation with the appropriate trigger and cancellation conditions or a guard link annotation with the appropriate condition*. Since interrupts and guards can only be used in CASL procedures, in iASR diagrams *the parent of a goal node must be a task node*.

Figure 4.23 illustrates how goal nodes are used with interrupt or guard link annotations in iASR diagrams. We will talk more about the trigger and cancellation conditions for interrupts associated with goal acquisition in Section 4.3.9.3.

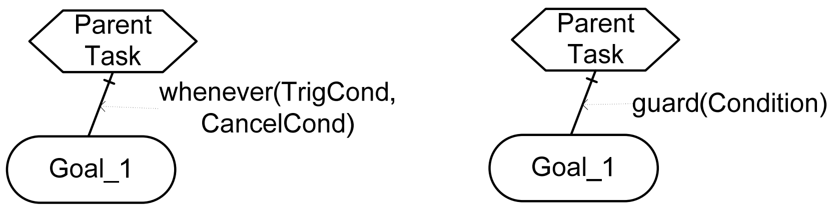


Figure 4.23. The use of the interrupt and the guard link annotations with goal nodes in iASR diagrams.

In Figure 4.23, the parent task is responsible for monitoring for the newly acquired instances of `Goal_1`. The parent task node, `ParentTask`, will be mapped into the appropriate CASL procedure that will contain the interrupt or the guard that is designed to monitor for `Goal_1`.

In summary, the presence of goal nodes in iASR diagrams indicates that the actor recognizes that it has the corresponding CASL goals in its mental state. In order for those goals to be achieved the modeler has to provide the necessary means of achieving them (task nodes), which will be placed below the goal nodes and connected to them by means-ends links. The parent nodes of the goal nodes will be tasks that are responsible for monitoring for the goals.

4.3.7 Acquiring Goals: Dependency-Based Goals

In our approach, agents acquire goals in two different ways: through requests coming from other agents in the system and by themselves using the `commit` action. The requests correspond to the inter-agent dependencies in which the agent receiving the request is the dependee, while the goals acquired by the agent on its own are self-acquired goals. We discuss goals coming from inter-agent dependencies here and then cover self-acquired goals in the next section.

In CASL, an agent can acquire a goal after receiving a request from some other agent. This is very important since it allows for agents acting together as a team to achieve certain goals together. It also allows agents to delegate some tasks or goals to other agents that are capable of performing/achieving them or can perform or achieve them better than the delegating agents. The ability to model this delegation of goals in CASL is also important for the support of *i** style of systems modeling in which the notion of dependency is of utmost importance. Goal acquisition through requests coming from other agents is a natural way to model inter-agent dependencies. In fact, in our approach goal dependencies and task dependencies are modeled through requests to achieve a goal or perform a task respectively. Resource dependencies are represented through either goal or task dependencies. The agent in need of a resource can ask the provider of that resource to use a specific means to supply it. In this case, we map the resource dependency into a task dependency. Alternatively, the depender may give the provider of the resource complete freedom in choosing the best way to supply it. This naturally maps into a goal dependency. Note this is more specific and operational than the dependencies in *i** (which do not specify how goals are delegated). It may be a bit restrictive in some contexts, where delegation is done without communications (e.g., a mother monitoring her child), but it seems quite appropriate for agent-based systems.

As previously discussed, we use interrupts (and guards) in CASL models to check for newly acquired goals. These interrupts are placed inside the procedures responsible for monitoring for the new goals. As can be seen in Figure 4.24, the node inside the dependee actor at which an intentional dependency terminates is the node responsible for fulfilling the dependency. It represents the goal the dependency must achieve or the task it must execute. In iASR diagrams, dependencies originate from task nodes. The details of the trigger conditions and cancellation conditions for the interrupts will be presented later in this chapter.

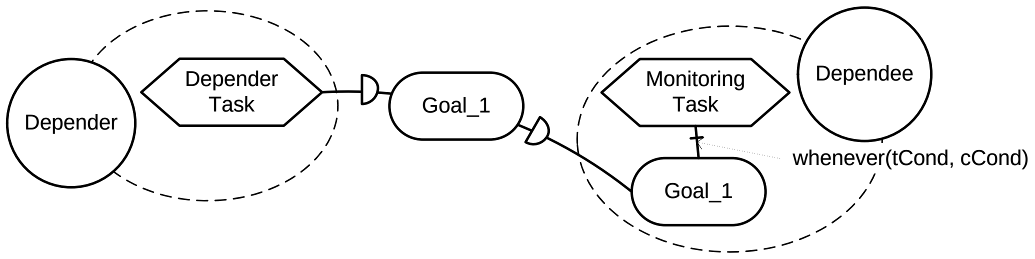


Figure 4.24. An example of inter-agent goal dependency.

To be successfully mapped into CASL models for simulation and verification, SR diagrams have to be sufficiently detailed. We try to preserve as much of the strategic and intentional aspects of SR diagrams as possible while mapping them into the corresponding CASL models. The CASL language is much more process-oriented than *i** and requires that we add the missing details to the iASR diagrams. The particulars of inter-agent dependencies are one of the areas that need to be modeled at a more detailed level. While at an early stage the inter-agent dependencies may look like (or be even less detailed than) the goal dependency depicted in Figure 4.24, before mapping the iASR diagram into CASL they need to be refined as in Figure 4.25.

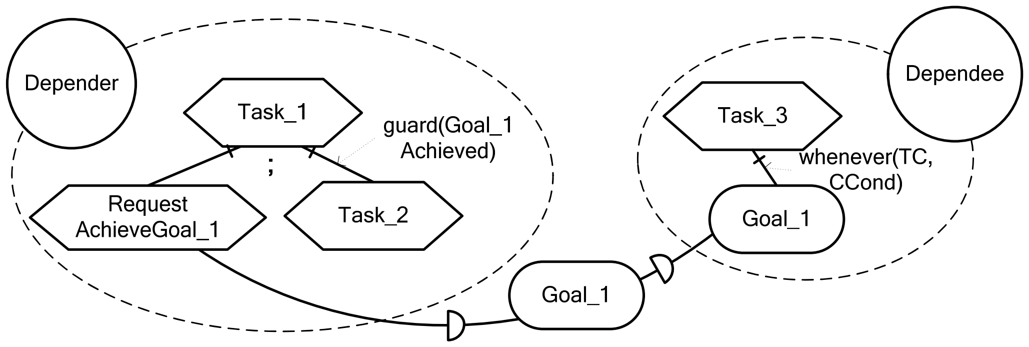


Figure 4.25. The iASR-level details of an inter-agent goal dependency.

As you can see in Figure 4.25 above, the idea is for every inter-agent dependency (of any type) to add a task that makes a request to another agent to supply the dependum. In Figure 4.25, this task is called `RequestAchieveGoal_1`. It is the origin of the inter-agent goal dependency in the iASR diagram. Usually agents continue executing their tasks after the dependee has performed the required task, achieved the goal, or provided the resource. If the depender made a synchronous request to the dependee, it will need to wait until the dependum is provided before continuing with other activities. This situation is illustrated by Figure 4.25. We added an additional task, `Task_2`, inside the Depender agent. This task is the one executed after the dependum (here, it is the goal `Goal_1`) is provided. One way to base that task's execution upon the supply of the dependum by the dependee is to add the guard link annotation, which blocks until the dependum is supplied. Alternatively, the Depender may continue with its activities without waiting for the achievement of `Goal_1` (asynchronous request). In this case, it may be concurrently monitoring for `Goal_1`. If the dependum is not observable by the depender, the depender may require the dependee to notify it after providing the dependum. The details of this refinement of goal delegation will depend on the application context.

In general, the origin of an inter-agent dependency in an iASR diagram is the depender's task that sends the request to provide the dependum to the dependee. The end of an inter-agent dependency is the node in the iASR diagram for the dependee that represents the goal the dependee must achieve or the task it must execute to provide the dependum.

Agents use the `request` communicative action to request services of other agents. In the request below, the Depender requests the Dependee to achieve φ, which can be either a goal in the CASL sense (a constraint on the future states of the world for the Depender) or a goal to perform some task.

```
request(Depender,Dependee,φ)
```

For a goal dependency, a request for some agent to achieve φ is usually represented as follows:

```
request(Depender,Dependee,Eventually(φ))
```

Note the use of **`Eventually`** — it gives the requested agent flexibility in terms of *when* the goal can be achieved. **`Eventually`**`(φ)` states that the goal φ has to be achieved by the depender at some time in the future. **`Eventually`** is not the only temporal predicate that can be used with `request` action. With the use of other temporal operators we can express, for example, goals that have to be maintained at all times (**`Always`**`(φ)`). However, in this thesis, we concentrate on achievement goals in the form of **`Eventually`**`(φ)`.

4.3.7.1 Task Dependency-Based Goals

While goal dependencies allow the dependee to select appropriate ways of achieving goals for the depender, quite frequently the depender wants to specify what exactly the dependee has to do in order to provide the dependum of the dependency. In this case, a task dependency is established. In our approach, in a task dependency the depender requests the dependee to execute a known procedure. A request by one agent to another to perform a procedure `SomeProc`, *which must be a properly defined procedure of the Dependee*, is represented as follows:

```
request(Depender,Dependee,DoAL(SomeProc,now,then))
```

Here, **DoAL**(δ,s,s′) (*Do At Least*) is an abbreviation for **Do**(δ||(πa.a)*,s,s′). This means that the program δ must be executed, but other concurrent activities may be executed as well. The dependee will have the flexibility to do other things while executing the requested procedure. After finishing the execution of `SomeProc`, the dependee will most likely notify the depender of this fact if the completion of the task is not observable by the depender.

Below we present an example of a task dependency (see Figure 4.26). Here, the Meeting Scheduler (MS) agent requests the intended meeting participant to send his/her available dates to it so that the MS can come up with possible mutually agreeable (for all the participants) dates for the meeting. `GetAvailDates` task of the Meeting Scheduler sends a request to the Participant to execute the Participant's task `SendAvailDates`:

```
request(MeetingScheduler,Participant,DoAL(SendAvailDates,now,then))
```

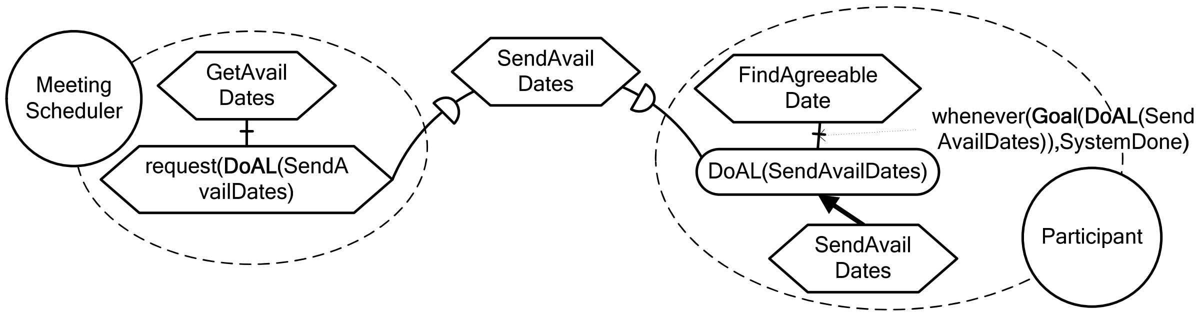


Figure 4.26. The detailed version of the iASR diagram for the `SendAvailDates` task dependency.

The `FindAgreeableDate` task of the Participant monitors for the acquisition of the goal **DoAL**`(SendAvailDates,now,then)`. This is not a goal to achieve some state in the system. Here, the depender asks the dependee to execute one of the dependee's procedures. Therefore, the number of alternative ways of achieving this goal is limited to exactly one — to execute the requested procedure.

In general, a means-ends decomposition of a goal to perform a task is limited to having just one means — that means being the execution of the task. By acquiring a goal to perform a task (see the note on the *Serves* relationship and how it can affect the acquisition of goals later in this section), the agent acquires the responsibility to perform the task. The task therefore must be performed unconditionally. This means that applicability conditions and link annotations are not allowed to be used with task nodes that are means to achieve such goals. For example, in Figure 4.26 the task `SendAvailDates` is the only means to achieve the goal **DoAL**`(SendAvailDates,now,then)` and is unaccompanied by any annotation.

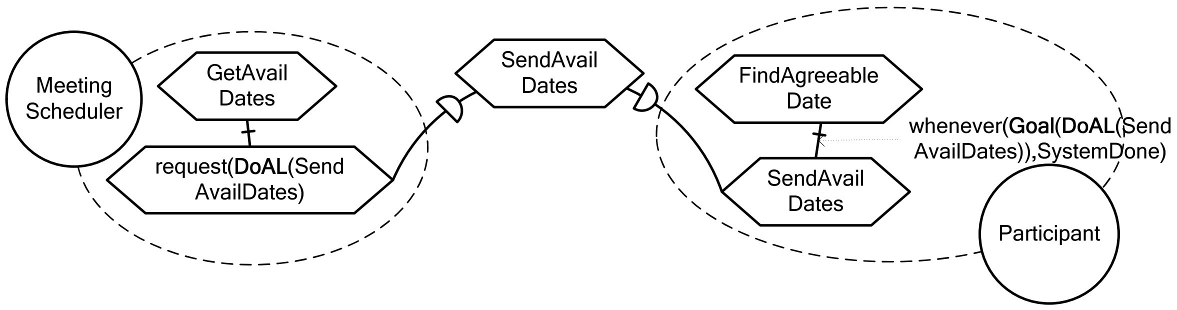


Figure 4.27. A simpler version of the iASR diagram for the `SendAvailDates` task dependency.

Figure 4.26 is a hypothetical diagram that presents the full details of the task dependency `SendAvailDates` including the acquisition of the goal by the Participant and the means-ends decomposition of the goal. Since the only means to achieve a goal that requests the dependee to perform a task is to execute this task, modeling means-ends decompositions in such cases is unnecessary. The designer can replace the diagram of Figure 4.26 with a simpler diagram, where the means-ends decomposition is removed. Since there is no means-ends decomposition anymore, we can even remove the goal node from the diagram as in Figure 4.27. In this simplified version of the diagram, the task `SendAvailDates` (the means to achieve the goal) becomes a subtask of the task `FindAgreeableDate` (the task responsible for monitoring for the goal) and is executed whenever the goal is acquired since it is now accompanied by the interrupt annotation, which was previously associated with the removed goal node. The resultant diagram is much simpler and much more straightforward to map into the corresponding CASL code. However, there may be cases where the dependee dos not want to execute the requested procedure unconditionally. In this case, the model of Figure 4.26 may be extended to include other conditions.

4.3.7.2 Resource Dependency-Based Goals

In *i**, a resource dependency is a separate type of dependency, different from goal and task dependencies. While modeling systems at high level of abstraction, we talk about one actor depending on another for some resource to be provided. At the iASR level, however, we can model resource dependencies more precisely. When one agent is in need of a certain resource that is provided by another agent, it can either ask that agent to furnish the resource without specifying the exact means of doing it, or it can instruct the provider of the resource on how exactly it is supposed to provide the resource. Therefore, resource dependencies are modeled through requests to either achieve a goal of making the resource (the dependum of the resource dependency) available to the requesting agent, or performing some task that provides the resource. A special case of resource dependency — information resource dependency — will be discussed in Section 4.4.3.

Below we have an example of an SR resource dependency that becomes a goal dependency at the iASR level. In this case, the resource is money. The SR diagram fragment shows the resource dependency where the actor representing a telecom company (Telecom) depends on its customer (Customer) for the payment of his bills. In this scenario, the Customer actor will have a reciprocal dependency on the Telecom actor to provide various communication services. This dependency is not shown in the example.

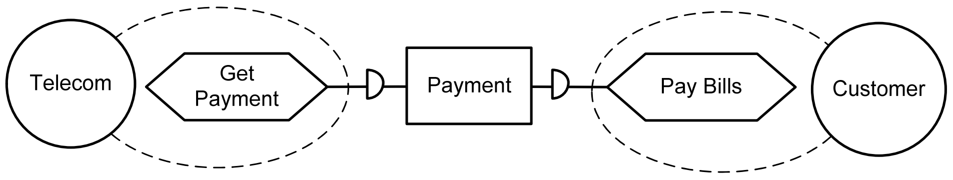


Figure 4.28. The SR diagram with the `Payment` resource dependency.

Figure 4.28 shows the SR diagram that represents the Telecom-Customer payment dependency. At the SR level, we show `Payment` as being a resource dependency originating at `GetPayment` task of Telecom. On the Customer end, the task `PayBills` is responsible for supplying the resource. When we go to the Intentional Annotated SR level however, we need more precision in modeling the dependency (see Figure 4.29).

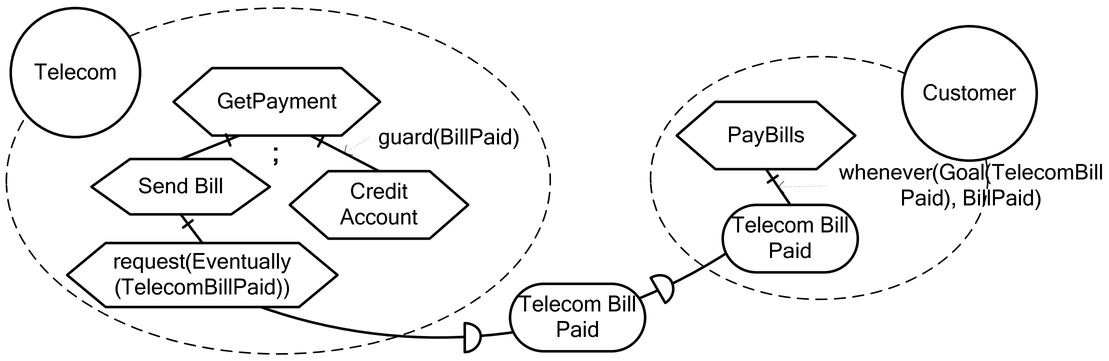


Figure 4.29. The iASR-level details of the `Payment` dependency.

Figure 4.29 presents the detailed version of the Payment dependency and is a typical example of the iASR-level refinement of inter-agent dependencies. First, we specify the Telecom task that is responsible for requesting the Customer to pay for the Telecom services. This task is called `SendBill` and as its subtask we have the appropriate `request` action, which is the origin of the dependency. We assume that the value of the predicate fluent `BillPaid` becomes true once the Customer pays its telecom bill. The task `CreditAccount` is executed by the Telecom once it becomes aware of the payment. Here, we assume that there are many alternative ways to pay the Telecom bill and the Telecom actor prefers to leave it up to the Customer to select the way to pay the bill. Therefore, the resource dependency `Payment` is converted to a goal dependency `TelecomBillPaid`. When this dependency is mapped into CASL code, the request will look like this (we omit the parameters of the goal here):

```
request(Telecom,Customer,Eventually(TelecomBillPaid))
```

On the Customer side, the above request causes the agent to acquire the goal `TelecomBillPaid`. As previously discussed, the Customer agent needs to have an interrupt that monitors for newly acquired goals of this type. The task `PayBills` contains the corresponding interrupt. Upon acquiring the goal `TelecomBillPaid`, the Customer agent will select one of the means (not shown in the figure) for achieving it.

4.3.7.3 The *Serves* Relationship

In the discussion so far, we have assumed that agents acquire goals (either to achieve some conditions or to perform some tasks) whenever they are asked to do so by other agents. In real-life scenarios, however, agents may not always be helpful and collaborative, or at least not to every agent in the system. There may be many reasons for that. Certain services offered by the agents could be too costly and the designer may want only to allow the use of such services by a restricted number of agents; due to security concerns, some capabilities may only be offered to some trusted agents. It is very easy to address the above concerns with the use of the *Serves* relationship. **`Serves`**`(Dependee,Depender,φ)` specifies that the Dependee is going to be helpful to the Depender in providing φ. Another possibility is to use a more general version of the

Serves relationship, **Serves**(Dependee,Depender), which specifies that the Dependee is willing to help the Depender in everything. One possible use of this version of *Serves* is the specification of the managerial structure of the company. We will return to this issue in Chapter 5.

In order to make use of the *Serves* relationship we need to make changes to the way agents acquire goals. This means that we have to modify the successor state axiom for the *W* relation, the accessibility relation on situations that specifies which situations are compatible with what the agent wants. Below is the modified version of the successor state axiom for *W* that takes the *Serves* relationship into consideration. It specifies that in order for agt to adopt the goal φ requested by depender, agt needs to be helpful to depender in achieving φ, i.e., **Serves**(agt,depender,φ) has to hold.

```
W(agt,then,do(a,s)) ≡ [W(agt,then,s) ∧
    ∀depender,φ,now (a = request(depender,agt,φ) ∧ K(agt,now,s) ∧
    now ≤ then ∧ Serves(agt,depender,φ) ∧ ¬Goal(agt,¬φ,s) ⊃
    φ[do(a,now),then])]
```

The above axiom states that a situation then is accessible from do(a,s) iff it is *W*-accessible from s and if the action a is a request action requesting that φ hold, and now is the current situation along the path defined by then, and the agent agt is helpful to depender, and the agent does not have the goal that ¬φ in s, then φ holds at [do(a,now),then)].

4.3.7.4 Reasoning about Inter-Agent Communication

Since the semantics of request and other communication acts available to agents is formally defined in CASL, the framework allows us to reason about the effects they have on agents' mental states without knowing the detailed content of those messages. For example, if we know that when some agent requests the agent `agt` to achieve the goal **`Eventually`**`(φ)` and later some other agent requests `agt` to achieve **`Eventually`**`(¬φ)`, we can conclude that `agt` may acquire both goals since they are not necessarily conflicting. On the other hand, if the goals are **`Always`**`(φ)` and **`Always`**`(¬φ)`, then the agent will not acquire the goal **`Always`**`(¬φ)` since the successor state axiom for *W* prevents agents from acquiring conflicting goals.

4.3.8 Acquiring Goals: Self-Acquired Goals

Sometimes an agent needs to acquire a goal without other agents requesting it. Most commonly, these goals are used in goal decompositions or to analyze several possible scenarios or design alternatives. By making the agent acquire a goal by itself the modeler makes sure that the agent's mental state reflects the fact that multiple alternatives exist in that particular place in the model of the agent's behaviour. Moreover, unlike task decompositions with alternative composition annotations, the presence of a goal node suggests that the designer envisions new possibilities of achieving the goal. This way the agents would be able to reason about various alternatives available to them or come up with new ways to achieve the goals. Self-acquired goals are the tool that can be used by the modelers to identify and analyze problems with multiple solutions and document the possible alternatives. These goals also add flexibility to the system models in that alternative approaches for solving the goals can be kept in the model and the designers will not have to operationalize the goals early.

The agent is able to self-acquire goals by performing a special action called `commit`. By executing `commit(Agent,φ)`, where `Agent` is the name of the agent and φ is the formula that the agent wants to hold, the agent adds the goal of achieving φ to its mental state. By executing the `commit(Agent,φ)` action the agent essentially makes a request to itself to achieve the goal φ.

Unlike the `request` action, which is only possible when the requestor agent has the corresponding goal in its mental state, the `commit` action does not have any preconditions. The agents can therefore try to acquire any goal that is consistent with its existing goals. The successor state axiom for the *W* accessibility relation is modified to take into consideration the new `commit` action. As is the case with the goals that are acquired through requests coming from other agents, the axiom will guard against adopting self-acquired goals that are inconsistent with the previously acquired goals:

```
W(agt,then,do(a,s)) ≡ [W(agt,then,s) ∧
        ∀depender,φ,now((a = request(depender,agt,φ) ∧
        Serves(agt,depender,φ) ∨ a = commit(agt,φ)) ∧
        K(agt,now,s) ∧ now ≤ then ∧
        ¬Goal(agt,¬φ,s) ⊃ φ[do(a,now),then])]
```

Similarly to goals coming from the inter-agent dependencies, upon the execution of the `commit` action the agent needs to recognize that the goal is in its mental state. Guard annotations are used with self-acquired goals in iASR diagrams: the `commit` action changes the mental state of the agent and a guard allows the behaviour of the agent to be modified to reflect the change in the agent's mental state. We do not need to use interrupts (which fire multiple times) to monitor for self-acquired goals since agents acquire up to one goal per every execution of the `commit` action (goals may not be acquired if they conflict with existing agent goals, so the agent may acquire no goals for some executions of the `commit` action). So, we use the guard annotation instead of an interrupt since the agent needs to achieve the goal once per each `commit` action.

The general iASR diagram pattern for handling self-acquired goals is shown in Figure 4.30. The `commit` action is presented in its own task node so that it is easily visible on the diagram. The `commit` action is then followed by a goal node with the corresponding `guard` link annotation. The guard condition becomes true when the goal is acquired by the agent. This happens immediately after `commit` is performed. The children of a goal node are the means to achieve this goal.

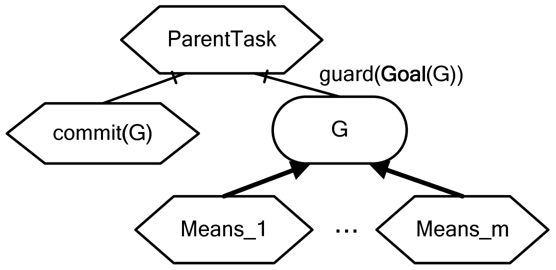


Figure 4.30. The iASR diagram pattern for handling self-acquired goals.

Let us look at the example that demonstrates the use of self-acquired goals (Figure 4.31). Suppose that after finishing the scheduling process, the Meeting Scheduler agent needs to inform participants about the time and location of the meeting. Since there are many possibilities of contacting participants, any many more may become available later, it is wise to include the goal `ParticipantInformed` into the iASR model for the scheduler instead of simply selecting a means for informing participants and modeling it as a task in the iASR diagram. The goal `ParticipantInformed` adds flexibility to the process specification, allows for documenting all the alternative ways of achieving the goal along with their applicability conditions (not shown in Figure 4.31), and enables the agent to select the best way to achieve this goal at runtime based on the most current context.

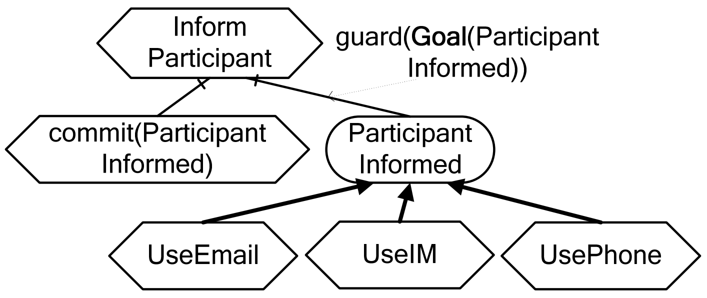

Inform Participant
guard(Goal(Participant Informed))
commit(Participant Informed)
Participant Informed
UseEmail
UseIM
UsePhone

Figure 4.31. Using self-acquired goals.

One of the main uses of designer goals is to model various types of goal refinement including AND-decompositions and OR-decompositions of goals. For example, now we can show the full details of the decomposition of the Figure 4.19b. In the case of goal decompositions, self-acquired goals enable the agent to go through its designer's reasoning at runtime.

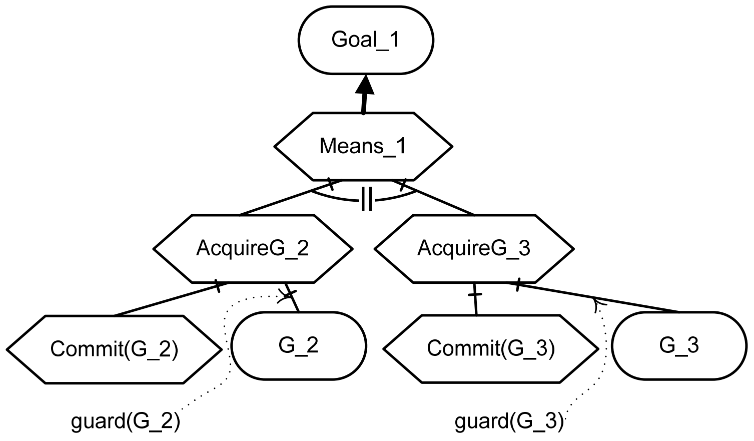


Figure 4.32. The fully detailed iASR diagram for the AND-decomposition of goals.

In Figure 4.32, `Goal_1` is AND-decomposed into `Goal_2` and `Goal_3`. `Goal_2` and `Goal_3` are the self-acquired goals — there are no inter-agent dependencies that make the agent acquire these goals. Since the goals have to be achieved in parallel, we have two tasks, `AcquireG_2` and `AcquireG_3`, which are executed concurrently and are responsible for the acquisition and achievement of `Goal_2` and `Goal_3` respectively. Each task is decomposed into the `commit` action that acquires the required goal. It is followed by the goal node, which is guarded by the guard that blocks until the goal is acquired. The above diagram illustrates the fact that at the iASR level the diagrams get quite detailed and complicated since they have to be easily mapped into CASL code. A lot of the diagram modifications are quite mechanical and could be automated. Note that in iASR diagrams, we usually omit the first parameter of the `commit` action since it is clear what agent is executing this action. We conclude that there is a price in complexity if one wants to model goals explicitly in our approach.

4.3.9 Mapping Goals into CASL

4.3.9.1 Introduction

One of the essential features of CASL is that it supports reasoning about agents' goals, which play a major role in the *i** approach. In this section, we describe the mapping of goal nodes of iASR diagrams and the associated means-ends decompositions into CASL models.

The presence of a goal node in an iASR diagram indicates that an agent has the corresponding goal in its mental state. We, therefore, need to make sure that the counterparts of the goals that the designer utilizes in the iASR diagrams are present in the corresponding CASL models. Designers use goal nodes in their iASR diagrams to model the agents' objectives that can be achieved by a number alternative approaches. Means-ends decompositions are used to encode various alternative means of achieving the goals of the agents. These means-ends decompositions will have to be encoded in CASL and associated with the goals that they are designed to achieve.

4.3.9.2 Mapping Goal Nodes

An iASR goal node will be mapped into a CASL formula that corresponds to the goal and a procedure that specifies the approaches that the agent will use to try to achieve the goal. In our approach, we mostly handle achievement goals. The formulae that we are mapping the iASR goals into are situation-dependent. They represent the desired states in the system: when the formulae are evaluated to false, this means that the corresponding goals are not yet achieved; the fact that the values of such formulae change to true signals the achievement of the goals. Alternatively, we can map goals into primitive fluents, if appropriate.

Let us see how the mapping is defined for an iASR goal node called `iASR_Goal` (see Figure 4.33). For now, we do not consider the parent and children nodes of `iASR_Goal`.

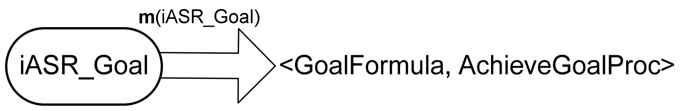


Figure 4.33. The mapping of the goal node `iASR_Goal`.

The mapping **m** will map the goal node `iASR_Goal` into the following tuple:

m`(iASR_Goal) = <GoalFormula, AchieveGoalProc>,`

where

- `GoalFormula` is a CASL formula with a free situation variable
- `AchieveGoalProc` ∈ $PName_c$ and `AchieveGoalProc` is defined in P_c

We remind here that $PName_c$ is a set of procedure names in the CASL theory C and P_c is a set of procedure definitions in this theory. The constraints on `AchieveGoalProc` prevent iASR goal nodes from being mapped into undefined procedures.

We use the following notation to access the goal formula and the achievement procedure of a goal node:

m`(iASR_Goal).formula = GoalFormula`

m`(iASR_Goal).achieve = AchieveGoalProc`

The achievement procedure for the goal encodes the ways the agent may achieve this goal. It may use the appropriate applicability conditions and/or link annotations to specify the means precisely. We assume that generally, the agent has means to achieve the goal that typically work, but it is rare that they will always work. Thus, we cannot guarantee that any of the means for achieving the goal that are represented in the achieve procedure for that goal are always able to achieve it. In order to be assured that `AchieveGoalProc` in fact makes `GoalFormula` true every time it is executed we need to carefully specify, for instance, the restrictions on the situation in which the procedure is invoked. Making sure that the applicability condition for the `AchieveGoalProc` holds when the agent starts executing the procedure cannot, however, guarantee that it successfully achieves the goal (the executability of `AchieveGoalProc` does not imply the achievement of the goal). The main difficulty here is the fact that we are modeling a multiagent system with lots of interacting concurrent processes in a shared environment. When the procedure is executing, other agents' actions may modify the parameters of the environment (and even the agent's own mental state), which are crucial for the successful execution of `AchieveGoalProc`. There are a number of ways to deal with the problem. One could divide the set of fluents in the system into subsets that are each controlled by one agent in the system, thus making it impossible for other agents to directly affect the values of the fluents and making sure that procedures relying on those fluents and executed by the 'owner' agent cannot be disrupted by other agents. Alternatively, we can set up a system for managing shared (for reading) and exclusive (for writing) locks on fluents to guarantee that only one agent at a time can change the value of a shared fluent. We leave this to the future work. We note here that while some of the approaches for concurrent systems verification were applied to the problem of multiagent systems verification, the problem remains largely open.

Here, we assume that `AchieveGoalProc` is designed to achieve the corresponding goal, but may not do that at all times. We therefore state that `AchieveGoalProc` will try to achieve the goal, instead of saying that it will in fact achieve the goal. The above semantics is expressed in the following formula:

$$\exists s,s'.\mathbf{Do}(\mathrm{AchieveGoalProc},s,s') \wedge \mathrm{GoalFormula}[s']$$

The above formula says that sometimes executing `AchieveGoalProc` in some situation `s` in the absence of concurrent actions will lead to its successful termination in situation `s'` and in that situation `GoalFormula` will hold (the goal will be achieved).

Agent capabilities, which are described later in this chapter, provide a greater level of assurance that the goal will be achieved. They should be used, among other things, as means of achieving critical goals.

4.3.9.3 The Use of Interrupts

Since goal acquisition in CASL is done declaratively through the use of successor state axioms and agent behaviours are specified procedurally, the agents need to monitor their mental states for changes and to modify their behaviour in response to these changes. The synchronization of the declarative part of the agent — its mental state — and the procedurally specified behaviour is achieved through the use of interrupts (see Figure 4.34) or guards.

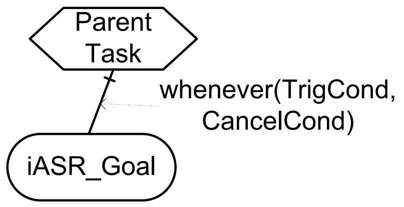


Figure 4.34. The use of interrupts for goal monitoring.

Let us now look more closely at the use of interrupts and interrupt annotations. In the iASR diagrams, goal nodes *always* have task nodes as their parents and are connected to them with the task decomposition links. These tasks are used to monitor for the newly acquired goals, therefore the interrupts must be placed in the goals' parent tasks. When the trigger condition becomes true, it signals that the goal has been acquired in the mental state of the agent. When the interrupt fires, the agent will attempt to achieve the newly acquired goal by executing the goal's achievement procedure. This is how the goal in the mental state triggers the appropriate achievement behaviour. For example, let us consider the mapping of the ASR diagram fragment in Figure 4.34. Suppose that the `ParentTask` task node is mapped into `ParentTaskProc` by the mapping **m** (**m**`(ParentTask) = ParentTaskProc`). When the trigger condition becomes true the achievement procedure for `iASR_Goal` is executed. Based on the task decomposition mapping rules presented earlier in this chapter the procedure for the parent node of `iASR_Goal` will look like this:

```
proc ParentTaskProc
   ...
     <m(trigCond) → m(iASR_Goal).achieve until m(CancelCond)>
   ...
endProc
```

In our approach, we use a new version of interrupts that provides us with more flexibility in mapping from iASR diagrams into CASL models. This new version of interrupts includes a cancellation condition that, unlike the original ConGolog version, allows the programmer to control when interrupts stop firing. We use cancellable interrupts mainly to specify under which circumstances the parent task of a goal node stops monitoring for the new instances of the goal. The reason for that is the fact that many goals have to be repeatedly achieved, some have to be achieved exactly once (these goals are normally used with guards), some have to be monitored for only during some specific activities, etc.

Let us discuss how our extended version of interrupts is defined. We first go over the way interrupts are handled in ConGolog [De Giacomo *et al.*, 2000]:

```
<φ → δ> ≡ while Interrupts_runnning do
      if φ then δ else False? endIf
```

Here, φ is the trigger condition, while δ is the program that is executed whenever φ becomes true. It is assumed that a special fluent `Interrupts_running` is initially *True*. When control is given to the interrupt, it repeatedly executes δ until becomes φ false. At this point, the interrupt blocks and gives up control to another part of the program. Based on the transition semantics of ConGolog, if φ becomes false, the process blocks right at the beginning of the loop. If later some action makes φ true, the loop is resumed. To terminate the loop (not just suspend it) a special primitive action called `stop_interrupts` is used. Its sole effect is to make `Interrupts_running` false. Therefore, the general pattern for a program δ1 that contains interrupts is `start_interrupts;(δ1>>stop_interrupts).` `start_interrupts` and `stop_interrupts` are automatically added by the ConGolog interpreter, so the programmer does not have to worry about them. This program would stop all the blocked interrupt loops in δ1 when there are no more actions in δ that can be executed (since `stop_interrupts` runs at the lower priority, it is executed only when δ1 blocks).

The cancellation condition in the new version of interrupts allows the user to decide when the interrupt should be stopped:

```
< TriggerCondition → Body until CancelCondition >
```

The above interrupt is in fact defined as a *while* loop:

```
while ¬CancelCondition do
        if TriggerCondition then Body else False? endIf
```

The cancellation condition is assumed to be false initially. The loop terminates when the cancellation condition becomes true. The fluents that correspond to the cancellation conditions are defined by the programmer. The system specification must ensure that the cancellation condition is false at the beginning and during the execution of the interrupt and that it eventually becomes false.

We also define the fluent `SystemDone`, which, when used as a cancellation condition, emulates the original ConGolog interrupts. The interrupts with `SystemDone` are not stopped until the program is terminated. So, the interrupt

```
<TriggerCondition → Body>
```

is viewed as an abbreviation for

```
<TriggerCondition → Body until SystemDone>.
```

Quite frequently, we need to use interrupts with parameters. The interrupt fires when there are bindings for the variables `v1`, `v2`, etc. for which `TriggerCondition` holds:

```
< v₁, v₂,…,vₙ: TriggerCondition → Body until CancelCondition >
```

The above interrupt is mapped into the following:

```
while ¬CancelCondition do
    if π $v_1,v_2,…,v_n$: TriggerCondition($v_1,v_2,…,v_n$) then
        Body($v_1,v_2,…,v_n$) else False? endIf
```

On the other hand, one quite frequently needs interrupts that fire only once. In our approach, it is possible to use the guard annotation, `guard(φ)`, instead of such interrupts. This approach saves the developer from defining the unnecessary cancellation conditions.

4.3.9.4 Mapping Dependency-Based Goals

Goals coming from inter-agent dependencies are acquired by agents through `request` communication actions. We assume that the agents that acquire these goals have the commitment to do everything possible in order to achieve them. The dependency-based goals will most likely have to be constantly monitored for and will have to be repeatedly achieved. These goals are generic (parameterized) ones: every time the depender agent makes a request to the dependee to achieve a goal, it actually requests an instance of the goal to be achieved.

The dependency-based goals are acquired through the actions of other agents — when an agent receives the appropriate request from some agent in the system, its mental state changes accordingly. The agent has to detect the change in its mental state and modify its behaviour in order to respond to the request by attempting to achieve the goal, perform the requested task, or supply the needed resource. The dependee's tasks that are responsible for monitoring for new goals acquired through inter-agent dependencies will therefore contain the interrupts that will fire when the new goals are acquired. The bodies of those interrupts will contain the achievement procedure for the goal.

In the example in Figure 4.35, we show the dependency `MeetingScheduled` between Meeting Initiator (MI) and Meeting Scheduler (MS). The MS's task `MSBehaviour` monitors for the goal `MeetingScheduled` and contains an interrupt with the trigger condition `TriggerC` and the cancellation condition `CancelC`. Furthermore, the MS has one means to achieve the goal `MeetingScheduled` — by using the task `ScheduleMeeting`.

As expected, `MSBehaviourProc` (the CASL procedure, into which the task `MSBehaviour` is mapped) will contain the following code that deals with the goal `MeetingScheduled`:

```
proc MSBehaviourProc
  ...
  <m(TriggerC) → m(MeetingScheduled).achieve until m(CancelC)>
  ...
endProc
```

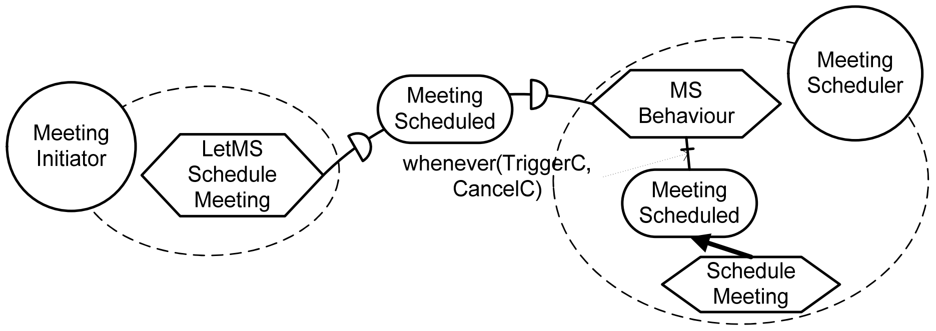


Figure 4.35. The `MeetingScheduled` goal dependency.

Let us now examine the trigger and cancellation conditions for the above interrupt. As mentioned earlier, many inter-agent dependencies exist throughout the life of system and therefore the dependee has to continuously monitor for the new goals coming from these dependencies. In the example diagram of Figure 4.35, `MeetingScheduled` is a goal dependency of this type. Here, achieving the goal `MeetingScheduled` is the main responsibility of the Meeting Scheduler agent and it acquires the goal whenever the Meeting Initiator needs to schedule meetings. In cases like this, goal-monitoring interrupts run until the system is terminated. We use the fluent `SystemDone` as a cancellation condition when we only need it to become true upon the termination of the system.

On the other hand, unlike the top-level dependencies (similar to `MeetingScheduled` dependency above) that are generally present throughout the life of the agent, there are many cases of dependencies that are present only in certain contexts. For example, when agents are using an interaction protocol to negotiate about something, request clarification, etc., the associated dependencies exist only in the context of some other, higher-level dependencies. Figure 4.36 shows a diagram with a dependency that only exists in a certain context.

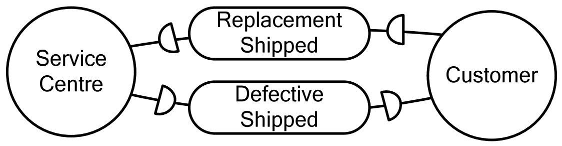


Figure 4.36. Top-level and context-only dependencies.

The diagram in Figure 4.36 models a common scenario of servicing products under warranty. In the above figure, we have two actors, Service Centre and Customer. Customer depends on Service Centre for the shipment of replacement products. At the same time, we have another dependency, this time going from Service Centre to Customer, where Service Centre requires Customer to ship the defective product back. While `ReplacementShipped` is clearly the top-level dependency here, `DefectiveShipped` dependency exists only in the context of `ReplacementShipped`. When the service centre attempts to solve the instance of the goal `ReplacementShipped`, it will in turn request the customer to ship the defective product back. For the Customer agent it does not make sense to monitor for the goal `DefectiveShipped` unless it has already requested the replacement product from Service Centre. Therefore, we can model the cancellation condition for the interrupt that monitors for the acquisition of `DefectiveShipped` with the fluent `ReplacementComplete`. It could be either a primitive or a defined fluent. This fluent is false initially and the program will make sure that it holds once the replacement is complete. Of course, in order to differentiate between instances of the goal `ReplacementShipped`, both the formula that the goal node is mapped into and the fluent `ReplacementComplete` need to be parameterized. We present the interrupt for handling `DefectiveShipped` goals below:

$$\langle \mathbf{m}(\texttt{TriggerC}) \rightarrow \mathbf{m}(\texttt{DefectiveShipped}).\texttt{achieve}\ \ \mathbf{until}\ \mathbf{m}(\texttt{ReplacementComplete})\rangle$$

Now we will discuss the trigger conditions. In CASL, once an agent acquires a goal, it cannot be removed from the agent's mental state unless the `cancelRequest` action is executed for the goal. Even if some goals are achieved, the agent still has them in its mental state in the sense that they are true in desirable histories for the agent. When monitoring for new, not yet achieved goals, we need to select only those instances of goals for which the formula they map into is false. Therefore, the triggering condition for the dependency-based interrupts will normally be (here we use the version of interrupts with parameters):

$$\langle \vec{t} :\ \mathbf{Goal}(\texttt{agt},\mathbf{Eventually}(\mathbf{m}(\texttt{G}).\texttt{formula}(\vec{t}),\texttt{now},\texttt{then})) \wedge \mathbf{Know}(\texttt{agt},\neg\mathbf{m}(\texttt{G}).\texttt{formula}(\vec{t}),\texttt{now}) \rightarrow \ldots\rangle,$$

where $\vec{t}$ is a parameter list and `G` is a goal node in an iASR diagram. Here, we assume that $\mathbf{m}(\texttt{G}).\texttt{formula}(\vec{t})$ (the CASL formula, which the goal is mapped into) is an achievement goal and therefore was requested as $\mathbf{Eventually}(\mathbf{m}(\texttt{G}).\texttt{formula}(\vec{t}),\texttt{now},\texttt{then})$ by the depender. Our assumption is that all goals are of this form.

In the trigger condition above we pick the parameter list $\vec{t}$ such that the goal $\mathbf{Eventually}(\mathbf{m}(\texttt{G}).\texttt{formula}(\vec{t}),\texttt{now},\texttt{then})$ is in the mental state of the agent `agt`. This means that the agent had to achieve the goal $\mathbf{m}(\texttt{G}).\texttt{formula}(\vec{t})$ some time after it acquired it. Additionally, we require that $\mathbf{Know}(\texttt{agt},\neg\mathbf{m}(\texttt{G}).\texttt{formula}(\vec{t}),\texttt{now})$ be true, meaning that the agent knows that the goal $\mathbf{m}(\texttt{G}).\texttt{formula}(\vec{t})$ has not yet been achieved. If there is an instance of the goal that has not been achieved, then the agent must try to achieve it. Here, we assume that once achieved goals stay true.

In the context of the diagram of Figure 4.36, the interrupt that the Service Centre agent will use for handling the inter-agent goal dependency `ReplacementShipped` will look like this (we omit the situation parameters here):

```
<sn: Goal(SC,Eventually(m(ShipReplacement).formula(sn))) ∧
            Know(SC,¬m(ShipReplacement).formula(sn)) →
                        m(ShipReplacement).achieve(sn) until SystemDone>
```

Here, we check if there is a serial number (we assume that it uniquely identifies the product) for which a customer requested a replacement and Service Centre has not yet shipped that replacement. In this case, the agent knows that the goal formula, which is supposed to be true when the goal is achieved, is, in fact, not true for the selected value of `sn`. The cancellation condition is `SystemDone` since this is a high-level dependency independent of any other dependencies.

Since most of the trigger conditions and many cancellation conditions are quite lengthy, we expect that many of them could be replaced with special labels (for example, `TCondShipRepl` is the label for the trigger condition of the interrupt that hands the goal `ReplacementShipped`), which are substituted by corresponding CASL formulae or fluents before we use the mapping **m** to map iASR diagrams into CASL models. The modeler must keep track of these labels.

4.3.9.5 Mapping Task Dependency-Based Goals

In a task dependency, the depender asks the dependee to perform a certain task. Since the request is made with the use of the `request` action, the dependee acquires the corresponding goal to perform the task. As described previously, task dependency-based goals are special kind of goals since there are no alternative solutions for these goals — the only way to achieve them is to execute the required task. Therefore, the CASL mapping for the task dependencies is simpler than for other dependencies. We now present the code that the Participant agent will use for handling the task dependency `SendAvailDates` (see Figure 4.27 for the iASR diagram for this example). The procedure `FindAgreeableDateProc` (the CASL procedure that the task node `FindAgreeableDate` of Participant is mapped into) will look like the following:

```
proc FindAgreeableDateProc
      ...
      <TC → SendAvailDates until SystemDone>
      ...
endProc
```

The trigger condition `TC` for the above interrupt should become true if the task `SendAvailDates` has not yet been performed by the Participant agent, hence we take it to be the formula below:

```
<Goal(Ptcp,DoAL(SendAvailDates,now,then)) ∧
          Know(Ptcp,¬∃s,s'(s≤s'≤now ∧ DoAL(SendAvailDates,s,s'))) → … >
```

The above expression becomes true if the Participant agent (`Ptcp` in the code above) has the goal to perform at least the task `SendAvailDates` and knows that it has not done that before the current situation. Note here that once the Participant agent performs the required task, the **`Know`** part of the trigger can never evaluate to *True* since the agent will know that it performed the task. Therefore, no matter how many requests to perform `SendAvailDates` it receives, it will simply ignore them. To remedy the situation the designer might want to incorporate some parameters (e.g., time, request ID, etc.) into the task so that different instances of the goal to perform the task can be distinguished from each other. [1]

In general, the agents could be asked to perform tasks with parameters. Below we have a parameterized version of the trigger condition for handling goals to perform tasks:

$$<\vec{t}: \mathbf{Goal}(\texttt{agt}, \mathbf{DoAL}(\texttt{SomeTask}(\vec{t}), \texttt{now}, \texttt{then})) \wedge$$
$$\mathbf{Know}(\texttt{agt}, \neg\exists s', s''[s' \leq s'' \leq \texttt{now} \wedge \mathbf{DoAL}(\texttt{SomeTask}(\vec{t}), s', s'')]) \rightarrow \ldots >$$

When using the parameterized version of the trigger condition, the agent is able to distinguish among various requests asking it to perform the task `SomeTask`, provided the parameters of the task are different. If it is required that some task be performed for each and every request (e.g., the task “check radiation level”), this task could get a time stamp parameter that will make every request different. The depender then can use the following code to request some other agent to perform a task with a time stamp:

$$\pi\, \texttt{t (t = Time(now))?; request(agt1,agt2,}\mathbf{DoAL}\texttt{(SomeTask(t),now,then)))}$$

4.3.9.6 Mapping Self-Acquired Goals

Self-acquired goals are the ones obtained without requests from other agents. They could be used by the designer to model alternative ways of achieving the required results. As previously described, designer goals are also needed to handle goal refinements if these refinements are to be analyzed at the CASL level.

[1] A simple account of time is presented in [Shapiro *et al.*, 1998]. A special functional fluent `Time(s)` is introduced, which maps a situation into the current time at the situation. It is also assumed that all the actions in the domain increment `Time` by the amount it takes to perform those actions.

In the example diagram below (Figure 4.37) we have a fragment of a task decomposition of the `ScheduleMeeting` task of the Meeting Scheduler agent (we are not showing all the subtasks of `ScheduleMeeting`). One of the responsibilities of the task is to get the information about participants' availability. There could be some number of ways to achieve this goal and the designer wants to keep his options open and therefore uses a self-acquired goal to capture the (potential) existence of alternatives. To acquire the designer goal `AvailableDatesKnown` we have to execute the `commit` action that adds the instance of the goal into the agent's mental state. Note that we use the parameter `mid` (meeting ID) to allow the MS to acquire and achieve instances of the goal `AvailableDatesKnown` for every meeting it schedules.

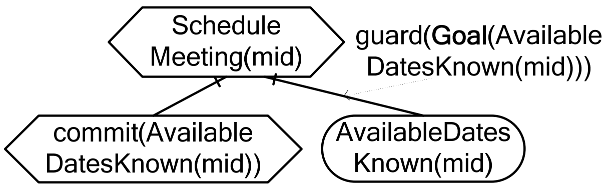

Schedule Meeting(mid)
guard(Goal(Available DatesKnown(mid)))
commit(Available DatesKnown(mid))
AvailableDates Known(mid)

Figure 4.37. The `AvailableDatesKnown` self-acquired goal.

We assume that **m**`(ScheduleMeeting)=ScheduleMeetingProc`. The guard condition in Figure 4.37 is mapped into the CASL formula **Goal**`(MS, AvailableDatesKnown(mid),s)`, however it is labelled as **Goal**`(AvailableDatesKnown(mid))`. We frequently omit some parameters in conditions, goal/task node labels, etc. in iASR diagrams for brevity or replace complex conditions with acronyms, etc. The CASL mapping of Figure 4.37 into CASL is quite simple:

```
proc ScheduleMeetingProc(mid)
 ...
 commit(MS,Eventually(m(AvailableDatesKnown(mid)).formula(mid)));
 guard Goal(MS,Eventually(m(AvailableDatesKnown(mid)).formula(mid))) do
```

```
      m(AvailableDatesKnown(mid)).achieve(mid)
 endGuard
endProc
```

We assume that the task node `ScheduleMeeting` is mapped into the CASL procedure `ScheduleMeetingProc(mid)`, which takes the meeting ID as parameter. The commit action is then executed for that specific meeting ID acquiring the goal `AvailableDatesKnown` for a specific meeting request. The guard condition is the presence of this goal in the mental state of the Meeting Scheduler. Since the goal has just been acquired for the meeting `mid`, we know that it has not yet been achieved. Therefore, we do not check whether the goal has been already achieved as we usually do when we map interrupts into CASL.

In fact, since the execution of the `commit` action makes an agent acquire a single instance of a goal, the guard condition can check only for that particular instance, thus making this condition much simpler than the usual trigger condition of an interrupt. We now present a generic procedure for handling the self-acquired goal `SomeGoal`:

```
proc ParentTaskProc($\vec{t}$)
    ...
    commit(agt,Eventually(m(SomeGoal).formula($\vec{t}$)));
    guard Goal(agt,Eventually(m(SomeGoal).formula($\vec{t}$))) do
          m(SomeGoal).achieve($\vec{t}$)
    endGuard
endProc
```

Here, the procedure for the parent task takes a parameter list $\vec{t}$. Somewhere in the code of the procedure, the agent acquires the goal `SomeGoal` with the parameters $\vec{t}$ by executing the `commit` action. Next, we have the guard operator, whose condition becomes true once this instance of `SomeGoal` is in the mental state of the agent `agt`. Since we know which instance of the goal has just been acquired and we also know that the instance could not have been achieved yet, the guard condition does not check whether the instance has been achieved or not.

In this discussion, we assume that the task node that is mapped into the commit action and the goal node representing the self-acquire goal are next to each other in an iASR diagram, as in Figure 4.37. If an agent is self-acquiring instances of a goal in one place, while monitoring for them in another, then the modeler must be careful in using guards since they block the process until their conditions become true.

4.3.9.7 Mapping Achievement Procedures

Achievement procedures for goals encode the ways in which these goals can be achieved. The achievement procedure for a goal is produced by mapping the means-ends decomposition of the goal node, which corresponds to that goal, into CASL. As discussed previously, in iASR diagrams, the means for achieving goals have to be task nodes and the only composition annotation available for the means-ends links is the alternative annotation. In coming up with a goal achievement procedure we must take into consideration the applicability conditions as well as the link annotations associated with the various means of achieving the goal.

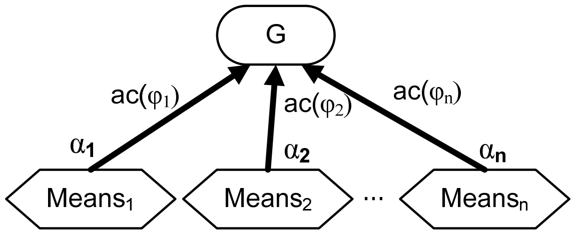


Figure 4.38. Generic means-ends decomposition.

Figure 4.38 shows generic means-ends decomposition for the goal G. There are *n* ways to achieve the goal G, each represented by the task Means_i. The means-ends links that connect the means with the goal G are each augmented with applicability conditions and link annotations that provide additional details on the ways the goal G could be achieved. Figure 4.38 is a diagram representation for the procedure that achieves the goal G. Suppose that **m**(G).achieve = G_Achieve. The code for G_Achieve is presented below:

```
proc G_Achieve
    guard m($\varphi_1$) do m($\alpha_1$)(m($\text{Means}_1$)) endGuard
  |
    guard m($\varphi_2$) do m($\alpha_2$)(m($\text{Means}_2$)) endGuard
  |
    …
  |
    guard m($\varphi_n$) do m($\alpha_n$)(m($\text{Means}_n$)) endGuard
endProc
```

For example, below (Figure 4.39) we have a case, which is a modification of the example in Figure 4.21. It displays three possibilities for notifying the participants of a meeting: phoning them, emailing them and coming up to them and notifying them personally.

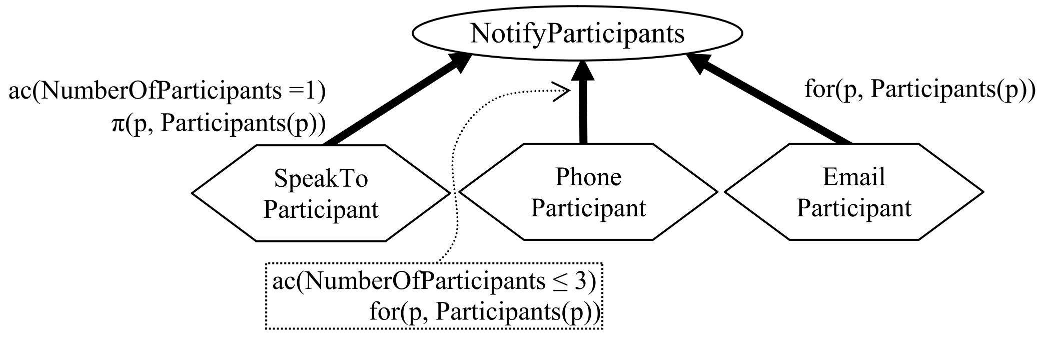


Figure 4.39. An example of a means-ends decomposition.

The means for achieving the goal `NotifyParticipants` in Figure 4.39 have a mix of applicability conditions and link annotations. The applicability conditions specify under which circumstances the means *can* be selected. If the number of meeting participants is one, all three alternatives can be considered; if the number of the participants is two to three, then only two means are considered, etc. The link annotations that are used in Figure 4.39 are *for* loops that indicate that each participant must be notified by the chosen method and the *pick* annotation, which is used with the first alternative and selects the only meeting participant. Let **m**`(NotifyParticipants).achieve` = `NotifyParticipantsProc`. We present the mapping of the means-ends decomposition of Figure 4.39 below. The absence of an applicability condition for a means to achieve a goal indicates that the means is applicable under any circumstances. So, the absence of the applicability condition for the task `EmailParticipants` means that unlike the two other tasks there is no guard operator in the mapping of the task procedure and the link annotation for that means:

```
proc NotifyParticipantsProc(Participants)
       guard NumberOfParticipants=1 do
             π p. Participants(p)?; m(SpeakToParticipant)(p)
       endGuard
      |
       guard NumberOfParticipants≤3 do
            for p: Participants(p) do m(PhoneParticipant)(p) endFor
       endGuard
      |
       for p: Participants(p) do m(EmailParticipant)(p) endFor
endProc
```

In mapping means-ends decompositions, we have to take into consideration the applicability conditions for the means and the link annotations associated with the means.

The mappings of every means to achieve the goal are joined by the non-deterministic choice operator (”|”) since the means are alternative ways to achieve the parent goal. The code obtained from mapping a generic means-ends link is shown below:

guard m(φ_i) **do m**(α_i)(**m**(Means_i)) **endGuard**,

which, based on the definition of the guard operator, is defined as follows:

if m(φ_i) **then m**(α_i)(**m**(Means_i)) **else** False? **endIf**

The conditional construct (and thus the condition of the guard operator) takes care of the applicability condition **m**(φ_i). If the applicability condition **m**(φ_i) holds, then we execute **m**(α_i)(**m**(Means_i), which is a mapping of the task Means_i augmented with the mapping of the link annotation α_i. Based on the semantics of the *if-then-else* operator, the test of the applicability condition and the first transition of the associated means are performed as a single atomic step. Therefore, we are sure that the applicability condition holds when the procedure for the means starts executing. If the applicability condition is false, then the above expression blocks without performing any transitions since the test False? can never succeed.

In general, if we have several means of achieving some goal, we will have the CASL code below (we removed the mapping **m** for brevity). The same discussion applies to any number of means.

guard AC_1 **do** Means_1 **endGuard** | … | **guard**(AC_n) **do** Means_n **endGuard**

Based on the semantics of the non-deterministic choice operator (a | b), either one of the actions has to be successfully executed for the whole construct successfully execute. In the code above, if several applicability conditions are true, then either of the means is executed. If only one applicability condition holds, then only the associated means can be executed. If none of the applicability conditions hold, then the whole construct blocks until something makes one of the applicability conditions true.

4.4 Knowledge

4.4.1 Introduction

Information exchange plays an extremely important role in human organizations. It is used for passing knowledge, coordination, etc. In modern society, information becomes more and more important and is frequently treated as a critical resource on which the lives of people and organizations depend. Information resources are becoming increasingly valuable and many are willing to pay a premium for convenient access to reliable information. It is hard to find cases where humans working together do not exchange information

Since multiagent systems mimic human societies in that they are systems with multiple autonomous and social agents that work together to achieve goals (alternatively, they can be in competition), they too benefit immensely from information exchange. Not unlike CASL's support for reasoning about goals, the ability of the CASL language to model agent knowledge and knowledge exchange allows us to model multiagent systems more naturally. Instead of talking about one software entity invoking a mutator method of another entity to modify the variable, which stores the price of some product, we say that one agent informs another of the change in price for the product they are negotiating about.

The support of CASL for reasoning about knowledge allows the designer to model information exchanges precisely and enables verification of information-related requirements. Information exchange is abundant in multiagent systems and communication, coordination, cooperation, etc. all involve the sending and receiving of information that can be formalized and reasoned about in CASL.

4.4.2 Acquiring Knowledge

The CASL formalism allows us to model two aspects of the mental state of the agents: their goals and their knowledge. Agents acquire knowledge about something when they are informed of it by other agents. There are several communicative actions that allow agents to exchange information with each other. These actions, `inform`, `informRef`, and `informWhether`, are used by the agents in the system to send information to other agents. To avoid the situation where agents lie, they can only tell others what they know. The mental state of the agents receiving the new information changes accordingly. We discussed how knowledge is represented in CASL in Section 2.2.2.

4.4.3 Mapping Information Resource Dependencies

Since CASL supports reasoning about agents' knowledge and goals, it is useful for modeling of information flow in distributed systems. Information exchange and interaction protocols play an important role in multiagent systems and we are able to model them well in CASL. As far as we know, the only other approach that explicitly represents agent knowledge is ALBERT-II [du Bois *et al.*, 1997]. In *i**, the flow of information can be modeled through information resource dependencies. We distinguish information resource dependencies based on the requests that the dependers make to the dependees. The request requires the dependee to inform the depender of the value of a functional fluent or of the truth value of a certain formula.

Figure 4.40 shows a generic information resource dependency example where the `Depender` agent asks the `Dependee` to inform it of the truth value of the formula φ. The task `AskAboutφ` is responsible for asking the `Dependee` agent for information about the truth value of φ. This is the SR diagram for the dependency and it does not contain enough information to be mapped into a CASL model.

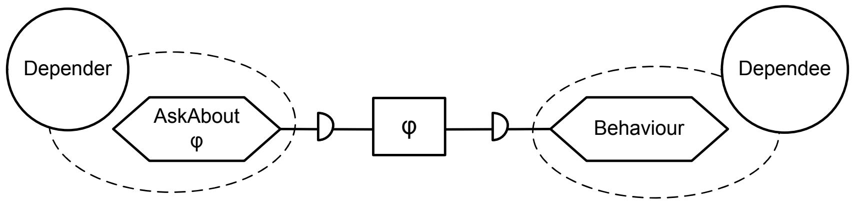


Figure 4.40. A generic info resource dependency.

At the iASR level, the above diagram will be modified to include more details about how the dependency is fulfilled. First of all, as is the case with all resource dependencies, an information resource dependency is turned into a task or a goal dependency. If the information dependency is turned into a goal one, the goal of that dependency is the fact that the depender eventually has the requested information. The dependee can achieve the goal by executing the appropriate inform action, thus passing the needed information to the depender. We will also include a depender task node that will actually make the request to the dependee (it will be mapped into the appropriate `request` action). The dependee model will include the means to achieve the information goal. To illustrate the above discussion let us look at the iASR diagram for the example from Figure 4.40 (in the diagram below, `EvKnowφ` is an abbreviation for **`Eventually`**`(`**`KWhether`**`(Depender,φ))`):

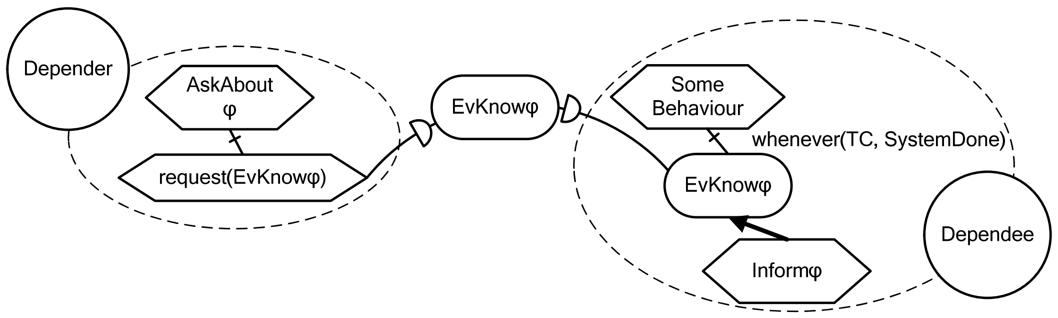


Figure 4.41. The iASR diagram for a generic info resource dependency.

The task `AskAboutφ` now includes a subtask, which is a `request` action, the new origin of the dependency. In order to receive the information that it needs, the `Depender` agent will execute the following `request` action (we omit the situation argument):

```
request(Depender,Dependee,Eventually(KWhether(Depender,φ)))
```

Here, the `Depender` agent asks the `Dependee` agent to be informed of the truth value of the formula φ. The dependee agent will then acquire the goal **`Eventually`**`(`**`KWhether`**`(depender,φ))` and will achieve it by executing the task `Informφ`. Since the `Depender` agent wants to know whether the formula φ is true or false, the `Dependee` task `InformPhi` might be the following (we omit the situation arguments for brevity):

```
proc InformPhi
    if Know(Dependee,φ) then
        inform(Dependee,Depender,φ)
    else
        if Know(Dependee,¬φ) then
            inform(Dependee,Depender,¬φ)
        endIf
    endIf
endProc
```

At first, it may seem that there is only one way of achieving an information goal, namely to provide the requested information. However, the agent may need to perform some computation to get the requested information first (or in turn ask other agents for the information), and then send it to the requestor. As seen from the above example, achieving information goals is not done by simply executing the appropriate `inform` action.

Alternatively, agents can ask to be informed of the value of functional fluents. The way for the `Dependee` agent to achieve the goal **`Eventually`**`(`**`KnowRef`**`(Depender,θ))` is to execute the corresponding `informRef` action. The consequence of an agent receiving one of the inform-type communications is that its knowledge base is updated with the new information. The details of the interactions will vary according to the application.

4.4.4 The *Trusts* Relationship

When an agent receives information from other agents through `inform` actions, its mental state changes to include the knowledge of the newly communicated information. This is done unconditionally in that no matter who the sender agent is, the new information is accepted. This may not be acceptable for all the applications. Trust plays an increasingly important role in the multiagent systems and it could be very useful for the modeling framework to support trust and distrust among agents. One of the ways to base information acquisition on the level of trust among the pair of agents is to introduce a *Trusts* relationship, which is similar to the *Serves* relationship for goals in that it filters out information coming from untrusted sources. A simple way to specify that an agent trusts another one is to say **`Trusts`**`(Depender,Dependee)`. This says that the `Depender` agent trusts all the information provided by the `Dependee` agent. We can fine-tune the relation to include the subject of trust in it: **`Trusts`**`(Depender,Dependee,φ)`, which says that the `Depender` agent trusts the `Dependee` for updates on the value of the fluent or formula `φ`.

In order to make use of the *Trusts* relationship we need to make changes to the way agents acquire information. Here is an example of the modifications that one needs to make to the successor state axiom for the *K* accessibility relation for knowledge (assuming that `inform` is the only knowledge-producing action):

$$
\begin{aligned}
&K(\mathtt{agt},\mathtt{s''},\mathtt{do(a,s)}) \equiv \\
&\quad \exists \mathtt{s'}\ (K(\mathtt{agt},\mathtt{s'},\mathtt{s}) \wedge \mathtt{s''} = \mathtt{do(a,s')} \wedge \mathtt{Poss(a,s')} \wedge \\
&\quad \forall \mathtt{informer},\varphi\ (\mathtt{a} = \mathtt{inform(informer,agt},\varphi) \wedge \\
&\qquad \underline{\mathtt{Trusts(agt,informer)}} \supset \varphi[\mathtt{s'}]))
\end{aligned}
$$

The above axiom states the conditions the situation `s″` is *K*-accessible from situation `do(a,s)`. For non-inform actions, the predecessor of `s″`, called `s′`, must be *K*-accessible from `s` and the action `a`, which takes `s′` to `s″`, must be executable in `s′`. If, however, the action is an `inform` action, then, provided that the agent `agt` trusts the informer, in addition to the above constraints, the subject of the communication, φ, must hold in situation `s′`.

4.4.5 Self-Acquisition of Knowledge

Agents acquire knowledge about something when they are informed of it by other agents. There could be cases when an agent needs to come to know something without being informed about it. For example, the agent might compute a value of a function and then communicate that information to other agents, etc. For these situations we propose the action `assume(agt,φ)`, whose effect is that `agt` knows that φ is true. Alternatively, the agents could use the action `assumeRef(agt,θ)`, whose effect is that the `agt` knows who/what θ is.

In CASL, the precondition of the `inform` action is that the agent knows what it is informing another agent about (`Poss(inform(informer,agt,φ),s) ≡ `**`Know`**`(informer,φ,s)`). The `assume` action does not have this precondition, however, we require that agents can only assume true propositions (perhaps unknown): `Poss(assume(agt,φ),s) ≡ φ(s).`

4.4.6 Specifying Interaction Protocols

There has been a lot of work on the specification of agent interaction protocols. For example, AgentUML [Odell *et al.*, 2000] is a rich, non-formal, UML-based notation, while commitment machines [Yolum and Singh, 2001] provide a formal way to specify protocols that can be executed flexibly. Let us show how goals and knowledge can be used in describing interactions among agents. More work has to be done to support intuitive, flexible and formal specifications of interaction protocols in the combined *i**-CASL approach, but even with the facilities we currently have, it is possible to model interaction protocols and execute them flexibly. Here, we look at the simplified version of the NetBill protocol discussed in [Yolum and Singh, 2001]. This protocol describes interactions between a customer and a merchant. In NetBill, once a customer accepts a quote, the merchant sends the encrypted goods (e.g., software) and waits for payment from the customer. Once the payment is received, the merchant sends the receipt with the decryption key. The interactions are shown in Figure 4.42.

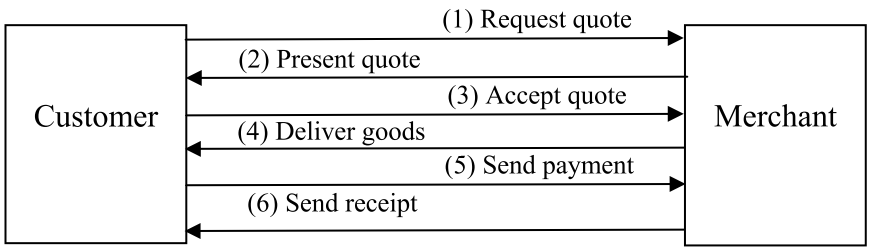


Figure 4.42. A simplified NetBill protocol.

For an agent playing a role of customer, a very simple specification of the protocol may look like the following:

```
proc CustomerBehaviour(Cust)
      set(¬Done);
      assume(¬Done);
      (<Goal(Cust,Eventually(KRef(Cust,Quote(product)))) →
         request(Cust,Merch,Eventually(KnowRef(Cust,Quote(product))))
      until Know(Cust,Done)>
      ||
      <KRef(Cust,Quote(product)) →
            if Know(Cust,GoodPrice(Quote(product))) then
                  inform(Cust,Merch,AcceptQuote(Cust,product)))
            else set(Done); assume(Done) endIf
      until Know(Cust,Done)>
      ||
      <Know(Cust,ReceivedGoods) →
            commit(PaymentSent);
            guard Goal(Cust,PaymentSent) do
                  SendPaymentProc
            endGuard
      until Know(Cust,done)>
      ||
      <Know(Cust,ReceiptReceived) →
            set(Done); assume(Done)
      until Know(Cust,Done)>)
endProc
```

In the above code, `Done` means that the interaction protocol has finished executing. We initially set `Done` to false. We change the value of `Done` by first executing the action `set`, which changes the value of the fluent, and then executing the action `assume`, which updates the mental state of the agent accordingly. There are four interrupts in the program. All of them are running concurrently and correspond to the states in the execution of the protocol, where action on the part of the customer is required. The first interrupt is triggered when the agent has the goal of knowing the merchant's quote for `product`. In the body of the interrupt the customer asks the merchant for the quote. Once it knows the quote, the second interrupt is triggered. The customer then evaluates the quote with the `GoodPrice` function (we omit its definition here). If the price is good, the customer informs the merchant that it accepts the quote, otherwise the protocol is terminated. Once the customer knows that it has received the goods, it commits itself to sending the required payment to the merchant. Note here that unlike the previous interrupts, the agent does not simply execute the body of the interrupt, but rather adopts the goal `PaymentSent`. This is an example of a self-acquired goal. Since there are a number of ways the customer can pay the merchant, hard-coding one of the methods into the protocol can greatly reduce the flexibility of the system. Therefore, the agent acquires the goal of paying the merchant through the `commit` action. The usual guard operator that blocks until the goal is in the mental state of the customer follows this action. The achievement procedure for the goal `PaymentSent`, `PaymentSentProc` is the body of the guard operator. Finally, the last interrupt is triggered when the customer receives the receipt from the merchant. At this point, the interaction protocol terminates.

Note that by using `assume` and `commit` appropriately to acquire goals or knowledge in the triggering conditions of the interrupts, an agent can execute this interaction protocol flexibly by jumping into it at any place.

4.4.7 Conditional Commitments in Interaction Protocols

When humans and human organizations are involved in joint business or other activities, they (if they are benevolent) are committed to fulfilling their obligations to the other parties involved in the activities. Quite frequently, the parties have a lot of conditional commitments related to some possible future situations. In a business setting, these commitments are usually explicit. For example, when a customer orders a car from a car dealer, the dealer commits to customizing the car according to the customer wishes (e.g., installing the optional equipment) and delivering the car to the customer. It also commits to fixing the car in case the car breaks down during the warranty period provided that the car is properly maintained by the customer.

When several agents are involved in some joint activity according to a particular interaction protocol, they are not only committed to the immediate action that is required by the protocol, but also are conditionally committed to fulfilling their obligations down the road, provided their counterparts do the same. In the NetBill protocol, for example, when the merchant sends the quote to the customer, it also commits to sending the goods upon the customer's acceptance of the quote. Moreover, the merchant is committed to sending the receipt upon receiving the payment for the delivered goods. Logically, all of these commitments have to be in the mental state of the merchant, but handling these conditional commitments declaratively will greatly increase the number of goals for the agents (a lot of the conditional commitments are many layers deep resulting in complex expressions with nested goals) and make the execution of the interaction protocols quite cumbersome. [Yolum and Singh, 2001] provide a way to solve this problem by introducing *commitment machines* that can be compiled into an easily traversable structure similar to a finite state automaton. While this approach is quite interesting and the applicability of it (or some of its ideas) to the specification of interaction protocols in CASL should be investigated, we leave this as a future work. Here, we adopt a simple approach for handling conditional commitments that can be summarized as follows:

- The agent commits to executing the required interaction protocol (as shown for the NetBill protocol earlier), thus *implicitly* acquiring all the conditional commitments associated with this protocol.
- At each step of the protocol, the agent *explicitly* acquires the appropriate commitments deemed appropriate by the designer (e.g., once the merchant

receives the payment, it acquires the commitment to send the receipt to the customer).

This way we can avoid the potentially huge expansion of the agent's goals due to the conditional commitments. In our approach, the designer determines which commitments should be in the mental state of the agent and which should be handled procedurally, without any changes to the mental state of the agent. For example, upon receiving the payment for the goods, the merchant should send the receipt to the customer. We can make the merchant acquire the goal `SentReceiptTo(Customer)`, thus changing the mental state of the agent:

```
<Customer: Know(Merchant,ReceivedPaymentFrom(Customer) →
            commit(SentReceiptTo(Customer))
until Know(Merchant,Done)>
```

Alternatively, the commitment can be handled purely procedurally by executing the procedure `SendReceiptProc(Customer)`:

```
<Customer: Know(Merchant,ReceivedPaymentFrom(Customer) →
            SendReceiptProc(Customer)
until Know(Merchant,Done)>
```

4.5 Capabilities

In this section, we discuss capabilities. Capabilities are a new notion introduced into the *i** modeling framework in this thesis. Goal capabilities and task capabilities map into epistemically feasible CASL procedures and are always guaranteed to achieve their goals/perform their tasks provided their *context conditions* hold when the capabilities are invoked and other agents in the environment do not interfere with the execution of the CASL procedures corresponding to the capabilities.

We introduce capabilities in Section 4.5.1, discuss physical executability and epistemic feasibility in Section 4.5.2, and describe the use of capability nodes in SD, SR, and iASR diagrams in Section 4.5.3. We talk about the mapping of goal capabilities into CASL and the formal restrictions on the CASL procedures corresponding to goal capabilities in section 4.5.4. The mapping of task capabilities is described in section 4.5.5. We discuss some capability-related issues in section 4.5.6.

4.5.1 Introducing Capabilities

In analyzing multiagent systems and the interdependencies among the agents in such systems, it makes sense to talk about the *capabilities* of agents. These capabilities are typically viewed as the services that the agents are able to provide. An agent with a certain capability is usually understood to have at least one plan to achieve the goal corresponding to that capability. Each plan may have preconditions associated with it or a *context*, which describes the circumstances under which the plan is to be used. Some researchers (e.g., [Sycara *et al.*, 1999]) use the term to describe what agents advertise to other agents in a multiagent system. Martin, *et al.* [Martin *et al.*, 1999] call generic goals that the agents are able to achieve *"solvables"* and describe them as being the capabilities of agents. These capabilities could be viewed as a high-level interface to the agents in a multiagent system.

We, on the other hand, use the term *capability* to refer to the goals that the agent can always achieve and the tasks that the agent can always perform in a certain context. Capabilities of an agent can be the ends of a top-level dependency, thus being part of the agent's interface to other agents in the system. Alternatively, capabilities can be used internally as sub-behaviours of agents. Capabilities will also include the context conditions and the specifications of other agents' compatible behaviour that specify when the capabilities are guaranteed to achieve their goals or perform their tasks. The capabilities have to be carefully specified to provide this guarantee. In a sense, we use Design by Contract [Meyer, 1997] here: if the user of a capability makes sure that its context condition holds (and the behaviours of other agents are compatible with the capability), the capability is guaranteed to succeed.

There are two types of capability nodes that we can use in diagrams: capabilities to achieve goals and capabilities to perform tasks. They are used for specifying goals that can always be achieved and tasks that can always be performed because the plans/procedures that the agent has for achieving the goals or performing the tasks are guaranteed to succeed in environments satisfying the context conditions of these capabilities and when no outside processes interfere with the capabilities. Note that the goal capability of achieving a goal `G` first acquires the goal using the appropriate `commit` action if the agent does not already have this goal in its mental state and then achieves it. Capability nodes can be used in both SD diagrams and in SR/iASR diagrams in *i** models.

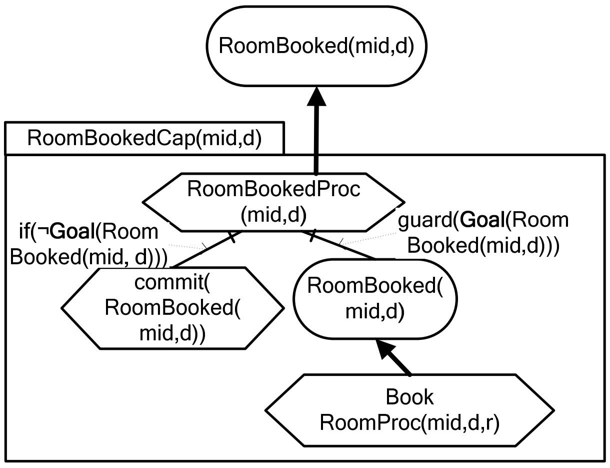


Figure 4.43. The goal capability `RoomBookedCap`.

Figure 4.43 is an example of a goal capability node used as a means to achieve a goal. It is taken from the meeting scheduling case study (see Chapter 6). Here, a goal capability `RoomBookedCap` is used as a means of achieving the goal `RoomBooked`. It is capable of booking a room for a meeting on a certain date. The top-level procedure of the capability is called `RoomBookedProc`. It is decomposed into a `commit` action and a goal node `RoomBooked`. The action is only executed if the particular instance of the goal `RoomBooked` with the parameters `mid` and `d` (meeting ID and date) is not in the mental state of the agent (see Section 4.5.4). Once the goal is in the mental state of the agent, `BookRoomProc` is executed. A thorough discussion of how capability nodes are used in iASR diagrams is presented in Section 4.5.3.

4.5.2 Physical Executability vs. Epistemic Feasibility

Recall that we imposed only very weak constraints on the achievements procedures for goals, i.e., that there be *some* situations where these procedures achieve the goal when no concurrent actions occurred (see Section 4.3.9.2). The key difference between regular goals and capabilities is that in the case of regular goals it is enough that the agent *tries* to achieve the goal and it is not required that the agent actually *succeed* in achieving the goal; in case of capabilities we require that that the agent always be able to achieve the goal or perform the task provided the corresponding context conditions hold. To ensure this, we require that the plans the agent has for achieving the goal/performing the task of the capability be both physically executable and epistemically feasible in all situations satisfying the context condition. Note that for ordinary goals, we do not even require physical executability of the means in all situations — we only require physical executability in some situation.

Since in our framework we assume that agents have incomplete knowledge, there may be plans that the agents are not able to execute due to the lack of knowledge. Thus, it is important that the ability of CASL to reason about knowledge provides us with the means to evaluate the *epistemic feasibility* of agents' plans [Lespérance, 2002] (see also Section 2.2.4). While some researchers (e.g., [Padham and Lambrix, 2000]) have proposed new mental state modalities to account for agent capabilities, in this work we characterize capabilities through the notion of epistemic feasibility. Here, agent capabilities are sub-behaviours that are carefully scripted to be both physically executable and epistemically feasible, meaning that the agent has to have enough knowledge to be able to successfully execute the behaviours. This should increase the designer's confidence in the routines' workability and the workability of other routines and dependencies that rely on these capabilities.

To illustrate the difference between the physical and the epistemic abilities to execute a program let us look at a small example. Suppose we want the agent to open a safe by dialling its combination, i.e.:

$$\pi\, \texttt{number((combination(safe1,number))?; dial(safe1,number))}$$

While the above program is physically executable, i.e:

$$\forall s\ \exists s'.\mathbf{Do}(\pi\, \texttt{number((combination(safe1,number))?;}\ \texttt{dial(safe1,number))}, s, s'),$$

it is not epistemically feasible if the agent does not know the combination, i.e.:

$$\neg\exists \texttt{number}.\mathbf{Know}(\texttt{agent,combination(safe1,number)},s).$$

The capability notation we use in *i** (see Figure 4.44) diagrams is inspired by the way packages are represented in UML [Rumbaugh *et al.*, 1999]. Similarly to UML packages, capabilities aggregate modeling elements into conceptual wholes. When an agent has a goal or a task capability, it means that the agent has an epistemically feasible as well as physically executable plan to achieve the goal or perform the task. All capabilities have context conditions that must be true for the capabilities to be guaranteed to succeed (see Sections 4.5.4 and 4.5.5 for more on this). These context conditions are static properties of capabilities. In most cases to guarantee successful execution of goal capabilities (i.e., to assure that the goal achievement procedure actually achieves the goal of a goal capability) one also needs to specify what actions by other agents are allowed to be executed concurrently with the execution of the CASL procedure corresponding to the capability. The specification of the allowed behaviour of other agents is also a static property of capabilities. The notation of Figure 4.44 shows just the goal/task that the capability achieves/executes. A more detailed notation showing the internals of the capability can also be used (see Figure 4.43).

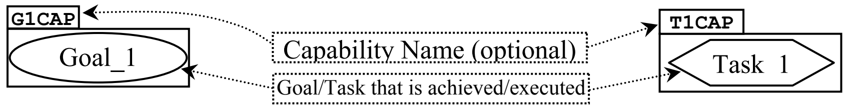

G1CAP
Goal_1
Capability Name (optional)
Goal/Task that is achieved/executed
T1CAP
Task 1

Figure 4.44a. A goal Capability.

Figure 4.44b. A task capability.

4.5.3 Using Capability Nodes in SD, SR and iASR diagrams

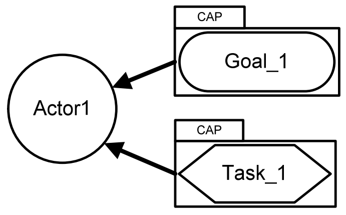


Figure 4.45. Using capability nodes in SD diagrams.

Capability nodes proposed in this thesis can be used in SD, SR, and iASR diagrams. Capability nodes in SD diagrams are used mostly to model the abilities of agents that are part of the environment of the system-to-be. These include humans, hardware devices, or legacy software systems. Figure 4.45 illustrates the use of capabilities in SD diagrams.

Here, the actor Actor1 has capabilities to achieve `Goal_1` and to perform `Task_1`. See Section 5.2.3 for more details on using capabilities in SD diagrams.

Capability nodes can be used anywhere task nodes are used in iASR diagrams, namely as subtasks of other tasks and means to achieve goals. The goal and task capability nodes in Figure 4.44, which simply show the task that the capability performs or the goal it achieves, can be used in SD, SR, and iASR diagrams, while capability nodes that show the internals of the capabilities, i.e., the detailed plan to achieve a goal or perform a task, can be used only in iASR diagrams (see Figure 4.46).

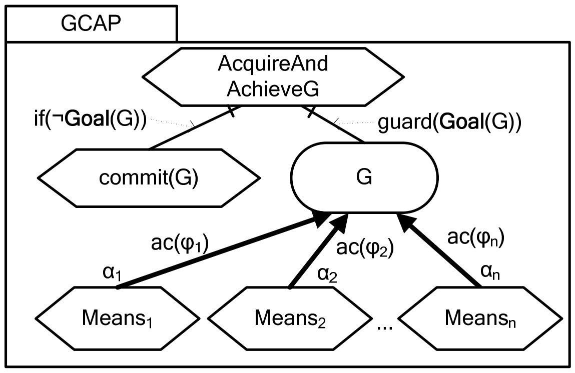


Figure 4.46 A goal capability node that shows the internals of the capability.

This more detailed view of the capabilities allows the designer to specify various means of achieving the goal of the goal capabilities and the detailed decompositions of the tasks that the task capabilities are able to execute. The task and means-ends decompositions are specified as usual. The usual iASR rules for means-ends decompositions apply, meaning that the means to achieve the goal of the capability can only be tasks and (nested) capabilities, that the only composition annotation available is the alternative, that the means can be labelled by the appropriate link annotations and applicability conditions, etc. (see Figure 4.46).

The ability to use capability nodes both with and without internal details allows the designer to concentrate on certain parts of the model while leaving other parts sketchy: on an iASR diagram, the capabilities that are not pertaining to the specific aspect of the agent behaviour being analyzed can be represented as single nodes showing just their goals or tasks, while other capabilities can be fully described through the usual goal or task decomposition facilities (see Figure 4.47):

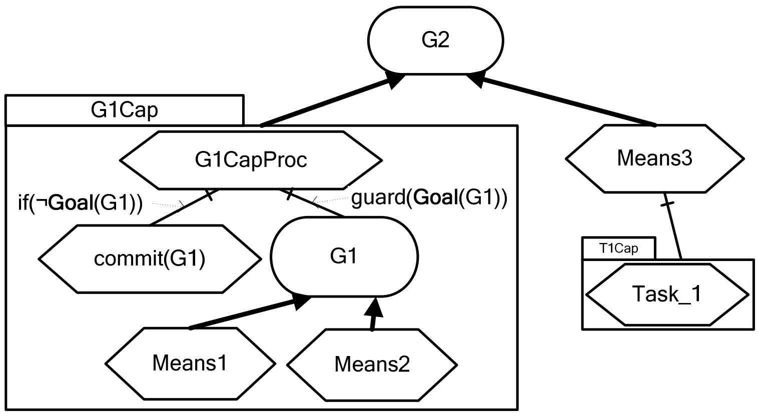


Figure 4.47. An iASR diagram with both types of capability nodes.

In Figure 4.47, the goal `G2` has two means of achieving it, a task and a capability. The capability `G1Cap` is modeled in full detail, showing how the goal `G1` is acquired and then achieved using two means, the tasks `Means1` and `Means2`. Here, the assumption is that achieving `G1` implies the achievement of `G2`. Another means of achieving the goal `G2` is the task `Means3`, which is decomposed into the task capability `T1Cap`. Note the iASR diagram does not show the details of the task capability (e.g., whether the task `Task_1` is decomposed into subtasks or not). It just shows that the capability executes the task `Task_1`.

As previously noted, a goal capability, which achieves the goal `G`, may acquire this goal by using the `commit` action. The reason for this is that we would like to treat capabilities as modules that contain not only the agent behaviour specification, but also all the necessary changes to the mental state of the agent. Since we want an agent to be aware of the goals that it is achieving, we would like it to have the goal that `G1` holds in its mental states if it is executing the capability `G1Cap` that achieves the goal `G1`. When an agent is executing one of its goal capabilities, it may or may not already have the goal of the capability in its mental state. For example, if the capability is used as a means of achieving some goal that has been previously acquired either through an intentional dependency or by executing a `commit` action, then the goal is already in the mental state of the agent and does not need to be reacquired.

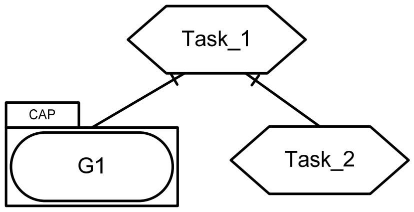


Figure 4.48a. Goal capability as a subtask.

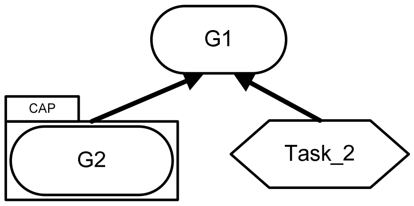


Figure 4.48b. Goal capability as a refinement of a parent goal.

On the other hand, there are cases that require the agent to acquire the goal of the capability by executing a `commit` action. For example, if a goal capability is used as a task in a task decomposition (Figure 4.48a), then the agent must acquire the goal of the capability since it is not in the mental state of the agent. Also, the goal capability that achieves `G2` can be used as a means of achieving the goal `G1` (it makes sense if by achieving `G2` the agent automatically achieves `G1`). In this case (Figure 4.48b), the agent must explicitly acquire the goal `G2` by using the appropriate `commit` action.

Therefore, the `commit` action in goal capabilities is annotated with an *if*-annotation that lets the agent acquire the goal of the capability only when it does not already have it in its mental state. The explicit overall structure of a goal capability is shown in Figure 4.49.

There, the goal capability `G1CAP` with its top task, `AcquireAndAchieveG1`, is responsible for both acquiring the goal `G1` when necessary and providing the means for achieving it. As it is usual with self-acquired goals, the goal node, which indicates the presence of a goal in the mental state of an agent, follows the task node for the `commit` action together with the corresponding guard annotation (self-acquired goals are discussed in Section 4.3.8). Finally, the means for achieving the goal `G1` are specified by the tasks $\texttt{Means}_i$, connected to the goal node by the means-ends links with the appropriate link annotations and applicability conditions (not shown). Since a goal capability has a task at its top level, it can be used as a task in a larger diagram.

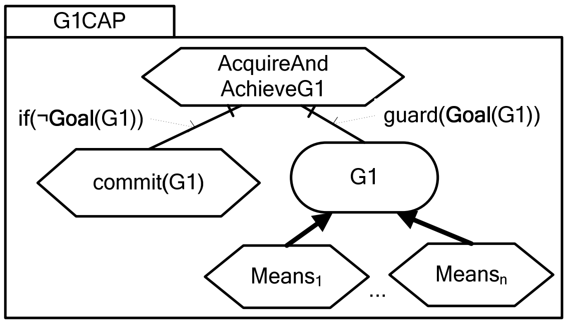


Figure 4.49. The full details of a goal capability node.

4.5.4 Mapping Goal Capability Nodes

A goal capability node will be mapped into a CASL formula that corresponds to the goal, a procedure that acquires and achieves the goal, a CASL formula that specifies the context condition (i.e., the circumstances under which the capability is guaranteed to achieve its goal), and a program that specifies compatible behaviour of other agents (Figure 4.50).

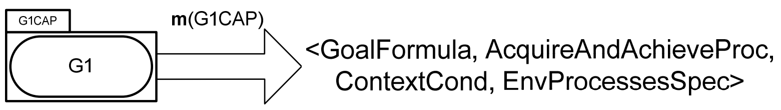

G1CAP
G1
m(G1CAP)
<GoalFormula, AcquireAndAchieveProc, ContextCond, EnvProcessesSpec>

Figure 4.50. The mapping of a goal capability node into CASL.

The following is the mapping rule for goal capabilities. The mapping **m** will map the goal capability `G1CAP` into the tuple:

```
        m(G1CAP) = <GoalFormula, AcquireAndAchieveProc,
                                   ContextCond, EnvProcessesSpec>,
```

where

- `GoalFormula` is a CASL formula with a free situation variable
- `AcquireAndAchieveProc` $\in$ *PName*$_c$ and `AcquireAndAchieveProc` is defined in P_c
- `EnvProcessesSpec` is a program that specifies the template of behaviour of other agents in the environment that is compatible with the achievement procedure of the capability
- `ContextCond` is a CASL formula with a situation variable

Thus, the mapping of a goal capability node is quite similar to the mapping of a goal node. The differences are the addition of the context condition and the specification of the allowed processes in the environment. Also, note that instead of `AchieveGoalProc` here we have `AcquireAndAchieveProc`, which is responsible for both acquiring the goal and achieving it. We think of a goal capability as a module, which is responsible for both acquiring and achieving the goal.

We use the following notation to access the goal formula, the achievement procedure, the compatible behaviour of other agents, and the context condition:

```
m(G1CAP).formula = GoalFormula
m(G1CAP).achieve = AcquireAndAchieveProc
m(G1CAP).context = ContextCond
m(G1CAP).envProc = EnvProcessesSpec
```

Constructing Achievement Procedures

Achievement procedures for goal capabilities are constructed in a similar way to the achievement procedures for goal nodes. Each achievement procedure is produced by mapping the means-ends decomposition of the goal that the capability achieves. The decomposition specifies the ways that the goal can be accomplished. As is the case with goal nodes, the mapping of the achievement procedures for the goal capabilities will take into consideration the applicability conditions that may accompany the task nodes, which represent various means for achieving the goals of the capabilities (see Figure 4.51).

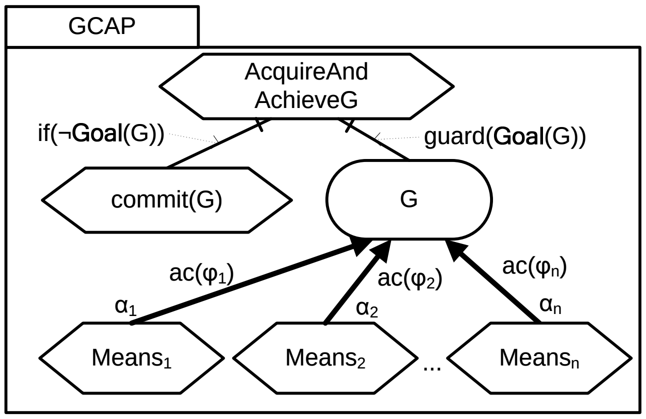


Figure 4.51. Specifying means to achieve the goal of a goal capability.

Figure 4.51 shows a generic goal capability `GCAP` that achieves the goal `G` and its means-ends decomposition that models various approaches for achieving the goal `G`. We show the full details of the goal capability here, including the acquisition of the goal `G`. Each of the means to achieve the goal, represented by the task $\texttt{Means}_i$, is accompanied by the applicability conditions φ_i and link annotations α_i. Suppose that **`m`**`(GCAP).achieve = GCAP_achieve`, then the code for `GCAP_achieve` is as follows:

```
proc GCAP_achieve($\vec{t}$)
    if ¬Goal(agt,m(G1CAP).formula($\vec{t}$)) then
            commit(agt,m(G1CAP).formula($\vec{t}$)) endIf;
    guard Goal(agt,m(G1CAP).formula($\vec{t}$)) do
        guard m($\varphi_1$) do m($\alpha_1$)(m(Means$_1$) endGuard
        |
          ...
        |
        guard m($\varphi_n$) do m($\alpha_n$)(m(Means$_n$) endGuard
    endGuard
endProc
```

The above procedure did not make use of the context condition of the capability GCAP. The condition is not used in the achievement procedures of goal capabilities or the procedures that perform the required tasks of task capabilities. Instead, the context condition for a capability can be used to guard the selection of the capability as a means to achieve a goal or a way to perform a task (similar to the applicability condition for means-ends links). In addition, a context condition can be used as an assertion that would help in proving properties about the system. When its context condition does not hold, a capability is not guaranteed to execute successfully. We use the generic example below (Figure 4.52) to illustrate this.

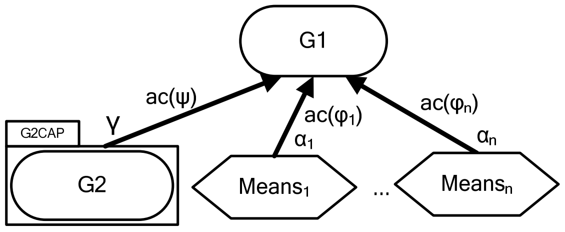


Figure 4.52. A goal capability used as a means to achieve a goal.

Figure 4.52 shows a means-ends decomposition of the goal G1. There are n+1 means for achieving G1: the goal capability G2CAP and n tasks Means_i. Each of the means for achieving G1 is accompanied by a link annotation: the goal capability has the link annotation γ, while each task Means_i has the link annotation α_i. Additionally, all of the means to achieve the goal G1 have applicability conditions: the capability G2CAP has the applicability condition ψ, while the tasks Means_i each have the condition φ_i. The applicability conditions specify, as usual, when it makes sense to use the means to try to achieve the goal G1. The achievement procedure for the goal G1 will then look as follows (suppose that **m**(G1).achieve = G1_achieve):

```
proc G1_achieve
    guard m($\psi$) do m($\gamma$)(m(G2CAP).achieve) endGuard
  |
    guard m($\varphi_1$) do m($\alpha_1$)(m($\text{Means}_1$) endGuard
  |
    …
  |
    guard m($\varphi_n$) do m($\alpha_n$)(m($\text{Means}_n$) endGuard
endProc
```

The context conditions of capabilities are mostly used to prove properties of the system. However, we do not require that the capabilities only be used when their context conditions hold. One may want to use a capability even though its context condition is not satisfied. In cases like this, there is no guarantee that the capability successfully achieves its aim. When the designer wishes to make sure that a capability is used only when its context condition is satisfied, he should make it part of the applicability condition. This will guarantee that the capability is not invoked when its context condition is not satisfied. The context condition for the goal capability G2CAP in Figure 4.52 will become the following:

```
m(G2CAP).context ∧ m(ψ)
```

Achievement Procedure Constraints

A CASL procedure that achieves the goal of a goal capability must be written much more carefully than a regular goal achievement procedure. We require that the procedure (provided that the context condition for the capability holds) be both physically executable and epistemically feasible. There is an important restriction on the achievement procedures for goal capabilities:

```
∀s.contextCond(s) ⊃
 AllDo(Subj(agt,AcquireAndAchieveProc ; Formula?) || EnvProcessesSpec,s)
```

The above constraint states that in all situations satisfying the context condition, every subjective execution (see Sections 2.2.4.1 and 2.2.4.2 for the discussion of subjective and deliberative execution of programs) of the achievement procedure of a capability by the agent in an environment in which processes specified by `EnvProcessesSpec` may occur terminates with the goal being achieved. We note here that for smart agents that deliberate and use lookahead, the **`Delib`**`(agt,δ)` notation can be used instead of **`Subj`**`(agt,δ)`.

The designer must determine exactly what behaviour guarantees the successful execution of the achievement procedure for a goal capability. One of the extreme cases is when `EnvProcessesSpec` is empty (i.e., `EnvProcessesSpec = nil`). In this case, no outside processes are permitted if one wants a guarantee that the achievement procedure succeeds. On the other hand, if `EnvProcessesSpec` is as follows:

$$\texttt{EnvProcessesSpec} = (\pi\ \texttt{act.(Agent(act)} \neq \texttt{agt)? ; act})^{\leq k}$$[2],

then we are saying that the goal will be achieved no matter what the other agents do. Of course, the specification for most capabilities will identify concrete behaviours for the agents in the environment, which assure the successful execution of the achievement procedure. For example, suppose an agent has a goal capability to fill a tank with water. It is guaranteed to succeed unless the tank's valve is opened. Here is the corresponding specification for the outside processes compatible with the capability:

$$\texttt{EnvProcessesSpec} = (\pi\ \texttt{act.(act} \neq \texttt{openValve)? ; act})^{\leq k}$$

As we mentioned previously, a goal capability that achieves the goal `G2` can be used to achieve another goal, `G1`, if `G2` is a refinement of `G1`. In other words, we can put the goal capability node `G2CAP` beneath a goal node `G1` as a means of achieving that goal if the following holds:

$$\forall \texttt{s}.\mathbf{m}\texttt{(G2CAP).context(s)} \wedge \mathbf{m}\texttt{(G2CAP).formula(s)} \supset \mathbf{m}\texttt{(G1).formula(s)}$$

4.5.5 Mapping Task Capability Nodes

The mapping rules for task capabilities are simpler than the rules for goal capabilities described above. Task capabilities are mapped into the corresponding CASL procedures that execute the task, the context conditions that, as before, describe the circumstances under which the capabilities are guaranteed to succeed, and the specifications of other agents' behaviour, which are compatible with them succeeding. It should always be possible to successfully execute the CASL procedures that task capabilities are mapped into, provided the corresponding context conditions hold and the actions of other agents

[2] $\delta^{\leq k}$ specifies that the program δ is iterated nondeterministically up to *k* times.

in the environment are restricted to the ones allowed by `EnvProcessesSpec`. These procedures are also required to be epistemically feasible.

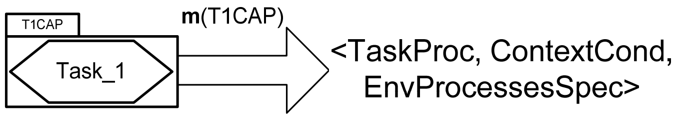


Figure 4.53. The mapping of a task capability node into CASL.

A task capability node will be mapped into a CASL procedure that corresponds to the task, a context condition that must hold for the procedure to be successfully executed, and a CASL program that describes the behaviour of other agents that is compatible with the procedure that the capability maps into (see Figure 4.53).

```
        m(T1CAP) = <TaskProc, ContextCond, EnvProcessesSpec>,
```

where

- `TaskProc` ∈ *PName*$_c$ and `TaskProc` is defined in P_c
- `EnvProcessesSpec` is a program that specifies the behaviour of other agents in the environment that is compatible with `TaskProc`
- `ContextCond` is a CASL formula with a free situation variable

The following notation is used to access the CASL procedure associated with the capability, the context condition, and the compatible behaviour of other agents:

```
m(T1CAP).procedure = TaskProc
m(T1CAP).context = ContextCond
m(T1CAP).envProc = EnvProcessesSpec
```

A task capability can be used anywhere a task node can be used in *i** diagrams. Figure 4.54 is an example of a task capability used as a subtask of another task.

In Figure 4.54, we have the task capability `STCAP` as a subtask of `ParentTask`. It is accompanied by the link annotation γ. The code for `ParentTask` is not different from the CASL code that the task would map into if none of its subtasks were capabilities. Below we show the part of the CASL procedure for `ParentTask` that involves the capability `STCAP` (suppose that **m**`(ParentTask) = ParentTaskProc`):

```
proc ParentTaskProc
   m(γ)(m(STCAP).procedure);
   …
endProc
```

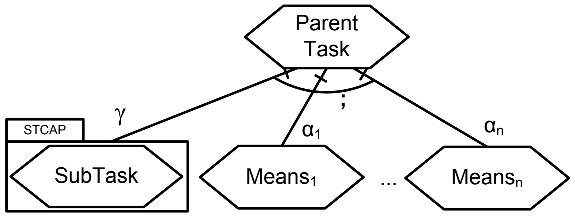


Figure 4.54. Task capability used as a subtask.

Constructing Procedures for Task Capabilities

The CASL procedure that the task capability is mapped into, `TaskProc`, is constructed just like a procedure for an ordinary task, taking into consideration the decomposition of the main task of the capability, along with the corresponding composition and link annotations. See Section 4.2.3 for details on the mapping of task decompositions into CASL.

Task Procedure Constraints

The procedure of a task capability must be successfully executable whenever the context condition holds at the time of its invocation and the actions by other agents in the environment are restricted to the ones specified by `EnvProcessesSpec`. The procedure must also be epistemically feasible. Thus, the constraint that guarantees the successful execution of the procedures of task capabilities is very similar to the constraint for goal capabilities:

```
∀s.ContextCond(s) ⊃ AllDo(Subj(agt,TaskProc) || EnvProcessesSpec, s)
```

4.5.6 Discussion

Goal capabilities have to be able to achieve their goals whenever their context condition holds at the time of invocation and provided that other agents in the environment are executing actions compatible with the achievement procedure of the capability. Therefore the designer needs to be careful in defining the goals of the capabilities, their context conditions, and the specification of compatible processes in order to avoid the situations where goal capabilities are supposed to achieve goals that are not, in fact, always achievable with the available plans. For example, the capability of an agent to book a meeting room might be conditional upon the availability of meeting rooms. Such goals must be properly *deidealized* (relaxed to make them always achievable).

While the context condition of a capability specifies the restrictions on the starting situation (where the capability is invoked), more research needs to be done on how to formalize some assumptions on what activities can and cannot be performed by other agents while the capability is running so as to avoid interference. However, our formalization of capabilities seems to be compatible with many ways of specifying allowable outside processes.

Using capabilities

Generally, there are agents that are being created as part of the system-to-be and there are agents that already exist in the organization or in the environment in which the system is to be situated. Therefore, it is necessary for our analysis to model the environment where our system is to be integrated. Capabilities provide a reasonable abstraction for describing agents that are part of the environment. This environment may include legacy systems that we want to interface with as well as human agents and we will need to include the relevant agents from the environment into our models. We may have quite detailed information about some of these agents, while having no information about the internals of the others. Either way, capabilities seem to be an appropriate formalism to model these agents. If we know that an agent that is part of some legacy system is able to achieve a certain goal or perform a certain task, we can model that by putting a capability box corresponding to the goal or the task inside the agent even if we don't know the internals of this capability. By doing this we state that we are confident that under certain conditions these agents will be guaranteed to succeed in executing their capabilities.

For the agents that are being designed as part of the system-to-be, turning some behaviours into capabilities requires the designer to be very careful in modeling those behaviours. On the other hand, capabilities guarantee that they can be successfully executed provided their context conditions hold at the time of invocation and no agent executes actions incompatible with the capability. This could be viewed as an instance of the Design by Contract principle [Meyer, 1997].

Capabilities as Components

The problem of software reuse is very important in software engineering. Software components are currently in use in all sorts of applications, from consumer-oriented to enterprise-level ones. The Internet opened the doors to an even wider use of component technologies (e.g., Enterprise JavaBeans [Sun Microsystems, 2002]). The domain of open, dynamic multiagent systems with agents that adapt to changes in the environment and switch their roles depending on their objectives is an ideal domain for componentization. Capabilities are good candidates for the specification of components. Agent capabilities seem to be at the right level of abstraction for the analysis of multiagent systems and exploration of software reuse possibilities. Pluggable behaviours could be analyzed and verified separately and building agents will involve loading modules with the required behaviour. Components could also very naturally support agents that change roles — the agents will just load the behaviours that correspond to the new role and start using them.

Based on the above discussion, it is quite desirable to have a formal framework that can support the notion of agent capability as a pluggable behaviour. The capabilities will then have an interface that clearly identifies the goal that the capability is able to achieve, the preconditions for the use of the capability, or the context of the capability. The capabilities will essentially be plans that achieve certain goals or perform certain tasks, possibly with their own local fluents and a fragment of the knowledge base in order to isolate the modules as much as possible. The capabilities then become "black boxes" that have precisely defined input and output and hide their internals. However, this is a potential problem since in many applications it is important to optimize the agents' activities, construct plans to achieve very high-level goals, and so on, which could be hard without the ability to peek inside the plans the agents are executing as part of their capabilities. It seems, therefore, that it makes sense to turn capabilities into "grey boxes", which reveal at least their high-level structure and can be tuned according to individual agents' demands. These are the topics that require further exploration.

4.6 Synchronizing Procedural and Declarative Components of CASL Agents

As discussed in Section 4.3.6, CASL agents have two components: the procedural specification of their behaviour and the declarative specification of their mental states. iASR diagrams represent the procedural component of CASL agents. Thus, the presence of a goal node in an iASR diagram states that the agent is aware of the goal being in its mental state and is prepared to deliberate on whether and how to achieve it. For the agent to modify its behaviour in response to the changes to its mental state, it must synchronize its procedural and declarative components.

Figure 4.55 is an overview of how the mental state and the process specification of a CASL agent can be synchronized. The process specification can be used to acquire goals and knowledge through the `commit` and the `assume` actions respectively. On the other hand, in order to modify the behaviour of the agent based on the changes to its mental state, one needs to use interrupts or guards.

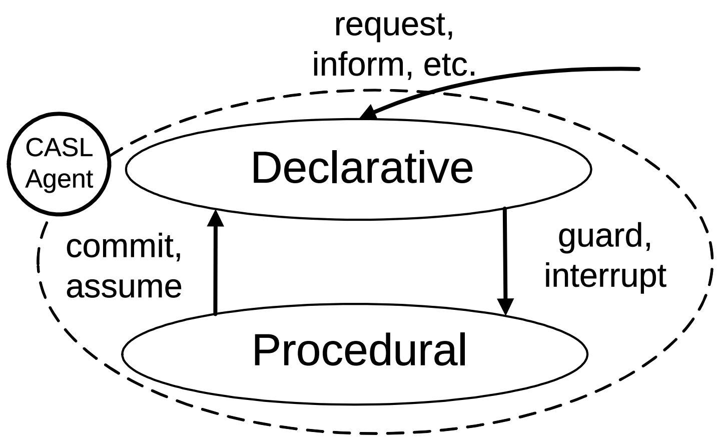


Figure 4.55. Synchronizing procedural and declarative components of CASL agents.

5 The Combined *i**/CASL Methodology

In this chapter we describe our requirements engineering methodology that combines the *i** modeling framework [Yu, 1995] with the Cognitive Agents Specification Language (CASL) [Shapiro and Lespérance, 2001]. The standard *i** modelling framework with its Strategic Dependency and Strategic Rationale diagrams is used first for the early phase and some of the late phase of the requirements engineering process, while the Intentional Annotated SR (iASR) diagrams are used for the late phase of RE. The iASR models are then mapped into CASL and the CASL agent programming language is used for validation and verification of the iASR models.

5.1 Introduction

*i** is an informal graphical framework very well suited for both early and late requirements analysis stages. It has powerful facilities for capturing the intentional and strategic aspects of the operational environment of the system-to-be as well as for describing the intentional aspects of the system-to-be itself. The network of inter-actor dependencies produced by the *i** approach is a tool for analyzing various (possibly alternative) responsibility assignments in the system. *i** supports reasoning about functional and non-functional dependencies among actors as well as the analysis of goal and softgoal decompositions aimed at finding the best alternative for achieving these goals and softgoals. We think that CASL naturally complements *i** since it is amenable to formal analysis and supports reasoning about the goals and knowledge of the agents. It provides agent-oriented verification and simulation environments that support the intentional notions of *i**, thus allowing us to incorporate these notions further in the software engineering process.

While a certain portion of the *i** modeling framework has been formally represented in the modeling language *Telos* [Mylopoulos *et al.*, 1990], many aspects of *i** models still do not have precise semantics (e.g., a task decomposition). In the previous chapter, we presented an approach that aims at disambiguating Strategic Rationale diagrams of *i** and increasing the detail level of the diagrams to the point that they can be straightforwardly mapped into the corresponding CASL models. These models can then be formally verified using the CASLve tool [Shapiro *et al.*, 2002] in order to discover inconsistencies within the requirements. The formal analysis of software requirements greatly increases the chances that errors and inconsistencies in the requirements for software systems are found and fixed, thus avoiding the need for costly software modifications that result from these errors propagating into the design and implementation of the system.

The assumption that our requirements engineering methodology makes is that modeling and analysis cannot be performed adequately in isolation from the organizational and social context in which a new system will have to operate. Our proposal generally follows the *i** methodology for requirements engineering [Yu, 1995] and the *i**-ConGolog methodology [Wang, 2001], but includes several important enhancements. There are two types of enhancements that we make to the standard *i** approach. The modeling improvements include, among other things, the use of self-dependencies, capability nodes, the *System* agent, and, most importantly, the iASR diagrams and their mapping into CASL models. This RE methodology can be integrated quite easily into the Tropos agent-oriented development methodology [Castro *et al.*, 2002]. Not unlike the Tropos methodology, we are advocating the use of RE notions throughout the development process. The emphasis of the methodology is on the phase of the requirements engineering process that takes place before the formulation of the initial requirements. Our goal here is to understand the "whys" of the requirements [Yu and Mylopoulos, 1994]. We analyze why the system is needed, how the interests of the stakeholders can be addressed, how the system would meet its organizational goals, what the alternatives for the system are and how they affect the stakeholders [Yu, 1997]. Since stakeholder goals are the origin of requirements for the system-to-be, we see requirements-driven software engineering as well as goal-based Requirements Engineering as extremely promising approaches. We use the *i** notation to capture the intentions of stakeholders, the responsibilities of the system towards these stakeholders, and some aspects of the architecture of the system. CASL is used for the formal analysis of the system.

To illustrate the methodology we will use some generic examples as well as examples from a system for scheduling meetings within an organization. It is a variation of the system described in [Yu, 1995]. For each meeting request the system will try to find a date from the preferred date range that satisfies all of the intended participants. Once such date is found, the meeting is confirmed and an appropriate room for it is booked. The system is presented in detail in Chapter 6 of this thesis. Below we present an overview of the methodology.

5.1.1 Methodology Overview

Let us briefly survey the main elements of our methodology. We provide the details in the rest of the chapter.

1. *Early Requirements* (Section 5.2). The analysis of the organizational environment for the system-to-be is performed with the help of the *i** notation. Stakeholders and their goals are identified (Sections 5.2.1, 5.2.2) together with the intentional dependencies that exist in the organization (Section 5.2.3). SD diagrams are used to help in collecting, representing and analyzing this information. SR diagrams may also be used during the early phase of RE, providing more details about individual stakeholders and their relationships. The importance of non-functional requirements is stressed in Section 5.2.4. New modeling elements, *capability* nodes, are proposed for use in both SD and SR

models (Section 5.2.3). Additional organizational details can be added with the help of the *Serves* and *Trusts* relationships, which are discussed in Section 5.2.5.

2. *Late Requirements with SD Models* (Section 5.3). The system-to-be is introduced into SD models through one or more actors (Section 5.3.1). The intentional dependencies are adjusted. The possible system-environment boundaries are analyzed. The use of the new special *System* agent is proposed in Section 5.3.2.

3. *Architectural Analysis* (Section 5.4). Organizational architectural style is selected as in [Fuxman *et al.*, 2001a] and analyzed using the NFR framework [Chung *et al.*, 2000]. The system architecture is then analyzed. Agent may be decomposed into sub-agents to reduce the load, improve the communication patterns, etc. Architectural patterns may also be used as suggested in [Castro *et al.*, 2002]. The agent-oriented system architecture may be modified based on the feedback from the SR-level analysis.

4. *Late Requirements Using SR Models* (Section 5.5). Agents' processes are analyzed. SR diagrams are used to model the rationale for and motivation of individual agents. Alternative process configurations are identified and analyzed based on their contributions to softgoals. Intentional Annotated SR (iASR) diagrams are then produced (Section 5.5.2). Annotations are added to the SR diagrams to provide more details and precision and goal decompositions are modified (Section 5.5.6) for easy mapping into CASL specifications. Goal dependencies are not abstracted out (Section 5.5.3), but converted into the corresponding interactions between agents, while goals are deidealized and softgoals are suppressed (Section 5.5.4). Agent interactions are then detailed (Section 5.5.5). The use of capability nodes is described in Section 5.5.7.

5. *Requirements Verification Using CASL* (Section 5.6). iASR diagrams are mapped into CASL (Section 5.6.1) and analyzed using CASLve (Section 5.6.2) and possibly other tools. Since the methodology supports requirements traceability quite well (Section

5.6.3), problems found during the formal analysis of the CASL specifications can be used to appropriately modify the SD and SR models.

5.2 Early Requirements: Understanding the Organizational Setting

The first step in coming up with a correct, unambiguous, and precise requirements document is to analyze the environment in which the system-to-be is intended to be situated. The phase of the methodology during which these activities are performed is called the early requirements phase.

In most software projects, the goal is to create a system that improves the processes in the organization in some way or another. In order to create a system that achieves this improvement and meets the expectations of its users, we need to understand the environment of the proposed system, the existing processes, and the goals of the players in the organization. Therefore, the first step is to analyze the relationships among the stakeholders in the organizational environment as it exists before the introduction of the system-to-be. The way we approach the problem of analyzing the environment of the system as well as the system in its environment is through *modeling*. Modeling is the process of constructing abstract descriptions, possibly formal, that are amenable to interpretation. Modeling has many benefits. It facilitates in requirements elicitation by guiding it and helping the requirements engineer look at the domain systematically. Models can also be a measure of progress since usually the more complete the model is, the more complete the elicitation is. Moreover, inconsistencies in the model are indicative of conflicting or infeasible requirements, disagreements among stakeholders, or confusion over terminology. Models are an invaluable tool for checking the understanding of the environment/system by the requirements engineer. Formal models can be tested to determine whether they have the desired properties and consequences and whether they are consistent; they can also be animated to help in validation of the requirements. In this phase of the requirements engineering process, we use the *i**'s Strategic Dependency models to model the intentional aspects of the organization.

The early phase of requirements engineering deals with enterprise modeling. Here, we model the organization in which the development takes place and/or in which the system will operate. Enterprise modeling and analysis deals with understanding an organization's structure, "the business rules that affect its operation, the goals, tasks and responsibilities of its constituent members, and the data that it needs, generates, and manipulates" [Nuseibeh and Easterbrook, 2000].

5.2.1 Identifying Stakeholders

We first identify the stakeholders in the environment of the system-to-be. There are many definitions of the term *stakeholder* in the literature. Nuseibeh and Easterbrook [Nuseibeh and Easterbrook, 2000] define stakeholders as "individuals or organizations who stand to gain or loose from the success or failure of the system", while one of the definitions presented in [Kotonya and Sommerville, 1998] states that "system stakeholders are people or organizations who will be affected by the system and who have a direct or indirect influence on the system requirements".

In general, the stakeholders are customers who order the system, developers who design the system, and users who interact with the system to achieve their goals. The stakeholders are modeled as *actors* in the *i** framework. The actors could be entities (i.e., people, software agents) or roles (e.g., "meeting participant"). The distinctions among an agent, a role, and a position are discussed in Section 2.1.1.

The identification of stakeholders is an informal process, which is usually assumed to be fairly straightforward and most requirements engineering methods (e.g., KAOS

[Dardenne *et al.*, 1993]) do not support the stakeholder identification activities per se. However, in many cases coming up with a set of stakeholders is not a trivial task. [Sharp *et al.*, 1999] attempt to devise a domain-independent, effective, and pragmatic approach to discovering the relevant stakeholders of a specific system. This problem is out of the scope of this thesis, so we assume that an appropriate technique for discovering stakeholders is used at this stage of the methodology.

In our meeting scheduler system, we identify the Meeting Initiator (MI), which is a role within an organization. This means that many individual agents can play that role and schedule meetings. The Meeting Initiator is the actor that is most interested in the meetings taking place and thus will initiate the process of arranging a meeting. The second stakeholder is the Meeting Participant (MP), a role played by agents that are requested to participate in meetings by the Meeting Initiator. The third stakeholder is an agent called the Meeting Room Booking System (MRBS). This represents a software system that already exists in the organization and whose aim is to track the availability of meeting rooms. Figure 5.1 shows the above stakeholders using the *i** notation:

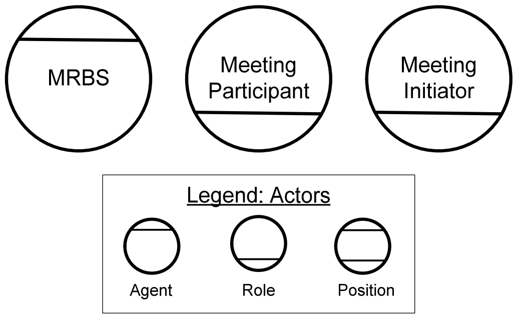


Figure 5.1. The stakeholders of the Meeting Scheduler System.

The MRBS agent in the above figure is an example of a legacy software system that we model as a stakeholder in the organizational model. Depending on the application, it may be useful to model legacy systems, people, and even hardware devices as stakeholders. Generally, a person or an entity can be an actor in an *i** Strategic Dependency model if it can be assigned responsibility for goals/subgoals of other actors in the environment of the system-to-be or the system-to-be itself, or depends on these actors.

5.2.2 Discovering Stakeholder Goals

At this stage, we model the existing processes in the organization since analyzing the way the current system/organization works is crucial in determining the requirements for the system-to-be. Upon identifying the stakeholders in the system we can analyze each stakeholder in terms of its goals. Whether the actors participating in the system are self-interested or altruistic, they all have their objectives and strive to achieve these objectives. Identifying the stakeholders' goals is therefore an important step towards understanding the system rationale. Eliciting goals focuses the RE engineer on the needs of the stakeholders, rather than on the possible solutions satisfying these needs. Once we have identified the stakeholders, we can start analyzing what each stakeholder brings to the organization and what it expects from the other actors in the organization. This is described in terms of the delegated goals/tasks/resources — the goals the actor wants others in the organization to achieve for it, the tasks it wants performed, and the resources it wants furnished.

In order to help the modeler gradually document the goals of the stakeholders we propose the use of the following notation (Figure 5.2):

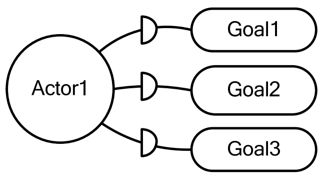


Figure 5.2. An actor, whose goals are not yet delegated to other actors.

Here, we have a stakeholder called Actor1 with three goals: `Goal1`, `Goal2`, and `Goal3`. These goals have been identified by the modeler, but have not yet been made part of any intentional dependency. Upon finding the right actor to delegate a stakeholder goal to, the modeler will draw an arrow from the goal node to another actor, thus completing the dependency (note that here we are modeling the *existing* system, so finding the actor to delegate a stakeholder goal to amounts to discovering the existing business processes within the organization). This minor addition to the *i** notation is useful in some of the modeling activities. It can be used during the early requirements phase while discovering the goals of stakeholders. We can also use the same notation later when the system-to-be is introduced into the models. At that point, the dependencies in the organization are changed to include the actors of the system-to-be. The notation of Figure 5.2 will help by showing the goals of the actors graphically. Figure 5.3 shows the goals of Meeting Initiator. It needs to schedule a meeting and also book a conference room for that meeting.

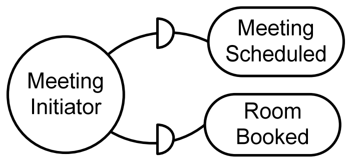


Figure 5.3. The Meeting Initiator agent with two of its goals.

5.2.3 Discovering Intentional Dependencies among Actors

In *i**, intentional dependencies are extremely important. When an actor cannot achieve one of its goals or cannot achieve it as efficiently as some other actor in the organization, it can choose to delegate the achievement of the goal to another actor capable of achieving it. While this may be advantageous to the delegating actor, the actor becomes vulnerable if the delegated actor fails to achieve this goal.

Dependencies represent such patterns of social relationships in the system. They act as the "glue" that binds the system together. Without dependencies, the system does not exist as a whole. The absence of inter-actor dependencies in the system/organization indicates that the actors are capable of achieving their goals, performing their tasks, and providing the needed resources on their own, without any help from others. This means that there is no cooperation, communication, coordination, etc. among the individuals. But these notions are fundamental for human organizations. In the domain of human organizations (and modern human society in general), no individual has all the capabilities and resources to satisfy his needs, so he is forced to cooperate with others by delegating some of his goals to other actors in the society while taking on responsibilities to assist others with the achievement of their own goals. Thus, the dependencies (both incoming and outgoing) are what makes an individual/organization/etc. part of the society. Modeling inter-actor dependencies is therefore of foremost importance to a requirements engineer.

The *i** approach allows us to analyze the goals of the actors and their strategic dependencies, thus capturing the intentional aspects of the system-to-be and its organizational environment. *i** has four types of dependencies among actors: a goal dependency, where a depender delegates the achievement of a certain goal to a dependee, a task dependency, where the execution of the task is delegated to a dependee, a resource dependency, which indicates that the dependee is expected to provide a resource for the depender, and a softgoal dependency, where the dependee achieves a goal that is expressed qualitatively. As mentioned above, by depending on other actors an actor gains the opportunity to achieve some of its goals that it cannot achieve otherwise, or to achieve them goals more efficiently. At the same time the depender becomes vulnerable if the dependee is unable/unwilling to provide the dependum (achieve the requested goal, provide the resource, or perform the task).

At this stage, the modeler analyzes the organization (the environment for the system-to-be) and determines what dependencies exist among the stakeholders. The Strategic Dependency model of the organization is updated accordingly. To discover the intentional dependencies in an organization we analyze how the goals of the actors in the organization are achieved through delegation to other actors. As already mentioned, the network of intentional dependencies that exists in an organization represents the existing business processes in that organization.

The figure below (Figure 5.4) is a modified version of Figure 5.2 where one of the goals of Actor1 is delegated to Actor2. This diagram shows that the modeler has determined that Actor1 depends on Actor2 for the fulfillment of the goal dependency `Goal1`, while the dependencies involving the other two goals of Agent1 have not yet been modeled in the diagram.

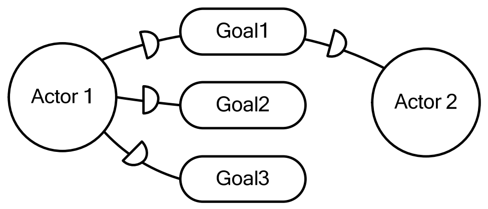


Figure 5.4. An actor with one goal delegated to another actor.

Self-dependencies are a useful addition to the *i** notation, one which we think will help modelers analyze intentional dependencies among the actors in the environment and later reorganize them after the system-to-be is introduced. Self-dependencies indicate that a stakeholder needs to achieve a certain goal, get a certain resource, or perform a certain task, and that currently it is the stakeholder itself who is responsible for achieving the goal, furnishing the resource, or performing the task. Figure 5.5 shows a possible model of the environment of a meeting scheduling system. Meeting initiators need intended meeting participants to attend meetings. Also, the initiators need to schedule their meetings and book the conference rooms themselves.

Another addition to SD diagrams that we would like to propose is the use of *capability nodes* (see Section 4.5 for a thorough discussion). Capabilities are meant to indicate that the agent can always achieve the goal or successfully execute the task specified within the capability node, provided that certain context conditions hold. This is quite similar to Design by Contract [Meyer, 1997] in the sense that if the depender makes sure that the corresponding context condition holds, the dependee in turn guarantees to provide the dependum. Thus, capabilities provide higher level of assurance to the depender than regular goals or tasks. Capability nodes at the SD level are used primarily for specifying the capabilities of the agents that are part of the environment of the system-to-be. These could be humans, hardware devices, or legacy software systems. During the late requirements stage one of the tasks of the modeler is to identify the actors of the system-to-be and determine what their responsibilities are going to be. In contrast to this, the actors that are part of the environment of the future system already have certain behaviour built into them. This behaviour is most easily identified for hardware devices and software systems. To model the capabilities of the environment-based agents it is useful to use task or goal capability nodes (see Figure 5.6).

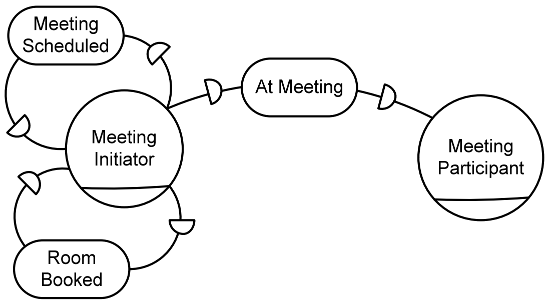


Figure 5.5. The model of the environment for the meeting scheduler system.

Figure 5.6 shows the actor Actor1, which has the capability to achieve `Goal_1` and the capability to perform `Task_1`. We use capabilities to describe the services of the environment agents when we are sure that these services can always be delivered provided certain specified context conditions hold. A good example of a capability is a method that is part of the interface of some software component. Provided that certain preconditions hold at the time this method is invoked, it is guaranteed (assuming the supplier of the component verified it properly) to deliver the expected result. Alternatively, capabilities can be used with the agents of the system-to-be to specify the goals/tasks that those agents must always be capable of achieving/executing.

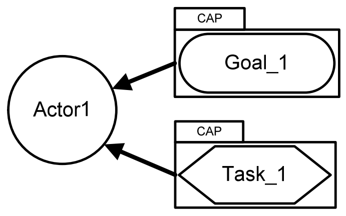


Figure 5.6. The use of capability nodes in SD diagrams.

While a lot of the information in the early requirements phase can be modeled and analyzed using just the SD diagrams, to get the complete understanding of the organization modelers might need to use the Strategic Rationale diagrams for more detailed look at the environment of the system-to-be. The SR models help in analyzing individual actors, their rationale and their intentions, and can reveal previously unidentified dependencies among the stakeholders. This information can be used to update the corresponding SD diagrams.

5.2.4 Analyzing Non-Functional Dependencies

An important aspect of requirements engineering is the analysis of non-functional requirements (NFRs). These are sometimes called quality requirements. *i** supports reasoning about non-functional requirements through the use of *softgoals*. There is no clear-cut satisfaction condition for a softgoal. Softgoals are related to the notion of *satisficing* [Simon, 1981]. Unlike regular goals, for softgoals one needs to find solutions that are “good enough”, where softgoals are satisficed to a sufficient degree. High-level non-functional requirements are abundant in organizations and quite frequently the success of systems depends on the satisficing of their non-functional requirements. While there are frameworks that deal exclusively with NFRs and allow their systematic treatment (e.g., the NFR-Framework [Chung *et al.*, 2000]), *i**'s support for the modeling of NFRs and their effect on systems and organizations is quite strong. During the early requirements engineering phase the modeler identifies the stakeholder goals that can be either regular goals, which lead to functional requirements, or softgoals, which lead to non-functional requirements.

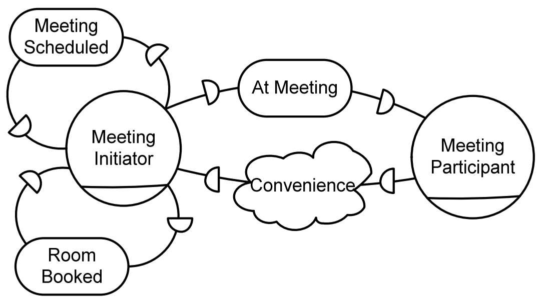


Figure 5.7. Modeling Non-Functional Dependencies.

Figure 5.7 shows an example of a softgoal dependency among the actors in the environment of the system-to-be. Here, the Meeting Participant actor wants the meeting, which is being scheduled by the Meeting Initiator agent, to be convenient in terms of time and place. `Convenience` is clearly a goal that can be achieved to various degrees. If one has many softgoals with the same name (e.g., `Accuracy`) related to different processes, it might be convenient to indicate which goal/task/resource the softgoal is related to (e.g., `Accuracy(RoomBooked)`).

At this stage, the objective of the modeler is to identify the softgoal dependencies among the actors in the environment. These dependencies will remain in the model with the introduction of the system-to-be. Softgoals, whose achievement is assigned to the system later in the process, will naturally become non-functional requirements.

Generally, softgoals guide the modeler in the selection of different system configurations or design/process alternatives. Softgoals will be either abstracted out at some point in the requirements engineering process having helped in such selection, or will be operationalized into some set of regular goals (whose satisfaction is easy to monitor for) and/or tasks, whose achievement/execution approximates the softgoal. *i** supports reasoning about non-functional requirements during the early and late requirements phases through the use of softgoal dependencies in Strategic Dependency diagrams. Strategic Rationale diagrams provide facilities to show positive or negative contributions of various alternative configurations to the softgoals.

5.2.5 More Organizational Details: The *Trusts* and *Serves* Relationships

While most effort at the early requirements stage in our methodology is concentrated on modeling stakeholders, their intentions, and the intentional dependencies in the organization using the *i** notation, modeling other organizational details could be useful as well. There are many organizational and social aspects in an organization that may influence the design, architectural, deployment, etc. choices for the system-to-be. A lot of these aspects (e.g., security, efficiency, etc.) will give rise to non-functional requirements, which we will talk about in the next section. Taking these aspects into consideration during the design of the new system helps in building a system that matches the organization as closely as possible and satisfies the needs of the stakeholders in the best possible way. Some properties of organizations can be captured (albeit in a simplified form) by models. In the previous chapter, we have introduced the *Trusts* (Section 4.4.4) and *Serves* (Section 4.3.7.3) relationships. The use of these relationships was described in conjunction with CASL models, where they can be used to address concerns such as security. On the other hand, we believe that the *Serves* and *Trusts* relationships can come into play earlier in the requirements engineering process, during early requirements phase. The *Serves* relationship can be used to approximate the managerial structure of an organization, while the *Trusts* relationship can be used to coarsely model some of the privacy and information security policies of an organization. The *Serves* and *Trusts* relationships are directed relations that could be represented as directed graphs. The

Serves relationship specifies, for each actor, which other actors it is willing to help in the achievement of their goals. Similarly, the *Trusts* relationship specifies, for each actor, which actors in the organization it believes. While in the early requirements phase, the *Trusts* and *Serves* relationships are intended as a rough approximation of the state of affairs in the organization, much more detailed versions of these relationships could be introduced later. The requirements engineer can start modeling by stating, for example, that one particular actor named Actor1 generally trusts Actor2, or that Actor1 is generally helpful to Actor3. The relationships can be refined when more details are needed: Actor1 trusts Actor2's information if it related to some specific topic, and Actor1 is helpful to Actor3 in achieving a certain set of goals. We do not insist on any particular modeling notation to represent these relationships. For example, the figure below shows a graph with three actors with arrows representing the *Trusts* or *Serves* relationships among them. We can see that Actor1 is helpful to Actor2 and Actor2 and Actor3 are mutually helpful to each other.

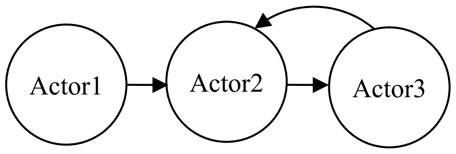


Figure 5.8. A possible graphical representation for the *Trusts* or *Serves* relationships.

This simple graphical representation for the *Serves* relationship applies only in the case of actors being *generally* helpful to other actors. Representing more detailed relationships among actors as described above requires more complex diagrams or multiple diagrams. For example, one can have as many directed graphs as there are stakeholder or agent goals. On the other hand, one can label the arrows in the graph with particular goal(s). The same applies to the *Trusts* relationship: its general form can be easily represented by a directed graph, while more detailed versions will require more complex notations. We acknowledge that more work needs to be done to come up with a simple graphical notation for representing the *Serves* and *Trusts* relationships, but this is outside of the scope of this thesis. As we demonstrated in Sections 4.3.7.3 and 4.4.4, these relationships, however detailed, can be easily mapped into CASL code to allow for more detailed analysis of the CASL model of the system-to-be.

While modeling the *Trusts*/*Serves* relationships is optional, we feel that doing this early on may allow the modeler to understand the organization better. The level of detail of the models of these relationships can be adjusted to the modeler's needs: they can be very high-level during the early requirements phase and get more and more detailed as the RE process moves closer to requirements validation.

5.3 Late Requirements Using SD Diagrams

While in the early requirements phase we concentrated on modeling the existing system/organization, in the late requirements phase of the software engineering process we essentially model the required future system. Nuseibeh and Easterbrook [Nuseibeh and Easterbrook, 2000] note that the distinction between modeling an existing system, and modeling a future system is an important one, and is often blurred by the use of the same modeling technique for both. They also remind us that early structured analysis methods suggested that one should start by modeling how the work is currently carried out (the current physical system), analyze this to determine the essential functionality (the current logical system), and finally build a model of how the new system shall operate (the new logical system). While all three models are rarely built, we stress the importance of distinguishing the model of the current state of the organization built in the early requirements phase from the model of the future state of the organization, which is built in the late requirements phase and incorporates the new system.

The output of late requirements analysis is the requirements specification, which describes all the functional and non-functional requirements of the desired system [Castro *et al.*, 2002]. In this phase of the requirements engineering process, the system-to-be is introduced into our models as one or more actors that help in the achievement of stakeholder goals.

5.3.1 Introducing the System-To-Be

At this point in the requirements engineering process, there is a model of the organization or the environment of the system-to-be, which is agreed-upon by the stakeholders. As we mentioned before, there must be a perceived need for improvement to the current state of affairs in the organization as well as some room for such change or improvement. The improvement will come from the introduction of the new system into the organization. During this phase, we continue to use *i**'s Strategic Dependency models.

The system-to-be can be introduced as either one actor or a collection of actors. This may depend, among other things, on the level of understanding that the modeler has about the new system: the structure of the system may or may not be already apparent to the requirements engineer. Thus, the system is one or more actors who contribute to the fulfillment of stakeholder goals [Castro *et al.*, 2002]. In later analysis, the system-to-be may be reorganized into a new set of actors.

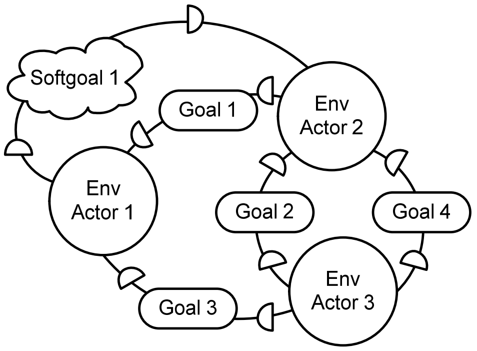


Figure 5.9a. System environment with three actors.

We use an example to illustrate how intentional dependencies in the environment can change with the introduction of the system into an SD model. Figure 5.9a shows a model for a system environment consisting of three actors, Env Actor1 through Env Actor3. There are four goal dependencies among the agents as well as one softgoal dependency.

Figure 5.9b shows what might happen to the model after the actor representing the system has been added to it. The reconfiguration of the intentional dependencies that existed in the environment is the main activity at this stage. Some of the dependencies among the stakeholders may disappear since these actors are now depending on the system-to-be for the achievement of their goals, while other dependencies between the pairs of stakeholders will remain intact. Moreover, the system-to-be can also depend on the actors in the environment.

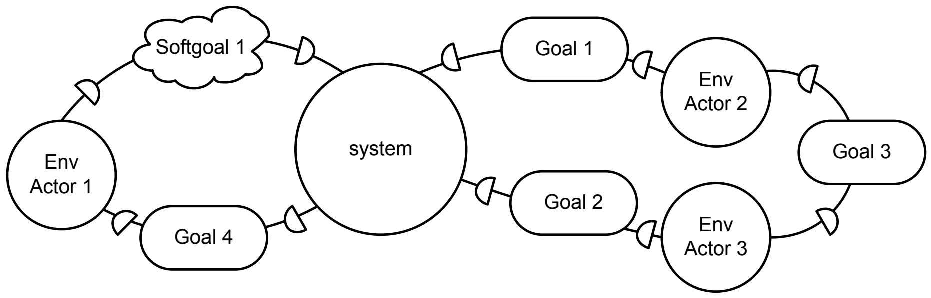


Figure 5.9b. A system is introduced into the environment.

It is important to note that the boundary between the environment and the system is floating during this phase of the RE process. This is one of the key ideas of the *i** approach since it allows the modeler to explore all the possible configurations of the combined system consisting of the environment and the system-to-be to find the one which satisfies stakeholder goals in the best possible way.

While analyzing the system and its environment, we must take into consideration the point of view of the environment and the stakeholders as well as the point of view of the system. Putting too much functionality into the new system may make the agents in the environment extremely vulnerable should the system fail to achieve their expectations. On the other hand, expecting too much from the environment may be unrealistic [Yu, 1997]. The solution is to look at the opportunities that the system-to-be offers for the stakeholders as well as the vulnerabilities that the stakeholders will expose themselves to if they start depending on the new system, and try to find the configuration that utilizes the most opportunities while mitigating against the vulnerabilities. Such analysis may shift the boundary of the system and the environment and, in the words of [Yu, 1997], "redistribute the pattern of intentionality". When the redesigned network of inter-agent dependencies is agreed upon, it will assign certain responsibilities to the agents of the system-to-be and to the actors in the environment. The former will become the requirements for the new system, while the latter will become the assumptions about the environment.

At the later stages of the requirements engineering process, Strategic Rationale diagrams are used to model the reasoning that the actors go through in achieving their goals as well as the goals delegated to them by other actors. The goals are decomposed using means-ends or task decompositions, or AND/OR goal refinements. As more details of the agent processes are introduced, new inter-agent dependencies may appear. The network of intentional dependencies may change due to the new understanding of the system-to-be gained through the detailed SR analysis. Therefore coming up with the right network of intentional dependencies in the system is an iterative process.

5.3.2 The *System* Agent

It is customary for the analyst using the *i** framework to first introduce the system as a single agent and then refine it into a collection agents. This way, the stakeholders are first modeled as being dependent on a single system agent as illustrated by Figure 5.9b. Once more system sub-agents are introduced, the intentional dependencies from the stakeholders to the system are adjusted to utilize these sub-agents. Figure 5.10a below illustrates this.

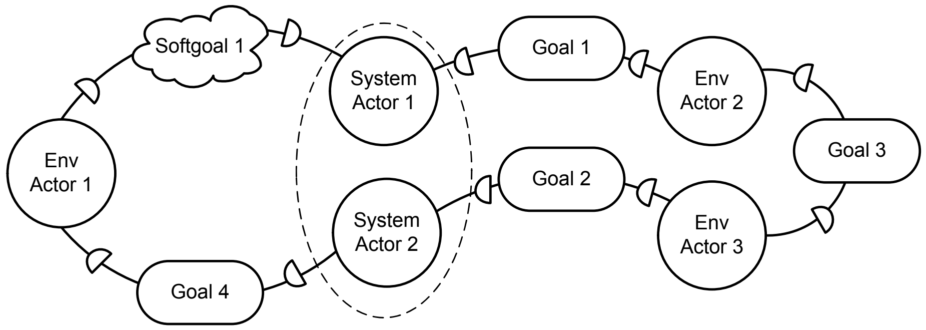


Figure 5.10a. The system is refined into several sub-agents.

This approach usually works, but frequently it is quite unnatural for a stakeholder to depend on a system sub-agent. The reason is that the stakeholder depends on the system as a whole and is not (or does not want to be) aware of the exact system architecture. For example, the user of an automated meeting scheduling system depending on it for the scheduling of the meetings and the booking of the appropriate conference rooms does not know that the booking of the rooms might be the responsibility of a special system sub-agent. This user depends on the system *as a whole*, not on its component, and it is the system that in turn depends on its component for the booking of the rooms. Thus, having a stakeholder depend on a sub-agent of the system-to-be may not be the correct modeling of the situation.

We propose the use of a special agent called *System*. This agent represents the system as a whole and can be used, among other things, to remedy the situation described above. In such situations, it may be more natural to make the stakeholder depend on the *System* agent and then let the *System* agent analyze the goal and delegate the subgoals/subtasks to the system sub-agents. To illustrate this use of the *System* agent, we show the modification of the example in Figure 5.10a below. Here, instead of depending directly on the sub-agents of the system, environment actors depend on the *System* agent. This agent in turn delegates the goals to its sub-agents.

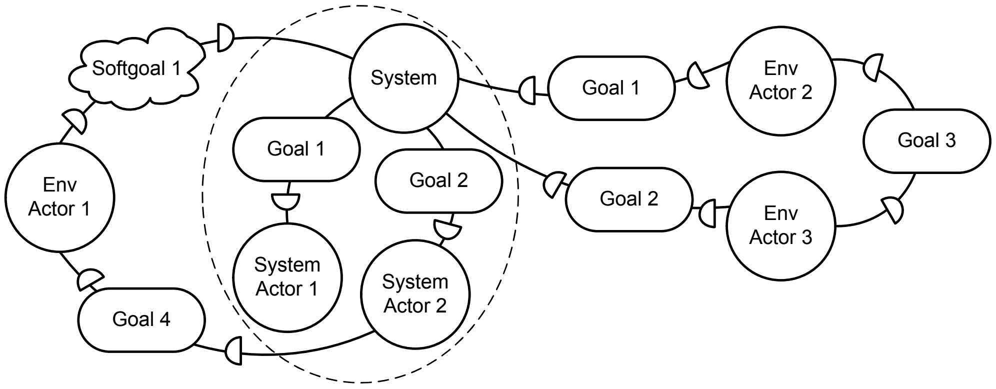


Figure 5.10b. Using the *System* agent.

Similarly, the *System* agent can be employed when stakeholders depend on the system for the achievement of goals that are very complex or too high-level to be assigned to an individual sub-agent of the system-to-be. These goals can be assigned to the *System* agent and decomposed into the low-level goals/tasks using the usual goal/task decompositions of the Strategic Rationale diagrams. Once these goals and tasks are simple enough, they can be assigned to the individual sub-agents of the system-to-be. In a sense, this is quite similar to the goal decomposition and assignment process employed by KAOS [Dardenne *et al.*, 1993]: the *System* agent provides a place to decompose complex system goals before assigning the simpler subgoals to the agents of the system-to-be (or of the environment). The difference from the KAOS framework is that any goal, not just leaf-level ones, can be assigned to agents. The *System* agent can be just an analysis tool and may be removed from the system in subsequent phases of the software development process. This allows the designer to simplify the SR/iASR models of the behaviour of the agents of the system-to-be by offloading the goal/task decompositions to the *System* agent. The modeler can still perform the necessary goal analysis and preserve requirements traceability while the SR-level models for the system agents and the subsequent implementation will become simpler. If the *System* agent is still present at runtime, it can act as a dispatcher/broker/manager and incorporate the intelligence to guide the achievement of the goals that the system is responsible for by decomposing those goals and creating a dynamic agent network of system and non-system agents for achieving these goals at runtime. This approach avoids the hardcoding of the decompositions of these complex goals into the model since such hardcoding may lead to inflexible systems.

The *System* agent can also be assigned the goals that do not come from the immediate stakeholders of the system, but rather from laws (including physical laws), regulations, business rules, etc. The analyst may choose not to include stakeholders such as "the government" into their models. In this case, these goals will become the goals of the *System* agent since they are the responsibility of the system as a whole. The *System* agent can also be used when the system needs special control over the achievement of the stakeholder goals (e.g., adaptable and customizable systems).

Analysts can also use the *Organization* agent, which is quite similar to *System* and represents the organization for which the system-to-be is being developed. *Organization* can, for example, have the goals of the owner/management of the company, the government (for the state-owned organizations), etc. This agent can be depended upon for the achievement of high-level organizational goals such as "be profitable" and "efficient management". By in turn depending on actors that are part of the organization, this agent can enforce these high-level organizational goals. One of the reasons for using the special *Organization* agent in *i** models is the fact that business goals are quite often extremely complex and require cooperation of all of the members of an organization. One can delegate these goals to *Organization*, decompose them within this agent using SR/iASR models, and then delegate the subgoals to the appropriate members of the organization. Also, a lot of business and organizational goals are meant to be achieved by the organization as a whole. Thus, depending on the special *Organization* agent for the achievement of these high-level goals may be a better way of modeling certain aspects of the domain.

5.4 Architectural Analysis

In the area of architectural analysis we mostly follow the Tropos approach. It allows us to analyze the architecture of the system in several ways. Firstly, we can select the architectural style [Kolp and Mylopoulos, 2001] that best match the organization being investigated. This analysis is done from the point of view of organizational theory. Secondly, we can analyze the architecture from the usual software engineering standpoint.

5.4.1 Organizational Architecture

Architectural analysis performed in the usual software development process focuses on modules, subsystems, communication protocols and other technical details of the system. It does not normally take into consideration the business processes in the organization and non-functional requirements. Fuxman et al. [Fuxman *et al.*, 2001a] defined a number of organizational architectural styles to remedy the situation — flat structure, pyramid, joint venture, structure-in-5, etc. These architectural styles are intended to guide the design of the system architecture for cooperative, dynamic, and distributed applications, and are naturally applicable to multiagent systems. The styles come not from the literature on software engineering, but from organization theory, the theory of the firm, etc.

The selection of the most appropriate organizational architectural style is done as follows. The relative importance of software quality attributes such as security, adaptability, modularity, etc. is identified for the desired software architecture. These qualities are used as criteria for the selection of the organizational architecture. The architectural styles mentioned above were evaluated [Fuxman *et al.*, 2001a] with respect to the following nine software quality attributes: security, adaptability, coordinability, cooperativity, availability, integrity, modularity, aggregability. The selection of the appropriate architecture is made through an analysis done with the help of the NFR framework [Chung *et al.*, 2000] (some of the quality attributes are decomposed further to allow for more detailed evaluations).

5.4.2 The System Architecture

While the appropriate organizational style allows the designer to select the high-level architecture that best suits the needs of the organization, there are still a lot of details that need to be worked out. A thoroughly selected organizational architectural style is just the first step in coming up with the complete software architecture for the system-to-be.

It is well known in the software engineering practice that large unstructured systems are hard to develop, deploy, and maintain. They are usually hard to scale up, inefficient, and expensive. Good software architectures are instrumental in developing flexible, scalable, and easily maintainable software systems. In our approach, the modelers are expected to use common software engineering principles in coming up with the software architecture for the system. In agent-oriented software engineering, architectural analysis is used to determine which agents are going to be developed as part of the system, what responsibilities are going to be assigned to these agents, and what interdependencies will exist among them. During this phase, the high-level architecture of the system is explored and requirements for all the system's sub-agents are specified. The papers on the Tropos methodology (e.g., [Castro *et al.*, 2002]) suggest using lower-level patterns such as broker, matchmaker, monitor, etc. within the framework set by the enterprise-level patterns such as joint venture.

We do not anticipate that modelers will devise the system architecture right away using SD models. One of the reasons for this is that the modeling of the details of agent processes and inter-agent interactions is done using the Strategic Rationale diagrams, which focus on finer aspects of the system. The details revealed during the SR-level analysis very often influence the high-level system architecture. Therefore, the number and configuration of system agents cannot be fixed at the end of the SD modeling phase. It is expected that new agent configurations, which are better architecturally, will emerge as more details about agent interactions and agent responsibilities are identified. During this early analysis or, more likely, during the SR-level analysis, it may become clear that some agents should be replaced with subsystems of agents to be able to handle their responsibilities better. The analysis of agent processes performed with the help of SR/iASR diagrams may reveal other problems, whose solutions require architectural changes. Thus, in our approach, the development of multiagent system architectures is an iterative process. We start off with Strategic Dependency models for the environment and the system with its environment. We then explore using SR diagrams the reasoning each agent goes through in order to achieve its goals and fulfill its responsibilities. We do not abandon SD diagrams that were developed earlier. Rather, we update them using the information obtained from analyzing the detailed agent interactions.

5.5 Late Requirements Using SR Diagrams

Strategic Dependency diagrams allow the designer to model intentional dependencies among the agents to see how the agents of the environment and system agents cooperate in achieving their goals. These diagrams provide a high-level, architectural view of systems and their environments. This view is external to the agents. Strategic Rationale diagrams, on the other hand, allow the modeler to analyze goals, plans, and inter-agent dependencies from the point of view of each actor. These diagrams are used to specify the reasoning each agent goes through in order to achieve its goals and help other agents depending on it in achieving their goals. The intentional elements such as goals, softgoals, tasks, and resources appear both as dependencies external to the agents and as modeling elements inside the agents. Strategic Rationale diagrams help in exploring the motivations of the agents and rationale behind their decisions.

In our approach, the SR-level analysis is done in two phases. We use the standard *i** Strategic Rationale diagrams first. The next step is to provide more details by adding the necessary annotations to the SR diagrams as well as adjusting these diagrams for an easy mapping into CASL models according to the rules specified in Chapter 4, thus producing iASR diagrams.

5.5.1 Building Strategic Rationale Diagrams

The way we use the standard Strategic Rationale diagrams is no different from the usual *i** process (e.g., [Yu, 1995]). Therefore, we will not get into details about how these diagrams are used. Having said that, we must note that in our methodology the standard SR diagrams are the first of two SR-level types of diagrams that the modeler is supposed to produce, along with the iASR diagrams. Therefore, SR diagrams are intended as first approximations of the agent process models.

As usual, to model the reasoning each agent (or the designer for that agent) goes through while attempting to achieve its goals and the goals that have been delegated to it by other agents, the modeler has means-ends relationships and task decompositions available to him. Means-ends analysis is mainly used to specify alternative means of achieving goals, while task decompositions are used to decompose complex tasks into more manageable ones. In the standard SR diagrams, task decompositions may include goals as well. Figure 5.11 presents a small SR diagram.

The task/goal decomposition stops when all the leaf tasks are sufficiently simple to be easily implemented. During the goal decomposition process, the designer will specify various means of achieving agent goals. Some alternatives for achieving a certain goal will involve delegating the goal or some of its subgoals to other agents in the system or the environment, while others will not involve obtaining any outside help.

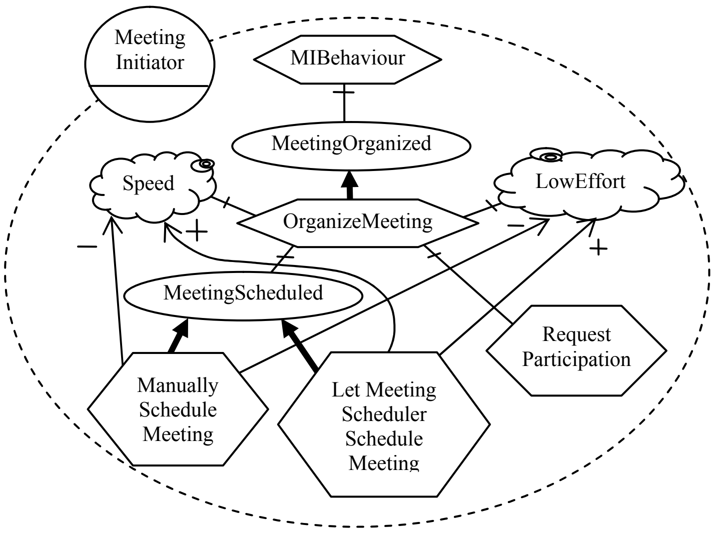


Figure 5.11. An SR diagram showing softgoals and softgoal contributions.

AND/OR decompositions [Giunchiglia *et al.*, 2003] can be helpful in decomposing goals into more manageable subgoals. These decompositions define the parent goals in terms of conjunctions or disjunctions of other (presumably simpler) goals. While modelers can use AND/OR decompositions early in their models, we do not directly support them in iASR diagrams. A mapping from AND/OR goal decompositions to the iASR diagrams is proposed in Section 4.3.2.

Figure 5.12 presents a small AND/OR goal decomposition. The top level goal is `Email Somebody`. To achieve it one needs to achieve the subgoals `Select Recipient`, `Prepare Content`, and `Send`. Since these are related through AND-decomposition, they all need to be achieved for their parent goal, `Email Somebody`, to be achieved. Alternatives in the way goals are achieved are introduced through OR-decompositions. For example, one can either type in the email address of the recipient, or select the address from an address book.

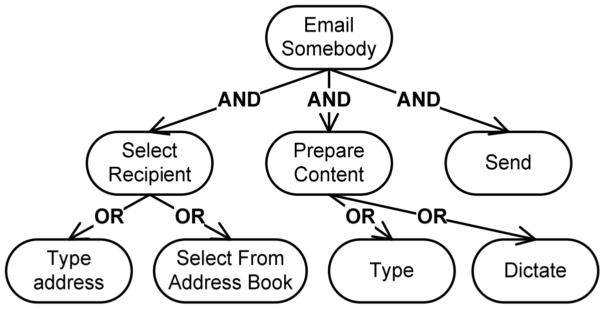


Figure 5.12. An example AND/OR goal decomposition.

Using Softgoals

SR diagrams provide ways to show how various process alternatives contribute (positively or negatively) to the satisficing of the softgoals that the agents have. Certainly, requirements engineers should not ignore non-functional requirements and must strive to select the alternatives that contribute positively to the agents' softgoals. Thus, softgoals together with their contribution links document the selection of alternatives. In our approach, it is in producing standard SR diagrams that the modeler analyzes non-functional requirements and evaluates process alternatives with respect to their contribution to the softgoals.

In Figure 5.11, we have an SR diagram for the Meeting Initiator agent with the alternative ways to achieve the goal `MeetingScheduled` being analyzed. There are two alternative ways to schedule a meeting: to schedule it manually or to use a meeting scheduler. Two softgoals are identified: `Quick` and `LowEffort` with the appropriate contribution links going from the alternatives to these softgoals.

5.5.2 Building iASR Diagrams: Adding Annotations

Intentional Annotated SR (iASR) diagrams are designed to add precision to the standard SR diagrams by providing link annotations, composition annotations, and applicability conditions, as well as to streamline the diagrams for an easy mapping into CASL models. The standard *i** SR diagrams can be quite ambiguous. For instance, SR diagrams do not provide any information on whether the subtasks in task decompositions are supposed to be executed sequentially or concurrently and whether all the subtasks are to be executed unconditionally or only under certain circumstances. Similar questions apply to means-ends links linking several means to achieve a certain goal with that goal. There is no way to tell whether all these means are applicable at all times or not.

While it could be argued that the level of detail provided by the regular SR models is enough to analyze the intentional aspects of agents' behaviour, we feel that more detailed specifications of agent processes help modelers understand the requirements better. Besides, quite often the sequencing of the subgoals in a goal decomposition is a property of the environment of the system (e.g., one must go to the airport, then go to the gate, and then finally board the plane). Also, adding annotations to SR models brings them to the level of detail required for successfully mapping these models into CASL.

There are three annotation types available to the modeler. Firstly, there are composition annotations that are mostly used with task/goal decompositions and specify whether the subtasks/subgoals are to be performed sequentially (";"), concurrently ("||"), with prioritized concurrency (">>"), or are alternatives ("|"). The default composition annotation is sequence. Secondly, there are link annotations that are attached to decomposition and means-ends links to specify under which conditions and how the subtasks/subgoals (for decomposition links) and means (for means-ends links) are to be executed by the agent. There are six link annotations (see Section 4.1.3 for details): *while* loop, *for* loop, pick, if, interrupt, and guard. The default is no link annotation — this means that the subgoals/subtasks are executed unconditionally. The third type of annotations is the *applicability condition* (see Section 4.3.3). These annotations accompany means-ends links and specify under what conditions each means to achieve a certain goal can be attempted. They are used to filter out the ways to achieve goals that are not feasible/efficient/etc. All of these annotations can be added gradually: at some point in time, it may be the case that the details of the agent processes in one part of the diagram have been fully specified, while more analysis is required in other parts of the model.

5.5.3 Building iASR Diagrams: Keeping Goal Dependencies

Quite a few of the agents' goals can be operationalized, i.e., replaced with tasks/procedures that achieve them. This is a standard practice since most software development processes and programming languages do not support goals. Therefore, goals are abstracted out before the late RE phase is over. Similarly, softgoals are frequently *metricized* [Davis, 1993] for easy evaluation. For example, suppose an agent has the goal "communicate message A to agent B". An operationalization of this goal will most likely involve choosing a communication protocol, the format of the messages, etc. The problem with this approach is that the details of the operationalization of the goals as well as the underlying assumptions are frozen into the requirements of the system-to-be. This replacement of goals with procedures (agent plans) that achieve them compromises the evolvability of the system and makes it more fragile [Castro *et al.*, 2002].

In contrast, one of the key ideas of the *i** process is to leave many goals and goal dependencies around. This is a new feature of *i** compared to the other notations. Goals in *i** diagrams indicate that it is up to the agent (or the designer for the agent) to select the best way to achieve them. It is important to document these as goals and keep them around during the software engineering process since they indicate the places in the system where evolution is most likely to happen. Goal nodes could be viewed as specifying alternatives in the way the system functions. It is conceivable that new ways to achieve goals may become available at some point in the life of the system. Thus, clearly documenting (and possibly implementing) agent goals as such will make the system more flexible. Whether the process alternatives are chosen at design-time by the designer or at runtime by the agents themselves (if the implementation is agent-oriented) is dictated mostly by the development platform chosen and therefore is irrelevant for the RE process. The position that we take in this thesis is similar to the position of the *i** modeling framework and the Tropos methodology: the implementation platform should not influence the methods used in RE. It is quite the opposite: the intentional concepts from Requirements Engineering should be pushed farther in the software development process. This will result in more flexible systems, which meet their requirements better. In addition, this thesis proposes the use self-acquired goals (see Section 4.3.8) that modelers can employ to allow the agents to reason about various ways of achieving their goals, which do not come from inter-agent dependencies, but from the designer of the agent. Thanks to the CASL's support for agent goals, we can go further than before in avoiding early operationalization of agent goals. The designer now has the tools to formally verify the models containing intentional elements such as agent goals and knowledge.

Self-acquired goals and the support of CASL for reasoning about these goals provide the analyst with an interesting new ability — the SR-level goal decompositions can now become a runtime tool, as opposed to being just a design-time facility available to modelers. SR goal decompositions are used to decompose high-level goals into simpler goals and/or tasks. In some cases, the intermediate goals are only used to discover the lower-level goals and tasks and can be abstracted out before goal decompositions are translated to some formal notation for verification and/or animation. Using self-acquired goals one can map the whole goal decomposition tree or a part of it (it could either be a standard SR goal decomposition or an AND/OR decomposition) into CASL (see Figure

4.32). In this way, the agents in the corresponding CASL model will be able to follow the reasoning that the analyst did while creating the goal decomposition. Their mental states will be appropriately updated. This makes the agents more aware of their processes and could allow them to make important decisions, which previously had to be made at design-time by the modeler, at runtime. Since the agents are able to retrace the goal decompositions at runtime, they will make the choices based on the current context, which can make them more adaptable to the changing environment.

5.5.4 Building iASR Diagrams: Deidealizing Goals and Suppressing Softgoals

Below we will identify several important steps that one has to go through while developing iASR diagrams for the system-to-be. These steps are similar to the ones proposed in [Wang, 2001] and are intended to help the modeler in simplifying Strategic Rationale diagrams and adding details to the agent processes. The steps that we outline in this section are intended to be done before the diagrams are converted into iASR form.

5.5.4.1 Suppressing Softgoals

Our approach does not provide tools for formal analysis of softgoals. Therefore, softgoals and softgoal dependencies are only used to help in the evaluation of process alternatives and software architectures. They must be abstracted out from SR diagrams before annotations are introduced. The softgoal nodes will be dropped from the diagrams along with the accompanying contribution links. Also, alternative ways of achieving agent goals — the ones that were found to contribute less to the softgoals than the best alternative — can be removed (see Figures 5.11 and 5.13a). Alternatively, some softgoals may be metricized (i.e., turned into regular goals) so as to have some quantitative approximation of these softgoals in the model for further analysis. An analysis tool could provide support for these types of simplifications.

We use Figures 5.13a and 5.13b to illustrate some of the ideas presented in several preceding sections. The figures show two possible diagrams for the Meeting Scheduler's task `OrganizeMeeting` after all the softgoals (see Figure 5.11) have been removed. The first diagram (Figure 5.13a) illustrates the operationalization of goals. Here, the least promising alternative for achieving `OrganizeMeeting` has been removed together with the goal `MeetingScheduled`. This decision is now built in the diagram. Suppose that the underlying assumption was that the number of meeting participants was going to be quite large, so scheduling meetings manually would be slow and would require high effort from the meeting organizer (see the softgoal contributions in Fugure 5.11). This assumption is now also frozen into the model. On the other hand, one can keep the goal in the model to indicate that there are alternative ways of achieving that goal (Figure 5.13b). This has all the benefits described in Section 5.5.3. The applicability condition documents that if the number of meeting participants is up to 3, it makes sense to schedule meetings manually, otherwise it should be better to let the Meeting Scheduler schedule meetings.

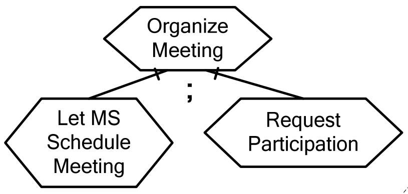


Figure 5.13a. `OrganizeMeeting` with the second alternative removed.

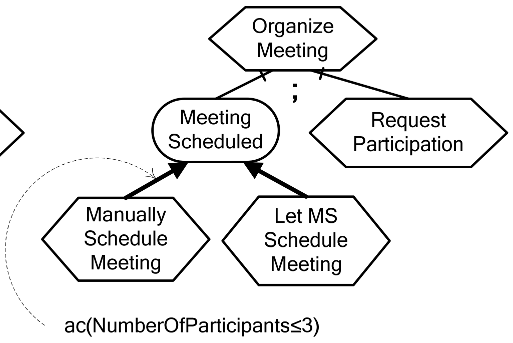


Figure 5.13b. `OrganizeMeeting` with the goal dependency and the two alternatives.

5.5.4.2 Deidealizing Goals

Very frequently, initial goals are too ideal. They are not achievable by the system and need to be *deidealized* [van Lamsweerde *et al.*, 1995] to become achievable. One possibility is to weaken goals that cannot always be achieved. For example, suppose that the Meeting Scheduler agent has the goal `ArrangeMeeting`. This goal may be too ideal since the intended meeting participants may not have any common available dates for the meeting, making the scheduling of the meeting impossible. Therefore, the goal must be weakened to account for that possibility. It is important to think about the achievability of agent goals early on since the reformulation of a goal will most likely lead to changes in the means of achieving it. Of course, the achievability of goals is most easily verified if one has formal definitions of these goals available. The formal analysis of agent goals using CASL models later in the process may uncover more unachievable goals compared to the informal analysis done at this stage of the methodology. Nevertheless, thinking about the achievability of goals throughout the RE activities is very helpful since the necessary changes may be introduced much earlier.

5.5.5 Building iASR Diagrams: Adding Agent Interaction Details

In *i**, intentional dependencies are extremely important. These dependencies become agent interactions in an agent-oriented setting. Speech act-based communication in a multiagent system is quite different from remote method invocation-style communication among software components. For example, in order for an agent to use the services of other agents, these services must be requested (usually with the "request" performative). The requested agents may acknowledge the request and, if they are helpful, perform the requested service. While attempting to supply the dependum, the dependee may request additional information, thus making the interactions more complex. Once the service is performed, the requesting agent will either be notified of that fact or will have to monitor for the successful rendering of the service.

*i** usually abstracts over modeling detailed agent interactions. Since specifying detailed interactions among cooperating agents in a multiagent system is essential and since our goal is to animate and verify *i** models using the CASL agent programming language, we require much more detailed specification of agent interactions than the usual *i** process provides. Section 4.3.7 provides all the details on how to deal with dependency-based goals in iASR diagrams. It shows how agents can delegate goals and tasks to other agents and also notes that a resource dependency must be converted to either a goal or a task one depending on the level of freedom the dependee has in providing the required resource for the depender.

For instance, when providing details for inter-agent communication that takes place when one agent is requesting the services of another, the modeler needs to update the iASR diagram for the depending agent with the tasks that request these services for the depender. On the other hand, the iASR diagrams for the dependees will be updated with the goal nodes with appropriate interrupt annotations to model the acquisition of goals. Figure 4.41 illustrates this.

Since CASL supports reasoning about knowledge, analysts can model information exchanges in their iASR diagrams and these interactions can be formally analyzed. Information dependencies are discussed in Section 4.4.3.

Careful analysis of inter-agent dependencies may reveal that new sub-dependencies are needed. For example, a dependee may need to request some additional information or clarification from the depender. Also, some complex interaction protocols may be needed to replace certain dependencies. For instance, finding the best price for a product may require the buyer (the depender) and sellers (the dependees) to use the Contract Net protocol [Smith and Davis, 1981], which involves more interactions than a simple request. Additional notations can be used to carefully specify agent interaction. Designers may wish to use protocol diagrams, which are part of Agent UML (AUML) [Odell *et al.*, 2000] to specify all the details of the interactions (this may be more appropriate for the later phases of the software development process).

We note that some communication exchanges do not have to be modeled (e.g., one agent asking again for some information it did not receive the first time). This could be handled at the implementation level. CASL models are quite idealistic in that many of the usual communication-related problems such as missed messages, etc. are not dealt with. What we are interested in is high-level message exchanges. Since the dependencies are not mapped into CASL per se, the only way to establish the dependencies is for dependers to send requests to dependees and for dependees to monitor for these requests. These details must be provided on iASR diagrams.

5.5.6 Building iASR Diagrams: iASR-style Goal Decompositions

As described in Section 4.3.2, Intentional Annotated SR diagrams differ from SR diagrams, besides having various annotations, in the use of goal nodes and means-ends links. Since goal nodes are intended to be utilized as means of exploring alternatives in models, they are used only with means-ends links, which specify alternative ways of achieving these goals (thus, they are available with the alternative composition annotation only). Moreover, means-ends links are used only with goal nodes and these goal nodes must be accompanied by the interrupt link annotations. Additionally, the means for achieving goals can only be task/capability nodes, which streamlines the mapping of iASR diagrams into CASL models. We discussed why this is required in Section 4.3.2. This means that SR diagrams where some of the means for achieving goals are goals themselves must be modified to comply with the above rule by utilizing *self-acquired goals* (see Figures 4.19 and 4.32).

5.5.7 Using Capability Nodes in iASR diagrams

Capability nodes (Section 4.5) are a new modeling concept introduced into the *i** modeling framework in this thesis. Modelers can use task and goal capability nodes in iASR diagrams to indicate that agents have epistemically feasible plans that guarantee the successful achievement of the corresponding goals or the execution of the corresponding tasks provided that their context conditions hold when they are invoked and there is no outside interference while they are executing. Capabilities can be a part of the interface of an agent — that is, they can be used as means of achieving the goals or performing the tasks at the requests of other agents. On the other hand, capabilities can be used internally by an agent itself. We expect that capabilities will be used by the agents, among other things, as means of fulfilling critical dependencies as well as for the internal processes where failure cannot be tolerated.

There are two forms of capability nodes: the first one shows only the goal that the capability achieves or the task that it performs, while the more detailed view of capabilities shows the details of how the goals are achieved or the tasks are performed. Abbreviated capability nodes can be used in both SD (see Section 5.2.3) and SR-level diagrams, while the detailed view is only allowed in the SR-level diagrams (see Figure 4.47).

5.6 Requirements Verification Using CASL

5.6.1 Mapping iASR Diagrams into CASL

Once all the necessary details have been introduced into an iASR diagram, it can be mapped into the corresponding CASL model. This model provides the formal semantics for the otherwise informal *i** model, thus making iASR diagrams amenable to formal analysis.

CASL is a much closer match for the *i** models that specify multiagent systems and their environments than ConGolog, which was used in conjunction with *i** in [Wang, 2001]. The key here is the support of CASL for reasoning about agents' mental states, goals and knowledge. This, among other things, allows for much more thorough analysis of inter-agent communication. Modelers can also use CASL to reason about epistemic feasibility of agent plans as well as about trust and privacy issues. The process of adding details to the SR diagrams and their subsequent mapping into CASL could be considered exploratory prototyping.

The mapping process will map every element (with the exception of dependencies) of iASR diagrams into CASL as described in Chapter 4. Specifically, leaf-level tasks will be mapped into CASL procedures or primitive actions, and task decompositions will be mapped into procedures taking into consideration all the annotations used with these decompositions. Goals will be formally defined by CASL formulae and associated with CASL procedures encoding the ways the goals can be achieved. Capabilities are associated with epistemically feasible procedures; their context conditions are formally defined. The agents are mapped into CASL procedures that describe their behaviour. Dependencies are not mapped into CASL per se. The dependencies at CASL level are established by the agent procedures that request services from other agents as well as by agents monitoring for such requests. Data is passed around mostly through procedure parameters. Additional auxiliary CASL code may need to be added to the model to describe the initial situation, initial knowledge of agents, some properties of the environment, etc.

5.6.2 Verifying the System with CASLve

CASL is a formal specification language and coupled with the appropriate tools provides powerful facilities for the formal analysis of CASL models. CASLve [Shapiro *et al.*, 2002] is a verification environment for CASL based on the PVS theorem prover [Owre *et al.*, 1996]. CASLve defines an encoding of CASL programs in the input language of PVS. The verification environment also provides a library of proof methods for proving various types of results. One can use CASLve to prove properties such as liveness, safety, and termination. In general, many interesting properties of multiagent systems specified in CASL can be analyzed using CASLve. If expected properties of the system are not entailed by the CASL model, it means that the model is incorrect and needs to be fixed. The source of an error found during verification can usually be traced to a portion of the CASL code for the system and/or to some part of its iASR diagram. After the necessary changes have been made, the verification can be repeated, thus making the process iterative. As outlined in the next section, we believe that our approach has a high level of requirements traceability support. Therefore, it is quite easy to change the CASL code for the system in attempt to fix an error and then synchronize the code and the corresponding iASR diagram. Changing the iASR diagram first and then mapping it to CASL to get the updated version of the CASL model is also possible. In both cases, the changes are easily localized so that the complete remapping from iASR to CASL or from CASL back to iASR is not necessary. A model editing tool could support this synchronization.

5.6.3 Supporting Requirements Traceability

The requirements engineering process is aimed at identifying the needs of stakeholders, coming up with system requirements that refine these needs, resolving conflicts among requirements, and specifying these requirements in an easily understandable form that is the basis of the system design and implementation [Castro *et al.*, 2003]. Therefore, it is important to keep track of which stakeholder goals give rise to which requirements and what the origins of all the requirements in the system are. In addition, as Ramesh and Jarke [Ramesh and Jarke, 2001] note, it is very important to keep track of bi-directional relationships between requirements and the development process artefacts in order to facilitate the maintenance and verification of the system.

Castro et al. [Castro *et al.*, 2003] state that a requirement is traceable if one can discover its origin, why it exists, what other requirements relate to it, and how that requirement relates to systems designs, implementations and user documentation. Consequently, requirements traceability refers to the ability to follow the lifecycle of a requirement in both forward and backward directions — i.e., from its origins, through its specification and development, to its subsequent deployment and use, and through periods of ongoing refinement and iteration [Gotel and Finkelstein, 1994]. Since our work mainly deals with requirements engineering phase of the software development process and does not go beyond high-level design, our concern is the traceability in the early requirements, the late requirements, and the high-level design phases of software development process. In particular, we would like to show that our approach supports tracing a requirement from its origin as a stakeholder goal through various *i**-provided analysis steps (i.e., strategic dependencies, softgoal analysis, and goal decompositions) to the corresponding CASL code and back.

*i** allows the requirements engineer to capture the goals of stakeholders — the origins of system requirements. These goals are achieved through the network of inter-agent dependencies and/or goal and task decompositions. The achievement of a stakeholder goal may or may not involve a delegation of the goal or some of its subgoals to the actor that is a part of the system-to-be. In any case, following goal/task decomposition links and inter-agent dependencies allows us to see how high-level stakeholder goals are being achieved by the system-to-be and its environment. If a change is introduced to some nodes in the model, the affected nodes are easily identified — they are the modified nodes' successors in the graph of goal/task decompositions. The successors may also include the nodes related to the modified ones through inter-agent dependencies. On the other hand, by examining decomposition and dependency links going up the goal/task decomposition graph from a certain goal or task node, we can easily see why the task has to be performed or the goal has to be achieved (as well as all the conditions under which this particular iASR diagram node becomes active). Therefore, it is easy to trace requirements back and forth within annotated *i** diagrams.

The next step of the method, after the iASR diagrams have been developed, is to map them into the corresponding CASL models. The modeler is required to define a mapping `m` from a completed iASR model into CASL code as outlined in Chapter 4. This allows for formal analysis of the *i** model. The mapping `m` is itself a very important source of traceability information as it bridges between the *i** and CASL models. It maps every element of an iASR diagram into CASL. Every agent/position/role, every goal and task node, every annotation and applicability condition, and every goal and task decomposition is mapped, thus giving the ability to trace requirements from iASR models to CASL code. Intentional dependencies are mapped into the corresponding agent interactions defined using performatives. Tracing from the CASL code back to the corresponding iASR models is quite easy as well, since we expect that most of the CASL code is produced by mapping elements of the iASR diagrams, and this is documented by the mapping `m`, which is defined by the modeler. If the modeler is using *Trusts*/*Serves* relationships in the models (Section 5.2.5), then the directed graphs or other, more complex models used to represent these relationships, could be easily mapped into the associated CASL fluents (see Sections 4.3.7.3 and 4.4.4). There could be some auxiliary CASL code to help with the animation of the CASL model or formal analysis of the requirements. This CASL code is part of the needed CASL infrastructure and is not directly related to the elements of iASR diagrams. It cannot, therefore be traced back to requirements. It is imperative, however, that this code be appropriately documented for ease of use and modifiability.

5.8 Automating the Mapping into CASL

Given the mapping rules presented in Chapter 4, the mapping between iASR diagrams and the procedural components of the corresponding CASL agents must be very close. The code for CASL procedures that iASR goals, capabilities, and non-leaf-level tasks map into is completely determined by these tasks' decompositions in iASR diagrams and by the composition and link annotations used in these decompositions. The flow of control in the procedural component of a CASL specification is based on the link and composition annotations used by the modeler in the corresponding iASR models (see the mapping of the model in Figure 4.14). Therefore, most of the code for the procedural component of a CASL agent can be produced directly from the agent's iASR diagram.

A modeling tool can be used to facilitate the mapping of iASR diagrams into CASL specifications. As discussed above, most of the mapping is quite routine and can be automated. In this thesis, we allow leaf-level task nodes to be mapped into either primitive actions or CASL procedures. In this way, the modeler can decide to reduce the size of iASR diagrams by not decomposing agent processes all the way to primitive actions. On the other hand, forcing the modeler to fully decompose agent tasks (i.e., by requiring that leaf-level tasks be mapped only into primitive actions with the same name and parameters) will help a modeling tool map iASR diagrams into CASL by automatically constructing CASL procedures from primitive actions and composition and link annotations.

The following changes to the mapping process can be made to allow for more automated mapping of iASR diagrams into CASL:

- Goals must be mapped into defined fluents with the same name.
- Achievement procedures for goals must be labeled *GoalName*Proc, where *GoalName* is the label for the goal. The same applies to capabilities.
- Conditions in annotations must have descriptive names and must be mapped into defined fluents with the same name.
- Goal and task parameters must be fully specified in iASR diagrams.

The modeler will have to define the fluents that goals/annotation conditions are mapped into as well as the context conditions and the allowable process specifications for agent capabilities. Also, action preconditions and successor state axioms for fluents must be defined by the modeler. In addition, specifications for individual CASL agents must be combined together to produce the specification for the whole system – concurrent composition is mostly used for this. Then, CASL specifications can be constructed from the corresponding iASR diagrams.

6 The Meeting Scheduling Process Case Study

In this chapter, we examine the approach proposed in the thesis through the use of a case study. We apply our methodology to the process of scheduling meetings in an organization. This is a simple case study that, nevertheless, allows us to demonstrate most of the features of our approach. The meeting scheduling scenario was previously explored and modeled using *i** [Yu, 1995] and Wang's *i**-ConGolog approach [Wang, 2001].

6.1 Introduction

In the case study discussed in this chapter we model the process of scheduling meetings in some organization. In the context of the *i** modeling framework the process was first analyzed in [Yu, 1997]. The initial requirements for the process were stated as "For each meeting request, to determine a meeting date and location so that most of the intended participants will be able to effectively participate" [Yu, 1997]. We will modify the scenario in several ways as well as make certain assumptions about the process and the stakeholders involved in it in order to make our models more manageable and to better illustrate our methodology.

In our view of the meeting scheduling process, for every meeting we have a single person initiating the meeting scheduling process. We take the length of the meetings to be the whole day. We do this to simplify our model and note that this model can be easily extended to handle multiple meetings per day. The meeting can be successfully scheduled if for *all* the intended participants there is a date that fits their schedule. We also assume that in the environment of the system-to-be there is a legacy software system called the Meeting Room Booking System that handles the booking of meeting rooms.

In the scenario that we study in this chapter, the possibility of introducing an automated meeting scheduling system is explored and compared (at a high level) to the current manual meeting scheduling process. The automated meeting scheduling system is then explored in great detail using iASR diagrams. Finally, the iASR model of the system is formalized by mapping it into CASL using the mapping process proposed in this thesis.

6.2 Early Requirements Analysis

The first step in the requirements engineering methodology presented in Chapter 5 is to analyze the environment of the system-to-be, the stakeholders, and the current processes. This analysis helps in determining the needs of the stakeholders and the relationships among them, which is crucial in understanding why the new system is needed and how it can improve the current situation.

6.2.1 Identifying Stakeholders

We first identify stakeholders in the current meeting scheduling system. In the words of Nuseibeh and Easterbrook [Nuseibeh and Easterbrook, 2000], stakeholders are "individuals or organizations who stand to gain or loose from the success or failure of the system". In the meeting scheduling scenario, the process of stakeholder identification is quite easy. However, there may be cases where this step of the methodology is not trivial. Using the *i** terminology we call stakeholders *actors*. The following actors are participating in the process of scheduling meetings:

- *the Meeting Initiator (MI)*, who initiates the scheduling of a meeting;
- *the Meeting Participant (MP)*, an actor who is requested to participate in meetings;
- *the Meeting Room Booking System (MRBS)*, a legacy software system that is used to reserve meeting rooms.

These are the actors that are part of the meeting scheduling process. However, we can identify another actor — we will call it *Disruptor* — that arranges meetings with meeting participants (thus changing their schedules) outside of the process modeled here. The reason we need this actor is that not all meetings are managed by the organization's meeting scheduling process. For example, while we assume that everybody within the organization is using the meeting scheduling process, arranging meetings with people that are not part of the organization takes place outside of the current scheduling process.

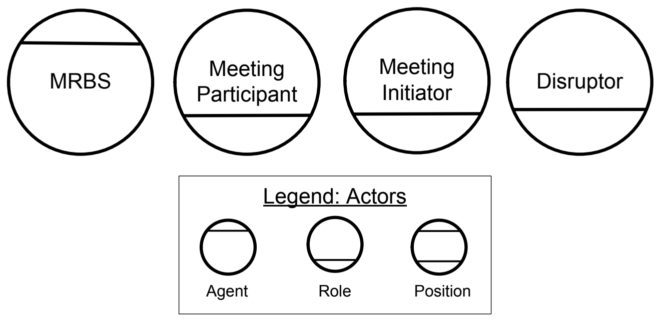


Figure 6.1. The stakeholders in the meeting scheduling process.

Most of the stakeholders shown above can be classified as *roles* in *i**. Naturally, the roles of Meeting Initiator, Meeting Participant, and Disruptor can be played by many different agents. However, since the MRBS represents a concrete legacy system, we consider it an *agent* in the *i** sense.

6.2.2 Identifying Stakeholder Goals and Intentional Dependencies

We can describe the current meeting scheduling process in the organization in terms of the newly identified stakeholders and their relationships. This will help in identifying the goals and the needs of the stakeholders, and their rationale for being part of the meeting scheduling process. We can describe the current process as follows. The MI schedules meetings by contacting each Meeting Participant and getting the list of its available dates. It then chooses a date that is convenient for all meeting participants (including the Meeting Initiator) and attempts to book the meeting on that date. The Meeting Initiator then uses the MRBS to book a room for the meeting. At the same time, the Disruptor can also arrange meetings with Meeting Participants outside of this meeting scheduling process. We can now start modeling the meeting scheduling process using Strategic Dependency (SD) models. The first step in modeling the current processes in an organization is to identify the intentions/goals of the stakeholders and to discover the intentional dependencies that exist among these stakeholders.

Let us look at the Meeting Initiator actor. It is the driving force behind the meeting scheduling process, so the intentional dependencies that help in achieving the Meeting Initiator's goals make up the core of the process. The Meeting Initiator needs to schedule a meeting with meeting participants, to book a room for the meeting, and it also needs the participants' presence at the meeting. Figure 6.2 below shows these goals using the notation proposed in Section 5.2.2:

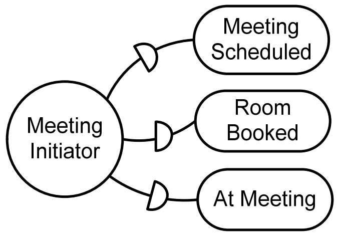


Figure 6.2. Meeting Initiator with three unassigned goals.

The above figure presents the Meeting Initiator actor with three (yet unassigned) goals. For now we will not concern ourselves with the goal parameters. By looking at the current process in the organization, we can determine how the goals of Meeting Initiator are presently handled, whether they become part of some intentional dependencies or are achieved by the actor itself. The booking of the meeting rooms is handled by the MRBS software system, so we can add the goal dependency `RoomBooked` to our SD diagram that models the scheduling process (see Figure 6.3). Note that we also add a goal capability node `RoomBookedCap` to the MRBS agent. The MRBS is a good actor to model with capabilities since it is a legacy software system and always provides the expected results when it is used properly and not being interfered with. `AtMeeting` represents the Meeting Initiator's goal of having the intended participants present at the meeting. This goal is delegated to the Meeting Participant through a goal dependency. As for the goal `MeetingScheduled`, it is presently handled by the Meeting Initiator itself. We can, therefore, add a goal self-dependency `MeetingScheduled` to the model. We note here that `RoomBooked`, `AtMeeting` and `MeetingScheduled` are goal dependencies (as opposed task dependencies). This means that the depender does not prescribe a specific course of action to the dependee, but instead gives the dependee the freedom to choose the right means for the achievement of the goal.

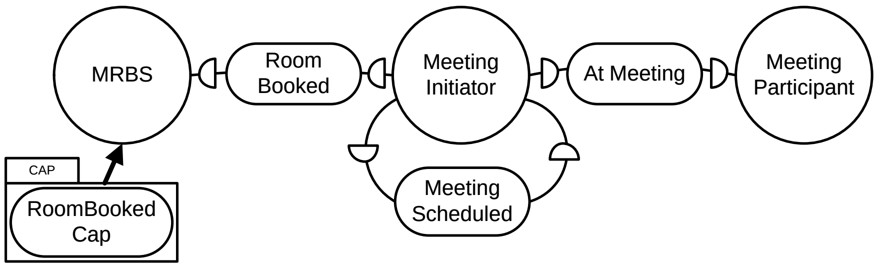


Figure 6.3. Delegating the Meeting Initiator's goals

Figure 6.3 shows the core intentional dependencies for the Meeting Initiator, which are exercised during meeting setup. By looking at the current meeting scheduling process we can add details to the above diagram by further analyzing the goals and needs of the stakeholders and their relationships. First, we consider the Meeting Initiator actor again. In order to be able to schedule meetings it needs the list of available dates from each of the intended participants. We can model this relationship using an *i** resource dependency. The subject of this dependency is the information resource `AvailableDates`. Also, the MI needs to know which room was booked for the meeting by the MRBS agent, so it depends on the MRBS for that information, thus giving rise to another resource dependency with the dependum being the information about the room number of the booked room. The MP, on the other hand, needs to know whether and when the meeting is scheduled. This is another information resource dependency that we add to the model (see Figure 6.4). We also add the Disruptor actor to the SD model of the meeting scheduling process. Not unlike the Meeting Initiator, it depends on the Meeting Participant for attending meetings. We model this with the goal dependency `AtMeeting`.

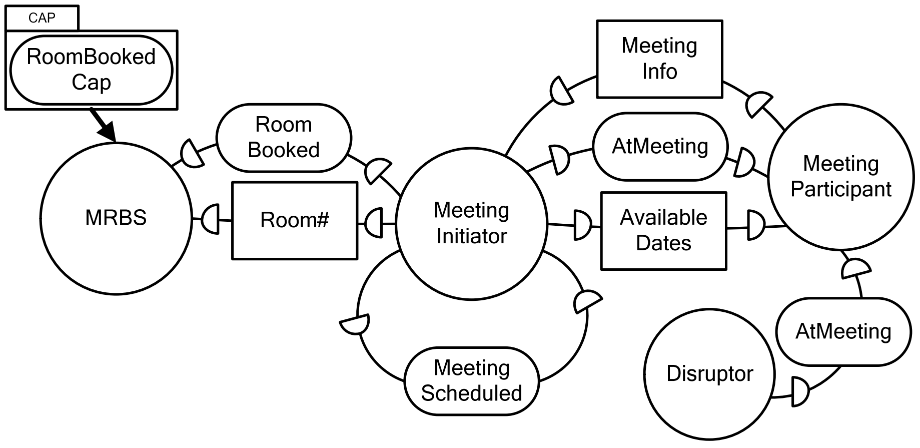


Figure 6.4. Adding details to SD model.

6.2.3 Modeling Additional Organizational Details

Section 5.2.5 proposed to model additional organizational details using the *Serves* and *Trusts* relationships. The *Serves* and *Trusts* relationships could be represented as directed graphs. The *Serves* relationship specifies, for each actor, which other actors it is willing to help in the achievement of their goals. Similarly, the *Trusts* relationship specifies, for each actor, which actors in the organization it believes. We will now sketch how the relationships can be modeled in case of the meeting scheduling process.

Most of the actors in our SD model of the process are roles, which can be played by many different agents. The *Serves* relationship can be used to effectively limit the agents that can play the role of the Meeting Initiator. Suppose that one of the constraints that exist in the organization is that only managers or team leaders are allowed to schedule meetings and book meeting rooms. One way to model that constraint with the *Serves* relationship is to declare that the MRBS agent representing the room booking system is only available to a restricted set of agents (managers and team leaders). If the directed labelled graph notation is used (see Figure 5.8), then we can use Figure 6.5 below to specify that only the agents playing the role of a Manager or a Team Lead can use the MRBS system to book meeting rooms:

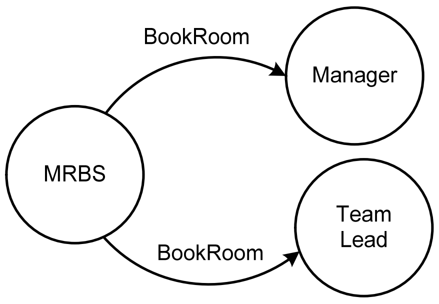


Figure 6.5. Using the *Serves* relationship.

We can similarly use the *Serves* relationship to define the actors that can request information about available dates from the Meeting Participants. Note that we are not proposing any particular notation for the *Serves* and *Trusts* relationships.

6.2.4 Modeling Non-Functional Dependencies

Modeling non-functional requirements during requirements analysis is very important. Non-functional requirements are sometimes called *quality* requirements and address such concerns as efficiency, ease of use, security, quality, etc. In the *i** terminology, non-functional requirements are called *softgoals*. There is no clear-cut satisfaction condition for a softgoal, instead softgoals are *satisficed* [Simon, 1981] — achieved to an acceptable degree. In our approach, softgoals are used as selection criteria for choosing the best alternatives among multiple process configurations. In *i**, softgoals can be the subjects of intentional dependencies where the depending actor delegates the satisficing of a softgoal to a dependee actor.

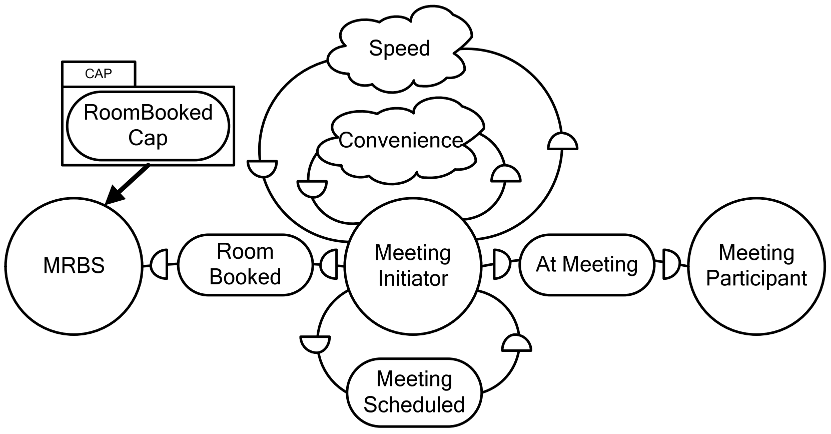


Figure 6.6. Modeling softgoal dependencies.

In our meeting scheduling process model, we can identify a number of different softgoal dependencies. In this chapter, we will model a few of them. For example, the Meeting Initiator has a couple of non-functional requirements related to the process of scheduling meetings. The first non-functional requirement is the convenience of the scheduling process. It is modeled by the softgoal `Convenience`. Also, the Meeting Initiator wants the scheduling process not to be very time consuming. This is modeled by the softgoal `Speed`. Since in the current meeting scheduling process the MI is responsible for scheduling meetings, we model the softgoal dependencies `Speed` and `Convenience` as self-dependencies involving the Meeting Initiator. Figure 6.6 shows the core dependencies of the meeting scheduling process with the newly introduced softgoal dependencies.

6.3 Late Requirements with SD Diagrams

In this section, the new automated meeting scheduling system is introduced into an organization. As this system is introduced, the existing processes in the organization will change. This will result in the reconfiguration of intentional dependencies in the model.

6.3.1 Introducing the System-To-Be

After exploring the organization, the processes within this organization, the stakeholders and their needs, we have a good understanding of the environment where the new automated meeting scheduling system will be introduced. Thus, the next step in the analysis of the meeting scheduling process is to bring the system-to-be into its organizational environment and analyze how the combined system (comprised of the environment and the system-to-be) is going to change. The system is represented by one agent (concrete actor) called the Meeting Scheduler (MS). It may be the case that the system is decomposed into several sub-agents at a later stage of the requirements engineering process.

With the introduction of the Meeting Scheduler agent, the network of intentional dependencies in the system changes significantly (see Figure 6.7). The Meeting Initiator now depends on the Meeting Scheduler for scheduling meetings. The initiator also needs to know whether the meeting is scheduled, when it is scheduled, and which room it is going to be held in. Since the Meeting Scheduler is scheduling meetings, it must have this information. Therefore, there is an information resource dependency `MeetingInfo` going from the Meeting Initiator to the Meeting Scheduler. It is now the responsibility of the Meeting Scheduler agent to book meeting rooms, so it depends on the MRBS for the achievement of the goal `RoomBooked`. The scheduler is now the depender of the information resource dependency `Room#`, which is, as previously, fulfilled by the MRBS.

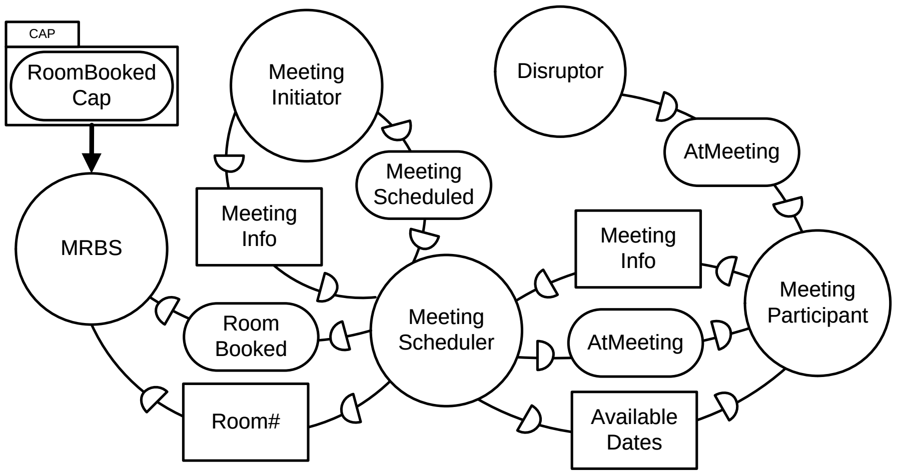


Figure 6.7. Introducing the system-to-be.

The figure above also shows that there are significant changes in the Meeting Participant's dependencies. Since the Meeting Scheduler agent is responsible for scheduling meetings, the dependencies `MeetingInfo`, `AtMeeting`, and `AvailableDates` now involve the Meeting Participant and the Meeting Scheduler. The only unchanged intentional dependency in the model is the goal dependency `AtMeeting` from the Disruptor to the MP. The dependency remains the same since the Disruptor actor is not using the organization's meeting scheduling process and therefore is not affected by any changes in this process.

6.3.2 Introducing the *System* Agent

The *System* agent was introduced in Section 5.3.2 as a tool that can be used, among other things, to collect the goals that come not from the stakeholders in the organization, but from regulations, laws, etc. In order to demonstrate the use of the *System* agent in our case study we will assume that in the organization that we are modeling there exists a regulation requiring that no meetings be scheduled on weekends. This regulation could come from a company policy or a labour regulation. The exact source of this regulation is not important. What is important is the fact that it comes from the outside of the meeting scheduling process that we are modeling.

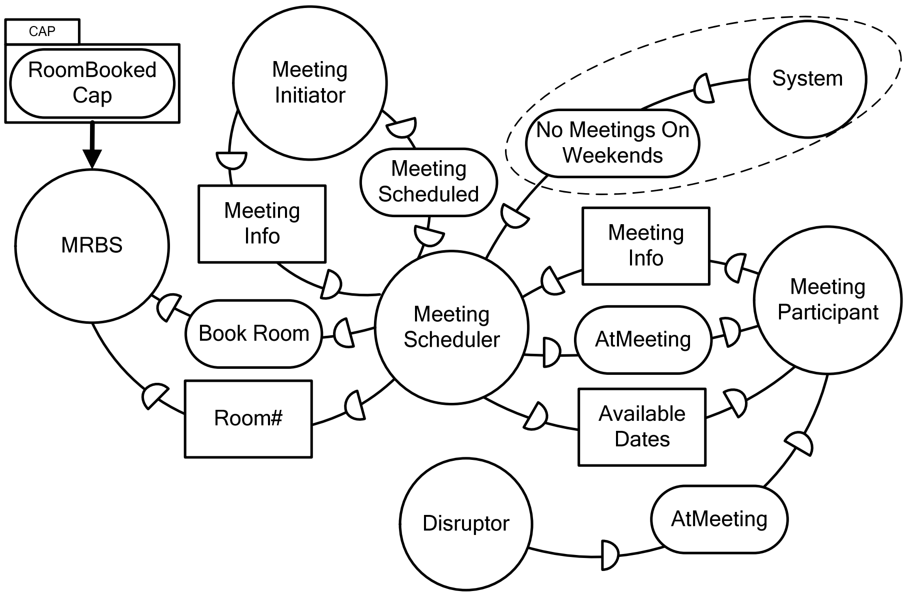


Figure 6.8. The *System* agent in introduced into the model.

We can capture requirements coming from this kind of regulations by assigning them to the *System* agent. The *System* agent represents the combined system as a whole and its goals are thought of as the goals the whole system must strive to achieve. The modeler can then look at the goals of the *System* agent, and decompose them if necessary, up to the point where the subgoals can be handled by single agents, and then assign these subgoals to the selected agents, thus distributing the responsibilities for the achievement of its goals throughout the system. By assigning the goal `NoMeetingsOnWeekends` to the *System* agent we declare that the new automated meeting scheduling system as well as its environment must make sure that no meetings are scheduled on weekends. The next step is to analyze this goal and see whether it needs to be decomposed into simpler subgoals to be assigned to the actors in our model. In our methodology, Strategic Rationale (SR) and Intentional Annotated Strategic Rationale (iASR) diagrams are used to handle goal and task refinement. Thus, in order to refine the goals of the *System* agent one needs to use SR/iASR diagrams. However, in our case the goal `NoMeetingsOnWeekends` can easily be assigned to the Meeting Scheduler agent since that agent is responsible for scheduling meetings and it can make sure that no meetings are scheduled on weekends. This assignment of responsibility for the system goal is modeled by the goal dependency `NoMeetingsOnWeekends` from the *System* agent to the Meeting Scheduler (see Figure 6.8).

6.3.3 The System's Architecture

In this chapter, we will not discuss the architectural analysis of the system since the meeting scheduling system we are modeling is quite simple and is of the right size to be modeled by just one agent, the Meeting Scheduler. The need to split the system up into several sub-agents could arise if there was a requirement, for example, to schedule meetings for hundreds of people. In this case, the system could be split into a variable number of agents handling communication with meeting participants (e.g., one agent per 100 participants) as well as the agent responsible for communicating with the initiator and selecting potential meeting dates. This would improve the system's scalability.

6.4 Late Requirements with SR Diagrams

Once we have identified the intentional dependencies among the actors in the environment of the system-to-be and the system-to-be itself, we can look "inside" the environment actors and the system agents and model their internal processes. This step of the methodology gives the modeler a great insight into the way the actors behave, the choices they face, their intentions, etc. New, more fine-grained intentional dependencies are frequently discovered during SR/iASR analysis. In this section, we will use the SR notation to model the internals of the actors that are part of the meeting scheduling process. These are the initial high-level SR diagrams that will be refined later and turned into iASR diagrams for mapping into CASL.

6.4.1 Modeling the Meeting Initiator: Using Softgoals for Alternative Selection

One very important investigation that designers can do using SR/iASR models is the analysis of various alternatives using their contributions to softgoals. In Section 6.2.4, we modeled several non-functional requirements (NFRs), `Convenience` and `Speed`, that exist in the meeting scheduling system. These NFRs are modeled as softgoal self-dependencies of the Meeting Initiator. We do not require that all the choices in terms of alternative process configurations be made at the requirements analysis stage. In fact, we suggest quite the opposite — keeping goals and alternative means of achieving them around until (and during) the design phase of the software development process. However, in order to concentrate on the modeling of the automated meeting scheduler and to keep the model simple, we will immediately use the softgoal analysis to select the best alternative and remove other alternatives.

The way that alternative process configurations are related to softgoals (NFRs) is through softgoal contributions. The analyst identifies the elements of the model that affect (positively or negatively) the softgoals of the actors and models these influences using softgoal contribution links. Figure 6.9 is the SR diagram for the Meeting Initiator with two alternative ways of scheduling meetings. One way to do it is to use the old manual scheduling process. The other way is to use the new Meeting Scheduler agent, which is under development.

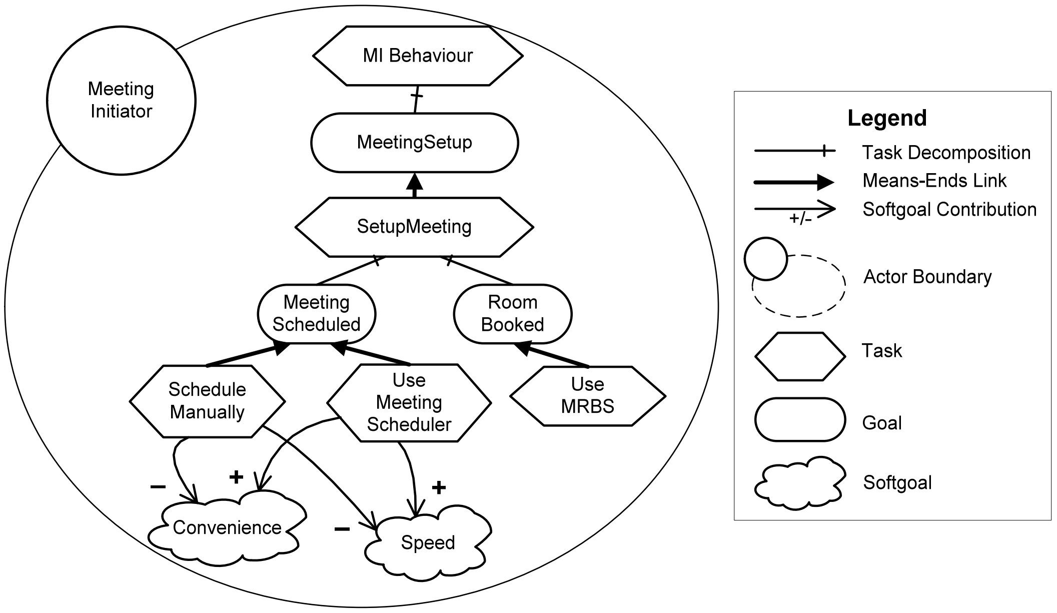


Figure 6.9. Modeling the softgoal contributions.

The two non-functional requirements identified for the Meeting Initiator are used to rank these two alternatives. The task `UseMeetingScheduler` naturally represents the alternative that uses the new MS agent to schedule meetings. Since all the work to schedule meetings is offloaded to an automated scheduling system, it contributes positively to the softgoal `Convenience`. Therefore, we add a positive softgoal contribution from `UseMeetingScheduler` to `Convenience`. Also, the automated system is going to be faster than the current manual process, so we add another positive contribution from `UseMeetingScheduler` to the softgoal `Speed`. On the other hand, the analysis of the manual meeting scheduling process with respect to the two non-functional requirements shows that the manual process contributes negatively to both NFRs. This is modeled by the negative softgoal contributions from the task `ScheduleManually`. To concentrate on modeling the system using the automated scheduler we assume that at this point a decision is made to abandon the manual scheduling process in favour of using the Meeting Scheduler agent.

One of the requirements on the iASR models before one can map them into the corresponding CASL specifications is the absence of softgoals, softgoal dependencies, and softgoal contribution links. Softgoals can be used as above to select the best alternative and then be abstracted out. Alternatively, softgoals can be approximated with hard goals. In any case, they must be removed before the mapping to CASL is done.

Let us look at the SR model for the Meeting Initiator actor after the removal of the softgoals and softgoal contributions (see the Figure 6.10). The model presents a high-level view of the internal process for the actor using goals, tasks, resources, and intentional dependencies.

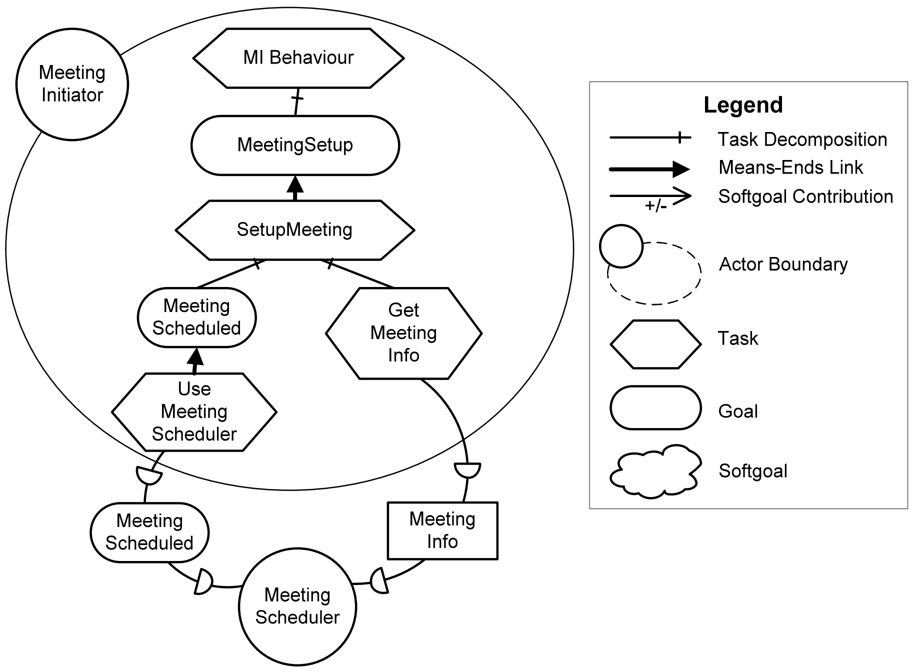


Figure 6.10. The SR diagram for Meeting Initiator.

The top-level node in the diagram is the task `MIBehaviour`, which models the overall behaviour of the actor. While the SR diagram notation does not require this, we always have a task as the top node in every actor's SR diagram in order to simplify the conversion of SR diagrams into iASR diagrams. The task `MIBehaviour` has one subgoal, `MeetingSetup`. This is the main goal of the Meeting Initiator. The goal has one means of achieving it. In order to achieve the goal `MeetingSetup`, the MI must execute the task `SetupMeeting`, which is in turn decomposed into a subgoal and a subtask. Setting up meetings involves scheduling them (including the booking of meeting rooms). This is modeled by the goal `MeetingScheduled`. As per the discussion in the previous section, the scheduling of meetings is done by delegating it to the Meeting Scheduler. Thus, the only means to achieve the goal `MeetingScheduled` is to request help from the automated scheduler. At this stage, we model this as the task `UseMeetingScheduler`. The details of the request will be provided in iASR models. After delegating the scheduling of a meeting to the Meeting Scheduler, the initiator must know the outcome of the scheduling process: whether the meeting was scheduled, when it was scheduled, and in what room it is going to be held. The Meeting Initiator must get this information from the Meeting Scheduler. This is modeled by the task `GetMeetingInfo`. This task requests the needed information from the Meeting Scheduler agent. Thus, there is a resource dependency from the MI to the MS with the dependum `MeetingInfo`.

6.4.2 Developing an SR Model for the MRBS

As mentioned before, the Meeting Room Booking System is a legacy software system that is used in the current manual meeting scheduling process. This system is deemed adequate for the new automated scheduling system, so it is going to be used in conjunction with the Meeting Scheduler agent. Therefore, it is represented as an actor in our diagrams and we need to model its behaviour, dependencies, etc. Since it is a legacy system, it is useful to model the services that the MRBS provides as capabilities. The purpose of the MRBS system is to manage the booking of meeting rooms and we have all the confidence that if the system is used properly and not interfered with, it can successfully book rooms. Therefore, a goal capability `RoomBookedCap` can be used to represent the service of the agent as illustrated in the SD diagrams in the previous sections (e.g., Figure 6.8). At this stage of the requirements engineering process, our goal is to model the processes inside the MRBS agent. While modeling legacy systems, hardware devices, etc. one needs to concentrate on the incoming and outgoing intentional dependencies/interactions and on approximating the internal system processes.

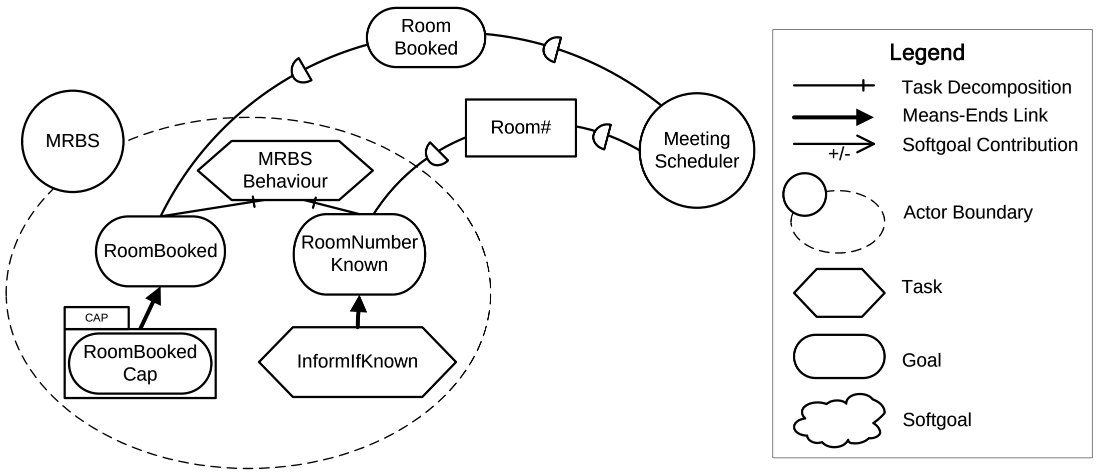


Figure 6.11. Modeling the MRBS agent.

Figure 6.11 is a high-level SR model of the MRBS system. There are two incoming dependencies from the Meeting Scheduler (the MS is the only actor now using the services of the booking system), the goal dependency `RoomBooked` and the resource dependency `Room#`. The top-level task of the MRBS agent is `MRBSBehaviour`, which models the overall behaviour of the agent. It is decomposed into two goals: `RoomBooked` and `RoomNumberKnown`. These goals represent the services that the MRBS agent provides: it can book rooms and notify its user whether it was successful and which room was booked for a meeting. The means of achieving the goal `RoomBooked` is a goal capability with the name `RoomBookedCap`, while the task `InformIfKnown` is a means to achieve the goal `RoomNumberKnown`.

6.4.3 Developing an SR Model for the Meeting Participant

We assume that the Meeting Participant is a role played by the personal agents of the employees in the organization. They are participating in the meeting scheduling process on behalf of their owners. It can be seen from Figure 6.8 that the Meeting Participant actor depends on/is depended upon by the Meeting Scheduler, and by the actor called the Disruptor, which represents the outside forces that make the members of the organization block their calendars with external meetings. Let us analyze the Meeting Participant actor more closely. We start by developing a high-level SR diagram for it (see Figure 6.12). The top-level task representing the overall behaviour of the actor is called `ParticipantBehaviour`. It is decomposed into subgoals and subtasks that take care of incoming and outgoing dependencies of the actor. These dependencies are the same as in the SD diagram in Figure 6.8. The first subgoal the task is decomposed into is called `MSInformedOfAvailableDates`. This goal exists because of the incoming resource dependency `AvailableDates`. Here, the MS needs to know which dates on the schedule of this Meeting Participant are still available for meetings. Unlike goal and task dependencies, which specify the degree of freedom the dependee has in providing their dependums, resource dependencies do not specify that information. Therefore, in dealing with resource dependencies, one needs to determine whether the process of providing a particular resource is better modeled with goals or tasks. In fact, in our approach, all resource dependencies are eventually replaced with goal or task ones. In case of the resource dependency `AvailableDates`, we are inclined to treat it as a goal from the point of view of the Meeting Participant — there may be many possible ways of letting the MS know about available dates. For example, participants can inform the scheduler directly or they can upload their available dates to some secure shared storage. Therefore, the dependency `AvailableDates` is connected to the goal node `MSInformedOfAvailableDates`. This goal has one means of achieving it. It is the task `InformMS`, which represents the process of sending the list of available dates directly to the scheduler. Later, other means could be identified and added to the model.

The second subgoal in the task decomposition of `ParticipantBehaviour` is the goal `AtMeeting` (see Figure 6.12). Two actors depend on the MP for the achievement of this goal — the Meeting Scheduler and the Disruptor. To achieve this goal the participant must attend the meeting, so the means of achieving the goal in the model below is the task `AttendMeeting`.

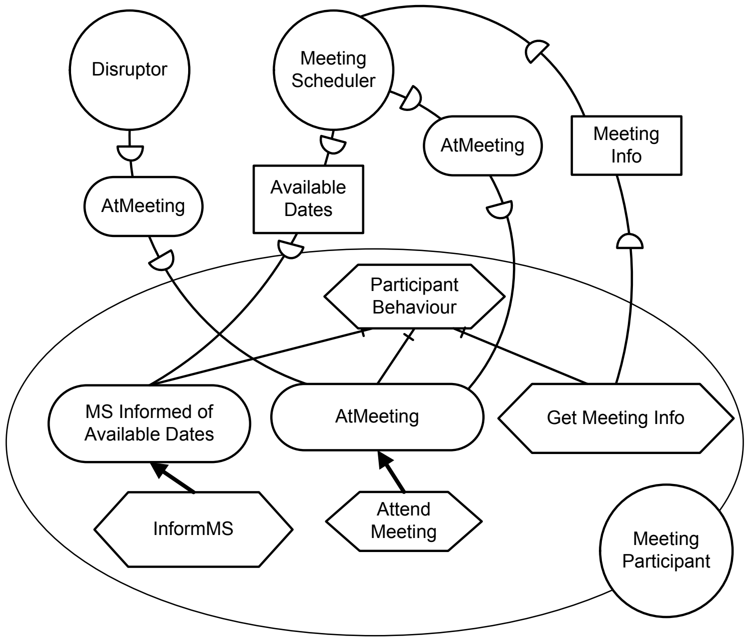


Figure 6.12. The SR model for the Meeting Participant.

The MP needs to know the details (date, room, etc.) of the meetings that it is supposed to participate in after they have been scheduled. This is modeled by the resource dependency `MeetingInfo` in the SD diagram in Figure 6.8 with the MS being the dependee and the MP being the depender. In the SR model above, we must identify the origin of this dependency inside the participant. This origin is the task `GetMeetingInfo`. It specifies how to get the information about the meeting. `GetMeetingInfo` can be replaced by a goal (e.g., `MeetingInfoKnown`), for which several achievement alternatives (e.g., asking the scheduler or looking up the info on a message board) can be identified. We will stay with the task `GetMeetingInfo` to simplify the model.

6.4.4 Developing an SR Model for the Disruptor

As mentioned before, the Disruptor actor represents the agents outside of the organization that want to arrange meetings with members of the organization. This actor creates obstacles for the meeting scheduling process by making the meeting participants occupy their schedules, thus reducing the number of available dates and making it more difficult to schedule meetings among the members of the organization.

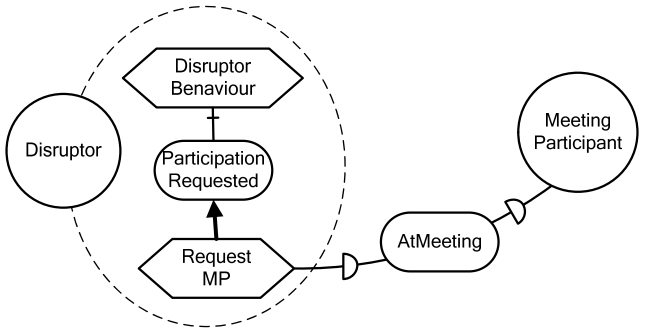


Figure 6.13. The SR model for the Disruptor.

The initial SR model for the actor is very simple. It has just one objective, the objective of requesting the Meeting Participant to attend a meeting with the Disruptor. In the diagram above (Figure 6.13), it is modeled by the goal `ParticipationRequested`. The goal is achieved by the task `RequestMP`, which requests that the Meeting Participant attend the meeting with the Disruptor, thus establishing the goal dependency `AtMeeting`.

6.4.5 Modeling the Meeting Scheduler

The Meeting Scheduler agent is the centerpiece of the new automated meeting scheduling system. This agent is involved in many dependencies and interacts with most of the actors in the system. Let us look inside this agent (see Figure 6.14). The top-level task that models the overall behaviour of the agent is called `MSBehaviour`. It is decomposed into a subgoal and a subtask. They represent the activities that the MS does in order to provide the dependums for two of its incoming dependencies. These dependencies come from the Meeting Initiator, which triggers the meeting scheduling process. The dependency `MeetingScheduled` ends inside the MS at the goal node `MeetingScheduled`. The way the MS agent fulfills the resource dependency `MeetingInfo` is by executing the task `SendMeetingInfoToMI`.

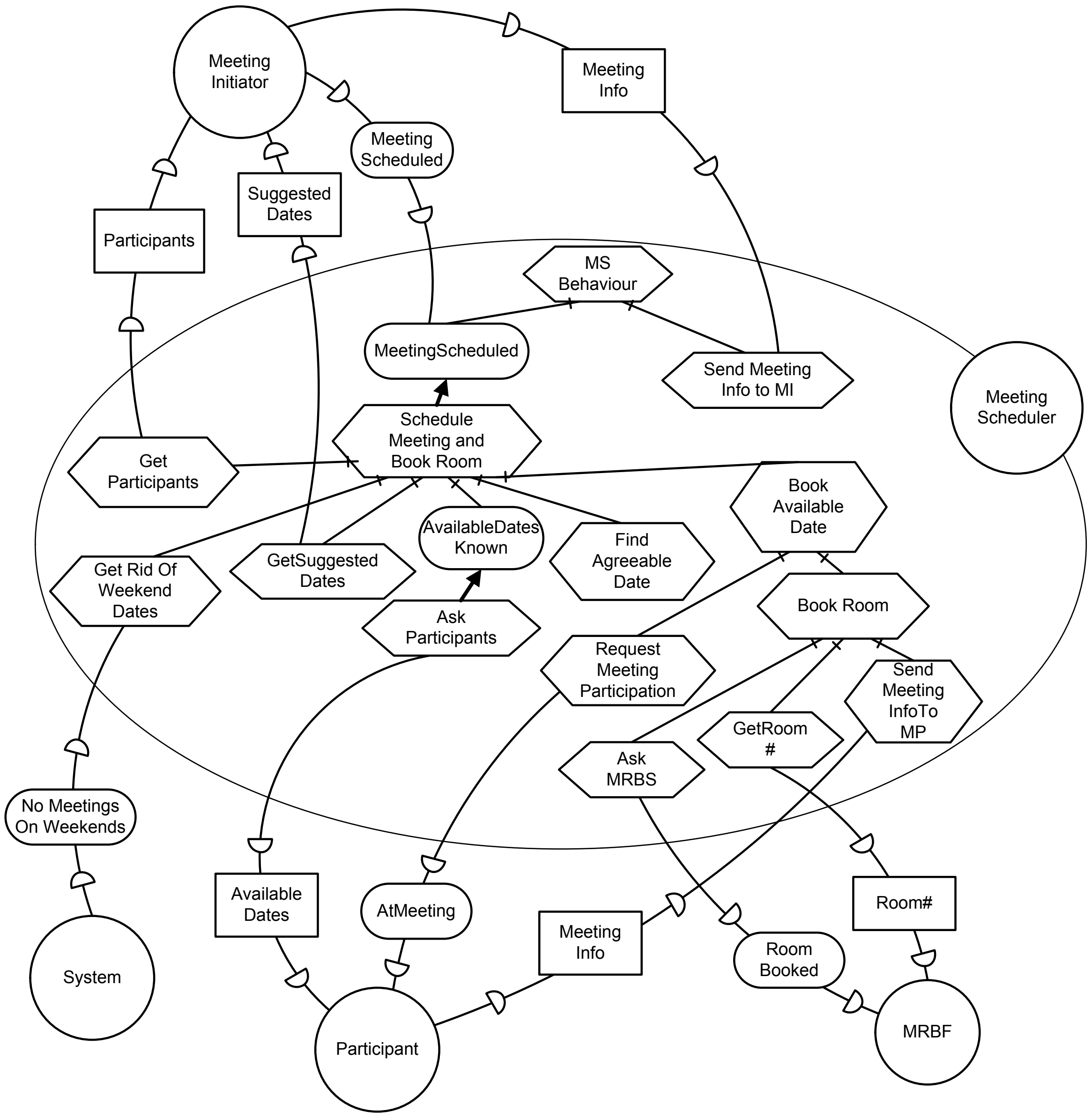


Figure 6.14. The initial SR diagram for the Meeting Scheduler.

The goal `MeetingScheduled` has one means of achieving it, the task `ScheduleMeetingAndBookRoom`. This task represents the main activity of the scheduler. It must schedule the meeting with all the participants as well as book a room for the meeting. We use task decomposition links to decompose the task into a number of subtasks and a subgoal. First, in order to be able to schedule a meeting the MS needs to know the identities of the meeting participants and the dates suggested by the Meeting Initiator. For this information the MS agent depends on the Meeting Initiator. The tasks `GetParticipants` and `GetSuggestedDates` model the activities needed to request that information from the initiator. These tasks are the origins of the two new resource dependencies, `Participants` and `SuggestedDates`. A deeper look at the meeting scheduling process inside the MS agent made the analyst realize the need for these two dependencies (they were not modeled in the initial SD diagram in Figure 6.8). The corresponding modifications to the SR diagram for the Meeting Initiator are discussed in the next section.

After receiving the list of the intended meeting participants from the Meeting Initiator, the MS needs to get available dates from all these participants in order to find a mutually unoccupied time slot for the meeting. This is represented by the goal `AvailableDatesKnown`, which is a subgoal of the task `ScheduleMeetingAndBookRoom`. Only one means for achieving this goal is modeled for now, the task `AskParticipants`. It may be the case that in the future the scheduler could get access to securely stored schedules of all the participants and get the information on available dates from there. So, we model `AvailableDatesKnown` as a goal and keep it as such through the rest of the RE process in attempt to make the system more flexible/adaptable later on. The task `AskParticipants` represents the activity of requesting the information on available dates from participants. It is, therefore, the origin of the information resource dependency `AvailableDates`.

After available dates are known, the task is to find a common date that is free for all of the participants and is one of the dates suggested by the initiator. The task `FindAgreeableDate`, also a subtask of `ScheduleMeetingAndBookRoom`, models this activity. Upon finding an agreeable date the Meeting Scheduler requests a meeting on that date and books a meeting room. This is modeled by the task `BookAvailableDate`, which is decomposed into subtasks `RequestMeetingParticipation` and `BookRoom`. By requesting participation from meeting participants the MS delegates the achievement of the goal `AtMeeting` to the MPs. Thus, the task `RequestMeetingParticipation` is the origin of the goal dependency `AtMeeting`. The booking of a meeting room is modeled by the task `BookRoom`, which is the origin of the goal dependency `RoomBooked` that goes from the scheduler to the MRBS. The MS also needs to know which room the MRBS books for the meeting, so the next subtask in the task decomposition of `BookRoom` is `GetRoom#`. It again requests this information from the MRBS, thus giving rise to the resource dependency `Room#`. Once all of the details of the scheduled meeting are known, the MS provides this information to the participants.

Let us take a look at the *System* agent with its goal dependency `NoMeetingsOnWeekends`. As it was decided in Section 6.3.2, the best way to achieve this goal is to delegate it to the Meeting Scheduler agent since it is responsible for scheduling meetings and therefore can be made to avoid scheduling them on weekends. In the SR model of the MS agent (Figure 6.14), the goal dependency `NoMeetingsOnWeekends` ends at the task `GetRidOfWeekendDates`, which throws away the dates suggested by the initiator that fall on weekends. This ensures that no meeting is going to be scheduled on one of those dates.

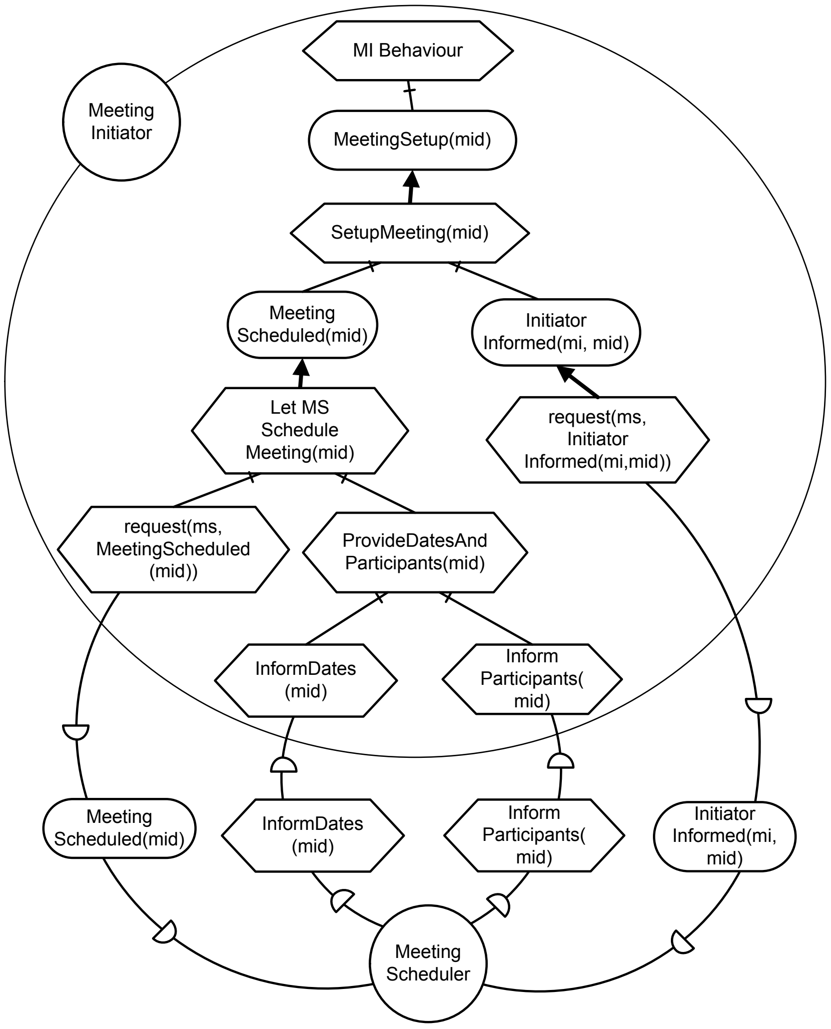
6.5 Toward iASR Diagrams

In this section, we look at how the some of the more complex SR diagrams developed above are refined and how the details are filled in to set the stage for their conversion into iASR diagrams. This involves adding parameters to goal and task nodes, adding details to interactions, replacing resource dependencies with goal/task ones, as well as refining goal/task decompositions. The simpler SR models are directly turned into iASR diagrams in Section 6.6.

6.5.1 Refining the Meeting Initiator Model

Our goal is to gradually refine the SR model for the Meeting Initiator so that it becomes detailed enough to be easily mapped into a formal CASL specification. Process details are added to the model at this stage. Some changes resulting from a more thorough analysis of the MI process are also introduced into the goal/task decompositions. The second version of the SR diagram for the MI in Figure 6.15, therefore, brings it closer to the iASR model.

When thinking about agent process specification, it is helpful to identify the context/subject of the goals/tasks/resources in SR models. This subject can be expressed with parameters. Adding parameters to goals and tasks is a way to increase the precision of the model. This precision is necessary for the formal analysis of the model using CASL. For example, instead of having a goal `MeetingSetup` (as in Figure 6.10), we add the parameter `mid` to the goal label and it becomes `MeetingSetup(mid)`. `mid` stands for "meeting ID", a unique identifier for a meeting. It allows the modeler to distinguish instances of the process that refer to different meetings. Parameters are also a way to pass information around. Therefore, the introduction of parameters for goals and tasks makes the analyst think about what data is needed for successfully executing a task or for achieving a goal. In the model below (Figure 6.15), once the goal `MeetingSetup(mid)` is identified, all the nodes that appear further down in the decomposition tree are passed this parameter to indicate that they refer to the same meeting.



MI Behaviour
Meeting Initiator
MeetingSetup(mid)
SetupMeeting(mid)
Meeting Scheduled(mid)
Initiator Informed(mi, mid)
Let MS Schedule Meeting(mid)
request(ms, Initiator Informed(mi,mid))
request(ms, MeetingScheduled (mid))
ProvideDatesAnd Participants(mid)
InformDates (mid)
Inform Participants(mid)
Meeting Scheduled(mid)
InformDates (mid)
Inform Participants(mid)
Initiator Informed(mi, mid)
Meeting Scheduler

Figure 6.15. The second version of the SR model for the Meeting Initiator.

Some changes to the goal/task decompositions inside the MI actor were also made in new the SR model above. The new information resource dependencies that were discovered during the initial SR-level analysis of the Meeting Scheduler (Section 6.4.5) are included in the diagram above. Since the MS needs the information on the meeting participants as well as the list of suggested meeting dates from the initiator, the goal `MeetingScheduled` is now decomposed differently (see Figure 6.15). The means to achieve the goal is the task `LetMSScheduleMeeting`, which includes the subtask `ProvideDatesAndParticipants`. Two subtasks of `ProvideDatesAndParticipants`, `InformDates` and `InformParticipants`, are used to inform the MS about the dates and participants respectively.

The task `GetMeetingInfo` from the first version of the SR diagram for the MI actor (it was a subtask of `SetupMeeting`, see Figure 6.10) is replaced with the goal `InitiatorInformed`. This change in the diagram is done in order to improve the flexibility of the process specification. The task `GetMeetingInfo` was to ask the Meeting Scheduler for the information on scheduled meetings. Since we anticipate that there will potentially be other means of getting this information (e.g., from a message board or through an announcement), it is better to replace the task with the goal `InitiatorInformed` because the goal models the intention of the initiator to acquire the information and the task that is asking the MS agent for this information is just a means of doing so.

Another addition to the model is the introduction of the task nodes that model the requests/informs that the MI performs. As discussed in Section 5.5.5, agent interactions are very important in multiagent systems. They are the primary means of goal/task delegation and information exchange in these systems. High-level agent interactions are also well supported by CASL. In order to be easily mapped into a formal CASL model, agent interactions must be specified in detail in iASR diagrams. One needs, for example, to specify the tasks that request a service or information, the tasks that provide the information, etc. When appropriate, we use suitable communicative actions of CASL to label the tasks that delegate goals and tasks to other actors or provide information to other actors. In particular, in the model in Figure 6.15, `LetMSScheduleMeeting` has a subtask labelled `request(ms,MeetingScheduled(mid,mi))`. This task delegates the achievement of the goal `MeetingScheduled` to the Meeting Scheduler. `request(fromAgt,toAgt,someGoal)` is a primitive action in CASL that results in the agent `toAgt` acquiring `someGoal`. Similarly, the means of achieving the goal `InitiatorInformed` is the task `request(ms,InitiatorInformed(mi,mid))`. Notice that for achievement goals in CASL one needs to request that **`Eventually`**`(φ)`. Since all of our goals in the case study are achievement goals, we leave out **`Eventually`** in the goal/task labels. Also, the `fromAgt` parameter (the requesting actor) is known from the context, so we do not specify it in the diagrams. Not all of the requests can be replaced with simple communicative actions. For example, the tasks `InformDates` and `InformParticipants` that inform the Meeting Scheduler of the meeting participants and preferred meeting dates are more complex and each simply cannot be replaced by an `inform`. We will eventually map them into CASL procedures.

As discussed in Section 4.3.7.2, all resource dependencies in iASR diagrams are replaced with either goal or task ones depending on the degree of freedom that the depender gives the dependee. In the first version of the SR model for the initiator we had three information resource dependencies. First, the MI depended on the scheduler for `MeetingInfo` (see Figure 6.10). This dependency has been replaced with a goal dependency `InitiatorInformed` reflecting the freedom that the MS agent has in choosing the means to provide this information. On the other hand, the two resource dependencies from the MS to the initiator that were identified in the initial SR model for the scheduler in Figure 6.14 are replaced with task dependencies `InformDates` and `InformParticipants`. This means that the scheduler does not give the initiator the freedom to choose how to communicate the required information; specific tasks must be instead executed.

6.5.2 Refining the SR Model for the Meeting Scheduler

In this section, we discuss the second version of the SR model for the Meeting Scheduler agent. First, as we have done in the previous section, we add parameters to the goal/task labels that specify the subject of goals/tasks. Since the activities of the meeting scheduler have to do with the scheduling of meetings, most goals and tasks in the diagram in Figure 6.16 now have the parameter `mid`, which represents the unique ID of the meeting. This indicates that the goals are achieved and the tasks are performed in the context of a particular meeting. The task `BookAvailableDate` and its subtasks have an additional parameter called `date`. The task represents the activities that the MS agent must go through to organize a meeting on a date that is available for all the participants. This additional parameter specifies that date.

After more detailed analysis of the meeting scheduling process, the decomposition of the task `BookAvailableDate` is modified to account for the possibility that meeting participants may not adopt the goal of attending a meeting on an available date if that date has since been allocated/occupied due to some external meeting request (through the Disruptor actor) while the meeting was being scheduled by the MS. Our first attempt to deal with this problem is to add a subtask to `BookAvailableDate` called `GetConfirmation(mid,date)`, which will request confirmation from all the participants that they are in fact intending to attend the meeting. This task becomes the origin of the new dependency `AttendanceConfirmed(mid,date)` where the MS depends on the participants for confirmation of attendance (see Figure 6.16). A more complex solution is introduced in the iASR diagram for the Meeting Scheduler in Section 6.6.5.

In the SR model in Figure 6.16, we also replaced all resource dependencies with goal/task ones depending on the level of freedom that the depender gives the dependee in fulfilling the dependency. Figure 6.14, the first version of the SR diagram for the MS agent, featured six resource dependencies. Let us review these dependencies. Once it got a meeting request, the MS depended on the initiator for the information on the meeting participants as well as on the suggested meeting dates. These were initially modeled as resource dependencies. As already discussed in Section 6.5.1, the two dependencies are then replaced with task dependencies `InformParticipants` and `InformDates`. These are complex tasks that communicate the list of intended meeting participants and the list of suggested meeting dates to the MS. Because of this complexity, the MS is not willing to give the initiator the freedom to provide the information as it sees fit, but rather the MS agent instructs the Meeting Initiator to perform the tasks that provide the necessary information to it.

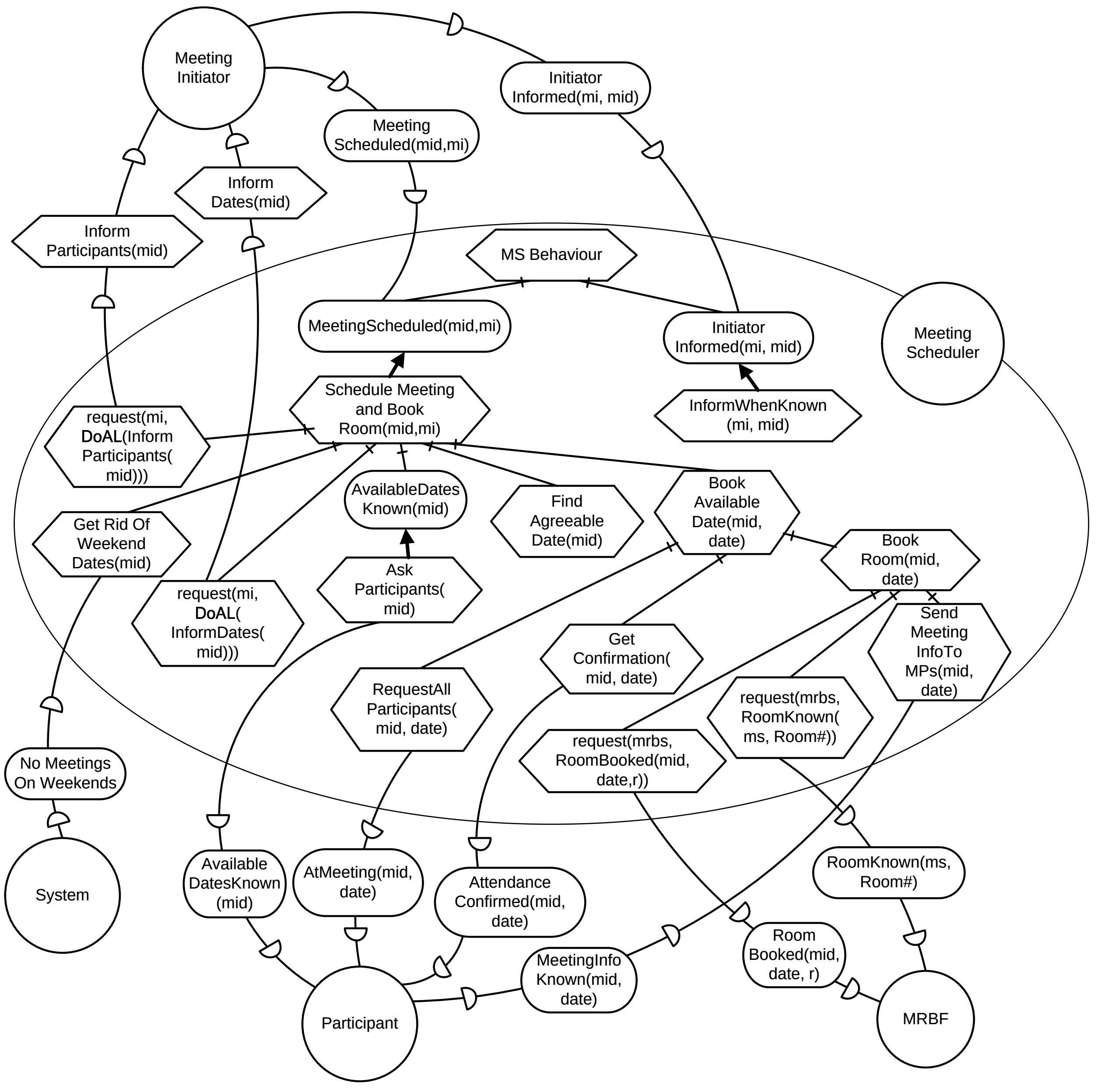

Meeting Initiator
Initiator Informed(mi, mid)
Meeting Scheduled(mid,mi)
Inform Dates(mid)
Inform Participants(mid)
MS Behaviour
MeetingScheduled(mid,mi)
Initiator Informed(mi, mid)
Meeting Scheduler
Schedule Meeting and Book Room(mid,mi)
InformWhenKnown (mi, mid)
request(mi, DoAL(Inform Participants(mid)))
AvailableDates Known(mid)
Find Agreeable Date(mid)
Book Available Date(mid, date)
Book Room(mid, date)
Get Rid Of Weekend Dates(mid)
request(mi, DoAL(InformDates(mid)))
Ask Participants(mid)
Send Meeting InfoTo MPs(mid, date)
Get Confirmation(mid, date)
RequestAll Participants(mid, date)
request(mrbs, RoomKnown(ms, Room#))
request(mrbs, RoomBooked(mid, date,r))
No Meetings On Weekends
System
Available DatesKnown (mid)
AtMeeting(mid, date)
Attendance Confirmed(mid, date)
RoomKnown(ms, Room#)
Room Booked(mid, date, r)
MeetingInfo Known(mid, date)
Participant
MRBF

Figure 6.16. The second version of the SR diagram for the Meeting Scheduler.

The resource dependency `MeetingInfo` from the initiator to the scheduler in Figure 6.14 is now replaced with the goal dependency `InitiatorInformed`. Here, the Meeting Initiator depends on the scheduler to be informed of the outcome of the meeting scheduling process. The MS is free to choose a means to achieve this goal. The parameter, `mi`, is the name of a particular meeting initiator. The rest of the resource dependencies from Figure 6.14 are replaced with corresponding goal dependencies: `AvailableDates`, `MeetingInfo`, and `Room#` are replaced by `AvailableDatesKnown`, `MeetingInfoKnown`, and `RoomKnown` respectively.

The initial version of the Meeting Scheduler's SR diagram (Figure 6.14) already provides a lot of details about interactions among actors. For example, it has tasks that represent requests for information or services from other actors and tasks that provide information to other actors. In order to increase the precision of the diagram, in the second version of the SR model for the Meeting Scheduler these tasks are labelled (where appropriate) with the corresponding CASL communicative actions. For example, requests to the MRBS agent for booking a meeting room on an available date and for information on which room is booked are modeled by two request actions that are subtasks of `BookAvailableDate`. Also, the requests that the Meeting Initiator execute the procedures `InformParticipants` and `InformDates` are modeled by the tasks labelled by `request(mi,`**`DoAL`**`(InformParticipants(mid)))` and `request(mi,` **`DoAL`**`(InformDates(mid))` (the parameter corresponding to the requesting agent, `ms`, is omitted for brevity). Here, **`DoAL`** (see Section 4.3.7.1) is an abbreviation that means that the task must be executed by the requested agent but that other concurrent activities are allowed. In our approach, this is a standard way to request an execution of a specific procedure. Thus, requests with **`DoAL`** are used to establish task dependencies among actors.

However, there are a lot of tasks that request or provide services that are still too complex to label with individual CASL communicative actions. For example, the task

`AskParticipants` is used to model the request for information on available dates from all the participants. It, therefore, must send requests to all the participants. This complex request cannot be replaced with a single `request` action. Similarly, the tasks `RequestAllParticipants` (modeling requests for meeting participation), `GetConfirmation` (that models requests for confirmation of meeting attendance), `SendMeetingInfoToMPs` (informing all the participants of the meeting details), and `InformWhenKnown` (that informs the initiator about the meeting details once they are known) are also too complex to express with a single communicative action and remain as they are until the next version (iASR) of the diagram. They will be easier to specify at that stage, where iteration and other control structures will be available.

6.6 Developing iASR Models

In this section, we look at how the SR diagrams developed in the previous sections are refined to get the corresponding iASR diagrams for all of the actors in the combined system. The process of evolving existing SR diagrams into iASR ones involves adding annotations and goal/task parameters, specifying interactions in details, deidealizing goals, replacing resource dependencies with goal/task ones, adding self-acquired goals, etc. Some of these activities (i.e., adding parameters, replacing resource dependencies, and providing details on agent interactions) have already been done for the more complex diagrams in the previous section.

Note that quite a few annotations (e.g., the sequencing of tasks) in the iASR diagrams presented below come from the environment, so while a lot of the activities at this stage of the methodology could be attributed to high-level design, considerable amount of work done during the creation of iASR diagrams is spent on creating a more detailed model of the environment and the requirements for the system-to-be.

6.6.1 Developing the iASR Model for the Meeting Initiator

In this section, we show how the SR model for the Meeting Initiator is turned into an iASR model (see Figure 6.17). Here, we add composition and link annotations as well as handle self-acquired goals in detail. We will now go over the changes to the model.

We start by observing that the MI actor must be capable of initiating many meetings. This was noted during the development of the second version of the SR diagram for the actor and the `mid` parameter, which uniquely identifies the meeting, was added to all the goals and tasks in the model. The goal `MeetingSetup` is the main goal of the MI and represents the need to schedule a specific meeting. In order for the MI to be able to acquire instances of that goal repeatedly we add an interrupt link annotation to the goal node `MeetingSetup` (see Figure 6.17). The interrupt label reads `whenever(`**`Goal`**`(MeetingSetup(mid),sD)`. It will fire whenever the MI has an unachieved goal that `MeetingSetup(mid)` for some binding of the parameter `mid` in its mental state. We omit some parameters in the annotations for brevity (e.g., the parameter that indicates which agent has the goal). The cancellation condition `sD` stands for `SystemDone`, which remains false as long as the system is running, thus making the interrupt fire a potentially unlimited number of times during a run of the system. Since in our approach we only handle achievement goals of the form **`Eventually`**`(`**`Goal`**`(agt,φ))`, we omit **`Eventually`** in the annotations.

In the analysis of the meeting scheduling process, it was determined that the initiator's goal `MeetingScheduled` is not always achievable. There may be situations when, for example, the intended meeting participants do not have any common available dates. Therefore, `MeetingScheduled` is an example of a goal that is too ideal. The modeler must then decide on how to relax this goal to make it always achievable or change the process model so that it can handle failure to achieve the goal. Formal goal definitions can substantially help in determining whether a goal is achievable and in relaxing a goal to make it achievable — this is called goal *deidealization*. However, at this point in the methodology, formal goal definitions are not always available, so the analyst can just re-label the goal node to show that is not always achievable. Here, we rename the goal node `MeetingScheduled` with `MeetingScheduledIfPossible`.

In the previous SR model of Figure 6.15, the task `SetupMeeting` was decomposed into two subgoals, `MeetingScheduled` and `InitiatorInformed`. These are examples of self-acquired goals that do not come from inter-agent dependencies, but are used to enhance the models. Models with self-acquired goals must be modified to show the goal acquisition explicitly (see Section 4.3.8 for details). The acquisition of the goal is done by the execution of the `commit` action, whose effect is the addition of the goal to the mental state of the agent. It is followed by a goal node annotated with the guard link annotation labelled by the goal. This allows the agent to recognize the presence of the goal in its mental state and modify its behaviour accordingly. Thus, the goal node `MeetingScheduledIfPossible` is preceded by the task `commit( MeetingScheduledIfPossible(mid))` that acquires the goal for the agent. The goal node now has a guard link annotation `guard(` **`Goal`**`(MeetingScheduledIfPossible(mid)))`.

A similar transformation applies to the goal `InitiatorInformed`. The corresponding goal node now has a guard link annotation and is preceded by the appropriate `commit` task. Note that the `commit` tasks must precede the goal nodes since they have to be executed before the process blocks while waiting for the guard condition to become true. Since the default composition annotation is sequence, it is enough to order the tasks/goals in the correct sequential order left to right. Note that the means for achieving the goal `InitiatorInformed` is a request action that asks the Meeting Scheduler to achieve the goal `PartricipantInformed(mi,mid)`. We make this change to the model since it was determined that the information the initiator requires is the same information that the meeting participants need. Thus, we can simplify the model for the MS and provide the information uniformly for all the interested parties (see Section 6.6.5).

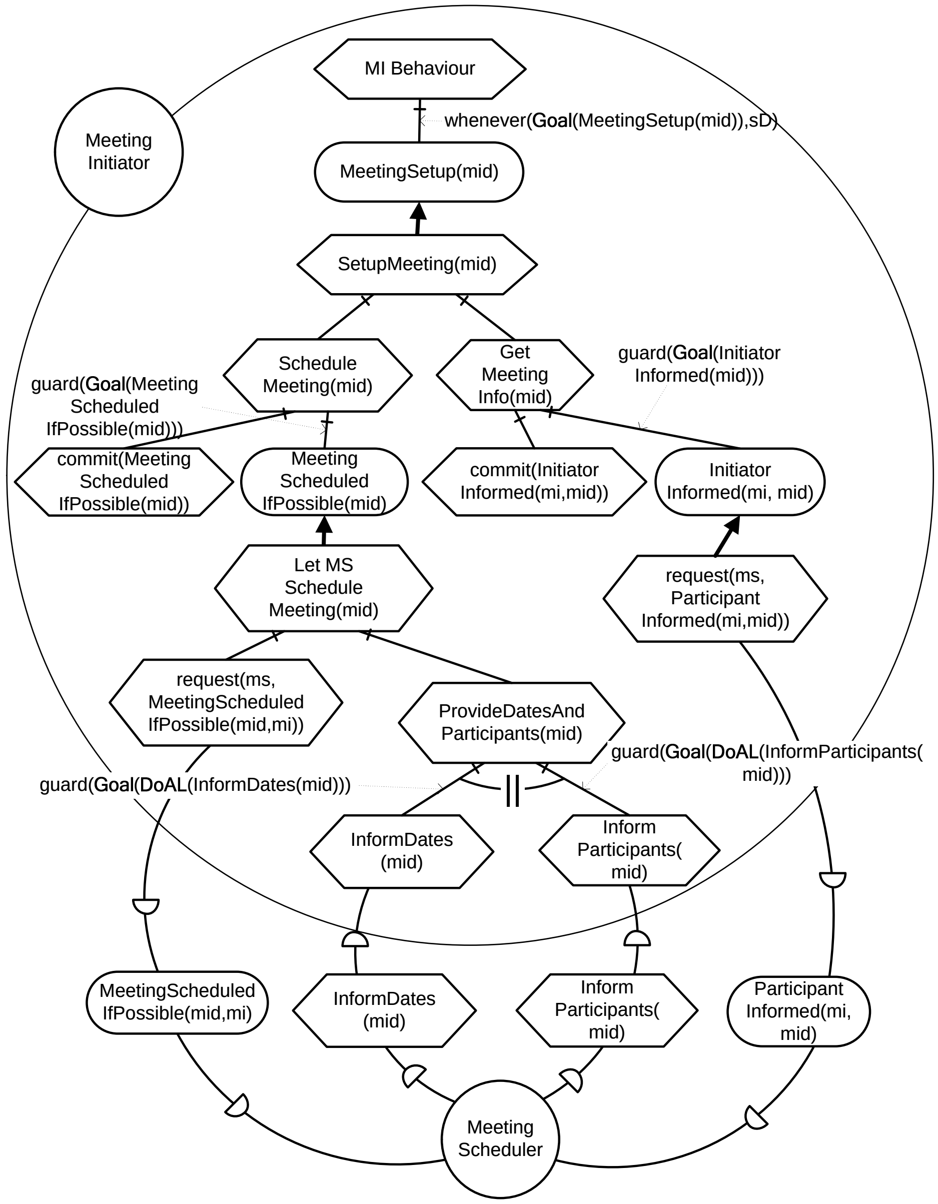

MI Behaviour
whenever(Goal(MeetingSetup(mid)),sD)
Meeting Initiator
MeetingSetup(mid)
SetupMeeting(mid)
Schedule Meeting(mid)
Get Meeting Info(mid)
guard(Goal(Initiator Informed(mid)))
guard(Goal(Meeting Scheduled IfPossible(mid)))
commit(Meeting Scheduled IfPossible(mid))
Meeting Scheduled IfPossible(mid)
commit(Initiator Informed(mi,mid))
Initiator Informed(mi, mid)
Let MS Schedule Meeting(mid)
request(ms, Participant Informed(mi,mid))
request(ms, MeetingScheduled IfPossible(mid,mi))
ProvideDatesAnd Participants(mid)
guard(Goal(DoAL(InformParticipants(mid)))
guard(Goal(DoAL(InformDates(mid)))
InformDates (mid)
Inform Participants(mid)
MeetingScheduled IfPossible(mid,mi)
InformDates (mid)
Inform Participants(mid)
Participant Informed(mi, mid)
Meeting Scheduler

Figure 6.17. The iASR diagram for the Meeting Initiator.

The task `LetMSScheduleMeeting` is the means for achieving the goal `MeetingScheduled`. It is decomposed into two subtasks that request the meeting to be scheduled by the MS and provide the information on the intended participants and the suggested meeting dates. The second subtask is `ProvideDatesAndParticipants`. It is decomposed into two subtasks that provide the MS agent with the suggested dates and intended participants. These subtasks, `InformDates` and `InformParticipants` are executed in parallel (note the concurrency link annotation). These tasks are executed when the MI agent acquires the goals to execute them. This is done with the appropriate guard annotations, `guard(`**`Goal`**`(`**`DoAL`**`(InformDates(mid))))` and `guard(`**`Goal`**`(`**`DoAL`**`( InformParticipants(mid))))`, that block the execution until the goals are in the MI's mental state. Guard annotations are used instead of interrupts, since the procedures must be executed just once for each meeting.

6.6.2 Developing the iASR Model for the MRBS

In this section, we develop a detailed iASR model that represents the legacy MRBS system. We add parameters to tasks/goals, show the internals of the `RoomBookedCap` goal capability, and add link and composition annotations.

In order to label the tasks that communicate with other actors with the appropriate CASL communication actions and provide the necessary link annotations that describe conditions under which the task are to be executed and the goals are to be achieved, one needs to incorporate certain elements of the formal domain model such as predicate and functional fluents that model domain properties. Therefore, the formalization of the *i** process model in our approach, in fact, starts before the iASR model is mapped into the corresponding CASL specification. It starts during the development of iASR diagrams: the modeler needs to develop the formal model (or at least some parts of it) while introducing annotations, specifying interactions, etc.

There are two high-level services that the MRBS agent is capable of providing. Both of them are means of supporting intentional dependencies coming from the Meeting

Scheduler agent. The first dependency makes the MRBS agent acquire the goal RoomBooked(mid,d), see Figure 6.18. Here, the second argument denotes the date on which the scheduler wants the room booked for the meeting mid. This goal is a subgoal of the task MRBSBehaviour that models the behaviour of this agent. In the SR diagram of Figure 6.11 we also had a resource dependency Room# where the MS agent depended on the MRBS for the information on whether a room was booked for a particular meeting and which room it was. This dependency is now replaced with a task dependency labelled ConfirmRoom(ms,mid,d), see Figure 6.18. The task ConfirmRoom is executed whenever requested by the MS agent.

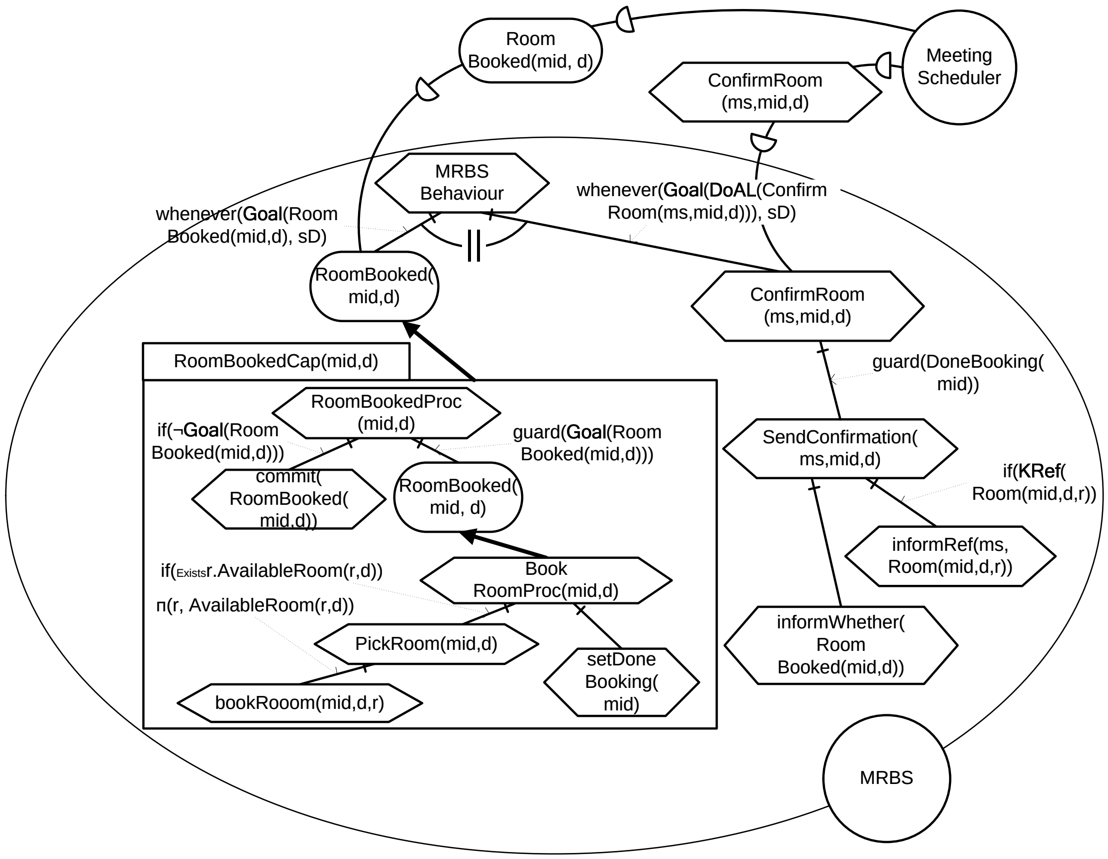


Figure 6.18. The iASR model for the MRBS agent.

The main goal and task are executed in parallel and each is accompanied by the appropriate interrupt link annotation that fires when the associated goal becomes part of the mental state of the agent. For the task dependency the goal that is acquired is `DoAL(ConfirmRoom(ms,mid,d))`. It is adopted as a result of a request to the MRBS to execute the specified task. The cancellation condition for both interrupts is `SystemDone`, which means that the interrupts will be firing until the system is terminated.

The only means for achieving the goal `RoomBooked` is the goal capability with the name `RoomBookedCap`. Unlike Figure 6.11, where the capability was shown in its abbreviated form, here it is presented in full detail. The top-level task of the capability is `RoomBookedProc(mid,d)`, which books a meeting room on the day `d` for a meeting with the meeting ID `mid`. It is decomposed into a task and a goal. The task is the `commit` action that acquires the goal `RoomBooked` for the requested parameter bindings (see Section 4.5.1 for a discussion of capabilities and how they are modeled). The `commit` action must be executed only if the required goal is not already in the mental state of the agent, therefore the task is used with a conditional link annotation. This is done so that a goal capability can be used anywhere a task can be used in an iASR diagram. The way it is used in the model for the MRBS, the `commit` action is not going to be executed since the goal is acquired through an intentional dependency. The subgoal `RoomBooked` is used with the guard annotation that unblocks when the agent has the goal `RoomBooked` in its mental state. The means of achieving the goal is the task `BookRoomProc(mid,d)`. This task is further decomposed into two tasks: `PickRoom(mid,d)`, which is executed if there is an available room to schedule the meeting `mid` on the date `d`, and the task `setDoneBooking(mid)`, which sets the flag `DoneBooking(mid)` to true after the MRBS attempts to book a room for the meeting `mid`. `PickRoom` has one subtask, `bookRoom(mid,d,r)`, which is used with a pick annotation (π) that non-deterministically selects a room `r` that is available on the day `d` (`AvailableRoom(r,d)` must hold).

The task `ConfirmRoom` has one subtask. When the MRBS tries booking a room for the meeting `mid` (note that the guard condition is `DoneBooking(mid)`), the task `SendConfirmation` is executed. It first informs the MS agent whether a room was booked for the meeting `mid` on the date `d`. This is true if the fluent `RoomBooked(mid,d)` holds. The CASL communicative action `informWhether` is used to send this information to the scheduler. The second subtask is executed conditionally. If the MRBS knows which room was booked for the meeting, it will send this information to the MS.

We note here that while the goal of the capability is called `RoomBooked` and the capability itself is called `RoomBookedCap`, the goal `RoomBooked` has, in fact, been deidealized. Since the supply of meeting rooms is limited, the MRBS cannot guarantee a successful booking at all times if it wants to maintain a consistent booking schedule. Therefore, if it cannot book a room, it will reply accordingly during the execution of the task `ConfirmRoom`.

6.6.3 Developing the iASR Model for the Meeting Participant

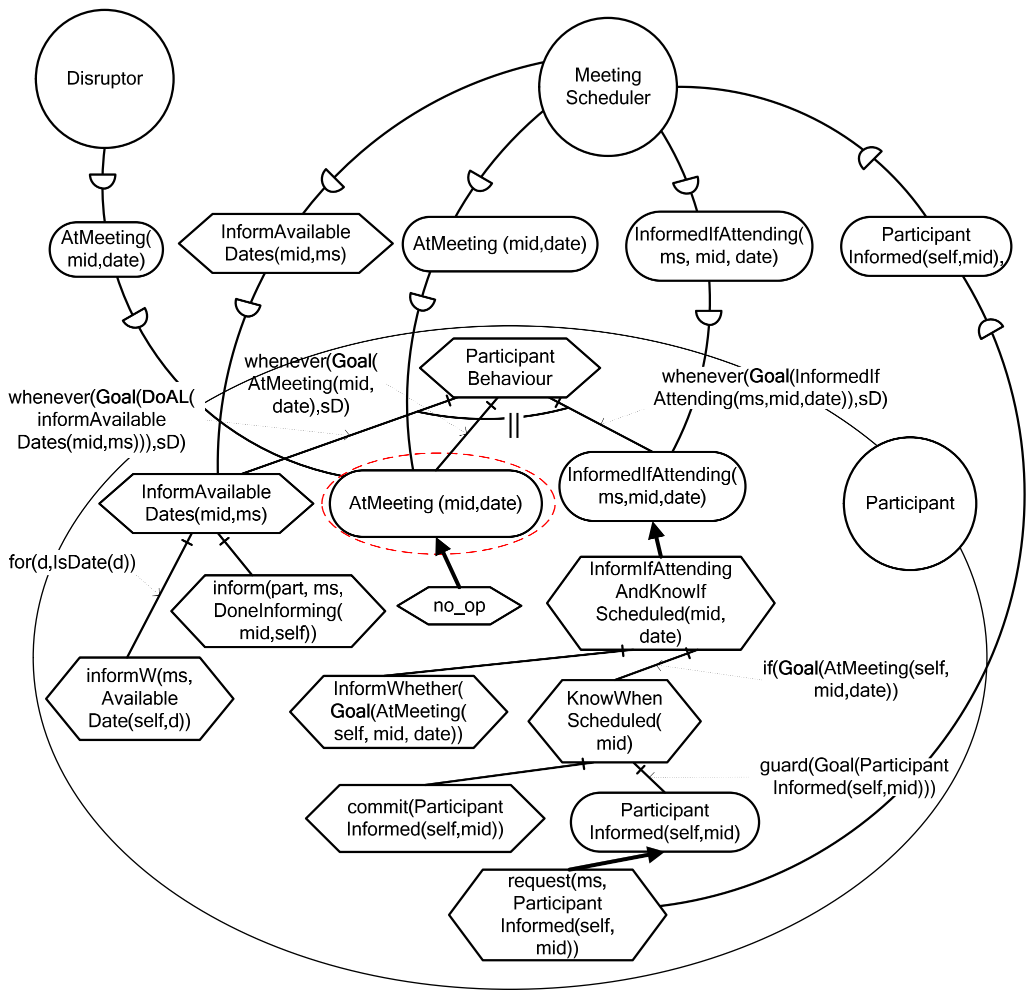
In this section, we provide the iASR model for the Meeting Participant actor. The diagram was reworked to reflect the presence of a new dependency where the scheduler needs a confirmation of meeting attendance from the participant; see Figure 6.19. The earlier SR diagram was in Figure 6.12.

The task `ParticipantBehaviour` is decomposed into a subtask and two subgoals corresponding to the main services provided by the participant. We added a concurrency annotation to this decomposition to show that the goals are to be achieved concurrently and with equal priority. The first task is `InformAvailableDates`. It is acquired from the MS through a task dependency. The task node is accompanied by the interrupt annotation that monitors for requests to execute the task `InformAvailableDates` and has a cancellation condition `SystemDone`. The task informs the scheduler about participants' availability. It first sends the availability of the participant for all the dates to the MS and then tells the scheduler that the participant has finished sending the requested information.

The first subgoal of `ParticipantBehaviour` is the goal `AtMeeting`. Participants are requested to participate in meetings by the MS agent as well as by the Disruptor actor. Note that in Figure 6.12 the only means of achieving the goal `AtMeeting` was the task `AttendMeeting`. Since we are mainly interested in the scheduling of meetings, we decided not to model the actual meeting attendance. We assume that once the participant has the goal `AtMeeting(mid,date)` for a particular meeting on a particular date, it will honour this commitment and attend the meeting. Thus, we could have omitted it from the diagram. However, we kept it there to illustrate that the goal dependency `AtMeeting` still exists. In order to respect the mapping rules for goal nodes, which require the presence of an achievement procedure, we added an empty procedure (`no_op`) as a means of achieving this goal.

The second subgoal of `ParticipantBehaviour` is the goal `InformedIfAttending(ms,mid,date)`, which is acquired from the MS and represents the need of the Meeting Scheduler to know whether the participant has, in fact, adopted the goal of attending the meeting `mid` on the date `date`. The means of achieving this goal is the task `InformAttendingAndKnowIfScheduled`. This task is responsible for informing the scheduler whether the participant intends to attend the meeting and also for finding out the details of the meeting from the MS. The task is decomposed into two subtasks. The first one is a communicative action `informWhether` that is used to inform the MS agent about the truth value of the expression **`Goal`**`(AtMeeting(self,mid,date))`, which means that the goal `AtMeeting` for a particular meeting ID and date is in the mental state of the agent. The second subtask is called `KnowWhenScheduled`. It is executed only if the participant is going to attend the meeting — note the *if* link annotation with the condition being the presence of the goal `AtMeeting` in the mental state of the agent. The self-acquired goal `ParticipantInformed(self,mid)` models the need of the participant for the meeting details. As a self-acquired goal, it is accompanied by the corresponding `commit` action.

The means of achieving the goal is to delegate it to the Meeting Scheduler agent through an appropriate request communication.



Disruptor
Meeting Scheduler
AtMeeting(mid,date)
InformAvailable Dates(mid,ms)
AtMeeting (mid,date)
InformedIfAttending(ms, mid, date)
Participant Informed(self,mid),
Participant Behaviour
whenever(Goal(AtMeeting(mid, date),sD)
whenever(Goal(DoAL(informAvailable Dates(mid,ms))),sD)
whenever(Goal(InformedIf Attending(ms,mid,date)),sD)
InformAvailable Dates(mid,ms)
AtMeeting (mid,date)
InformedIfAttending(ms,mid,date)
Participant
for(d,IsDate(d))
inform(part, ms, DoneInforming(mid,self))
no_op
InformIfAttending AndKnowIf Scheduled(mid, date)
informW(ms, Available Date(self,d))
InformWhether(Goal(AtMeeting(self, mid, date))
KnowWhen Scheduled(mid)
if(Goal(AtMeeting(self, mid,date))
guard(Goal(Participant Informed(self,mid)))
commit(Participant Informed(self,mid))
Participant Informed(self,mid)
request(ms, Participant Informed(self, mid))

Figure 6.19. The iASR model for the Meeting Participant.

6.6.4 Developing the iASR Model for the Disruptor

In this section, we develop the iASR diagram for the Disruptor actor, see Figure 6.20. Its process specification is very simple and the Disruptor's iASR diagram does not differ much from the SR one in Figure 6.13. Here, we add goal/task parameters, annotations, as well as provide details of the Disruptor's communication with the Meeting Participant.

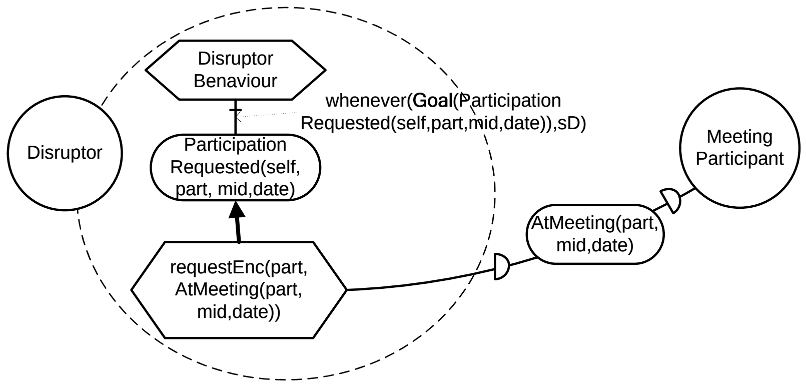

Disruptor
Disruptor Benaviour
whenever(Goal(Participation Requested(self,part,mid,date)),sD)
Participation Requested(self, part, mid,date)
requestEnc(part, AtMeeting(part, mid,date))
AtMeeting(part, mid,date)
Meeting Participant

Figure 6.20. The iASR model for the Disruptor.

The top-level task, `DisruptorBehaviour`, has one subgoal — `ParticipationRequested(ptcp,mid,date)`, which models the need of the Disruptor to invite the participant `ptcp` to the meeting `mid` on the date `date`. The goal node is accompanied by the interrupt link annotation that fires whenever there is a goal `ParticipationRequested` in the mental state of the Disruptor for some binding of the parameters. The means to achieve the goal is the task that delegates the goal `AtMeeting` to the Meeting Participant. Here, we label the task with the CASL communicative action `requestEnc`, an encrypted request, as introduced in [Shapiro and Lespérance, 2001]. We cannot use an ordinary request because in the formalization for the communicative actions of CASL all agents are aware of all actions; see [Shapiro *et al.*, 1998]. For simplicity, the communicative actions that we use in this case study are not encrypted, which means that all agents are aware of all the requests and know all the information that is being transmitted. However, in order to model the actions of the Disruptor correctly, i.e., the MS must be unaware of the fact that previously available dates have been booked by the Disruptor, we use `requestEnc` for an encrypted request. Only the sender and the receiver of the encrypted requests know the content of the message. We use this communicative action to make sure that only the meeting participant who is invited to attend a meeting by the Disruptor knows about it.

6.6.5 Developing the iASR Model for the Meeting Scheduler

This section presents the iASR diagram for the Meeting Scheduler. In this version of the model, we modify the process to make it more robust. This includes the use of a locking mechanism to avoid attempting to schedule several meetings on the same date and the handling of failures to find a meeting date and to book a meeting room.

Note that compared to the SR model in Figure 6.16 we are much less idealistic in specifying the iASR model for the Meeting Scheduler. In Figure 6.16, once an agreeable date was found, the meeting participants were requested to meet on that date and a meeting room was booked for that date. In the iASR diagram in Figure 6.21, on the other hand, we modify the model of the process to handle several types of possible failures. The kinds of failures that the new process is able to handle are the absence of common agreeable dates, the absence of available meeting rooms, and the participants refusing to commit to attending meetings. Also, we model the effects of the Disruptor actor more carefully. The Disruptor makes the meeting participants occupy their free time slots with meetings, thus reducing the number of such slots available for meetings organized by the Meeting Scheduler. Because of encrypted requests from the Disruptor (see Section 6.6.4), the MS is not aware of the changes to the participants' availability due to the actions of the Disruptor. It may be the case that after the participant had informed the MS about its availability on a particular date, the date became occupied by a meeting booked by the Disruptor. The scheduler may try to schedule a meeting on that date since it is unaware of the change. Since we assume that the participants' meeting slots are allocated on a first come, first serve basis, the participant with the date already occupied by the Disruptor's meeting will be unable to accept a new meeting on that same date. This is why the MS must ask for meeting confirmation after requesting meeting participation from participants. The MPs that had the date occupied by the Disruptor will reply negatively, thus making the date unavailable for the meeting. In this case, the scheduler will have to cancel the meeting on that date and try another common agreeable date. Similarly, if all the participants accepted the meeting date, but no meeting room can be found by the MRBS on that date, the meeting must also be cancelled and another date must be tried.

The Meeting Scheduler could be scheduling multiple meetings involving some of the same participants concurrently. In order not to attempt to schedule two meetings on the same date for the same participant, we use a simple locking mechanism. When the MS tries to schedule some meeting on a particular date, it is locked for all the participants of that meeting so that the scheduler does not attempt to schedule another meeting on that date for the same participants.

Let us take a look at the iASR diagram for the Meeting Scheduler (see Figure 6.21). The top-level task is still `MSBehaviour`, which represents the overall process executed by the MS. It is decomposed into two subgoals that model the major services that the agent provides. Both subgoals' nodes are accompanied by the usual interrupt annotations that fire when the goals are in the mental state of the scheduler. The cancellation condition for both interrupts is `SystemDone`. The two goals are to be achieved in parallel.

The first subgoal is `MeetingScheduledIfPossible`. This is a change from the previous diagram for the MS, which had the goal `MeetingScheduled` (see Figure 6.16). Here, we deidealized the goal `MeetingScheduled` after realizing that it cannot always be achieved. The means that achieve the new goal will attempt to schedule meetings, but it is also acceptable if they fail due to the absence of common time slots in participants' schedules or due to the absence of meeting rooms on the day of the meeting. The formal definition of this deidealized goal is presented during the mapping of the iASR model into CASL. The means of achieving the goal `MeetingScheduledIfPossible` is the task `TryToScheduleMeetingsAndBookRoom`. This task models the bulk of the Meeting Scheduler's process. Let us now look at how the task is decomposed. First, the MS needs to get the dates that the initiator is willing to meet on as well as the list of the meeting participants. To get that information, the MS asks the MI to execute the tasks `informDates` and `informParticipants` respectively. Thus, we have two task nodes labelled with `request` communicative actions that request the execution of these specific tasks (note the use of **`DoAL`** inside the requests).

Then the task `GetRidOfWeekendDates(mid)` is executed. This task is present in the decomposition since the MS is responsible for making sure that no meeting is scheduled on a weekend date. The reason for this is the goal dependency `NoMeetingsOnWeekends` from the *System* agent. In order not to schedule a particular meeting on a weekend, the scheduler can remove weekend dates from the list of suggested dates for a meeting. Thus, the task `GetRidOfWeekendDates` is decomposed into the task `removeWeekendDate(mid,d)`, which is accompanied by a *for* loop link annotation that iterates through the list of suggested dates and picks the ones that fall on weekends.

Next, the agent acquires the goal `AvailableDatesKnown(mid)`, which models the need of the MS to know the dates on which the intended meeting participants are available. A self-acquired goal is used here for flexibility. The `commit` action for the goal is executed once the MS knows all the participants of the meeting `mid`. The acquisition of the goal makes the condition of the guard annotation that accompanies the goal node `AvailableDatesKnown` true. The model specifies one means for the achievement of the goal. In order to know about the availability of the meeting participants, the MS asks each of the participants (note the *for* loop annotation) to execute the task `InformAvailableDates` that informs the scheduler about the availability of every date according to the participant's schedule. Thus, the request is the origin of the task dependency `InformAvailableDates`.

The next step in the meeting scheduling process is to determine the agreeable dates for the meeting. These are the dates on which all of the meeting participants and the meeting initiator are available. It is determined that the computation of available dates can be easily done declaratively through the use of the defined fluent

`AgreeableDate(mid,d,s)`, which holds if in the situation `s` the date `d` is available for all the participants and the initiator of the meeting `mid` (according to the MS's information). The formal definition of this fluent is presented in Section 6.7.5. Therefore, there is no explicit task for this step.

The next subtask in the task decomposition of `TryToScheduleMeetingAndBookRoom` is the task `TryAgreeableDates`, which is executed as soon as the MS has been informed about the availability of all the meeting participants (note the guard link annotation). This task is further decomposed into two subtasks. The first one, `TryDates(mid,d)`, is executed in a *for* loop iterating through all of the agreeable dates, while the second one, `ProcessFailure(mid)`, is executed when the meeting `mid` has not been scheduled after going through all of the agreeable dates. Here, the task is used with the conditional annotation `if(¬SuccessfullyScheduled(mid))`. `SuccessfullyScheduled(mid)` is a defined predicate fluent that holds if a meeting was successfully scheduled on some date and a room was booked for the meeting on that date (the situation argument is omitted here; see the definition of the fluent in Section 6.7.5).

The task `TryDates` has one subtask, `TryOneDate`, used with the conditional annotation `if(¬SuccessfullyScheduled(mid))`. This annotation prevents the scheduler from trying more agreeable dates for the meeting `mid` when the meeting has already been successfully scheduled. In that case, the fluent `SuccessfullyScheduled(mid)` holds and the task `TryOneDate` is not executed. The reason we need the two tasks, `TryDates` and `TryOneDate`, is that we only allow one link annotation per task/goal node. Therefore, if one wants a task to be executed in a loop and under a certain condition, one needs to use two task nodes, one being a supertask and another being a subtask, to be accompanied by a *for* loop and a conditional annotations respectively.

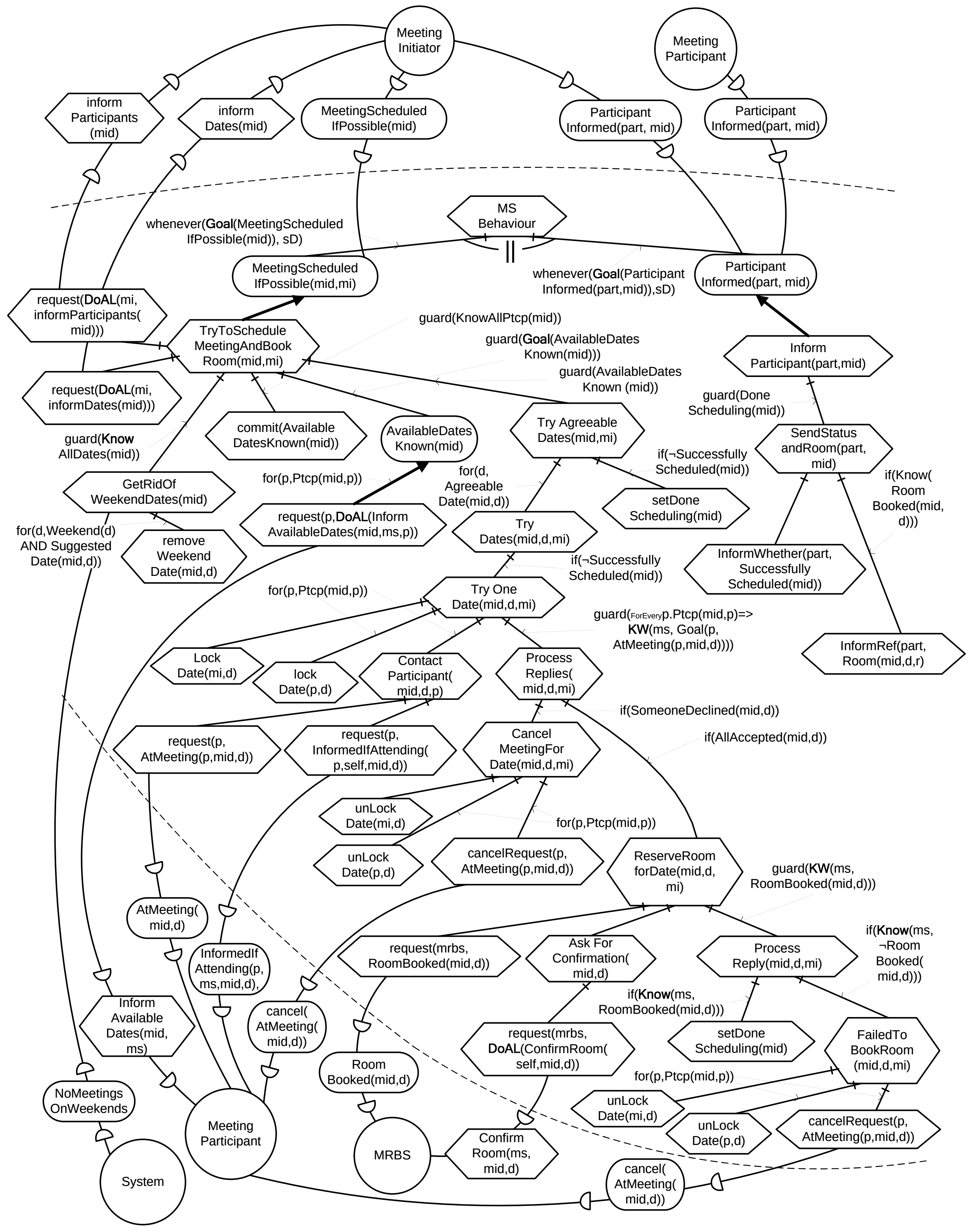

Meeting Initiator
Meeting Participant
inform Participants (mid)
inform Dates(mid)
MeetingScheduled IfPossible(mid)
Participant Informed(part, mid)
Participant Informed(part, mid)
MS Behaviour
whenever(Goal(MeetingScheduled IfPossible(mid)), sD)
MeetingScheduled IfPossible(mid,mi)
whenever(Goal(Participant Informed(part,mid)),sD)
Participant Informed(part, mid)
request(DoAL(mi, informParticipants(mid)))
TryToSchedule MeetingAndBook Room(mid,mi)
guard(KnowAllPtcp(mid))
guard(Goal(AvailableDates Known(mid)))
guard(AvailableDates Known (mid))
Inform Participant(part,mid)
request(DoAL(mi, informDates(mid)))
commit(Available DatesKnown(mid))
AvailableDates Known(mid)
Try Agreeable Dates(mid,mi)
guard(Done Scheduling(mid))
SendStatus andRoom(part, mid)
guard(Know AllDates(mid))
GetRidOf WeekendDates(mid)
for(p,Ptcp(mid,p))
request(p,DoAL(Inform AvailableDates(mid,ms,p)))
for(d, Agreeable Date(mid,d))
if(¬Successfully Scheduled(mid))
setDone Scheduling(mid)
if(Know(Room Booked(mid, d)))
for(d,Weekend(d) AND Suggested Date(mid,d))
remove Weekend Date(mid,d)
Try Dates(mid,d,mi)
if(¬Successfully Scheduled(mid))
InformWhether(part, Successfully Scheduled(mid))
for(p,Ptcp(mid,p))
Try One Date(mid,d,mi)
guard(ForEvery p.Ptcp(mid,p)=> KW(ms, Goal(p, AtMeeting(p,mid,d))))
InformRef(part, Room(mid,d,r)
Lock Date(mi,d)
lock Date(p,d)
Contact Participant(mid,d,p)
Process Replies(mid,d,mi)
if(SomeoneDeclined(mid,d))
request(p, AtMeeting(p,mid,d))
request(p, InformedIfAttending(p,self,mid,d))
Cancel MeetingFor Date(mid,d,mi)
if(AllAccepted(mid,d))
unLock Date(mi,d)
unLock Date(p,d)
for(p,Ptcp(mid,p))
cancelRequest(p, AtMeeting(p,mid,d))
ReserveRoom forDate(mid,d, mi)
guard(KW(ms, RoomBooked(mid,d)))
AtMeeting(mid,d)
InformedIf Attending(p, ms,mid,d),
request(mrbs, RoomBooked(mid,d))
Ask For Confirmation(mid,d)
Process Reply(mid,d,mi)
if(Know(ms, ¬Room Booked(mid,d)))
Inform Available Dates(mid, ms)
cancel(AtMeeting(mid,d))
if(Know(ms, RoomBooked(mid,d)))
request(mrbs, DoAL(ConfirmRoom(self,mid,d))
setDone Scheduling(mid)
FailedTo BookRoom (mid,d,mi)
Room Booked(mid,d)
for(p,Ptcp(mid,p))
NoMeetings OnWeekends
unLock Date(mi,d)
unLock Date(p,d)
cancelRequest(p, AtMeeting(p,mid,d))
Meeting Participant
MRBS
Confirm Room(ms, mid,d)
System
cancel(AtMeeting(mid,d))

Figure 6.21. The iASR model for the Meeting Scheduler.

The task `TryOneDate` is decomposed as follows (see Figure 6.21). The first subtask locks the date `d` for the Meeting Initiator, while the second one, which is executed in a *for* loop iterating through the meeting participants and is labelled `lockDate(p,d)`, locks the date `d` for the participant `p` (see above for the discussion). Once the date is locked for all the participants, they are contacted by the Meeting Scheduler. This is represented by the task `ContactParticipant(mid,d)`, which is also executed for each of the participants of the meeting `mid`. It has two subtasks. The first requests that the participant `p` acquire the goal `AtMeeting(p,mid,d)`, thus agreeing to attend the meeting `mid` on the date `d`. This is a simple request, so we label the task node with the appropriate `request` communicative action. The second subtask of `ContactParticipant` queries the participant as to whether he/she has in fact adopted the goal. This task is also labelled with the `request` action that makes each participant acquire the goal `InformedIfAttending(ms,mid,d)`.

Once the participants have been asked to send their meeting participation confirmations, the Meeting Scheduler can process their replies. To make sure that the scheduler has all the replies, the task `ProcessReplies(mid,d)` is only executed once its guard condition, which states that for all the participants of the meeting `mid` the MS knows whether they have adopted the goal to be at that meeting, is true. The task `ProcessReplies` has two subtasks, `CancelMeetingForDate(mid,d)` and `ReserveRoomForDate(mid,d)`, that are used to handle the situation where some participants declined the meeting and where all participants accepted the meeting respectively. The fluents `SomeoneDeclined(mid,d)` and `AllAccepted(mid,d)` are used in the conditional annotations that accompany the tasks. They are defined in Section 6.7.5. When some participant declines to meet on some date `d`, the Meeting Scheduler unlocks the date `d` for all the meeting participants and then cancels its requests to meet on that date (see the decomposition of `CancelMeetingForDate` in Figure 6.21).

If all meeting participants agree to meet on some date, the MS attempts to book a room for that meeting. This is modeled by the task node `ReserveRoomForDate`. The task is decomposed into three subtasks. The first one asks the MRBS agent to book a room for the meeting (we use the `request` action to make the MRBS adopt the goal `RoomBooked(mid,d)`), the second subtask requests confirmation from the MRBS of the room booking. Here, the scheduler asks the legacy system to execute the task `ConfirmRoom`, a procedure which sends information to the MS on whether the room was booked as well as the number of the booked room. The third subtask of `ReserveRoomForDate`, the task `ProcessReply`, is executed when the MS has the information on whether the room was booked (the guard link annotation with the condition **`KWhether`**`(ms,RoomBooked(mid,d))` is used). `ProcessReply` has two subtasks for handling the positive and negative replies about the room booking. If a room is booked, then the task `setDoneScheduling(mid)` is executed. If the MS is executing this task, it means that the meeting `mid` has been successfully scheduled on some date. This task is responsible for setting the flag `DoneScheduling(mid)` to true. The fluent `DoneScheduling(mid)`, when true, denotes that the scheduling process has been completed for the meeting `mid` (with either positive or negative result).

On the other hand, if the MRBS has been unsuccessful in booking a room, the scheduler executes the task `FailedToBookRoom(mid,d)` which unlocks the date `d` for all the participants (note the *for* loop annotation) and the initiator of the meeting `mid` and cancels its requests to meet on the date `d` since no room is available on that date. The fluent `SuccessfullyScheduled(mid)` remains false and the MS must either try another date or give up scheduling this meeting if there are no more agreeable dates left. This fluent (defined in Section 6.7.5) holds if the meeting is scheduled successfully and there is a room booked for it.

Now that we have explored the process of trying to schedule a meeting on a particular date, we show what happens if the meeting is not scheduled by the time the scheduler runs out of agreeable dates for the meeting `mid`. This is modeled by the task `ProcessFailure(mid)`, which is a subtask of `TryAvailableDates` and is executed on condition that the fluent `SuccessfullyScheduled(mid)` is false. If `setDoneScheduling(mid)` is executed in this place of the process, it means that the Meeting Scheduler has tried all agreeable dates and failed to schedule the meeting `mid`. Here, the fluent `DoneScheduling(mid)` must be set to true since the scheduling process is complete. This concludes the description of the process for achieving the goal `MeetingScheduledIfPossible`.

The second top-level goal of the MS, besides the goal `MeetingScheduledIfPossible`, is the goal `ParticipantInformed(ptcp,mid)`, which is acquired through intentional dependencies from the Meeting Initiator and the Meeting Participant. Here, the MS needs to inform the interested parties of the result of the meeting scheduling process. Thus, the means of achieving the goal is the task `InformParticipants`. It has one subtask called `SendStatusAndRoom(ptcp,mid)`. The scheduler informs the meeting participants whether the meeting was successfully scheduled and what room was booked for the meeting. However, we want this information sent only when the outcome of the process is known. So, the task `SendStatusAndRoom` is executed only when the MS has completed the scheduling process — it is accompanied by the annotation `guard( DoneScheduling(mid))`. When the MS successfully completes the scheduling of a meeting or when there is no chance of meeting being scheduled (no agreeable dates or all such dates have been tried), the fluent `DoneScheduling(mid)` is set to true. Only then can the MS inform the participants about the result of the scheduling. The task `SendStatusAndRoom` is decomposed into two subtasks. The first one informs participants about whether the meeting was scheduled (we use the CASL communicative action `informWhether` here). The truth value of the fluent `SuccessfullyScheduled(mid)` is sent to the participants (including the meeting initiator). The second subtask of `SendStatusAndRoom` informs the requesting participant which room was booked for the meeting. The task is executed on condition that the room is booked — it is accompanied by the annotation `if(`**`KRef`**`(ms,RoomBooked(mid,d)))`, which holds if the MS knows that a room was booked for the meeting `mid` on the date `d`.

6.7 Mapping iASR Diagrams into CASL

In this section, we show how the iASR diagrams developed in the previous sections are mapped into the corresponding formal CASL specification. First, we look at the basic domain model specified in CASL. Then we show how the iASR diagram for the Meeting Initiator (Figure 6.17) is mapped into CASL in detail. Most of the iASR diagrams for other agents are mapped in a similar fashion, so the mapping of these diagrams is presented in Appendix A. The rest of the section concentrates on showing the mapping of the MRBS' goal capability and on discussing the aspects of the formal CASL model for the system that involve new issues such as the definitions of goals, conditions, etc.

6.7.1 Specifying the Basic Domain Model

Here, we specify the foundation of the formal model for the meeting scheduler domain. We present the predicates used to model dates, participants, etc., some primitive and defined fluents that model certain aspects of the domain, formal definitions of agent goals, as well as the primitive actions executed by the agents in the system together with their preconditions and effects.

6.7.1.1 Modeling Meeting Dates

First, we describe the predicates that represent some of the most basic entities in the model — the meeting dates. They do not change their value and therefore are modeled as non-fluents.

- `IsDate(d)` specifies that `d` is a date. This means that it can potentially be used for scheduling meetings.
- `Weekend(d)` specifies that `d` is a weekend date. In the current meeting scheduling process, the MS cannot schedule meetings on such dates.

6.7.1.2 Meeting Participants and Suggested Meeting Dates

The next two predicate fluents represent meeting participants and meeting dates suggested by the initiator for each meeting.

- `Ptcp(mid,p,s)` specifies that the agent `p` is an intended participant of the meeting `mid` in situation `s`.
- `SuggestedDate(mid,d,s)` specifies that `d` is one of the suggested meeting dates for the meeting `mid` in situation `s`.

In the current model, we assume that these fluents are set by the appropriate initial state axioms. The fluent `Ptcp(mid,p,s)` does not change throughout the execution of the system. Note that it must nonetheless be made a fluent because its extension is not known by the MS initially:

```
Ptcp(mid,p,do(a,s)) ≡ Ptcp(mid,p,s)
```

The predicate fluent `SuggestedDate(mid,d,s)` can only be changed by the primitive action `removeWeekendDate(agt,mid,d)`, which is used to remove the MI-suggested meeting dates that fall on weekends:

```
SuggestedDate(mid,d,do(a,s)) ≡ SuggestedDate(mid,d,s)
                                        ∧  ∃agt[a≠removeWeekendDate(agt,mid,d)]
```

These predicates are extensively used in the meeting scheduling process for selecting meeting participants and meeting dates.

The task `removeWeekendDate(agt,mid,d)` can be executed only if the date `d` is one of the suggested meeting dates for the meeting `mid` and it is also a weekend date. It can only be executed by the Meeting Scheduler:

```
Poss(removeWeekendDate(agt,mid,d),s) ≡
        SuggestedDate(mid,d,s) ∧ Weekend(d) ∧ agt = MS
```

Note that in all the primitive actions in our model the first argument is the agent performing the action. We indicate that using the following axiom (we omit these axioms for other primitive actions):

```
Agent(removeWeekendDate(agt,mid,d)) = agt
```

6.7.1.3 Flags for Managing the Scheduling Process

The following predicate fluents are used as flags in the meeting scheduling process. Their values are changed manually by the MS agent.

- `DoneScheduling(mid,s)`. When it holds, `DoneScheduling` indicates that the scheduling process for the meeting `mid` has been completed by the Meeting Scheduler in situation `s`.
- `Locked(ptcp,d,s)` indicates whether the date `d` is locked for the participant `ptcp` in situation `s`. If the value of this fluent is true for some date and some agent, it means that the scheduler is currently trying to schedule a meeting that requires the agent's participation on the specified date and no scheduling of another meeting should be attempted on that date until it is unlocked.

Let us now look at the primitive actions executed by the MS agent that change the values of the above fluents. First, we look at the action `setDoneScheduling(agt,mid)`, which is used by the MS agent to set the fluent `DoneScheduling(mid,s)` to true. Below are the precondition axiom for the action as well as the successor state axiom for the fluent. The action `setDoneScheduling` can only be executed by the MS:

```
Poss(setDoneScheduling(agt,mid),s) ≡ agt = MS
```

The predicate fluent `DoneScheduling(mid,s)` is only affected by the action `setDoneScheduling`. The axiom below says that `DoneScheduling(mid)` is true after executing some action `a` in the situation `s` if the action is `setDoneScheduling(agt,mid)` or the fluent was true before the execution of the action:

```
DoneScheduling(mid,do(a,s))≡
 ∃agt[a=setDoneScheduling(agt,mid)] ∨ DoneScheduling(mid,s)
```

The fluent `Locked(ptcp,date,s)` indicates whether a date is locked for a particular participant. There are two primitive actions executed by the MS agent that lock and unlock dates. Here are the precondition axioms for the actions:

```
Poss(lockDate(agt,ptcp,date),s) ≡ ¬Locked(ptcp,date,s) ∧ agt = MS
Poss(unlockDate(agt,ptcp,date),s) ≡ Locked(ptcp,date,s) ∧ agt = MS
```

It is possible to lock a date for some participant if it is not already locked. Similarly, it is possible to unlock a date if it is locked. The successor state axiom for the fluent `Locked(ptcp,date,s)` shows that the fluent can only be changed by the above actions:

```
Locked(ptcp,date,do(a,s)) ≡ ∃agt[a=lockDate(agt,ptcp,date)] ∨
        (Locked(ptcp,date,s) ∧  ∀agt[a≠unlockDate(agt,ptcp,date)])
```

This successor state axioms states that for the fluent `Locked` for some participant `ptcp` and date `date` to be true after the execution of some action either the action was `lockDate` for `ptcp` and `date` or the fluent was already true for `ptcp` and `date` before the execution of the action and the action was not the `unlockDate` action for that participant and date.

6.7.1.4 Managing Participants' Schedules

The key primitive fluent used to manage the schedules of meeting participants is `AtMeeting(ptcp,mid,date,s)`. This fluent indicates that a meeting participant `ptcp` participates in the meeting `mid` on the date `date`. We are not modelling meeting attendance, so there are no actions that change this fluent's value and the successor state axiom for this fluent states that it never changes. It must nonetheless be a fluent since its value is not known to all agents:

$$\texttt{AtMeeting(ptcp,mid,date,do(a,s))} \equiv \texttt{AtMeeting(ptcp,mid,date,s)}$$

When scheduling meetings, the MS is interested in the participants' commitment to attend meetings. The presence of a goal that `AtMeeting(p,mid,d,s)` in the mental state of the participant `p` indicates such commitment. The assumption is that once committed to attending a meeting, a participant will honour this commitment. As mentioned earlier, the actual meeting attendance is out of the scope of the meeting scheduling system and therefore is not modeled. See [Shapiro *et al.*, 1998] on how to model this.

In order to maintain consistency of the participants' schedules we have the following axiom for the Meeting Participant:

$$\forall \texttt{agt}[\textbf{Know}(\texttt{agt},\forall \texttt{p,mid1,mid2,date}[\texttt{AtMeeting(p,mid1,date,now)} \wedge \texttt{AtMeeting(p,mid2,date,now)} \supset \texttt{mid1=mid2}],S_0)]$$

This axiom states that there could only be one meeting per participant per day in the initial situation. This axiom prevents participants from acquiring goals to participate in more than one meeting per day, thus maintaining the consistency of their schedules. Another axiom makes sure that no meeting can span more than one date:

$$\forall \mathtt{agt}[\mathbf{Know}(\mathtt{agt}, \forall \mathtt{p}, \mathtt{mid}, \mathtt{date1}, \mathtt{date2}[\mathtt{AtMeeting}(\mathtt{p}, \mathtt{mid}, \mathtt{date1}, \mathtt{now}) \wedge \mathtt{AtMeeting}(\mathtt{p}, \mathtt{mid}, \mathtt{date2}, \mathtt{now}) \supset \mathtt{date1}=\mathtt{date2}], S_0)]$$

Since there are no actions to change the fluent `AtMeeting`, the schedules retain these properties. We define the availability of a participant on a particular date as the absence of any meeting commitments:

$$\mathtt{AvailableDate}(\mathtt{p}, \mathtt{d}, \mathtt{s}) \overset{def}{=} \neg\exists \mathtt{mid}[\mathbf{Goal}(\mathtt{p}, \mathbf{Eventually}(\mathtt{AtMeeting}(\mathtt{p}, \mathtt{mid}, \mathtt{d}, \mathtt{now}), \mathtt{now}, \mathtt{then}), \mathtt{s})]$$

The above formula states that the date `d` is available for scheduling meetings for the participant `p` if there are no meetings that the participant is already committed to attend on date `d`.

6.7.1.5 Booking Rooms

In order to distinguish entities that are rooms from other entities, we introduce the predicate `Rooms(r)` that specifies that `r` is a meeting room. To model rooms being booked for meetings, we have the predicate fluent `Room(mid,d,r,s)`, which holds if the room `r` is booked on the date `d` for the meeting `mid`.

Rooms are booked by the MRBS agent using the action `bookRoom(agt,mid,date,room)`, which is executed inside its goal capability `RoomBookedCap`. To insure consistency of booking rooms, the precondition of the action `bookRoom` that books the room `r` for the meeting `mid` on the date `d` states that the room must not already be booked for another meeting on the same date:

```
            Poss(bookRoom(agt,mid,d,r),s) ≡
                 ¬∃mid2[Room(mid2,d,r,s)] ∧ agt = MRBS
```

The primitive fluent `Room` can only be changed by the execution of the action `bookRoom`:

```
Room(mid,d,r,do(a,s)) ≡ Room(mid,d,r,s) ∨ ∃agt(a=bookRoom(agt,mid,d,r))
```

The MRBS sets the predicate fluent `DoneBooking(mid,s)` to true once it is done booking a room for the meeting `mid`. It uses the action `setDoneBooking(agt,mid)` to do that. The action can only be executed by the MS. The successor state axiom for this fluent is presented below. It says that after executing some action `a` the fluent holds if it held in the previous situation or the action executed was the `setDoneBooking` action:

```
      DoneBooking(mid,do(a,s)) ≡

           DoneBooking(a,s) ∨ ∃agt(a=setDoneBooking(agt,mid))
```

In order to create an instance of the system, one needs to specify what dates are available for the scheduling of meetings using the predicate `IsDate(d)`, which dates fall on weekends using `Weekend(d)`, and what rooms are available for holding meetings using the predicate `Rooms(r)`. For each meeting to be scheduled, its participants and the list of suggested dates for it must be specified using the fluents `Ptcp(mid,p,`S_0`)` and `SuggestedDate(mid,d,`S_0`)` respectively (note that the situation parameter refers to the initial situation). To make the instance more interesting, one can block certain dates in participants' schedules using the fluents `AtMeeting(p,mid,date,`S_0`)`.

Note that the *System* agent is not being mapped into CASL since the Meeting Scheduler takes responsibility for making sure that no meetings are scheduled on weekends.

6.7.2 Creating the CASL Model for the Meeting Initiator

In this section, we map the iASR diagram for the Meeting Initiator (Figure 6.17) into the corresponding CASL model. At this point, the modeler needs to map the entities in the iASR diagram for the MI into the elements of the formal CASL model using the mapping rules presented in Chapter 4 of this thesis.

We remind here that composition annotations are mapped into the corresponding CASL operators — we mostly use the default annotation that maps into the sequence operator and the concurrency annotation that maps into the concurrency with equal priority operator. Link annotations are mapped into the corresponding CASL programming constructs; the conditions featured in the link annotations are mapped into CASL formulae. Each goal node is mapped into a CASL formula and an achievement procedure. Task nodes are mapped into CASL procedures or primitive actions. The mapping of a task decomposition must take into account the composition and link annotations.

Let us look at the mapping of the top-most task in the MI's iASR diagram in Figure 6.17. Since the Meeting Initiator is a role, the CASL procedure that the task node is mapped into needs to have a parameter for the agent that is playing the role of the Meeting Initiator. We call this parameter `mi`. Also, as mentioned in the previous sections, only the most important parameters were added to the goal and task labels in the iASR diagrams. In the CASL code, on the other hand, we are not only concerned with specifying, for example, what meeting the task is used for scheduling (the `mid` parameter in iASR diagrams), but also with making sure that all the goals/tasks have all the necessary information passed to them through parameters. Here, we worry not only about the information needed for a particular task, but also about the information needed for the tasks/goals that this task is decomposed into. That is why the number of parameters in the CASL procedures/formulae may increase with respect to the corresponding iASR task/goal nodes (see, for example, the fluent `MeetingSetup` in the code for `MeetingInitiatorBehaviour` below). In our model, we assume that the names of the Meeting Scheduler and the MRBS agent are constants, so they do not have to be passed as procedure and action parameters.

As can be seen from Figure 6.17, the task `MeetingInitiatorBehaviour` has one subgoal accompanied by the interrupt link annotation — `MeetingSetup`. In order to map this part of the iASR diagram into CASL, we need to provide the mapping for the goal `MeetingSetup`. Based on the mapping rules presented in Section 4.3.9.2, a goal node is mapped into a CASL formula and an achievement procedure. The goal node `MeetingSetup` is therefore mapped as follows:

```
m(MeetingSetup(mid))=<MeetingSetup(mid,mi),AchieveMeetingSetup(mid,mi)>,
```

where `MeetingSetup(mid,mi,ms)` is the formal definition of the goal (we omit the situation parameter here), a defined fluent, and `AchieveMeetingSetup(mid,mi,ms)` is the achievement procedure for the goal. Therefore, using the mapping function **m** (see Section 4.3.9.2), we can now show the code for the `MeetingInitiatorBehaviour` procedure, composed as required by the mapping rules:

```
proc MeetingInitiatorBehaviour(mi)
      <mid: Goal(mi,Eventually(m(MeetingSetup(mid)).formula(mid,mi))) ∧
            Know(mi,¬m(MeetingSetup(mid)).formula(mid,mi)) →
            m(MeetingSetup(mid)).achieve(mid,mi)
      until SystemDone>
endProc
```

We can then substitute expressions involving the mapping function **m** with the CASL formula `MeetingSetup(mid,mi)`, which expresses the goal `MeetingSetup` formally, and with the achievement procedure for this goal, which is the procedure `AchieveMeetingSetup(mid,mi)`:

```
proc MeetingInitiatorBehaviour(mi)
     <mid: Goal(mi,Eventually(MeetingSetup(mid,mi))) ∧
                Know(mi,¬MeetingSetup(mid,mi)) →
          AchieveMeetingSetup(mid,mi),
     until SystemDone>
endProc
```

Therefore, as per the mapping rules for goal nodes presented in Section 4.3.9.2, the CASL code for the task involves an interrupt, which is triggered when for some binding of the meeting ID `mid` there is a goal `MeetingSetup(mid,mi)` that the initiator knows has not yet been achieved. When the MI has such a goal, the interrupt fires and the achievement procedure for the goal `MeetingSetup` is executed for the newly bound parameter `mid`.

Let us now look at the procedure `AchieveMeetingSetup`. It encodes the way the goal `MeetingSetup` can be achieved. In the model in Figure 6.17, there is just one means for achieving that goal, the task `SetupMeeting`. So, the procedure `AchieveMeetingSetup` is quite simple:

```
proc AchieveMeetingSetup(mid,mi)
     m(SetupMeeting(mid))(mid,mi)
endProc
```

Here, we use the mapping function **m** to map the task node labelled `SetupMeeting(mid)` into the corresponding CASL procedure and supply the meeting ID and initiator parameters to it. As mentioned in Section 4.2.1, iASR task nodes are either mapped into procedures or primitive actions. Non-leaf task nodes are always mapped into procedures since they must be further decomposed. We map the task node `SetupMeeting` into a CASL procedure with the same name. Substituting the actual procedure name for **m**`(SetupMeeting(mid))` we get the following:

```
proc AchieveMeetingSetup(mid,mi)
      SetupMeeting(mid,mi)
endProc
```

Let us see what CASL formula the goal node `MeetingSetup` is mapped into. By comparing the decomposition of the task `SetupMeeting` with the decomposition of the task `Means_1` in Figure 4.32 that shows a generic iASR diagram with AND-decomposition of goals, one can see that the modeler AND-decomposed the goal `MeetingSetup` into the goals `MeetingScheduledIfPossible` and `InitiatorInformed`. So, formally the goal `MeetingSetup` is a conjunction of the two goals:

$$\mathtt{MeetingSetup(mid,mi,s)} \overset{def}{=} \mathtt{MeetingScheduledIfPossible(mid,mi,s)} \wedge \mathtt{InitiatorInformed(mid,mi,s)}$$

From here on, we will map iASR diagrams directly into CASL code, without going through intermediate steps as shown above. The mapping details are fully discussed in Chapter 4. All the task nodes will be mapped into CASL procedures/primitive actions with the same name (the number of parameters may increase as per the discussion above).

As can be seen in Figure 6.17, the task `SetupMeeting` is decomposed into two subtasks, `ScheduleMeeting` and `GetMeetingInfo`. Here is how the procedure `SetupMeeting` looks like in CASL (the two procedure calls in the above code are executed in sequence, note the "`;`" operator):

```
proc SetupMeeting(mid,mi)
      ScheduleMeeting(mid,mi);
      GetMeetingInfo(mid,mi)
endProc
```

The CASL procedure corresponding to the task `ScheduleMeeting` is more complex since the task decomposition involves the self-acquired goal `MeetingScheduledIfPossible`. The goal is acquired using the `commit` action (see Figure 6.17). The achievement procedure for the goal is executed once the goal is in the mental state of the initiator.

```
proc ScheduleMeeting(mid,mi)
      commit(mi,Eventually(MeetingScheduledIfPossible(mid,mi)));
      guard Goal(mi,Eventually(MeetingScheduledIfPossible(mid,mi))) do
            AchieveMeetingScheduledIfPossible(mid,mi)
      endGuard
endProc
```

The procedure for the task `GetMeetingInfo` is quite similar to the above procedure. Here, the initiator acquires the goal `InitiatorInformed` and then executes its achievement procedure:

```
proc GetMeetingInfo(mid,mi)
      commit(mi,Eventually(InitiatorInformed(mid,mi)));
      guard Goal(mi,Eventually(InitiatorInformed(mid,mi))) do
            AchieveInitiatorInformed(mid,mi)
      endGuard
endProc
```

The goal `MeetingScheduledIfPossible` has just one means that achieves it (see Figure 6.17), so its achievement procedure is also simple (note that the name of the Meeting Scheduler, `MS`, is a constant):

```
      proc AchieveMeetingScheduledIfPossible(mid,mi)
            LetMSScheduleMeeting(mid,mi,MS)
      endProc
```

The goal `InitiatorInformed` has one means for achieving it as well — the Meeting Initiator asks the MS agent to schedule the meeting for it. Here, we use the communicative action `request`:

```
      proc AchieveInitiatorInformed(mid,mi)
            request(mi,MS,Eventually(ParticipantInformed(mi,mid)))
      endProc
```

Note that the MI requests that the MS adopt the goal `ParticipantInformed` as the information required by the initiator is the same as for any participant. The achievement of the latter goal will automatically achieve the former. The formal definitions of the goals `ParticipantInformed` and `MeetingScheduledIfPossible` will be presented in Section 6.7.5.

The task `LetMSScheduleMeeting`, which is the means for achieving the goal `MeetingScheduledIfPossible`, is decomposed into a task that requests the achievement of the goal and a task that informs the MS about suggested meeting dates and meeting participants:

```
proc LetMSScheduleMeeting(mid,mi)
      request(mi,MS,Eventually(MeetingScheduledIfPossible(mid,mi)));
      ProvideDatesAndParticipants(mid,mi)
endProc
```

The task `ProvideDatesAndParticipants` is mapped into a procedure with the same name, which is responsible for monitoring for requests from the MS to execute the tasks `InformParticipants(mid,mi,MS)` and `InformDates(mid,mi,MS)` and for executing these tasks when such requests are received. Because of the presence of the concurrency annotation in Figure 6.17, the two tasks are executed in parallel:

```
      proc ProvideDatesAndParticipants(mid,mi)
            guard Goal(mi,DoAL(mi,InformDates(mid,mi,MS))) do
                  InformDates(mid,mi,MS)
            endGuard
            ||
            guard Goal(mi,DoAL(mi,InformParticipants(mid,mi,MS))) do
                  InformParticipants(mid,mi,MS)
            endGuard
      endProc
```

The tasks `InformDates` and `InformParticipants` are leaf-level tasks in the iASR diagram in Figure 6.17. The modeler chose not to decompose them further. However, full details must be provided in the CASL model for the Meeting Initiator. The procedure that the task `InformDates` is mapped into first sends to the MS the suggested meeting dates and then informs the scheduler that it now knows all the suggested dates for a particular meeting:

```
proc InformDates(mi,mid,ms)
      for d: SuggestedDate(mid,d) do
            inform(mi,ms,SuggestedDate(mid,d))
      endFor;
      inform(mi,ms,∀d.SuggestedDates(mid,d) ⊃
                                          Know(ms,SuggestedDate(mid,d)))
endProc
```

In the above procedure, the *for* loop iterates through all the suggested dates executing the `inform` action for each of them. After that, another `inform` action is executed telling the MS that for each suggested date the scheduler knows that it is a suggested date. Thus, if the MS does not know that some date is a `SuggestedDate`, then it is not.

The procedure that the task `InformParticipants` maps into is very similar to `InformDates` and we will not discuss it:

```
      proc InformParticipants(mi,mid,ms)
            for p: Ptcp(mid,d) do
                  inform(mi,ms,Ptcp(mid,d))
            endFor;
            inform(mi,ms,∀d.Ptcp(mid,d) ⊃ Know(ms,Ptcp(mid,d)))
      endProc
```

This concludes the mapping process for the Meeting Initiator. Note that the agent is modeled as being very much aware of what it is doing: the agent's goals and knowledge are set and updated appropriately and trigger the relevant behaviour.

We also have to specify the goals and the knowledge that the agent has in the initial situation. Since the Meeting Initiator is the driving force behind the meeting scheduling process, it needs to initiate it by trying to achieve its goals. The top-level goal of the MI is the goal `MeetingSetup(mid,mi,s)`. In order to indicate which meetings it will try to schedule, we will specify the initial goals of the Meeting Initiator as follows:

```
Goal(mi,Eventually(MeetingSetup(MID_1,mi,now),now,then),S0)
    ...
Goal(mi,Eventually(MeetingSetup(MID_n,mi,now),now,then),S0)
```

This says that in the initial situation, the initiator has the goals to schedule *n* meetings with meeting IDs of `MID_1` to `MID_n`. The participants and suggested dates for these meetings must be specified using the fluents `Ptcp` and `SuggestedDate`.

6.7.3 Creating the CASL Model for the MRBS

In this section, we show how the mapping of a capability node into a CASL model differs from the mapping of other types of nodes. The rest of the model for the MRBS can be found in Appendix A.1.

Based on the mapping rules for capabilities presented in Section 4.5.4, a goal capability is mapped into a CASL formula/fluent, a CASL procedure that acquires and achieves the goal of the capability, a specification of the processes that may occur in the environment without affecting the success of the capability, and a context condition for the capability under which it is guaranteed to succeed. Let us see how the goal capability RoomBookedCap (see Figure 6.18) is mapped into CASL.

Since the capability achieves the goal RoomBooked, it maps into is the defined fluent RoomBooked:

$$\mathbf{m}(\texttt{RoomBookedCap}).\texttt{formula} = \texttt{RoomBooked(mid,date,s)}$$

The defined fluent RoomBooked(mid,d,s) tells whether a room has been booked for the meeting mid on the date d. It is a fluent that is defined in terms of the fluents Room and AvailableRoom. The goal RoomBooked that is delegated to the MRBS by the scheduler also maps into this fluent:

$$\texttt{RoomBooked(mid,date,s)} \overset{def}{=} \exists \texttt{r[Rooms(r)} \wedge \texttt{Room(mid,date,r,s)]} \vee$$
$$\forall \texttt{r[Rooms(r)} \supset \neg \texttt{AvailableRoom(r,date,s)]}$$

This is the deidealized definition of the goal RoomBooked. This fluent holds if there a room was booked for the meeting mid on the date or if there is no available room on that date. AvailableRoom is defined as a room that is not booked:

$$\texttt{AvailableRoom(room,date,s)} \overset{def}{=} \neg\exists \texttt{mid[Room(mid,date,room,s)]}$$

The specification for the allowable processes in the environment that guarantees that the capability succeeds in achieving its goal (booking a meeting room if there is one available while maintaining the consistency of the booking schedule) states that all the actions are allowed except for the action bookRoom. Effectively, the specification bans other agents from executing this action (note the non-deterministic iteration operator):

$$\texttt{RoomBookedCap.envProc} = (\pi\, \texttt{act[(act} \neq \texttt{bookRoom)?; act])}^{\leq k}$$

The context condition for the capability states that in order for the capability to be successful in booking rooms the MRBS must know initially about the availability of all rooms on all dates:

```
RoomBookedCap.context = ∀r,d[(Rooms(r) ∧ IsDate(d)) ⊃
                    KWhether(MRBS,∃mid.Room(mid,d,r,now),S0)]
```

The achievement procedure for the capability is the procedure `RoomBookedProc`.

```
RoomBookedCap.achieve = RoomBookedProc
```

At this point, we have the complete mapping of the goal capability `RoomBookedCap`. The complete CASL code is presented in Appendix A.1. In Section 4.5.4 we presented a constraint on the achievement procedures for goal capabilities. It states that in all situations satisfying the context condition every subjective execution of the achievement procedure of a capability by the agent in an environment in which processes specified by `envProc` may occur terminates with the goal being achieved. In case of `RoomBookedCap`, the constraint states that if the MRBS knows the status of all rooms initially and no other agent is booking rooms, then after the procedure `RoomBookedProc(mid,d)` is executed for some meeting ID, either a room is booked for the meeting `mid` on the date `d` or there is no available room on that date. Since the MRBS is personally maintaining the status of all the meeting rooms, it always has enough information to know whether a room is available or not and thus will be able to achieve the deidealized goal `RoomBooked`.

When creating an instance of the system, one needs to make sure that the context condition of the `RoomBookedCap` capability holds. Therefore, `RoomBookedCap.context` must hold in the initial situation. This means that the agent must know for every room and every date whether the room is occupied by some meeting on that date. For example, one can specify that the MRBS knows that the room `r1` is booked for the meeting `mid1` on the date `d1`:

$$\mathbf{Know}(\mathtt{MRBS},\mathtt{Room}(\mathtt{mid1},\mathtt{d1},\mathtt{r1},\mathtt{now}),S_0)$$

Also, the modeler can say that the MRBS knows that no meeting is taking place in the room `r1` on the date `d1`:

$$\forall \mathtt{mid}[\mathbf{Know}(\mathtt{MRBS},\neg\mathtt{Room}(\mathtt{mid},\mathtt{d1},\mathtt{r1},\mathtt{now}),S_0)]$$

6.7.4 Creating the CASL Model for the Meeting Participant

In this section we discuss selected aspects of the formal CASL model for the Meeting Participant actor. The complete CASL specification for the MP is presented in Appendix A.2. Note that the goal node `AtMeeting` is not mapped into CASL as per discussion in Section 6.6.3.

The Meeting Participant actor acquires some goals from intentional dependencies (see Figure 6.19). The formal definition of the goal `AtMeeting` was discussed in Section 6.7.1. Another dependency-acquired goal of the MP is `InformedIfAttending(ms,mid,d)`. Here, the Meeting Scheduler wants to know whether the participant `p` is going to attend the meeting `mid`. This amounts to checking whether the goal `AtMeeting(p,mid,d)` is in the mental state of the participant `p`. Therefore, the goal `InformedIfAttending` maps into the following defined fluent with the same name:

$$\mathtt{InformedIfAttending}(\mathtt{p},\mathtt{ms},\mathtt{mid},\mathtt{d}) \overset{def}{=}$$
$$\mathbf{KWhether}(\mathtt{ms},\mathbf{Goal}(\mathtt{p},\mathbf{Eventually}(\mathtt{AtMeeting}(\mathtt{p},\mathtt{mid},\mathtt{d},\mathtt{now}),\mathtt{now},\mathtt{then}),\mathtt{now}),\mathtt{s})$$

Let us now look at how the MP informs the scheduler about its available dates. As can be seen from Figure 6.19, the task `InformAvailableDates` first iterates through all the dates and reports their status to the MS. It then informs the MS that the participant is done by using the fluent `DoneInforming(mid,p,s)`. This fluent is defined as follows:

```
    DoneInforming(mid,ptcp,s) ≝
        ∀d[IsDate(d) ⊃ InformedAvailable(mid,ptcp,d,s)]
```

`DoneInforming` is true for some meeting `mid` and participant `ptcp` if for all dates it is true that `InformedAvailable(mid,ptcp,d,s)`. The fluent `InformedAvailable(mid,ptcp,d,s)` is true for a particular date `d` if after receiving a request from the MS to send available dates for scheduling the meeting `mid`, the participant `ptcp` has sent his/her availability status for the date `d` to the scheduler. The participant is finished informing the MS when it has sent the status of all the dates to the scheduler. The formal definition of `InformedAvailable` is as follows:

```
InformedAvailable(mid,p,d,s) ≝
    ∃s′,s″[S0 ≤ do(request(ms,p,DoAL(
        InformAvailableDates(mid,MS,p),now,then)),s′) ≤
        do(informWhether(p,MS,Available(p,d,now)),s″) ≤ s]
```

Here, we want to make sure that when scheduling the meeting `mid`, the MS has some recent availability information — the information sent by the participants after they received the request to execute the task `InformAvailableDates` for the meeting `mid`.

6.7.5 Creating the CASL Model for the Meeting Scheduler

In this section, we list all the definitions for goals and conditions found in the iASR diagram for the Meeting Scheduler (Figure 6.21). The complete CASL specification for this agent is presented in Appendix A.4.

After requesting the Meeting Initiator to provide the list of suggested meeting dates and the list of meeting participants for some meeting `mid`, the MS needs to make sure that it has that information before proceeding to scheduling the meeting. We have two guard link annotations with conditions `KnowAllDates(mid)` and `KnowAllPtcp(mid)` that make the program block until the all the suggested dates and participants (respectively) are known. These conditions are defined as follows:

$$\texttt{KnowAllDates(mid,s)} \overset{def}{=} \textbf{Know}\texttt{(MS,}\forall\texttt{p[SuggestedDate(mid,p,now)} \supset$$
$$\textbf{Know}\texttt{(MS,SuggestedDate(mid,p,now),now)],s)}$$
$$\texttt{KnowAllPtcp(mid,s)} \overset{def}{=} \textbf{Know}\texttt{(MS,}\forall\texttt{p[Ptcp(mid,p,now)} \supset$$
$$\textbf{Know}\texttt{(MS,Ptcp(mid,p,now),now)],s)}$$

The self-acquired goal `AvailableDatesKnown` is mapped into the defined fluent with the same name that holds when all the participants have informed the MS of their available dates since it requested that information during the scheduling of the meeting `mid` (see the definition of `DoneInforming` in Section 6.7.4):

$$\texttt{AvailableDatesKnown(mid,s)} \overset{def}{=}$$
$$\forall\texttt{p[Ptcp(mid,p,s)} \supset \texttt{DoneInforming(mid,p,s)]}$$

After all the participants, suggested dates, and available dates are known to the Meeting Scheduler for some meeting `mid`, it can start the meeting scheduling process. First, it must determine which dates are good for scheduling meetings. We call these *agreeable* dates. These dates must be on the list of suggested dates sent by the Initiator, they must be known by the scheduler to be available (since the Disruptor can at any time request participants to attend meetings outside of the organization, this knowledge may not be entirely accurate), and they must not be locked for any participant of the meeting (meaning that the MS is not currently trying to schedule other meetings on those dates). Hence, the definition of the fluent `AgreeableDate` is as follows:

$$\begin{aligned}&\mathtt{AgreeableDate(mid,d,s)} \overset{def}{=}\\ &\qquad \mathtt{SuggestedDate(mid,d,s)} \wedge\\ &\qquad \forall \mathtt{p[Ptcp(mid,p,s)} \supset \mathtt{LastInformedAvailable(p,d,s)]} \wedge\\ &\qquad \forall \mathtt{p[Ptcp(mid,p,s)} \supset \neg\mathtt{Locked(p,d,s)]}\end{aligned}$$

As mentioned above, the presence of the Disruptor with its encrypted meeting requests may invalidate the information the MS has about the availability of the participants. Therefore, the scheduler cannot just check the truth value of the fluent `Available(ptcp,date,s)` for some participant and some date to make sure that participant is available on that date — it does not know the value of that fluent at the current time (in the situation `s`). Instead, it must refer to the time it got that information from the participant `ptcp`. So, we formally define the dates that the MS thinks (to the best of its knowledge) are available as the dates for which the MS was last informed that they were available and has not since been informed otherwise:

$$\begin{aligned}&\mathtt{LastInformedAvailable(p,d,s)} \overset{def}{=}\\ &\qquad \exists s'.S_0 \le \mathtt{do(inform(p,MS,Available(p,d,now)),s')} \le s \wedge\\ &\qquad \neg\exists s''.s' \le \mathtt{do(inform(p,MS,\neg Available(p,d,now)),s'')} \le s\end{aligned}$$

The next two definitions allow the Meeting Scheduler to determine whether all the intended meeting participants agreed to meet on the proposed date or whether some declined. It is the case that everybody accepted to meet on the date `d` if they have the corresponding `AtMeeting` goal in their mental state:

```
AllAccepted(mid,d,s) $\overset{def}{=}$
      ∀p[Ptcp(mid,p,s)⊃
         Goal(p,Eventually(AtMeeting(p,mid,d,now),now,then),s)]
```

Someone declined to meet on the date `d` if there is an intended meeting participant without the goal in its mental state:

```
SomeoneDeclined(mid,d,s) $\overset{def}{=}$
      ∃p[Ptcp(mid,p,s) ∧
        ¬Goal(p,Eventually(AtMeeting(p,mid,d,now),now,then),s)]
```

If everybody accepted to hold the meeting `mid` on the date `d`, the MS goes ahead and tries to book a room for the meeting on that date. It delegates the goal `RoomBooked(mid,d)` to the MRBS and requests confirmation of the booking. If a room is booked, this means that the MS has successfully scheduled the meeting `mid`:

```
      SuccessfullyScheduled(mid,s) $\overset{def}{=}$
            ∃d[AgreeableDate(mid,d,s) ∧ AllAccepted(mid,d,s) ∧
            RoomBooked(mid,d,s)]
```

At this point, we can provide the formal definition of the goal `ParticipantInformed`. First, the MS must have gone through the scheduling process, so we require that the fluent `DoneScheduling(mid)` hold. After the scheduling process is complete, the participants (including the Meeting Initiator) need to know whether the meeting was scheduled (the truth value of the fluent `SuccessfullyScheduled(mid)`) and if the meeting was, in fact, scheduled successfully, which room was booked for it (the value of the fluent `Room`):

`ParticipantInformed(p,mid,s)` $\overset{def}{=}$

```
    DoneScheduling(mid,s) ∧
    (
        [Know(p,SuccessfullyScheduled(mid,now),s) ∧
        KRef(p,Room(mid,date,r,now),s)] ∨
        Know(p,¬SuccessfullyScheduled(mid,now),s)
    )
```

The last definition that we discuss in this section is that of the goal `MeetingScheduledIfPossible`. After it has been deidealized, there are four possibilities to achieve the goal. They are labeled *1* through *4* and are accompanied by comments in the formal definition below:

`MeetingScheduledIfPossible(mid,s)` $\overset{def}{=}$

{1. The meeting has been scheduled}

`SuccessfullyScheduled(mid,s) ∨`

{2. No agreeable dates}

`∀d[IsDate(d) ⊃ ¬AgreeableDate(mid,d,s)] ∨`

{3. Some participants cannot attend on all potential dates}

`∀d[AgreeableDate(mid,d,s) ⊃ SomeoneDeclined(mid,d,s)] ∨`

{4. No rooms available on all the dates that were accepted by the participants}

```
∀d[SuggestedDate(mid,d,s) ⊃ [AllAccepted(mid,d,s) ⊃
                                   ¬RoomBooked(mid,date,s)]]
```

The first case is when the meeting is successfully scheduled, so the fluent `SuccessfullyScheduled` holds. The second case is when there are no agreeable dates. This means that there is no common slot among the participants to schedule the meeting. The third case is when there are agreeable dates, but when the MS tries to schedule the meeting on those dates, some participant always declines. This could be happening if the participants' schedules are filling with outside appointments through the Disruptor's requests. The fourth possibility is that there are no rooms available for any of the dates that the participants are willing to meet on.

6.7.6 Creating the CASL Model for the Disruptor

In this section, we list the definition the goal `ParticipationRequested` found in the iASR diagram for the Disruptor (Figure 6.20). If sometime in the past the Disruptor requested that the participant `part` be at the meeting `mid` on the date `date`, then the goal `ParticipationRequested(disruptor,part,mid,date)` is achieved:

$$\mathtt{ParticipationRequested(disruptor,part,mid,date,s)} \overset{def}{=}$$
$$\exists s'[S_0 \leq \mathtt{do(requestEnc(disruptor,part,AtMeeting(part,mid,date)),s')} \leq s]$$

To create an instance of the system, the modeler needs to specify the initial goals of the Disruptor actor. An initial goal for the Disruptor can be specified as follows:

```
Goal(Disr1,Eventually(
     ParticipationRequested(Disr1,p1,mid1,d1,now),now,then),S0),
```

which says that the Disruptor `Disr1` wants the participant `p1` to participate in the meeting `mid1` on the date `d1`. The complete CASL specification for the Disruptor is presented in Appendix A.3.

6.7.7 Formal Verification

Formal verification of the CASL model for the meeting scheduling process is left for future work. After the iASR diagrams for all the actors in the system have been mapped into a formal CASL specification, a CASL verification tool can be used with great benefits to the modeler. With such a tool, a specification can be checked to see if it satisfies certain requirements (e.g., “no meetings on weekends”), the achievability of goals can be confirmed, and formal goal deidealization and decomposition can also be verified. In addition, the epistemic feasibility of agent processes — whether agents have enough knowledge to successfully execute their plans — can be checked. One such tool, CASLve [Shapiro *et al.*, 2002], has been used to verify a smaller version of the meeting scheduling system. Verifying systems of the size of the one presented in this chapter using theorem proving techniques is a large and complex job. Nevertheless, we expect that with further development the CASLve tool will be able to handle the verification of our system. The improvement of CASLve seems to be the best approach for obtaining a solid tool for verifying CASL specifications. Other avenues could be explored as well, for instance, simulation and model checking. However, most tools based on these techniques work with much less expressive languages than CASL. Therefore, CASL specifications must be simplified before these methods can be used on them. For example, mental states would have to be operationalized and an approach would have to be found to deal with incomplete information in CASL specifications.

However, even without a verification environment capable of formal verification of an example of this size, we found that creating iASR diagrams for the system and their mapping into CASL is extremely beneficial for the modeler. Our methodology requires the analyst to look more systematically at the domain, take goal decompositions more seriously, think about the achievability of goals and the epistemic feasibility of agent processes. This results in better understanding of the system requirements and leads to higher quality system analysis and design.

6.8 Discussion

In this chapter, we showed how our agent-oriented requirements engineering methodology is applied to the meeting scheduling case study. The meeting scheduling system discussed in this chapter has no account of time and a very simplistic treatment of meeting periods. This was done to simplify the model and both problems can be quite easily solved. The system can be modified to handle many meetings per day by, for example, replacing the predicate `IsDate(d)` with `Period(date,hour)` and modifying the rest of the model accordingly, which would give each meeting a one hour slot. Meetings could also be allowed to span several periods for further flexibility. However, in this approach, since the system does not have any account of the passage of time (there is no connection between the actual time and CASL situations), the execution of the system is separate from the real world, the dates and hours of the meetings. Thus, the system is not able to, for example, remind a participant that a meeting is starting soon. A more radical change in approach can be taken to address this. This approach is described in [Shapiro *et al.*, 1998]. It allows one to associate a time with each situation (the fluent `Time(s)` is introduced), assign duration to actions, etc. Using this approach, one would be able to remind participants of upcoming meetings, and integrate an account of meeting attendance into the model.

Traceability is a very important matter in software design. In Section 5.6.3 we discussed requirements traceability in our approach and concluded that it is well supported by our methodology. However, there is an issue that could be problematic for requirements traceability. This arises from the ability of analysts to model the achievement of agents' goals declaratively, not procedurally. For example, the *System* agent's goal `NoMeetingsOnWeekends` in Figure 6.21 is achieved by the Meeting Scheduler's task `GetRidOfWeekendDates`. Here, for every meeting date that is suggested by the Meeting Initiator and that falls on a weekend, the MS executes the action `removeWeekendDate`, which removes the date from the list of suggested dates. This is a purely procedural way of making sure that no meeting is scheduled on a weekend and we can easily show how the goal dependency `NoMeetingsOnWeekends` is supported by the MS. Another way of achieving this goal is to change the definition of the fluent `AgreeableDate`, which defines the dates that are suitable for scheduling meetings. One can change the definition presented in Section 6.7.5 in the following way:

$$
\begin{aligned}
&\mathtt{AgreeableDate(mid,d,s)} \overset{def}{=} \\
&\qquad \mathtt{SuggestedDate(mid,d,s)} \land \neg\mathbf{Weekend(d)} \land \\
&\qquad \forall \mathtt{p[Ptcp(mid,p,s)} \supset \mathtt{LastInformedAvailable(p,d,s)]} \land \\
&\qquad\qquad\qquad\qquad \forall \mathtt{p[Ptcp(mid,p)} \supset \neg\mathtt{Locked(p,d,s)]}
\end{aligned}
$$

The new definition has an additional conjunct, `¬Weekend(d)`, which makes sure that no date that is a weekend date becomes an agreeable date. This is a declarative way of achieving the goal `NoMeetingsOnWeekends`. While this new approach may be more elegant than the procedural one, there is one problem: definitions like this are not easily represented in iASR models. Thus, unlike the procedural approach in Figure 6.21, we cannot show visually how the goal is achieved by the MS when using the new definition of `AgreeableDate` since there is no node in the iASR diagram where the elimination of weekend dates is done and therefore no place to connect the goal dependency link from the *System* agent. Also, in order to maintain requirements traceability, one needs to show that only the conjunct `¬Weekend(d)` is present in the new definition of `AgreeableDate` because of the goal `NoMeetingsOnWeekends`. It is future work to determine whether and how declarative CASL facilities that achieve agents' goals can be effectively used in our models while maintaining requirements traceability.

We believe that the case study presented in this chapter shows the importance and benefits of formally modeling goal/task delegation and information exchange in a multiagent setting. Additionally, formal support for reasoning about goals can provide the analyst with a new modeling tool. For example, in the meeting scheduling case study we decided not to maintain schedules for meeting participants explicitly. Instead, we relied on the presence of `AtMeeting(p,mid,d,s)` goals in their mental states as an indication of the participants' intention to attend certain meetings on certain dates. With the axioms presented in Section 6.7.1.4 (and also assuming that agents do not drop their commitments), the consistency of participants' schedules can be easily maintained since the meeting requests conflicting with already adopted `AtMeeting` goals are automatically rejected.

7 Conclusion

In this thesis, we devised an agent-oriented requirements engineering approach with a formal component that supports reasoning about agents' goals (and knowledge), thus allowing for rigorous formal analysis of the requirements expressed as objectives of the agents in multiagent systems. We introduced Intentional Annotated Strategic Rationale (iASR) diagrams, which build on top of ASR diagrams of [Wang, 2001]. Using the set of rules provided in this thesis (Chapter 4), iASR diagrams can be mapped into the corresponding CASL specifications for formal analysis and/or simulation. A notion of agent capability based on the epistemic feasibility [Lespérance, 2002] of agent plans is formally defined and a capability node is added to the *i**-based diagrammatic notation. A methodology for requirements engineering using our approach is proposed (Chapter 5). A case study demonstrating the use of the methodology is also presented (Chapter 6).

7.1 Contributions

The main technical contributions of this thesis are listed below.

1) This thesis introduces Intentional Annotated SR (iASR) diagrams (based on the Annotated SR diagrams of [Wang, 2001]), which support modeling agents' goals and knowledge. The thesis also provides clear guidance on the use of goal nodes in iASR diagrams to simplify the mapping of these diagrams into CASL and describes ways of synchronizing the behaviour of agents with changes to their mental states.

2) New annotations for iASR diagrams are introduced. These include the interrupt with a cancellation condition, which provides more flexibility than the original interrupt

annotation of [Wang, 2001], the guard annotation, and the applicability condition annotation to specify when a means for achieving a goal can be used.

3) A set of mapping rules is proposed to map iASR diagrams into the corresponding CASL specifications while preserving requirements traceability and supporting the modeling of explicit goals and knowledge. The possibilities for an automated mapping are outlined (Section 5.8).

4) In this thesis, we introduce self-acquired goals (Section 4.3.8) that can be added to the mental state of the agents in the absence of requests from other agents. We show that these goals can be used as a tool to add flexibility to multiagent systems specifications and make agents aware of what they are doing. Also, self-acquired goals can be used to formally reason about goal decompositions at runtime.

5) The notion of an agent capability as an epistemically feasible agent plan is introduced (Section 4.5) together with capability nodes to be used in SD, SR, and iASR diagrams. The mapping of goal and task capabilities into CASL programs that are guaranteed to successfully execute under certain conditions is presented.

6) A methodology for the combined use of *i** and CASL is presented (Chapter 5). The methodology describes the steps of the requirements engineering process starting from the identification of stakeholders up to the formal verification of CASL specifications. We also propose some improvements to the *i** modeling process including the use of self-dependencies and the *System* agent (Sections 5.2.3 and 5.3.2 respectively). Self-dependencies help in modeling actors that depend on themselves for achieving their goals, etc., while the *System* agent is useful in modeling goals that the system-to-be as a whole is expected to achieve.

7.2 Benefits

The main advantage of the approach for the combined use of *i** and CASL proposed in this thesis is that it allows for the formal analysis of agents' goals and knowledge in the context of requirements engineering. This has the following benefits:

a) Goals are now represented as mental states. This is different from other approaches (e.g., KAOS and Tropos) where goals are treated as formulae that the whole system must satisfy. In our approach, goals are attributed to particular agents and become their motivations. Therefore, this approach is more agent-oriented and allows for precise modeling of stakeholder goals. With our approach, one can create and formally analyze a model with stakeholders that have conflicting goals, a common case in requirements engineering. Modeling of agent negotiations is also possible with our approach.

b) Goal delegation is an integral part of multiagent systems and the ability to represent and analyze intentional dependencies is one of the most important features of *i**. Our approach allows for formal analysis of goal delegation.

c) The ability to formally analyze agents' knowledge allows us to model knowledge exchange among agents and represent incomplete knowledge in the system. Formal reasoning about agents' goals and knowledge allows for rigorous analysis of complex inter-agent interactions. An example inter-agent protocol is analyzed (Section 4.4.6).

d) Agent processes can now be checked for their executability and epistemic feasibility. Agent goals can be checked for achievability.

e) Goals do not have to be abstracted out before the formal analysis is done.

f) Formal modeling and analysis of issues such as trust and privacy is now possible.

g) A formal CASL specification can be used both as a requirements analysis tool and as a formal high-level specification for a multiagent system.

7.3 Future Work

Here, we identify some of important areas for future work

- In this thesis, we only handled achievement goals of the form `Goal(agt,Eventually(φ))`. In the future, we would like to add support for other types of goals, e.g., maintenance goals and "achieve and maintain" goals.

- In Section 4.4.6 we sketched how an agent interaction protocol can be modeled in CASL. We would like to explore how such protocols could be visually modeled in *i** and then mapped into CASL specifications.

- We would like to investigate whether CASL could be used to formally analyze SD models. Since SD diagram cannot represent the details of agent processes, we expect that mainly the declarative facilities of CASL (including agent goals and knowledge) can be used at this stage of the requirements engineering process.

- We would also like to explore how privacy, security, and trust could be modeled in the formal CASL framework. This would involve relaxing some restrictions on knowledge exchange and goal delegation (e.g., we can allow false information to be communicated by agents or make all communications encrypted).

- More work needs to be done in order to explore how agent capabilities can be guaranteed to always succeed and how to make them into pluggable modules.

- Another area of future work is to see how advanced aspects of agent-based systems can be modeled in *i** and formally analyzed in CASL. This includes, for instance, modeling the ability of smart deliberating agents to come up with (previously unidentified) plans for achieving their goals at runtime.

- In Section 6.8 we discussed how a goal (`NoMeetingsOnWeekends`) can be achieved by using the declarative facilities of CASL. It remains to be investigated how this can be reflected in iASR diagrams and how requirements traceability can be preserved in this case.

Bibliography

F. Alencar, J. Castro, G. Cysneiros and J. Mylopoulos. From Early Requirements Modeled by the i* Technique to Later Requirements Modeled in Precise UML. In Proc. *Anais do III Workshop em Engenharia de Requisitos*, Rio de Janeiro, Brazil, 2000, pp. 92-109.

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.

G. Booch, J. Rumbaugh and I. Jacobson. The Unified Modeling Language: User Guide. Addison-Wesley, 1999.

F. Brazier, B. Dunin-Keplicz, N.R. Jennings and J. Treur. Formal Specification of Multiagent Systems: a Real-World Case. In Proc. *First International Conference on Multi-Agent Systems (ICMAS-95)*, San Francisco, CA, June, 1995, pp. 25-32.

J. Castro, M. Kolp and J. Mylopoulos. Towards Requirements-Driven Information Systems Engineering: The Tropos Project. *Information Systems*, 27(6), pp. 365-389, 2002.

J. Castro, R. Pinto, A. Castor and J. Mylopoulos. Requirements Traceability in Agent-Oriented Development. In A.F. Garcia, C.J.P.d. Lucena, F. Zambonelli, A. Omicini and J. Castro, Eds., *Software Engineering for Large-Scale Multi-Agent Systems, Research Issues and Practical Applications*, pp. 57-72, LNCS, Springer, 2003.

F. Chabot, J.F. Raskin and P.Y. Schobbens. The Formal Semantics of Albert II. Technical Report. Computer Science Department, University of Namur, Namur, Belgium, 1998.

L.K. Chung, B.A. Nixon, E. Yu and J. Mylopoulos. Non-Functional Requirements in Software Engineering. Kluwer, 2000.

A. Cimatti, E. Clarke, F. Giunchiglia and M. Roveri. NuSMV: A New Symbolic Model Verifier. In Proc. *International Conference on Computer-Aided Verification (CAV'99)*, Trento, Italy, July 7-10, 1999.

A. Collinot, A. Drogoul and P. Benhamou. Agent-Oriented Design of a Soccer Robot Team. In Proc. *Second International Conference on Multi-Agent Systems (ICMAS-96)*, Kyoto, Japan, 1996, pp. 41-47.

A. Dardenne, A. van Lamsweerde and S. Fickas. Goal-Directed Requirements Acquisitions. *Science of Computer Programming*, 20, pp. 3-50, 1993.

R. Darimont and A. van Lamsweerde. Formal Refinement Patterns for Goal-Driven Requirements Elaboration. In Proc. *4th ACM Symposium on the Foundation of Software Engineering*, October, 1996, San Francisco, USA.

A. Davis. Software Requirements: Objects, Functions and States. Prentice Hall, 1993.

G. De Giacomo, Y. Lespérance and H. Levesque. ConGolog, A Concurrent Programming Language Based on the Situation Calculus. *Artificial Intelligence*, 121, pp. 109-169, 2000.

P. du Bois. The Albert II Language: On the Design and the Use of a Formal Specification Language for Requirements Analysis. Ph.D. Thesis. Computer Science Department, University of Namur, Namur, Belgium, 1995a.

P. du Bois. Intuitive Definition of the Albert II Language. Technical Report RP-95-007. Computer Science Department, University of Namur, Namur, Belgium, 1995b.

P. du Bois, E. Dubois and J.-M. Zeippen. On the Use of a Formal Requirements Engineering Language: The Generalized Railroad Crossing Problem. In Proc. *Third IEEE International Symposium on Requirements Engineering*, 1997, IEEE Computer Society Press.

E. Dubois, P. du Bois, F. Dubru and M. Petit. Agent-Oriented Requirements Engineering: A Case Study Using the Albert language. In Proc. *Proc. of the Fourth International Working Conference on Dynamic Modelling and Information System DYNMOD-IV*, Noordwijkerhoud, The Netherlands, September 28-30, 1994.

E. Dubois and M. Petit. Using a Formal Declarative Language for Specifying Requirements Modelled in CIMOSA. In Proc. *European Workshop on Integrated Manufacturing Systems Engineering (IMSE '94)*, Grenoble, France, December 12-14, 1994, pp. 233-241.

E. Dubois, E. Yu and M. Petit. From Early to Late Formal Requirements: a Process Control Case Study. In Proc. *9th International Workshop on Software Specification and Design*, Isobe, Japan, 1998, pp. 34-42, IEEE Computer Society.

T. Finin, Y. Labrou and J. Mayfield. KQML as an agent communication language. In J. Bradshaw, Ed. *Software Agents*, MIT Press, 1997.

FIPA. The Foundation for Intelligent Physical Agents. 2001. Available at www.fipa.org.

A. Fuxman, P. Giorgini, M. Kolp and J. Mylopoulos. Information Systems as Social Structures. In Proc. *2nd International Conference on Formal Ontologies for Information Systems (FOIS'01)*, Ogunquit, ME, USA, October, 2001a.

A. Fuxman, M. Pistore, J. Mylopoulos and P. Traverso. Model Checking Early Requirements Specifications in Tropos. In Proc. *Fifth IEEE International Symposium on Requirements Engineering (RE'01)*, Toronto, Canada, August 27-31, 2001b.

F. Giunchiglia, J. Mylopoulos and A. Perrini. The Tropos Software Development Methodology: Processes, Models and Diagrams. In F. Giunchiglia, J. Odell and G. Weiss, Eds., *Agent-Oriented Software Engineering III, Third International Workshop, AOSE 2002, Bologna, Italy, July 15, 2002, Revised Papers and Invited Contributions*, pp. 162-173, LNCS 2585, Springer, 2003.

N. Glaser. Contribution to Knowledge Management in a Multi-Agent Framework (The CoMoMAS Approach). Ph.D. Thesis. L'Universtite Henri Poincare, Nancy, France, 1996.

O. Gotel and A. Finkelstein. An Analysis of the Requirements Traceability Problem. In Proc. *First Int'l Conference on Requirements Engineering*, 1994, pp. 94-101.

IEEE. IEEE Standard Glossary of Software Engineering Terminology, IEEE Std 610.12. 1990.

C. Iglesias, M. Garrijo and J. Gonzales. A Survey of Agent-Oriented Methodologies. In Proc. *5th International Workshop on Intelligent Agents: Agent Theories, Architectures, and Languages (ATAL-98)*, Paris, France, July, 1998, pp. 317-330.

F. Ingrand, M. Georgeff and A. Rao. An Architecture for Real-Time Reasoning and System Control. *IEEE Expert*, 7(6), December, 1992.

M. Jackson. Software Requirements and Specifications: A Lexicon of Practice, Principles and Prejudices. Addison-Wesley, 1995.

M. Jackson. The Meaning of Requirements. *Annals of Software Engineering*, 3, pp. 5-21, 1997.

N.R. Jennings. Agent-Oriented Software Engineering. In Proc. *9th European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW-99)*, Valencia, Spain, 1999, pp. 1-7.

N.R. Jennings and M. Wooldridge. Applications of Intelligent Agents. In N.R. Jennings and M. Wooldridge, Eds., *Agent Technology: Foundations, Applications, and Markets*, pp. 3-28, Springer-Verlag, 1998.

D. Kinny, M. Georgeff and A. Rao. A Methodology an Modelling Technique for Systems of BDI Agents. In W.V.d. Velde and J.W. Perram Eds., Proc. *Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World*, 1996, pp. 56-71, Springer-Verlag.

M. Kolp and J. Mylopoulos. Software Architectures as Organizational Structures. In Proc. *ASERC Workshop on "The Role of Software Architectures in the Construction, Evolution, and Reuse of Software Systems"*, Edmonton, Canada, August 24-25, 2001.

G. Kotonya and I. Sommerville. Requirements Engineering: Processes and Techniques. Wiley, 1998.

A. Lapouchnian and Y. Lespérance. Interfacing Indigolog and OAA: A Toolkit for Advanced Multiagent Applications. *Applied Artificial Intelligence*, 16(9-10), pp. 813-829, 2002.

Y. Lespérance. On the Epistemic Feasibility of Plans in Multiagent Systems Specifications. In J.J.C. Meyer and M. Tambe, Eds., *Intelligent Agents VIII - Agent Theories, Architectures, and Languages, 8th International Workshop (ATAL-2001), Revised papers*, pp. 69-85, LNAI 2333, Springer, 2002.

E. Letier and A. van Lamsweerde. Handling Obstacles in Goal-Oriented Requirements Engineering. *IEEE Transactions on Software Engineering, Special Issue on Exception Handling*, 26(10), October, 2000.

E. Letier and A. van Lamsweerde. Agent-Based Tactics for Goal-Oriented Requirements Elaboration. In Proc. *24th International Conference on Software Engineering*, Orlando, FL, USA, May, 2002, ACM Press.

D.L. Martin, A.J. Cheyer and D.B. Moran. The Open Agent Architecture: A Framework for Building Distributed Software Systems. *Applied Artificial Intelligence*, 13(1-2), pp. 91-128, January-March, 1999.

J. McCarthy and P. Hayes. Some Philosophical Problems From the Standpoint of Artificial Intelligence. In B.Meltzer and D. Michie, Eds., *Machine Intelligence, Volume 4*, pp. 463-502, Edinburgh University Press, 1969.

M. Merritt, F. Modugno and M. Tuttle. Time Constrained Automata. In *2nd Intl Conference on Concurrency Theory (Concur'91)*, LNCS 527, Springer, 1991.

B. Meyer. Object-Oriented Software Construction, 2 Ed. Prentice Hall, 1997.

R.C. Moore. A Formal Theory of Knowledge and Action. In J.R. Hobbs and R.C. Moore, Eds., *Formal Theories of the Common Sense World*, pp. 319-358, Ablex Publishing, 1985.

J. Mylopoulos. Requirements-Driven Software Development. Presentation at Catholic University of Louvain, October 26, 1999.

J. Mylopoulos, A. Borgida, M. Jarke and M. Koubarakis. Telos: Representing Knowledge about Information Systems. *ACM Transactions on Information Systems*, 8(4), pp. 325-362, October, 1990.

J. Mylopoulos, L.K. Chung, S. Liao, H. Wang and E. Yu. Exploring Alternatives during Requirements Analysis. *IEEE Software*, 18(1), January-February, 2001a.

J. Mylopoulos, M. Kolp and J. Castro. UML for Agent-Oriented Software Development: The Tropos Project. In Proc. *The Fourth International Conference on the Unified Modeling Language (UML'01)*, Toronto, Canada, October, 2001b.

B.A. Nuseibeh and S.M. Easterbrook. Requirements Engineering: A Roadmap. In A.C.W. Finkelstein, Ed. *The Future of Software Engineering. (Companion volume to the proceedings of the 22nd International Conference on Software Engineering, ICSE'00)*, IEEE Computer Society Press, 2000.

J. Odell, H. Van Dyke Parunak and B. Bauer. Extending UML for Agents. In Proc. *2nd Intl. Workshop on Agent-Oriented Information Systems (AOIS'00)*, Austin, TX, USA, July, 2000, pp. 3-17.

S. Owre, S. Rajan, J.M. Rushby, N. Shankar and M. K.Srivas. PVS: Combining Specification, Proof Checking, and Model Checking. In R. Alur and T.A. Henzinger Eds., Proc. *Computer-Aided Verification (CAV '96)*, New Brunswick, NJ, USA, July-August, 1996, pp. 411–414, Springer-Verlag.

L. Padham and P. Lambrix. Agent Capabilities: Extending BDI Theory. In Proc. *Seventeenth National Conference on Artificial Intelligence (AAAI-2000)*, Austin, TX, USA, August, 2000.

B. Ramesh and M. Jarke. Towards Reference Models for Requirements Traceability. *IEEE Transactions on Software Engineering*, 27(1), pp. 58-93, January, 2001.

A.S. Rao and M.P. Georgeff. BDI Agents: From Theory to Practice. In Proc. *First Intl. Conference on Multiagent Systems*, San Francisco, CA, USA, 1995.

R. Reiter. The Frame Problem in the Situation Calculus: A Simple Solution (Sometimes) and a Completeness Result for Goal Regression. In V. Lifschitz, Ed. *Artificial*

Intelligentce and Mathematical Theory of Computation: Papers in Honor of John McCarthy, pp. 359-380, Academic Press, 1991.

J. Rumbaugh, I. Jackobson and G. Booch. The Unified Modeling Language Reference Manual. Addison-Wesley, 1999.

R.B. Scherl and H. Levesque. The Frame Problem and Knowledge-producing actions. In Proc. *Eleventh National Conference on Artificial Intelligence*, Washington, DC, July, 1993, pp. 689-695.

A.T. Schreiber, R.J. Wieringa, J.M. Akkermans and W. Van de Velde. CommonKADS: A Comprehensive Methodology for KBS Development. University of Amsterdam, Netherlands Energy Research Foundation ECN, and Free University of Brussels, 1994.

J. Searle. Speech Acts: An Essay in the Philosophy of Language. Cambridge University Press, New York, 1969.

S. Shapiro and Y. Lespérance. Modeling Multiagent Systems with the Cognitive Agents Specification Language - A Feature Interaction Resolution Application. In C. Castelfranchi and Y. Lespérance, Eds., *Intelligent Agents Volume VII - Proceedings of the 2000 Workshop on Agent Theories, Architectures, and Languages (ATAL-2000)*, pp. 244-259, LNAI 1986, Springer, 2001.

S. Shapiro, Y. Lespérance and H. Levesque. Specifying Communicative Multi-Agent Systems with ConGolog. In W. Wobcke, M. Pagnucco and C. Zhang, Eds., *Agents and Multi-Agent Systems - Formalisms, Methodologies, and Applications*, pp. 1-14, Springer, 1998.

S. Shapiro, Y. Lespérance and H. Levesque. The Cognitive Agents Specification Language and Verification Environment for Multiagent Systems. In C. Castelfranchi and W.L. Johnson Eds., Proc. *First International Joint Conference on Autonomous Agents and Multi-Agent Systems*, Bologna, Italy, July 15-19, 2002, ACM Press.

S. Shapiro, M. Pagnucco, Y. Lespérance and H. Levesque. Iterated Belief Change in the Situation Calculus. In A.G. Cohn, F. Giunchiglia and B. Selman, Eds., *Principles of Knowledge Representation and Reasoning: Proceedings of the Seventh International Conference (KR-2000)*, pp. 527-538, Morgan Kaufmann, 2000.

H. Sharp, A. Finkelstein and G. Galal. Stakeholder Identification in the Requirements Engineering Process. In Proc. *10th International Workshop on Database and Expert Systems Applications*, Florence, Italy, September, 1999, pp. 387-391.

Y. Shoham. Agent-Oriented Programming. *Artificial Intelligence*, 60(1), pp. 51-92, March, 1993.

H.A. Simon. The Sciences of the Artificial, 2 Ed. MIT Press, Cambridge, MA, 1981.

R.G. Smith and R. Davis. Frameworks for Cooperation in Distributed Problem Solving. *IEEE Transactions on Systems, Man and Cybernetics*, 11(1), pp. 61-70, 1981.

Sun Microsystems. Enterprise JavaBeans Technology Specifications. 2002. Available at java.sun.com/products/ejb/docs.html.

K. Sycara, J. Lu, M. Klusch and S. Widoff. Matchmaking among Heterogeneous Agents on the Internet. In Proc. *1999 AAAI Spring Symposium on Intelligent Agents in Cyberspace*, Stanford University, Stanford, CA, USA, March, 1999.

A. van Lamsweerde. Requirements Engineering in the Year 00: A Research Perspective. In Proc. *22nd International Conference on Software Engineering*, Limerick, Ireland, June, 2000.

A. van Lamsweerde, R. Darimont and P. Massonet. Goal-Directed Elaboration of Requirements for a Meeting Scheduler: Problems and Lessons Learnt. In Proc. *Second IEEE International Symposium on Requirements Engineering (RE'95)*, York, UK, 1995, pp. 194-203, IEEE Compouter Society Press.

X. Wang. Agent-Oriented Requirements Engineering Using ConGolog and i* Frameworks. M.Sc. Thesis. Department of Computer Science, York University, Toronto, 2001.

X. Wang and Y. Lespérance. Agent-Oriented Requirements Engineering Using ConGolog and i*. In G. Wagner, K. Karlapalem, Y. Lespérance and E. Yu Eds., Proc. *3rd International Bi-Conference Workshop Agent-Oriented Information Systems (AOIS-01)*, 2001, pp. 59-78, iCue Publishing, Berlin.

R.J. Wieringa and E. Dubois. Integrating Semi-Formal and Formal Software Specification Techniques. *Information Systems*, 23(3-4), 1998.

M. Wooldridge. Agent-Based Software Engineering. *IEE Proceedings on Software Engineering*, 144(1), pp. 26-37, 1997.

M. Wooldridge and P. Ciancarini. Agent-Oriented Software Engineering: The State of the Art. In P. Ciancarini and M. Wooldridge, Eds., *Agent-Oriented Software Engineering*, LNAI Volume 1957, Springer-Verlag, 2001.

M. Wooldridge and N.R. Jennings. Intelligent Agents: Theory and Practice. *The Knowledge Engineering Review*, 10(2), pp. 115-152, 1995.

M. Wooldridge, N.R. Jennings and D. Kinny. The Gaia Methodology for Agent-Oriented Analysis and Design. *Journal of Autonomous Agents and Multi-Agent Systems*, 3(3), 2000.

P. Yolum and M.P. Singh. Commitment Machines. In Proc. *Agent Theories, Architectures, and Languages, 8th International Workshop (ATAL-2001)*, Seattle, WA, USA, August 1-3, 2001.

E. Yu. Modeling Strategic Relationships For Process Reengineering. Ph.D. Thesis. Department of Computer Science, University of Toronto, 1995.

E. Yu. Towards Modelling and Reasoning Support for Early-Phase Requirements Engineering. In Proc. *Third IEEE International Symposium on Requirements Engineering (RE'97)*, Annapolis, USA, 1997, pp. 226 -235, IEEE Computer Society Press.

E. Yu, P. du Bois, E. Dubois and J. Mylopoulos. From Organization Models to System Requirements - A “Cooperating Agents” Approach. In M.P. Papazoglou and G. Schlageter, Eds., *Cooperative Information Systems: Trends and Directions*, pp. 293-312, Academic Press, 1997.

E. Yu and J. Mylopoulos. Understanding "Why" in Software Process Modelling, Analysis, and Design. In Proc. *16th International Conference on Software Engineering*, Sorrento, Italy, May 16-21, 1994, pp. 159-168.

P. Zave. Classification of Research Efforts in Requirements Engineering. *ACM Computing Surveys*, 29(4), pp. 315-321, 1997.

Appendix A. The Complete CASL Specification for the Case Study

A.1 The CASL Specification for the MRBS

Refer to Figure 6.18 for the corresponding iASR diagram.

```
proc MRBSBehaviour()
      <mid,d: Goal(MRBS,Eventually(RoomBooked(mid,d))) ∧
                  Know(MRBS,¬RoomBooked(mid,d)) →
            RoomBookedProc(mid,d)
      until SystemDone>
      ||
      <mid,d: Goal(MRBS,DoAL(ConfirmRoom(MS,mid,d),now,then)) ∧
         Know(MRBS,¬∃s,s′(s≤s′≤now ∧ DoAL(ConfirmRoom(MS,mid,d),s,s′))) →
            ConfirmRoom(MS,mid,d)
      until SystemDone>
endProc


proc RoomBookedProc(mid,d)
      if ¬Goal(MRBS,Eventually(RoomBooked(mid,d))) then
            commit(MRBS,Eventually(RoomBooked(mid,d)))
      endIf;
      guard Goal(MRBS,Eventually(RoomBooked(mid,d))) do
            BookRoomProc(mid,d)
      endGuard
endProc
```

```
proc BookRoomProc(mid,d)
      if ∃r.AvailableRoom(r,d) then
            πr.AvailableRoom(r,d)?;
            bookRoom(MRBS,mid,d,r);
      endIf
      setDoneBooking(MRBS,mid)
endProc

proc ConfirmRoom(MS,mid,d)
      guard DoneBooking(mid) do
            SendConfirmation(MS,mid,d)
      endGuard
endProc

proc SendConfirmation(MS,mid,d)
      informWhether(MRBS,MS,RoomBooked(mid,d));
      if KRef(MRBS,Room(mid,d,r)) then
            informRef(MRBS,MS,Room(mid,d,r))
      endIf
endProc
```

A.2 The CASL Specification for the Meeting Participant

Refer to Figure 6.19 for the corresponding iASR diagram.

```
proc ParticipantBehaviour(mp)
   <mid: Goal(mp,DoAL(InformAvailableDates(mid,MS),now,then) ∧
   Know(mp,¬∃s,s′(s≤s′≤now ∧ DoAL(InformAvailableDates(mid,MS),s,s′))) →
      InformAvailableDates(mid,MS,mp)
   until SystemDone>
   ||
   <mid,date: Goal(mp,Eventually(AtMeeting(mid,date) ∧
               Know(mp,¬AtMeeting(mid,date)) →
      no_op
   until SystemDone>
   ||
   <mid,date: Goal(mp,Eventually(InformedIfAttending(MS,mid,date))) ∧
               Know(mp,¬InformedIfAttending(MS,mid,date)) →
      InformedIfAttendingAndKnowIfScheduled(MS,mid,date,mp)
   until SystemDone>
endProc

proc InformAvailableDates(mid,MS,mp)
      for d: IdDate(d) do
            informWhether(mp,MS,AvailableDate(mp,d))
      endFor;
      inform(mp,MS,DoneInforming(mid,mp))
endProc

proc  InformedIfAttendingAndKnowIfScheduled(MS,mid,date,mp)
      informWhether(mp,MS,Goal(mp,Eventually(AtMeeting(mp,mid,date))));
      if Goal(mp,Eventually(AtMeeting(mp,mid,date))) then
            KnowWhenScheduled(mid,mp)
      endIf
endProc

proc KnowWhenScheduled(mid,mp)
      commit(mp,Eventually(ParticipantInformed(mp,mid)));
```

```
    guard Goal(mp,Eventually(ParticipantInformed(mp,mid))) do
        request(mp,MS,Eventually(ParticipantInformed(mp,mid)))
    endGuard
endProc
```

A.3 The CASL Specification for the Disruptor

Refer to Figure 6.20 for the corresponding iASR diagram.

```
proc DisruptorBehaviour(disr)
  <p,mid,d:Goal(disr,Eventually(ParticipationRequested(
            disr,p,mid,d))) ∧
                Know(disr,¬ParticipationRequested(disr,p,mid,d)) →
            requestEnc(disr,p,Eventually(AtMeeting(p,mid,d)))
    until SystemDone>
endProc
```

A.4 The CASL Specification for the Meeting Scheduler

Refer to Figure 6.21 for the corresponding iASR diagram.

```
proc MSBehaviour()
  <mid,mi: Goal(MS,Eventually(MeetingScheduledIfPossible(mid,mi))) ∧
                Know(MS,¬MeetingScheduledIfPossible(mid,mi)) →
            TryToScheduleMeetingAndBookRoom(mid,mi)
  until SystemDone>
  ||
  <part,mid: Goal(MS,Eventually(ParticipantInformed(part,mid))) ∧
                Know(MS,¬ParticipatnInformed(part,mid)) →
            InformParticipant(part,mid)
   until SystemDone>
endProc


proc TryToScheduleMeetingAndBookRoom(mid,mi)
      request(MS,mi,DoAL(informParticipants(mid)));
      request(MS,mi,DoAL(informDates(mid)));
      guard KnowAllDates(mid) do
            GetRidOfWeekendDates(mid)
      endGuard;
      guard KnowAllPtcp(mid) do
            commit(MS,Eventually(AvailableDatesKnown(mid)
      endGuard;
      guard Goal(MS,Eventually(AvailableDatesKnown(mid) do
            for p: Ptcp(mid,p) do
                  request(MS,p,DoAL(InformAvailableDates(mid))
            endFor
      endGuard;
      guard AvailableDatesKnown(mid) do
            TryAgreeableDates(mid,mi)
      endGuard;
endProc


proc GetRidOfWeekendDate(mid)
      for d: Weekend(d) ∧ SuggestedDate(mid,d) do
            removeWeekendDate(MS,mid,d)
      endFor
endProc
```

```
proc TryAgreeableDates(mid,mi)
    for d: AgreeableDate(mid,d) do
        TryDates(mid,d,mi)
    endFor;
    if ¬SuccessfullyScheduled(mid) then
        setDoneScheduling(MS,mid)
    endIf
endProc

proc TryDates(mid,d,mi)
    if ¬SuccessfullyScheduled(mid) then
        TryOneDate(mid,d,mi)
    endIf
endProc

proc TryOneDate(mid,d)
    lockDate(MS,mi,d);
    for p: Ptcp(mid,p) do
        lockDate(MS,p,d)
    endFor;
    for p: Ptcp(mid,p) do
        ContactParticipant(mid,d,p)
    endFor;
    guard ∀p.Ptcp(mid,p) ⊃
        KWhether(MS,Goal(p,Eventually(AtMeeting(p,mid,d)))) do
        ProcessReplies(mid,d,mi)
    endGuard
endProc
```

```
proc ContactParticipant(mid,d,p)
      request(MS,p,Eventually(AtMeeting(p,mid,d)));
      request(MS,p,Eventually(InformedIfAttending(p,MS,mid,d)))
endProc

proc ProcessReplies(mid,d,mi)
      if SomeoneDeclined(mid,d) then
            CancelMeetingForDate(mid,d,mi)
      endIf;
      if AllAccepted(mid,d) then
            ReserveRoomForDate(mid,d,mi)
      endIf
endProc

proc CancelMeetingForDate(mid,d,mi)
      unLockDate(MS,mi,d);
      for p: Ptcp(mid,p) do
            unlockDate(MS,p,d)
      endFor;
      for p: Ptcp(mid,p) do
            cancelRequest(MS,p,Eventually(AtMeeting(p,mid,d)))
      endFor
endProc
```

```
proc ReserveRoomForDate(mid,d,mi)
      request(MS,MRBS,Eventually(RoomBooked(mid,d)));
      AskForConfirmation(mid,d);
      guard KWhether(MS,RoomBooked(mid,d) do
            ProcessReply(mid,d,mi)
      endGuard
endProc

proc AskForConfirmation(mid,d)
      request(MS,MRBS,DoAL(ConfirmRoom(MS,mid,d)));
endProc

proc ProcessReply(mid,d,mi)
      if Know(MS,RoomBooked(mid,d) then
            setDoneScheduling(MS,mid)
      endIf;
      if Know(MS,¬RoomBooked(mid,d)) then
            FailedToBookRoom(mid,d,mi)
      endIf
endProc

proc FailedToBookRoom(mid,d,mi)
      unlockDate(MS,mi,p);
      for p: Ptcp(mid,p) do unlockDate(MS,p,d) endFor;
      for p: Ptcp(mid,p) do
            cancelRequest(MS,p,Eventually(AtMeeting(p,mid,d)))
      endFor
endProc
```

```
proc InformParticipant(p,mid)
      guard DoneScheduling(mid) do
            SendStatusAndRoom(p,mid)
      endGuard
endProc

proc SendStatusAndRoom(p,mid)
      informWhether(MS,p,SuccessfullyScheduled(mid));
      if Know(MS,RoomBooked(mid,d)) then
            informRef(MS,p,Room(mid,d,r)) endIf
endProc
```