Chapter 6

Modelling Cooperative Work

The modelling approach in the previous chapter can be summarised in three steps: i) task analysis based on hierarchical data flow diagramming, ii) allocation of subtasks to human and machine, and iii) specification of the initiative for the data flows between human and machine. The three steps of the modelling approach themselves can be taken as the beginnings of a task decomposition of the "higher level" task of planning and organizing cooperative work in "lower level" application-specific tasks. The same task analysis techniques can be used to draft schemas at both levels.

From this stems the idea that flexible support systems may be modelled as a system component which manages, in negotiation with the user, the organization and execution of the cooperative execution of the domain task. It can create or change such a schema dynamically (e.g., in negotiation with the user), and it can interpret such schema to produce the behaviour of the system's role. For this we need a more fine-grained modelling or specification language. This is also needed to fill a lacuna in the model-based approach for designing computer support.

This chapter provides such specification language. It mediates between simple, high-level languages such as Winograd & Flores' networks and complex low-level implementation languages such as agent-process languages. It can serve as a tool for analysis and modelling of cooperation and collaboration and it can be used in model-based design of CSCW or Multi-Agent systems. The language plays an important role in a model-based design method comprising three steps. The first is to draft a W&F diagram. The second is to transform it to a specification in the proposed language and to add detail. Finally, this specification can be transformed to a set of agent programs.


Introduction

We are interested in methods agents can use to pool their knowledge, skill and information, regardless whether agents are human or computational, and without their contributions being determined beforehand. We are interested in how to specify and program these cooperation methods, so they can be used in multi-agent systems and in tools for Computer-Supported Cooperative Work. As a first step in this direction this paper presents a conceptual model for multi-agent cooperation. An interdisciplinary approach uses results from Distributed Artificial Intelligence (DAI), Computer Supported Cooperative Work (CSCW) and Discourse Theory.

Research in CSCW (Greif, 1988) has led to systems that support human-human cooperation by providing e-mail, electronic conferencing and shared workspaces, but these systems do not participate in the problem solving process.

In DAI (Bond & Gasser, 1988) human-like cooperation procedures have been imitated and formalized to support the cooperation between computers. Within DAI various cooperation methods have been described like negotiation, contract net and master-slave, to mention only the most popular ones. In DAI the term agent has been proposed which we will use throughout the paper to denote any participant, man or machine, in a cooperation process.

What is missing nowadays is a connecting link between these two research directions to support cooperation amongst humans and intelligent machines. Systems supporting the cooperation processes between humans and intelligent computers as assistants can be summarized under the label: 'Human-Computer Cooperative Work' (Steiner, Haugeneder & Mahling, 1990).

This type of system was pursued in the ESPRIT project IMAGINE. The ultimate goal of the project is to develop an environment for building applications that allow agents, both humans and DAI agents, to use cooperation methods. A prerequisite for such a system is a general framework and a language for describing cooperation methods. This paper presents the framework and proposes a specification language. The aim of our research is to design a language that is suitable as method/tool in the analysis and design of systems for support of human-human cooperation (CSCW), systems for cooperative problem solving (human-computer cooperation) and systems for integrated human-computer cooperative work (networks of many humans, many computer agents).

Our point of departure is an agent that owns and controls a body of knowledge and skills. The question we would like to answer is: what do we have to add to enable the agent to cooperate with other agents. With this research agenda, also proposed by Bannon & Schmidt (1989), it is not necessary to explain how an agent on its own can perform tasks to achieve goals. The only phenomenon to be explained is how agents can work together. In connection, the IMAGINE agent model distinguishes between a body that realizes functional capability in some application domain, a head that controls the body and interacts with other agents, and a communicator that provides the head with an interface to send and receive messages. The question then becomes: What is cooperation, how can agents cooperate and what must be in the head to allow for such cooperation? This paper formulates an answer in terms of cooperation methods.

Agents use certain methods or protocols to work together in a coordinated (i.e., productive) fashion. We need a high-level language or notation to describe such highly
interactive cooperative work processes among two or more agents. More specifically, there are five criteria such a language should meet.

1. **Cooperation and Collaboration.** The language must address the cooperative aspects of multi-agent interaction. A specification should be at the multi-agent level and not contain details about intra-agent aspects not relevant for the interaction.

2. **Transparency.** The language for cooperation methods must be easy to learn and to use by the human as analyst or designer or user of cooperation methods.

3. **Suited for N=2 Agents.** This requirement is typical for the IMAGINE project, but even for human-computer interaction it may be profitable to conceive of the system as a collection of agents.

4. **Sufficiently Detailed.** A model or specification of a cooperative work process must contain sufficient information about the rules of the game (i.e., about the rights and obligations of each agent at each point in time) and about the goals the agents may attempt to achieve by playing the game.

5. **Suited as a Representation Language.** An agent must be able to read a specification and get the information about what goals can be achieved and what it can do at each moment (i.e., the rules of the game). For software agents the specification language must be formalised and it must have a semantics in terms of agent behaviour (this paper provides a tentative semantics based on transformation of protocol specifications to agent programs in an existing language).

This paper presents a proposal for a representation language for cooperation methods based on various disciplines. Section 6.1 provides a cross-disciplinary survey, section 6.2 proposes the language and section 6.3 presents the conclusion.

### 6.1 Survey

Different problems may prevent a single agent to achieve what a group of agents could achieve (Bond & Gasser, 1988; Gasser 1991):

- No single agent possesses the knowledge and the skill and the information that is required to do the work.
- A single agent cannot be in different places at the same time.
- No single agent can do the work fast enough.

We are interested in methods that agents can use to circumvent especially the first problem: methods that allow agents to pool their knowledge, skill and information, without their contributions being determined beforehand. We view this kind of cooperation as a process initiated by one agent and in which at least one other agent engages. The process terminates when all agents have disengaged. One agent can be engaged in many cooperation processes in parallel.

Questions about this kind of cooperation are: how is a cooperation process initiated and entered, how do agents reach agreement about goals, methods and sub goals in the work, how is the assignment of goals to agents agreed on, how are conflicts resolved, how can agreements be adjusted during the process, how to deal with unexpected events, and how can cooperation processes be terminated?
This kind of cooperation is complex, not only because there are many different types of entities involved, but also because there is concurrency or parallelism which requires some form of synchronization. More specifically, there is a need for coordination among actions performed by different agents, to deal with interdependencies between actions. This results in a need for communication among agents which brings its own specific difficulties. In cooperative work there is a substantial overhead compared to individual work. Compared to individual work, many new problems arise and much can go wrong if the work is to be carried out cooperatively. Cooperation has been studied in various disciplines and organizational theorists (e.g., Mintzberg, 1983) point out that coordination is essential for cooperative work to be productive.

### 6.1.1 The Agent and Cooperation Methods

We assume agents are rational in Newell's sense: if an agent has knowledge that one of its actions will lead to one of its goals, it will select that action. Such a deliberative agent has been contrasted with a reactive agent that can deal with uncertainty in the outcome of actions, but in general an agent will require both faculties (cf., Ferguson 1992). Reactiveness and rationality can be combined in the framework of process control: the agent has goals, make plans (action specifications), executes actions while monitoring the effect of its actions, evaluates discrepancies between the observed and intended outcome of an action and, if necessary, changes its plan.

When pressed for a definition for cooperation method at this point we could now say that a cooperation method is any method, strategy, or plan that allows for distribution of rational reactive agent activities of planning, acting, executing and monitoring (Steiner et al., 1993).

### 6.1.2 The Cooperation Domain

The major assumption behind this research is the existence of a set of generic activities common across all cooperative tasks (Fikes, 1982; Kraut, 1988; Bhandaru, Croft and Mahling, 1990). Among these are planning, evaluation, establishing commitments, communication, negotiation, etcetera. From many application domains, e.g. air-traffic control or telecommunications management, one can make an abstract description. This abstracted model of the application domain, together with generic activities and concepts like agent and organizational structure, provide the concepts of the cooperation domain.

The main concept for describing work in application domains is that of a task:

**Task** Any piece of work that has to be done; something that one has to do

(The Shorter Oxford English Dictionary)

For cooperative work certain attributes of tasks are important across many application domains. These are found in project planning techniques (Pietras, 1990) such as PERT (Program Evaluation and Review Technique), CPM (Critical Path Method) or Gantt charts. The attributes are:

- **assignment.** (to one or more agents) A task has to be assigned to a certain agent (or agents).
- **allocation.** (to one or more resources) If a task requires resources, they have to be allocated to the task.
• **deadline.** (a point in time) A task can have a deadline before which the task has to be completed.

• **schedule.** (a time period) A task can have a schedule that says during which period the task will be performed.

• **result_to.** (one or more agents) If the task has a result, it may have to be shared with another agent.

• **report_to.** (one or more agents) There may be an agent that has to be informed about the outcome of the task (failure, success, etc.).

• **decomposition.** A task may have a decomposition into a set of subtasks and a control structure.

### 6.1.3 Communication

In communication, expressions do not only have a specific information content, i.e. topic and comment, but also an intended effect on the hearer. An agent may say something to another agent to inform, promise, direct, or declare the other agent something. In this respect, communicative actions have been compared to physical actions, and according to Searle (1969) the primitive elements of discourse are 'speech acts'.

A speech act consists of a proposition — the specific information — and an illocutionary act, which describes the intended effect of the utterance on the listener. The standard modes declarative, imperative, and interrogative are typical examples, Searle's fundamental types provide another set of examples, but the range of illocutionary acts is much wider than that. For example, in didactic discourse Winkels & Breuker (1991) identified 15 types of effects of 6 different types of illocutionary acts.

A general structure for specifying speech act utterances is:

- **Context**
  - Speaker
  - Hearer
  - Time
  - Place
- **Intention to be recognized by the hearer** (one of a small predefined set).
- **Propositional Content**
  - Topic
  - Comment
  - (optional) Propositional Content

The hearer should be able to identify unambiguously the intention, that is, whether an expression implies a request, or a statement of fact, or an assumption (cf. Campbell & D’Inverno, 1990).

Regarding the cooperation domain introduced in the previous section, we conclude that the topics of speech act will often be tasks. The comment may contain values or constraints for the task attributes presented above. In various negotiation scenarios (e.g. scheduling a meeting) constraints on values are used to avoid conflicts caused by premature commitment to a specific value.

The intentions in discourse may not immediately map onto a single utterance (speech act). For instance, if the intention is to help another agent who has just
made an error, it may involve a sequence of sub actions like an explanation of what has been the cause of the error, what could have been a correct action and what can be done to return to a state similar to that before the error was committed. This global sequence of help actions may be further refined until one arrives at elementary illocutionary actions.

Levin & Moore (1977) criticize speech act theory for its emphasis on individual utterances, which has led to a proliferation of speech act types. The emphasis should rather be on dialogues, encompassing multiple utterances and turn-taking. Winograd & Flores (1986) have a similar criticism: speech acts are not unrelated events, but participate in larger conversation structures.

Cohen & Perrault (1979) were the first to view discourse as a result of planning, with speech acts as operators. Nowadays many types of discourse planners have been described and applied. (see e.g. McKeown, 1985; Winkels & Breuker, 1991) These planners can create larger discourse segments but they do not account for dialogues, that is, interaction between the agents. Only the work on dialogue games of Levin & Moore (1977) and more recent work on formal multi-agent theories based on speech acts (Werner, 1989; Numaoka, 1990) tries to give a full account of interaction among agents. In this work, intentions and mutual knowledge are represented using a modal language with operators like ‘know’ and ‘intend’, leading to constructions like: ”speaker intends hearer to intend” and for mutual knowledge constructions like ”a knows that b knows that a knows ...” (see also section 6.1.5).

Grosz & Sidner (1990) use ‘Shared Plans’ as a central concept. Rather than expressing mutual knowledge in a logical language with modal operators, one can simply say that something is shared between agents. In task-oriented dialogues the agents talk about a shared plan and the goals of dialogues can be expressed as intended effects on shared plans.

The shared plan notion seems to be the most elegant way to view intentions involving other agents. It should be noted that examples of Grosz & Sidner (1990) are in fact shared multi-agent plans and the agents discussing the plan may be the same agents that appear as object in the plan.

The notion of shared plans can be integrated with the notion of cooperation domain: the concepts of the cooperation domain provide the building blocks for shared plans. In the agent model, each agent’s head stores its plans (intentions). When agents share a plan fragment, each agent has its own copy of the shared part and each knows with whom it is being shared. We assume that copies are the same and that each agent has perfect information about with whom it has shared the plan fragment.

### 6.1.4 Office Procedure Systems

Another source of understanding is the area of Computer Supported Cooperative Work (CSCW). Office procedure systems are CSCW systems avant la lettre. Figure 6.1 shows a typical office procedure modelled in Role Activity Theory (Holt, Ramsey and Grimes, 1983). The entire procedure is decomposed into a fixed network of sub processes, each belonging to one role or role-type (role with multiple replications, each to be filled by a different agent). The graph is neatly arranged such that the sub process of each role or role-type is in one column. The diagramming language has a well-defined semantics based on (coloured) Petri nets.
Ellis & Nut (1988) review the history of office procedure systems. The problem of office procedure systems is that they are too rigid. Even simple office tasks need a coordination mechanism that is more flexible than one that is based on fixed procedures. The result of the co-workers' efforts can be interpreted as the result of performance according to procedures, but in real offices it is the result of less predictable activities and negotiations among office workers: "As soon as the co-workers would start to work according to rules and procedures, the office would come to a grinding halt" (ibid.).

More flexibility is provided by the COORDINATOR (Winograd & Flores, 1986, Winograd, 1988). It is an e-mail system based on speech act theory. The system is used to try to reach an agreement about delegation of a task to another user. This involves negotiation about how to perform the task and communication after the task has been performed. The system keeps track of these 'conversations for action'. The system forces the user to stay in a rigid framework that can be represented as a diagram, see Figure 6.2. Users can send messages of predefined types (e.g. request, promise) After reading a 'request message' the user can press three buttons: 'promise', 'counter' or 'decline'.

This example differs from the previous not only in the diagramming technique, but, more importantly, also in its target. Whereas the report editing procedure in figure 6.1 is a domain-specific procedure, the procedure provided by the COORDINATOR in figure 6.2 is a general shared plan at the level of the cooperation domain, that can be used to create and maintain a shared plan at the level of the application domain.

Summarising, cooperation methods are shared multi-agent plans for creating and maintaining shared objects. A cooperation method provides a shared framework for the participating agents to discuss and modify, and if necessary unshare the shared object. It is a procedure prescribing how agents can conduct a discourse. The shared
object of a cooperation method can itself be a shared plan in the application domain.

![Conversation for action in COORDINATOR (after Winograd, 1988)](image)

**Figure 6.2: Conversation for action in COORDINATOR (after Winograd, 1988)**

### 6.1.5 Distributed AI and Multi-Agent Systems

In Distributed Artificial Intelligence various proposals have been made for protocols that allow computational agents to pool their knowledge and skills. A well-known example is the contract net protocol (Smith, 1980; Smith 
& Davis, 1981). The protocol is specified in detail as a set of message types and a BNF specification of the behaviour of the individual agent participating in the protocol.

In general, one can always specify a protocol by providing the procedures or programs for the agents participating in each role of the protocol. However, for purposes of specification, most programming languages would provide too much detail. Another disadvantage of programming the agents directly, compared to network specifications presented above, is that the insight in the reciprocity of roles is no longer obvious, and there is nothing to prevent errors that damage the reciprocity creeping in the agent programs.

An interesting language for agent programming is Hoare's language for concurrent sequential processes (CSP) (Hoare, 1985). It has been used in multi-agent systems to program agent behaviour (Connah 
& Wavish, 1990) and it has been used for modelling human-computer dialogues by specifying system behaviour as a function of external user-generated events (Alexander, 1987). Figure 6.3 provides an overview of the most important CSP constructs. Whereas CSP programs may be at an adequate level of detail, they do not help to guarantee reciprocity.

For analysis and modelling of multi-agent interaction, various authors pursue development of modal logics for reasoning about agents, goals and plans (Werner, 1989; Wieringa, 1993; Pitt, Anderton and Cunningham, 1995). Rosenschein and Kaelbling (1986), also use a modal language for expressing the beliefs of agents. They show how such a declarative representation can be compiled into a representation based on a
\( (e_1 \rightarrow P_1) \) prefix. Wait for event \( e_1 \), then activate \( P_1 \)

\( P_1 ; P_2 \) sequence. Activate \( P_1 \), after finishing \( P_1 \) activate \( P_2 \).
\( (P_1 ; P_2) \) finishes when \( P_2 \) finishes.

\( P_1 || P_2 \) parallel. Activate \( P_1 \) and \( P_2 \), synchronising on shared events.
\( (P_1 || P_2) \) finishes when both \( P_1 \) and \( P_2 \) have finished.

\( (e_1 \rightarrow P_1 \) event handler. Wait for \( e_1 \), then activate \( P_1 \) and \\( e_2 \rightarrow P_2) \) finish when \( P_2 \) finishes.

\( SKIP \) Finish immediately.

\( ABORT \) Stop execution.

\( Pname = P \) Name task \( P \) as "\( Pname \)".

(Defines a new process \( Pname \))

Figure 6.3: A subset of Hoare’s CSP language (after Alexander, 1987)

finite-state-machine. In an agent realized as such a machine rather than as a theorem prover the beliefs are not represented explicitly, but one can still say that the agent has the beliefs. In the same line Werner (1980) compares his (modal) logical theory of intentions and communication with protocol specifications in terms of networks showing which messages may be exchanged when, and procedures for generating and interpreting messages. He also regards the latter approach as providing implicit representations: “...the messages are given a pragmatic interpretation by indicating how they effect and are generated by the informational state of an agent. The procedures interpreting the protocol messages are implicit descriptions of linguistic and task intentional states” (see also section 6.1.3).

6.1.6 Conclusion

In the above we have achieved an understanding of cooperation and cooperation methods that can be summarized as follows:

- In order to be productive, cooperative work needs to be coordinated.
- Shared multi-agent plans or task structures provide a coordination mechanism. Figure 6.1 provides an example.
- Flexible cooperation requires that such shared multi-agent task structures can be created, extended, modified or unshared dynamically. This is in itself a cooperative work process, but at a “higher level”. To coordinate this cooperation, agents can use pre-defined and pre-shared (culturally shared) multi-agent task structures at the “higher level”. Figure 6.2 provides an example. These task structures or protocols qualify as cooperation methods: methods agents can use to pool knowledge and skills, without their contributions being determined beforehand.
- Agents must be capable to behave according to these shared plans.

For modelling cooperation methods there are different approaches:
- Protocols as networks. Role Activity Diagrams based on Petri Nets as in Figure 6.1 and networks of messages as in Figure 6.2. Let us, for the moment, ignore their differences. The are both network approaches, and either language or technique could be used to encode both the application domain work flow and the higher level conversation of action.
- Protocols as agent programs for each role, with, for example, CSP as a programming language.
- Modal multi-agent logics that allow reasoning about possible future agent actions, which use formulae in a mathematical language.

Table 6.1 compares these approaches to modelling cooperation methods on the criteria mentioned in the introduction.

The networks are attractive because they show the reciprocity of roles. Of these, the diagrams used by Flores and Winograd provide the easiest understanding of the constraints on behaviour of the agents. People find them easy to use, as we experienced in several workshops, and such a network can indeed be used to provide a partial specification of a cooperation method. This type of diagram concentrates on the control structure and ordering constraints among the messages to be communicated, whereas Role Activity Diagrams need to show the thread of execution for each role, which is often unnecessary and irrelevant to the cooperation process. Not only are W&F networks more succinct, they allow complex control constructs such as iteration and branching, while keeping a simple and overseeable network. These structures can be done in Role Activity Diagramming or Petri Nets, but they become very large and any natural appeal of simple networks is gone.

Whereas the W&F networks are easiest to understand among the three modelling approaches, there can be a problem in what such network means. W&F networks are often interpreted as finite state machines or state transition diagrams (e.g., Burmeister, Haddadi and Sundermeyer, 1993). Nodes are states and arcs are transitions between two state. The arcs are annotated with expressions like $A : request$ and $B : promise$. The COORDINATOR network in figure 6.2 shows states where both agents can take the floor. For example, at node 3, agent B can send a message "report completion" and then the dialogue moves to node 4. But, at node 3, agent A also can send a message ("cancel") and then the dialogue ends up in node 9. Nothing prevents A and B sending a message at the same time, and it is not clear what this might mean in a finite state framework. To give a clear semantics in a state-transition network, it is better to pose a restriction on networks: at each node, only one agent has the floor. In the remainder of this paper, the term W&F networks is used for networks that satisfy this restriction. Having this restriction, as will be shown later, does not forbid mixed initiative.

When W&F networks are tried for protocols involving more than two agents, there arises another ambiguity in what they mean. The question arises whether a message can be sent to all other agents or to a subset. This cannot be simply solved by extending the annotation of arcs (i.e., messages) with the names of the recipient agents. Then the state can no longer be interpreted as the state the entire dialogue or "multilogue" is in. Agents that received the message are in a different state than those that did not receive the message, and the sending agent is in two states. Unless this ambiguity is solved, W&F networks are not suited for protocols involving more than two agents.
Table 6.1: Comparing different approaches to modelling cooperation methods.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Role Activity Network</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Winograd &amp; Flores Netw.</td>
<td>++</td>
<td>+/-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Agent Programs</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Modal Logics</td>
<td>- -</td>
<td>+</td>
<td>+</td>
<td>-</td>
</tr>
</tbody>
</table>

Beside these deficiencies, the networks lack detail. Firstly, they cannot be used to specify data that is passed between the agents. Messages (arcs) could be extended with parameters or arguments to carry data between the agents. Secondly, they cannot be used to specify internal decision-making by the agents. The dialogue network assumes the *savoir faire* of a cooperation process, that is, the rules of the game, but it does not relate this with the individual agent processes. The language could be extended with arcs to represent internal transitions and with global variables to represent conditions. To model the decision making in more detail, it is necessary to annotate arcs with conditions. Thirdly, the networks do not capture the effect of external conditions or organizational structure on the protocol. In the example of two agents bargaining about a price (the haggling example, analysed in detail below) the seller is not supposed to initiate the protocol unless he owns the product or service or unless he is authorized by the owner. The buyer is not supposed to accept a price when he does not have that amount of money. Modelling these requires extending the network language with conditions for transitions. The W&F networks lack detail, but all kinds of extensions to the network language suggest themselves. Before the lack of detail is filled, the W&F networks cannot say enough about the rules of the game, and are not suited as a representation language.

Agent programming is much more powerful and provides virtually no restriction on what can be expressed. It can provide sufficient detail, and it suited as a representation language. There is only one negative aspect to agent programming. A set of agent programs is not easy to understand. First of all, having a separate program for each agent makes it more difficult to grasp how the behaviours of the agents fit together. Compared to W&F networks, the reciprocity of roles is no longer explicit. Secondly, having a set of programs will probably lead to specifications that do not fit on one sheet of paper. They are much larger than a simple network diagram. Specifications in a modal logic language are even harder to understand than sets of agent programs, and as they can be compiled into networks, there seems little advantage in using a modal language as specification language.

In conclusion, the literature survey did not provide a language that immediately satisfies the criteria listed in the introduction. The W&F networks provide a good starting point, but need quite a number of extensions before it provides sufficient detail about the rules of the game and the effects of messages. In the next section we propose a novel approach to modelling cooperation methods, one that combines the advantages of the network specifications and of agent programming.
6.2 A Language Proposal

The survey has yielded a number of approaches to modelling cooperation methods, but not one of these satisfies all criteria listed in the introduction. This section presents an approach developed in the IMAGINE project. It may be regarded as an extension or an improvement over the dialogue specifications using W&F networks. It can be used to describe cooperation methods with much more detail and with greater precision. The language is based on an existing programming language extended with:

1. allocation of agents or sets of agents to statements, and the use of agent variables that may bind to an individual or to a set of agents. These variables can be interpreted as roles (roles or role types in Role Activity theory).

2. the sending of messages between agents. Like in Role Activity theory, three cases are distinguished: 1 to 1, meaning one agent sends one message to another agent; 1 to N, meaning one agent sends each of N agents a copy of the same message; and N to 1, meaning that 1 agent collects similar messages sent by different agents.

3. temporal constraints. (scheduling constraints such as: before, after, while). These are necessary for deadlines or submission periods for messages.

4. sets and set operations to model roles and dynamic role changes. Sets are a convenient means of expression for cooperations involving sets of agents, where agents belong to certain roles and when they can change roles. For each role there is a set and role changes can be expressed as simple set operations. The use of sets to model roles and role changes has also been suggested for electronic conferencing systems DePaoli & Tisato (1991).

For an existing programming language we use Prolog, but the work in this section could be done in many other procedural languages. The role of the programming language is to provide a control structure that is equivalent to the control structure expressed in a W&F network. Moreover, the existing programming language provides the constructs for procedures that can be called and parameters that can be used to pass data between procedures. Whereas semantics of a declarative language like Prolog or of procedural languages is well investigated, the semantics of what one gets by extending such language with message exchange and allocation to agents is a surprisingly dark territory. This section explores whether such language could provide a better solution to modelling cooperation methods.

Below we first introduce Prolog, the extensions for allocation and simple message transfer. Two examples are provided of how these can be used to specify dialogues. Then we turn to protocols involving more than two agents. Contract-net and a meeting are presented as examples.

**Prolog Task Structures** Most programming languages can be used to define task structures for a certain type of goal. We start from the language Prolog, because it has a simple syntax. As a simple example of a Prolog program, one could write a task structure for calculating a percentage:

```prolog
percentagize(Part, Whole, Percentage) :-
```

122
divide(Part, Whole, Fraction),
multiply(Fraction, 100, Percentage).

Important features of Prolog are matching and unification. To explain these: suppose the Prolog interpreter is given the goal of \(<\text{percentagize}(20, 50, X)\>). The Prolog machine will search for a “clause” that matches this form. It will find the clause of the percentagize program of our example: \(<\text{percentagize}(20, 50, X)\>) matches with \(<\text{percentagize}(\text{Part, Whole, Percentage})\>) Unification of the variables yields: \(\text{Part} = 20, \text{Whole} = 50\). After this matching and unification, the Prolog machine will go through the steps of the percentagize program. First it will search for a clause that matches \(<\text{divide}(20, 50, \text{Fraction})\>). Suppose this is a predefined system program that can be executed. Then it will finish successfully, with a binding: \(\text{Fraction} = 0.4\). After that a similar process unfolds to solve \(<\text{multiply}(0.4, 100, \text{Percentage})\>) and finally the percentagize goal will finish successfully with \(\text{Percentage} = 40\), that is, it will finish as: \(<\text{percentagize}(20, 50, 40)\>.

At some point, the Prolog machine may fail to find a clause for a sub goal. It will then go back to the previous step and try to find an alternative matching and unification (backtracking). Figure 6.4 shows the Prolog constructs used below.

\[
\begin{array}{l}
G_1, G_2 \quad \text{conjunction.} \quad (G_1, G_2) \text{ succeeds if } G_1 \text{ succeeds and } G_2 \text{ succeeds.} \\
\text{In a procedural reading (i.e., first } G_1 \text{ and then } G_2\text{), conjunction is similar to sequence.}
\end{array}
\]

\[
\begin{array}{l}
(G_1|G_2) \quad \text{disjunction.} \quad (G_1|G_2) \text{ succeeds if } G_1 \text{ succeeds or if } G_2 \text{ succeeds.} \\
\text{In a procedural interpretation first } G_1 \text{ is tried. If } G_1 \text{ succeeds, } (G_1|G_2) \text{ succeeds. Otherwise, } G_2 \text{ is tried. If } G_2 \text{ succeeds, } (G_1|G_2) \text{ succeeds. Otherwise, } (G_1|G_2) \text{ fails.}
\end{array}
\]

\[
\begin{array}{l}
G_1 : \neg G_2 \quad G_2 \text{ is a method for } G_1. \text{ If } G_2 \text{ succeeds then } G_1 \text{ succeeds.}
\end{array}
\]

Figure 6.4: Prolog Constructs used in Specifications.

Two Agents: Allocation and Messages We can start from such a language and add a construct to assign subtasks or subgoals to agents or to agent variables. Each subtask or subgoal is associated with a variable that can take an agent or a list of agents as value. To communicate data from one agent to another we add a statement of the form

\[
\text{Sender : Message(Arg}_1, ..., \text{Arg}_N) \rightarrow \text{Receiver}
\]

It is intended as an abstract specification of communication between agents, which can be realized in different ways. For example, using a shared variable, a shared database or messages. It is a handshake involving both the sender and the receiver. For example, if the communication is based on messages, the sender must send the message and the receiver must receive it.

Using these, we can now write a multi-agent taskstructure for percentagizing:
Agent1, Agent2 : \text{percentagize}(\text{Part}, \text{Whole}, \text{Percentage}) : - \\
Agent1 : \text{divide}(\text{Part}, \text{Whole}, \text{Fraction}), \\
Agent1 : \text{message}(\text{Fraction}) \rightarrow \text{Agent2}, \\
Agent2 : \text{multiply}(\text{Fraction}, 100, \text{Percentage}).

The meaning of this code can be based on transformation to a set of programs for the individual agents: one program for each agent filling the role of \textit{Agent1} and one program for the agent filling the role of \textit{Agent2}. The control structure of both agent programs is the same as the control structure in the specification. To generate the agent program for \textit{Agent1}, lines of the form \textit{Agent1} : \text{Message} \rightarrow \text{Receiver} become a send statement in the agent program. Lines of the form \textit{Sender} : \text{Message} \rightarrow \textit{Agent1} become a receive action or an event handler in the agent program. Lines of the form \textit{Agent1} : \text{Something} become a line \text{Something} in the agent1-program. Lines of the form \textit{Agent2} : \text{Something} become a \text{SKIP} statement in the agent1-program. Below various cooperation method will be specified using these ideas.

\subsection{6.2.1 Examples}

\textbf{Haggle} Figure 6.5 shows a Winograd & Flores network for two agents haggling over a price. We start from this network specification, and we transform it to a a control structure in the extended Prolog-like language, and then we can add constraints on the contents of various messages:

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{haggle.png}
\caption{A Winograd and Flores network for haggling over a price.}
\end{figure}

For those not familiar with Prolog: the language uses recursion to express the iteration in the control structure of haggling. The multi-agent procedure for haggling has a subclause or subprocedure which represents the entire network starting from node 2 (Figure 6.5). This is necessary, in order to have a name to call recursively. In general, to transform a state transition diagram to code, each node that can be reached by more than one arc requires a sub clause or sub procedure in the encoding.

\begin{verbatim}
Seller, Buyer : haggle(Price) : - \\
(1) Seller : propose(P) \rightarrow Buyer,
\end{verbatim}
This example shows two improvements over the W&F network specification. The first is the use of arguments or parameters in the messages for the data exchanged. The second are the constraints on the values of parameters. These are true properties of haggling and these are now explicitly expressed.

Pause for Question In human discourse, when one participant informs or explains something to the other participant, the speaker transfers a part and then inserts a small pause to allow the other participant to take the initiative when he or she needs explanation. If the other does not act, the first participant, after a little while, continues. This example thus provides a model for a mixed initiative interaction:

```
Speaker, Hearer : transfer([First|Rest])) : —
(1) Speaker : inform(First) —> Hearer,
(2) | Hearing : question(Attribute) —> Speaker within 1sec,
(3) | Speaker, Hearer : inform(Attribute, Value)
(4) | not(Hearing : question(Attribute) —> Speaker within 1sec),
(5) ),
(6) Speaker, Hearer : transfer(Rest))
```

This example clearly illustrates how our language can deal with temporal aspects of agent interaction.

6.2.2 N>2 Agents

The examples above were limited to dialogues between two agents. A simple extension is to allow sets of agents. This is equivalent to role-types in Role Activity Theory. In general this means that each agent filling the same role will have its own copy of the agent program for that role.

In the exchange of messages, one has to be careful with sets of agents. The exchange of a message is written as:

```
Sender : Message(Arg1,..,ArgN) —> Receiver
```

If Sender is a single agent and Receiver a set, then this is a 1 to N broadcast which has a clear meaning. A message is sent to a set of receivers, meaning that copies of the same message will be sent to each receiver.
It is also possible to have a N to 1 construct: a receiver can collect similar messages from different senders. This would be written as:

\[ \text{msgs} = \{(S, Arg_1, \ldots, Arg_N) \mid S : \text{Message}(Arg_1, \ldots, Arg_N) \rightarrow \text{Receiver} \} \]  

Read this as: There is a set of terms \((S, Arg_1, \ldots, Arg_N)\) such that \(S\) has sent a message of type \(\text{Message}\) with arguments \((Arg_1, \ldots, Arg_N)\) to \(\text{Receiver}\). Collecting messages from anyone without a deadline would probably never finish. In general one cannot be sure that all agents will send a message or that all messages that have been sent will indeed arrive. To make sure that (2) terminates, one needs a submission period \((\text{within})\) or a deadline \((\text{before})\) and one would write the right-hand side of (2) as follows:

\[ \text{msgs} = \{(S, Arg_1, \ldots, Arg_N) \mid S : \text{Message}(Arg_1, \ldots, Arg_N) \rightarrow \text{Receiver before Time} \} \]

The value of \(\text{Time}\) is like a deadline, it is measured on the clock of the receiving agent, because the time between sending and receiving of messages is uncertain.

Set definitions like (3) can be made more specific, one may include arbitrary logical constraints that act as a filter on messages. For example, to collect messages having a single argument-value greater than 10 one would write:

\[ \text{Large}._\text{ Enough} = \{(S, Arg) \mid S : \text{message}(Arg) \rightarrow \text{Receiver, before Deadline, Arg} > 10 \} \]

The next example shows the use of these communication mechanisms in the specification of the well-known contract-net protocol (Smith, 1980, Smith & Davis, 1981), a cooperation method for the coordination of both planning and execution of tasks.

**Contract Net**  In the Contract Net Protocol there is a manager agent who has a task and who wants to contract it to another agent. The manager first requests bids from a set of agents. Some or all of these “Workers” reply with a bid, and the manager selects the bidder with the best bid as contractor. All are informed, and the contractor performs the task and returns the result.

\[ \text{Manager, Workers : contractnet(Task, Deadline, Contractor, Result) : } - \]

(1) \(\text{Manager : request(Task) } \rightarrow \text{Workers,}\)
(2) \(\text{Bids} = \{(B, V) \mid B \in \text{Workers}, B : \text{bid}(V) \rightarrow \text{Manager before Deadline}\},\)
(3) \(\text{Manager : select.bst.bid(Bids, (Contractor, V)), (Contractor, V) } \in \text{Bids,}\)
(4) \(\text{Manager : award } \rightarrow \text{Contractor,}\)
(5) \(\text{Manager : reject } \rightarrow \{B \mid (B, V) \in \text{Bids}, B \neq \text{Contractor}\},\)
(6) \(\text{Contractor : Task}\)
(7) \(\text{Contractor : report.result(Task) } \rightarrow \text{Manager.}\)

As a final example, below we show how the language can be used in the design of a Computer-Supported Cooperative Work (CSCW) application.
Meeting  The final example is a cooperation method for a meeting with a chair and a number of participants. The cooperation method represents a protocol where the chair is responsible for working through an agenda as a list of items. An item is first introduced by the chair and then each participant may request the floor to make statements about the item or to respond to statements made by another participant. The chair receives the requests to take the floor and appoints the floor holder. The cooperation method effectively represents a turn taking policy. When the chair judges that the participant's contributions lack newness, meaning the item has been discussed thoroughly enough, the chair concludes the discussion of the item, and introduces the next item, or finishes the meeting.

![Diagram](image)

Figure 6.6: An annotated W&F diagram for a meeting protocol.

Wielinga et al., (1995b) use this example to illustrate their approach for designing CSCW software. They provide a high level specification using a data flow diagram of the meeting process, somewhat similar to an inference structure in a model for expertise in their KADS approach to KBS design (see Schreiber et al., 1993). This specification provides no information about behaviour. It shows which activities may occur, but it provides no information about which activity may occur when nor how it effects future behaviour. This high-level specification is then transformed to an object-oriented implementation language. Only at this point the behaviour is specified.

In the approach of this paper, the specification of behaviour (i.e., the rules of the game) is addressed from the beginning of the design process. First a W&F network for the meeting protocol is drafted. This network (Figure 6.6) already incorporates behavioural constraints. Then the network is transformed to a specification in the representation language, which gives a more detailed and precise specification of the rules of the game (Figure 6.7). This high-level specification then can easily be transformed to agent programs to implement agents that can comply with the protocol.
Chair, Participants : meeting([First|Rest]) : —
(1) Chair : new_item(First) —> Participants,
(2) Chair, Participants : do_item_in_meeting(First),
(3) Chair, Participants : meeting(Rest).

Chair, Participants : do_item_in_meeting(Item) : —
(1) X ∈ Participants : request_floor —> Chair,
(2) Chair : assign_floor(X) —> Participants,
(3) Nonfloor = Participants — X,
(4) Chair, Nonfloor, X : do_statement_in_meeting().

Chair, Nonfloor, Floor : do_statement_in_meeting() : —
(1) Floor : statement(S) —> Chair + Nonfloor,
(2) Requests = {X | X ∈ Nonfloor, X : request_floor —> Chair, while
Chair : assess_newness(S, Verdict)},
(3) Verdict = new,
(4) Requests = [First|Rest],
(5) Chair : assign_floor —> First,
(6) NewNonfloor = Nonfloor + Floor — First,
(7) Chair, NewNonfloor, First : do_statement_in_meeting()
(8) | Verdict = new, Floor Requests = []
??? what to do in this condition ???
(9) | Verdict = old,
(10) Chair : conclude_discussion() —> Floor + Nonfloor

Figure 6.7: The meeting protocol in the specification language.

6.3 Conclusion

There is a need for a language to model cooperative work processes. The introduction stated five criteria such a language should meet. We did a survey and in section 6.1.6 a number of approaches were compared on the criteria. It was concluded that none is satisfactory on all criteria and that modal logic is least suited, simply because it is too difficult to learn and use.

We considered the advantages of the different approaches to modelling cooperation methods and synthesized a proposal for a language. This language allows specifications of protocols that are sufficiently detailed about goals that can be attained, about the rights and obligations of each agent at each point in time, and about the success or failure in attaining the goals give a particular unfolding of the protocol. We have demonstrated this with a few example specifications.

Coordination Mechanisms A protocol distinguishes between two or more different roles for agents. The simplest protocols only provide a control structure over the messages between different roles and a success/failure status on exit. The proposed language can be used to encode such protocols, but is also allows one to elaborate a role in a cooperation method in full detail. That is, it is allowed to specify all
meeting([First|Rest], Chair, Participants):-
    receive(new_item(First), Chair) --->
    do_item_in_meeting(First,Chair,Participants),
    meeting(Rest,Chair,Participants).

do_item_in_meeting(Item,Chair,Participants):-
    ( send(request_floor,Chair)
      | skip ),
    ( receive(assign_floor(self), Chair) --->
      Nonfloor = Participants - self,
      do_statement(Chair,Nonfloor,self,Statement),
      | receive(assign_floor(X), Chair), X /= self --->
      Nonfloor = Participants - X,
      do_statement(Chair,Nonfloor,X,Statement) ).

do_statement(Chair, Nonfloor, self, Statement):-
    send(statement(Statement),Chair + Nonfloor),
    ( receive(assign_floor(self), Chair) --->
      do_statement(Chair,Nonfloor,self,Statement)
      | receive(assign_floor(X), Chair), X /= self --->
      NewNonfloor = Nonfloor + floor - First,
      do_statement(Chair,NewNonfloor,X,Statement)
      | receive(conclude_discussion(Chair) ---> skip ).

do_statement(Chair, Nonfloor, Floor, Statement):- Floor /= self,
    receive(statement(S)) --->
    ( send(request_floor,Chair)
      | skip ),
    ( receive(assign_floor(self), Chair) --->
      Nonfloor = Participants - self,
      do_statement(Chair,Nonfloor,self,Statement),
      | receive(assign_floor(X), Chair), X /= self --->
      Nonfloor = Participants - X,
      do_statement(Chair,Nonfloor,X,Statement)
      | receive(conclude_discussion(Chair) ---> skip ).

Figure 6.8: The agent program for a participant.
meeting([First|Rest],Participants):-
    send(new_item(First),Participants),
    do_item_in_meeting(First, Participants),
    meeting(Rest, Participants).

do_item_in_meeting(Item,Participants):-
    (receive(X,request_floor),X in Participants: --->
        send(assign_floor(X),Participants),
        Nonfloor = Participants - X,
        do_statement(Nonfloor,X)
    ).

do_statement(Nonfloor, Floor):-
    (receive(Floor,statement(S)) --->
        ( collect(X,request_floor,Requests)
          | assess_newness(S,Verdict)
        ),
        ( Verdict = new, Requests = [First|Rest],
          send(assign_floor(First)),
          NewNonfloor = Nonfloor + floor - First,
          do_statement(NewNonfloor, First)
        | Verdict = new, Requests = []
        | Verdict = old,
          send(conclude_discussion, Floor + Nonfloor)
    )
).

Figure 6.9: The agent program for the chair.

intra-agent actions belonging to that role. Then transformation to agent programs yields a system agent capable of that role. In this way the language can also represent mechanism or devices for coordination as special agents (i.e., a role in coordination work to be fulfilled by an automated agent). In this manner all kinds of objects or mechanisms for coordination (e.g., a semaphore or traffic light, or a turn-taking mechanism, or a sector agent, or a user-interface agent) can be modelled as a role in a protocol.

Expressive Power. The expressive power of the proposed representation language derives not only from the multi-agent perspective, but also from the use of proven programming constructs and the use of set operations to model the effect of messages on role-membership. This allows one to specify complex examples (e.g., the meeting example) in a succinct way. The language is more explicit about behaviour than modal logic approaches, but it is still at a high level, higher than agent programs. It is of the same level as W&F networks, but it provides more precision and more detail, as logical and temporal constraints can be added, and operations on sets of agents are made explicit (cf., Pitt et al., 1995).

Semantics. Specifications in the language, except for extreme cases where all intra-agent actions are specified, do not provide a complete description of behaviour. This has a reason: agents can be involved in many different cooperations in parallel
and it is up to the individual agents to decide whether and how they will continue a certain cooperation. The protocols as specified provide the agent with legal moves, but they do not prescribe the specific move the agent should take.

The formal semantics of the language is not worked out, but it is indicated that an operational semantics can be defined as the transformation of a specification in the proposed language to a set of agent programs (i.e., an agent program for each role). This was illustrated in the meeting example. Pitt et al. (1995) confirm the observation that in practice the manual transformation to a set of agent programs appears not difficult.

To evaluate the language proposal, Table 6.2 shows how it compares to the network and agent programming approaches. They are ranked on each of the criteria. Regarding the first criterion, all modelling approaches can address cooperation among agents, they were selected for that reason. One the other criteria, they differ substantially.

Transparency. Regarding ease of use and learning, the proposed language ranks between networks and agent programs. Networks like Role Activity Diagrams or W&F networks provide a simple graphical technique that is easy to understand and use. The proposed representation language is more difficult to learn than networks: one has to learn programming language constructs. Compared to agent programming, the proposed language is easier to use, because a specification in the proposed language is more succinct than a set of corresponding agent programs (compare the size of the code for the cooperation method in figure 6.7 with the combined amount of code of the agent programs in figures 6.9 and 6.8). Moreover, in drafting specifications using the proposed language, it is easier to avoid errors in the reciprocity of roles compared to programming separate programs for the different agent roles. Even if one is an experienced agent-programmer, it may be difficult to oversee the set of agent programs and it may be difficult for the human programmer to avoid errors in the reciprocity of roles. Even if the final aim is a set of agent programs, the proposed specification language may be used as a step in-between. It is easier for the human to ensure that each agent program complies with the specification in the proposed language, than to ensure that the agent programs comply with each other.

N>2 Agents. All approaches considered can be used with more than two agents, but the W&F diagrams may become ambiguous (p. 120).

Sufficiently Detailed. Network approaches do not allow much detail. The proposed language is flexible in its level of detail. If desired, it is allowed to provide the same level of detail as agent programs, by specifying all intra-agent actions. Therefore, the proposed language gets the same rank as agent programs on this criterion.

Suited as a Representation Language. It appears that, given a set of choices or given the selection of a distributed architecture and programming paradigm, specifications in the representation language can be compiled to agent programs. Therefore, on this criterion, the proposed language is also ranked equal to agent programs.
Table 6.2: Comparing the proposed language with two existing approaches to modelling cooperation methods.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Role Activity Networks</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>W &amp; F Netw.</td>
<td>++</td>
<td>+/-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Agent Programs</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Proposed Language</td>
<td>+/-</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
</tbody>
</table>

It appears that our proposal satisfies all criteria, whereas none of the existing approaches does. It offers a bridge between W&F networks and implementation languages such as agent-process languages or languages for (distributed) object-oriented systems. Compared to network specifications according to Winograd and Flores, the representation language is more expressive. It is more precise and provides more detail about the rules of the game. And yet the specifications are succinct and they can easily be transformed to agent programs in an existing implementation language.

Acknowledgements I gratefully acknowledge the invaluable contribution of Keith Clark and Frank McCabe (Imperial College) in the development of the representation language. This research has been partly funded by the European Community as ESPRIT project 5362 (IMAGINE).