ROPE:
Role Oriented Programming
Environment for Multiagent Systems

M. Becht, T. Gurzki[1], J. Klarmann, M. Muscholl,

Institute of Parallel and Distributed High-Performance Systems

University of Stuttgart

Breitwiesenstrasse 20-22

D-70565 Stuttgart

{becht,klarmann,muscholl}@informatik.uni-stuttgart.de

Abstract

This paper introduces a programming environment and architecture for the development of agent based cooperative applications using a role based approach. We focus on the cooperative aspects by introducing cooperation processes (CP) as a concept of its own. CPs describe all and only the coordination and cooperation parts of an application. The explicit documentation of the coordination and cooperation mechanisms used in a MAS allows their evaluation and reuse. We are able to change existing and introduce new cooperation processes at runtime without modifying the existing agents. We specify the cooperative behaviour of an agent in a separate role description. The interconnection of these roles constitutes the CP. Describing cooperation independent from concrete agents allows to build heterogeneous, federative and transformable MAS. We show how agents decide what roles to accept and how the agent-role interaction works. Finally we present the ROPE framework and runtime environment.

1 Introduction

In todays rapidly changing markets companies have to be able to change their organizational structures, their size and to some degree the domain of their business. The ability to transform becomes a more and more important factor for success. It is the aim of the joint research project SFB 467 „Transformable Corporate Structures in Multi-Variant Serial Production"[2], supported by the Deutsche Forschungsgemeinschaft (DFG) to acquire theoretically well founded and interdisciplinary approved knowledge to describe, understand and build transformable corporate structures for the production domain.

A major drawback of existing systems is their lack of adaptability to structural change. To address these restrictions we are proposing a new approach. But before we move to the technical part we will give an idea of the problems we wish to address.

Let us assume a company that produces a variety of goods with different quantities with a set of autonomous performance units. Let us further assume that the company

---

[2] For more information on this project see
http://www.sfb467.uni-stuttgart.de

---

[1] Now at: Fraunhofer-Institut für Arbeitswirtschaft und Organisation, Nobelstraße 12, D-70569 Stuttgart

uses a production on demand strategy. Due to the complexity of the scheduling it might be possible that the company uses different planning strategies tailored to different product classes and quantities. The company may produce a state of the art product in a very high quantity with a planning strategy which e.g. takes care of machine maintenance issues or lets the customers place and customize their orders in an on-line ordering system. Additionally the company may produce single specialized products tailored to the needs of one customer using a planning strategy which optimizes for process quality instead of low cost. So instead of trying to implement one super intelligent planning strategy which is able to cope with all combinations of requirements which may ever occur, different planning algorithms which are optimized for different aspects and situations may be used and added to the system as necessary. For example it should be possible for each order to have its own customized cooperation process.

For satisfying these requirements in complexity and multiplicity in structures and tasks, MAS are increasingly used in the production domain (e. g.: InteRRaP [5]). These approaches use MAS with certain structures and agents which behave according to their position in the structure. They provide a multiagent based solution for the complex production planning problem. Nevertheless, there is still a lack in supporting flexible and rapidly changeable structures.

[6] propose a similar approach, were cooperating agents provide services which are controlled by interpreted scripts. An important difference is that for a cooperation process for each role there is a separate cooperation protocol. E.g. the manager (in a contract net) follows a different local protocol than the bidders. In our approach they all follow the same global protocol, but showing different behaviour in each stage of the CP.

To decouple the agent (task/production) capabilities from his cooperation capabilities [11] we introduce the concept of role. There is different work on roles in software engineering [7] and in the MAS community ([8], [3]). In [7] roles are used to "identify abstract actors" during the system analysis but the roles are mapped to classes by the software engineer later in the software development process. The work (with a strong software engineering background) presented in [8] has basically the same view on the importance of roles for MAS and the same view on what types of roles there would be in a MAS. The work differs from ours in that we provide a cooperation model, specification language, programming environment including a generator to derive an object oriented implementation from a given specification. Whereas in [8] the main focus is to identify generally applicable roles, role models and related design patterns and to extend object oriented modelling techniques.

There is more work referring to roles in MAS research but mostly the roles are not a concept of their own. They are either used to describe an agent's behaviour with respect to a certain task ("Agent A plays the role of the manager") or the role specific behaviour is not separated from the agent but included in his communication layer.

In the CSCW community (where human agents cooperate) there is no doubt about the importance of roles. Nevertheless our concept of role differs from the "common" CSCW role definition, in that it is finer grained and specific to activities. In [9] the ROPE cooperation model is applied to develop more flexible workflow schemes.

2 Overview of the ROPE project

Figure 1 shows the components developed in the ROPE project. Since we aim to describe cooperation we had to develop a general model of cooperation which will be described in section 3.
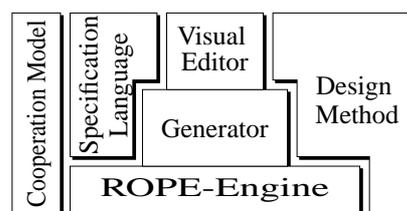

Figure 1. Components of ROPE.

To be able to describe cooperation processes graphically yet with well defined formal semantics we have developed a formal specification language extending high level petri nets [1].

A visual editor to design cooperation processes, a generator which transforms these cooperation processes described in the specification language to role implementations and the Rope Engine are prototypically implemented (see [2]).

In the present paper we focus on the ROPE engine. Our goal has been to develop a real distributed MA environment and not a simulation running on a single machine. The ROPE engine provides all the necessary code for the distributed execution of a cooperation process, thereby imposing very few requirements on the agents.

A detailed design methodology for developing specific cooperation processes is left for future work.

Figure 2 outlines the steps of a software development process with respect to ROPE. It shows which part of ROPE supports which part of a development process.
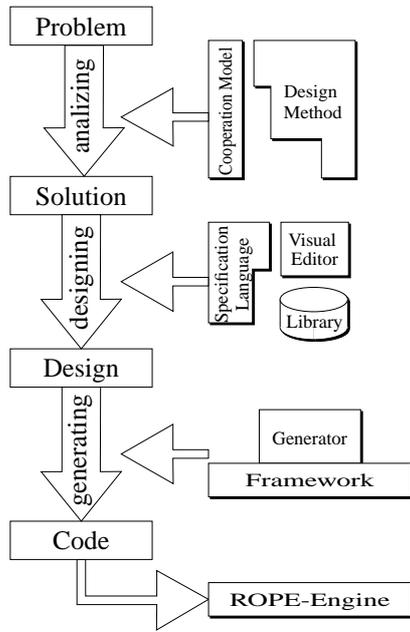
Figure 2. Parts of the design process supported by ROPE

3 Model and language

The scenarios in section 1 state that it is crucial to organizations to be transformable. Transformability requires that it must be possible to change the organizational structure without the need to restructure the tasks and vice versa. It should be possible to insert new cooperation processes without changing the agents and without changing the running cooperation processes or stopping and restarting the whole system. It should also be possible to replace agents taking part in a cooperation process by other agents and to add new or change running cooperation processes.

To fulfil these requirements we introduce our key concepts in section 3.1. Further design decisions lead to the implementation of ROPE. They are summarized in section 3.2.

3.1 Key concepts

The characteristic feature of ROPE is its strong emphasis on the role concept. It is introduced for satisfying the agent's requirements mentioned above.

Roles provide a well defined interface (Role-Agent Interface) between agents and cooperation processes, which enables an agent to read and follow the normative rules given by the cooperation process even if not known to the agent before.

In the specification of a cooperation process we introduce a role as an abstraction of an agent. The agent carries out the actions initiated by the role. The fundamental idea is to decouple the organization of the agents in the multi-agent system from the structure of cooperation processes. By that, changes in the agent organization do not affect the cooperation process specification and vice versa[3]. This is the prerequisite for more transformable agent cooperation.

We understand cooperation as a process which is controlled by normative rules to which the cooperating partners commit themselves, when accepting a certain role. The rules are defined with respect to the cooperation, using a global view which is independent from certain agents.

On a more concrete view we understand a cooperation process as a set of stages which model interactions. In a running cooperation process one or more stages are active. After the goal of a stage has been reached the control proceeds to one or more successive stages. The precondition of a stage is given by the goals of the preceding stages.

Agents taking part in a cooperation process play at least one role. A role requires the agent to have permissions and capabilities, whereas an agent playing a role commits itself to the obligations specified by the role. Every agent satisfying these requirements can execute the role.

During the execution of a cooperation process an agent can change its role. This allows to have roles designed for a particular purpose and are therefore small and easy to maintain.

We define a role as an entity consisting of a set of required permissions R, a set of granted permissions G, a directed graph of service invocations A and a state S visible to the runtime environment but not to other agents. The sets R, G and A may be empty. A describes the agents behaviour and may contain an arbitrary number of alternatives. The state of a role is used by the runtime environment to determine if the cooperation has reached the next stage. Roles may spawn sub cooperations and therefore have an associated set of sub-roles. Sub-roles inherit the granted permissions from their super-roles.

We define an agent as an entity consisting of a set of provided services. An agent a priori has no permissions.

3.2 Design decisions

Implementing the key concepts we had to make design decisions, which are explained in the following.

---

[3] In fact we see the organization structure of agents as long-term cooperation processes.

Because the functionality of an application domain is considered as stable we base our model on the existence of a service model which has to be developed specific to a certain application-domain. A service may be concrete, leaving no possibility for interpretation, or may be more abstract, requiring intelligence and autonomy for execution. Additionally services have primitives for asking the service provider to make a decision.

An agent then provides a set of services describing its capabilities. The service interface is independent of programming languages and allows to execute roles as clients remote from agents as servers.

The advantage of implementing the Role Agent Interface in this way is the possibility to map agents to roles using standardized technologies like e.g. CORBA or DCOM.

Other multi agent architectures using KIF/KQML based cooperation require more sophisticated mechanisms to embed agents with another technology. In our approach it is easier to provide a wrapper for existing applications and thereby include them in a cooperation process.

Capabilities are modeled as services. Therefore a role specifies the services needed by an agent. The obligation is specified by an action related to a role. The action describes how the required services have to be used. A role entering a stage executes its action on the services provided by the agent.

In detail an instantiated role runs asynchronously to its agent. It is responsible for the coordination part of a task and thereby controls the agent accordingly. The responsibility of the agent is to supply the role with knowledge and decisions that enables the role to fulfil its part.

The specification of permissions or more general rights is out of the scope of this paper. [10] addresses this topic in more detail.

The last design decision concerns the specification language. Because we want to describe cooperation processes and their normative rules prescriptively, we use a high-level petri net class (predicate-transition nets) which is extended by the role concept (see [1] for a formal definition).

Petri nets are known to be suitable for modelling discrete, event based, distributed systems. In our extension we model an agent playing a role as a token. Therefore we are able to describe the dynamic behaviour of interacting agents. Different types of tokens are used to represent different types of roles. Roles export certain local states of their action, so that guard expressions of transitions can be specified as boolean functions over these states. Guard expressions describe goals of interactions.

The advantage of this choice is the adequacy. Tokens allow a good visualization of the distributed state in a cooperation process and thereby simplify the design and test of agent cooperation. Communication between roles is either performed directly between roles in a stage or is done during the firing of a transition.

To summarize, a ROPE multiagent application consists of a multiagent layer implementing the functionality of the application and a cooperation layer on top specifying all cooperation networks and roles needed for cooperation (figure 3).
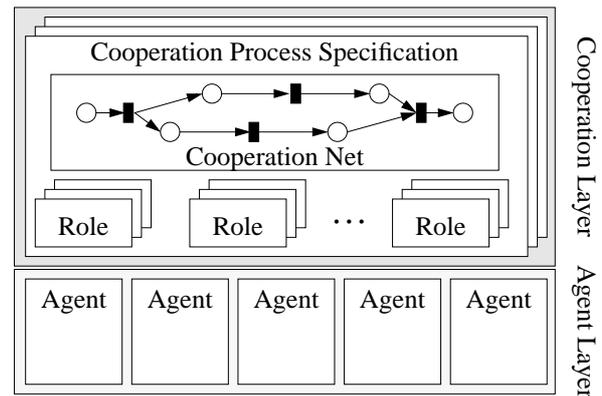

Figure 3. Basic architecture of a ROPE application.

Our design goal is to have an entirely distributed system avoiding a single point of failure.

3.3   Specification of a cooperation process

After having mentioned our key concepts and design decisions, we now introduce how a cooperation process is specified. The specification language is a graphical language to describe petri-nets, which has elements for annotating a place with roles, a role with required services, its action and its states, a transition with a goal and the whole net with a set of initial markings. We focus on the interaction of the model elements referring to the hot spots (Table 1) where application dependent behaviour can be inserted.

Agents playing some role in a cooperation process are represented by tokens. Tokens interact with each other in a stage represented by a place (1.). The part of an interaction performed by one token is specified by the action (3.) of its role (2.). Each token has well-defined local states (3.), which are tested by transitions.

The guard expression of each transition in the postset specifies one goal of the interaction (4.). To evaluate the goal expression the states of the tokens in its preset are tested by the transitions. Reaching the goal yields to a firing of the transition. Tokens are moved from the places

in the preset to places in the postset as specified by the edge annotations (5.) of the firing transition. While moving a token, a role change is performed.

A cooperation process terminates as soon as all tokens have disappeared. The starting roles are defined by the initial marking of a cooperation network (6.).

---

1. Model the interaction stages by a set of places and arrange them in a cooperation net.
2. Specify the collection of roles for each stage.
3. For each role define the required services, the action and the states to be tested by transitions.
4. For each goal of an interaction insert a transition in the postset of the related place and specify the goal through a guard expression.
5. Define the role changes through the edge annotations. This includes possible information propagation to new roles in the following stages.
6. Specify the possible initial markings.

Table 1: Steps to specify a cooperation process

3.4 Agent requirements

We do not make restrictive assumptions about the architecture of the agents. Agents in this layer may be deliberative, reactive or of any hybrid form.

To be compatible to any ROPE application an agent has to satisfy the following requirements:

- Each capability of an agent has to be implemented by one or more services.

- It has to be able to start instances of the ROPE-engine (local or remote).

- Given a set of services, it has to be able to decide which roles to accept.

The coordination part of the cooperation process is managed outside the multi agent layer in a separate cooperation layer. We assume that the agents always communicate via the ROPE engine. Nevertheless other communication will not be forbidden.

There is an initial cooperation process, all agents have to take part in. This CP defines the boundary of a multi agent application and has to define mechanisms to publish new cooperation processes to other agents.

There is no restriction that an agent is only part of one application. Thereby we are able to realize large federative multiagent systems, each with an own set of norma-

tive rules modelled by their cooperation processes. This enables us to support transformable corporate structures as mentioned in the introduction.

3.5 Role-agent mapping

The initial CP is responsible for the role-agent mapping whenever new cooperation processes are instantiated. More sophisticated role-agent mappings where agents negotiate for their role assignments can be achieved by an own cooperation process which starts the intended cooperation process.

In the following we explain how we realize the role-agent mapping at the beginning of a cooperation process. An example of the agent layer of a multiagent system is illustrated in figure 4. The Role-Agent Interface consists of seven service objects, depicted by structured circles. Suppose the system consists of four agents at some time, where the agents support those services that are shown in their bodies. Agent 1, 2 and 4 are generalists, whereas agent 3 is very specialized.
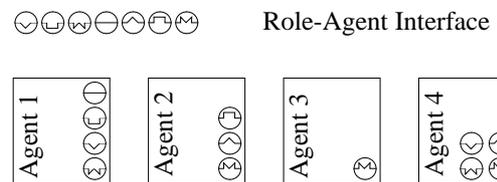

Figure 4. An example of an agent layer.

Suppose an example of a cooperation process as depicted in figure 5 is instantiated. The "shadows" behind each role visualize the stack of roles each agent may have to perform in this CP.

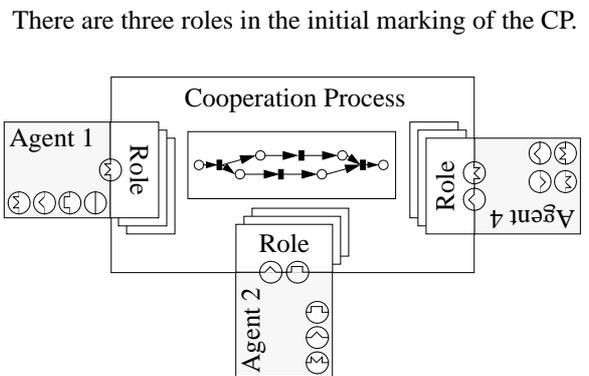There are three roles in the initial marking of the CP.

Figure 5. An example of a cooperation process.

Starting the cooperation process demands to link each role on top of a stack to an agent. The precondition for this linking is fulfilled by a valid mapping from the required services of all roles on top of each stack to the services supported by an agent. Given such a mapping a

selected agent connects to its first role, thereby starting the related action.

3.6 Execution of a cooperation process

Executing its action, the role instance (token) asks the related agent to make decisions, to perform some work or to transfer knowledge up to the role instance or down to the agent. Role instances which are located in the same stage may communicate with each other to exchange intermediate results. After the goal of a stage is reached, the related transition fires. While it fires final results are exchanged and each instance is moved to a succeeding stage, performing a role change. Reaching a new stage a role instance starts its action.

4 Framework

The ROPE Engine architecture relies closely on the model, discussed earlier. We will now take a closer look on how a cooperation executable is derived from the cooperation specification and how it is executed.

For easy code generation, the elements of a cooperation net (place, transition and role) correspond directly with abstract classes of the ROPE framework. The net annotations like guard expressions, edge annotations or the initial marking are used to generate methods for derived classes.

The ROPE engine offers a shared memory for cooperation to roles in a place. Each place object has its own shared memory. A unit of this memory is called slot. Each slot corresponds to the states exported by a role instance. Slots can be read and set by roles. At runtime each agent playing a role allocates slot-memory. These slots are read by transitions to evaluate their guard expressions as follows.

Each transition object consists of methods for evaluating the activation of the transition and for performing the switching of the transition. The activation evaluation is generated from the guard expression and uses backtracking to find a valid mapping from free variables in the expression to roles in the preset of the transition. As soon as such a mapping is chosen the transition performs the switching as generated. The new roles that are created in a following place are initialized and the generated action method is started. The framework provides easy access to manipulate the slots related to a role instance.

The evaluation method of transition objects is invoked within each update of the slot table of an associated place object. All updates to the slot table and switching of transitions are synchronized by the ROPE-engine using atomic broadcast semantics [4]. The ROPE-engine is completely decentralized. Each running cooperation process has its own ROPE-engine which is distributed between the participating agents.

4.1 Distribution of CPs and their execution

A further goal of the ROPE engine is the possibility to distribute cooperation executables to agents and execute them there, though agents may run on heterogeneous target platforms. Therefore the ROPE engine is implemented in JAVA. The implementation uses the standard JAVA classloader for loading cooperation process executables. Therefore it is open to use different distribution mechanisms like a network file system share or a proprietary mechanism for special requirements in the custom application usage (e.g. telematic services, production environment).

If the agent detects a need for cooperation, he is able to invoke a new instance of a cooperation process. A cooperation process includes the description for the other roles, which must be fulfilled by other agents. These agents are invited through an invitation mechanism defined by the initial CP. The framework requires the implementation of these basic services by the agent.

4.2 Accepting roles

The agent is encapsulated by the cooperation maintainer. The cooperation maintainer provides additional functionality, like communication access to executing cooperation processes and may also be extended to serve as a wrapper for other agent architectures.

According to the model, we presented before, an agent is able to take part in a cooperation process, when it is able to fulfil a role. A role is fulfilled by an agent, when he is able to offer all services the role requires. To match the required services with the offered services, the ROPE engine uses unique universal identifiers. An agent is able to take part in an unknown cooperation process, if he can fulfil a role with its services. The service itself is requested by a role. The agent returns a service object, implementing the service using primitives of the agent or encapsulating the agent's service.

5 Example

In this section we specify the contract net as an example cooperation process. The specification of the state, the required services and the behaviour of a role is is given in a java like language. For the contract net protocol we use a subset of the general service model "trade" which was specified as a package of java interfaces. The service model "trade" was also used for the specification of an auction-like protocol.

The complete graphical specification is shown in figure 6. The circles model the phases of the cooperation net whereas the rectangles model transitions between phases. Each phase is annotated with the available set of roles. As each phase provides its own namespace the role "Manager" may have different specifications in each phase. Although we provide the possibility to specify global roles we do not use this feature in the example. Phases 1 and 2 are the initial phases where the potential participant wait for the cooperation to start. Transition A specifies that the cooperation starts if the Manager from phase 1 has made a request and there are at least 10 potential contractors. If the condition evaluates to true all the contractors (specified by c[All]) are moved from phase 2 to phase 4 and are assigned the new role "Participant" and the manager which made the request is moved to phase 3 and is assigned the new role "phase3/Manager". The annotations at the transition-to-phase edges specify the role assignment and the initialisation of the new roles. The initialisations are used to take information about the negotiated service from one phase to the next.
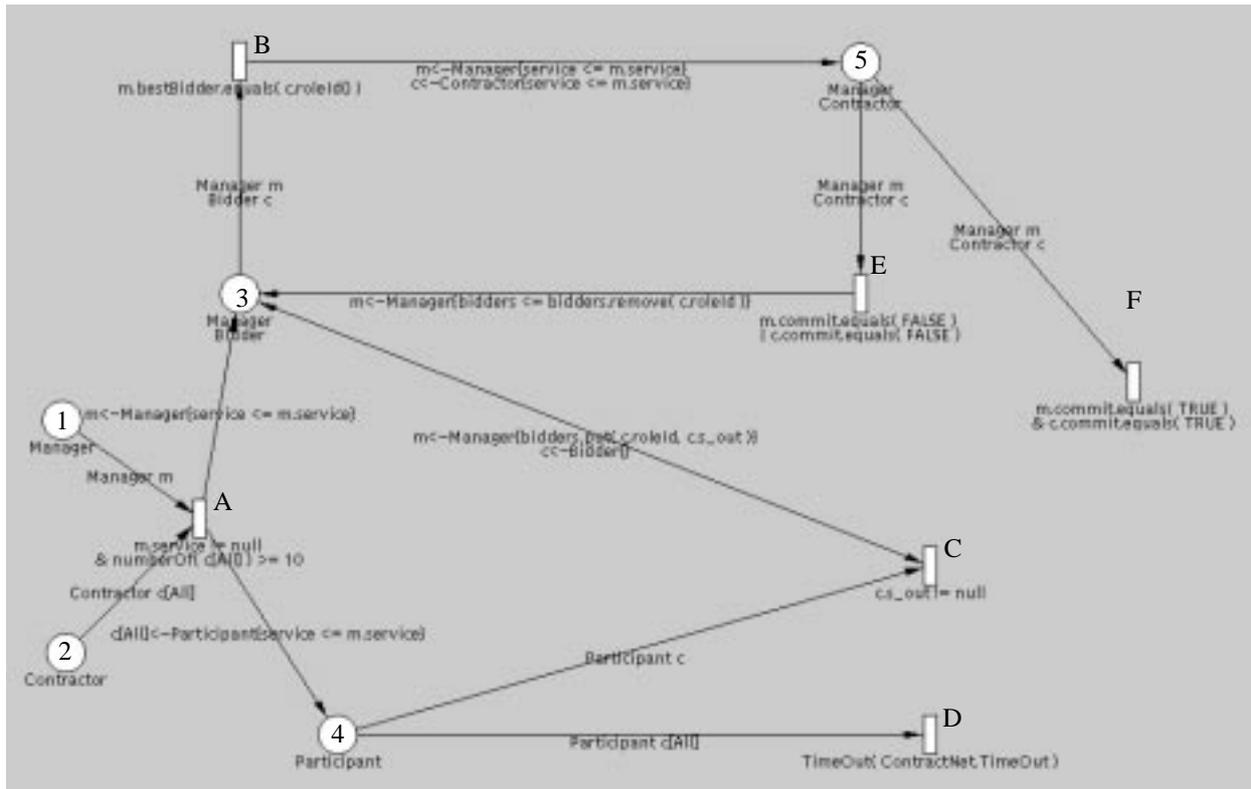

Figure 6. Graphical specification of the contract net

In phase 4 the role "Participant" asks the connected agent to make a bid for the requested service (the agent is referred in the behaviour part of the role by the keyword `agent`). Each time a participant makes a bid, this participant and the manager are removed from their phases and put into phase 3. The bidder (formerly the participant) does nothing in this phase. The manager receives the new offer, compares it to the previous ones and, if it is cheaper, asks the agent to accept it. If the agent accepts the manager and the according bidder are moved to phase 3, assigned the new roles "Manager" and "Contractor" and are both asked to commit to the contract. If they do, the contract net protocol is finished. If they don't, the contractor also leaves and the manager is put back into phase 3 to select another bid or to wait for new offers. Table 2 shows the complete specification of all roles. The `inout` keyword states that this state variable is not accessible for conditions, but for the edge annotations to transfer information. The behaviour-statement e.g. in the manager role of phase 1 is simply a cast from `agent` to an object that implements the `rope.trade.Request` interface. If this fails we'll get (at least in java) a `Class-castException`. So we leave it to the java interpreter to check if an agent provides the required interfaces.

Note that since the negotiated service is completely under the control of the cooperation net (resp. the runtime environment) we can make sure that the agents cannot change the service unless they are supposed to.

```
Phase1::Manager
STATE {
   rope.trade.Service service
   inout java.util.Hashtable bidders
}
REQUIRED SERVICES {
   rope.trade.Request
}
BEHAVIOUR {
   service=((Request) agent).request()
}
```

```
Phase2::Contractor
STATE {}
REQUIRED SERVICES {
   rope.provider.Offer
}
BEHAVIOUR {}
```

Table 2: Specification of all roles.

```
Phase3::Manager
STATE {
   rope.trade.Service service
   RoleId bestBidder
   inout java.util.Hashtable bidders
}
REQUIRED SERVICES {
   rope.trade.customer.Accept
}
BEHAVIOUR {
   Enumeration e = bidders.keys();
   RoleId bestId = null;
   RoleId id;
   while ( e.hasMoreElements() ) {
      id = e.nextElement();
      if ( bestId == null | bid-
ders.get( bestId ).intValue() > bid-
ders.get( id ).intValue() ) {
         bestId = id;
      }
   }
   service.setCost( bidders.get(
bestId ) );
   service.setProvider( bestId );
   service.setCustomer( agent );
   if ( ( (rope.trade.customer.Accept)
agent ).accept( service ) ) {
      bestBidder = bestId;
   }
}
```

```
Phase3::Bidder
STATE {}
REQUIRED SERVICES {}
BEHAVIOUR {}
```

```
Phase4::Participant
STATE {
   rope.trade.Service s_out
   inout rope.trade.Service s_in
}
REQUIRED SERVICES {
   rope.trade.Offer
}
BEHAVIOUR {
   s_out =((Offer)agent).offer(s_in)
}
```

Table 2: Specification of all roles.

```
Phase5::Manager
STATE {
   Boolean commit
   inout java.util.Hashtable bidders
   inout rope.trade.Service service
   inout RoleId bestBidder
}
REQUIRED SERVICES {
   rope.trade.Commit
}
BEHAVIOUR {
   commit=((Commit)agent).commit(serv-
ice,bestBidder)
}

Phase5::Contractor
STATE {
   Boolean commit
   inout rope.trade.Service service
   inout RoleId customer
}
REQUIRED SERVICES {
   rope.trade.provider.Commit
}
BEHAVIOUR {
   commit=((Commit)agent).commit(serv-
ice)
}
```

Table 2: Specification of all roles.

6   Conclusion

We have introduced a programming environment and architecture for the development of agent based cooperative applications using a role based approach. We focused on the cooperative aspects by introducing CP as a concept of its own. We specify the cooperative behaviour of an agent in a separate role description. The interconnection of these roles constitutes the CP.

Describing cooperation independent from concrete agents allows to build heterogeneous, federative and transformable MAS.

A detailed design methodology for developing specific cooperation processes is left for future work.

7   References

[1] Becht, M.; Muscholl, M; Levi, P.: Transformable Multi-Agent Systems: A Specification Language for Cooperation Processes. In: Proceedings of the World Automation Congress, ISOMA'98, May 10-14, 1998, Anchorage, Alaska, USA, Jamshidi, M.; de Silva, C. W.; Pierrot, F.; Fathi, Bien, Z.; and Kamel, M., 1998.

[2] Becht, M.; Klarmann, J.; Muscholl, M. Software Demo of ROPE: Role Oriented Programming Environment for Multiagent Systems. To appear in: Proc. Agents 99, May 1-5 1999, Seattle, Washington, USA.

[3] Omar Belakhdar and Jacqueline Ayel. Modelling Approach and Tool for Designing Protocols for Automated Negotiation in Multi-Agent Systems. In: Walter Van de Velde and John W. Perram, editors, *Agents Breaking Away,* volume 1038 of *Lecture Notes in Artificial Intelligence*, pages 100-115, Berlin, 1996. Springer.

[4] Chang, J.; Maxemchuck, M.: Reliable multicast protocols. In: ACM TOCS (Aug. 1984) No. 3, pp. 251-273.

[5] K. Fischer; J. P. Müller, M. Pischel. A pragmatic BDI Architecture. In: Intelligent Agents II. Agent Theories, Architectures, and Languages. (Eds. Wooldridge, Müller, Tambe). ATAL 95

[6] Fricke, S.; Albayrak, S.; Meyer, U.; Bamberg, B.; Többen, H. A Development and Test Environment for Agent-based Telematic Services. In: Intelligent Agents for Telecommunications Applications, Albayrak, S. (Ed.), IOS Press, 1998, pp. 225-251.

[7] Jacobson, I.; Christerson, M.; Jonsson, P.; Övergaard, G. Object-Oriented Software Engineering - A Use Case Driven Approach. Addison-Wesley 1996.

[8] Kendall, E. Agent Roles and Role Models: New Abstractions for MAS Analysis and Design. In: Intelligent Agents in Information and Process Management, Workshop at the 22nd German Annual Conference on Artificial Intelligence, KI-98, University Bremen, 1998, pp. 35-46

[9] Klarmann, J.; Becht, M; Muscholl, M.: Modellierung flexibler Workflows mit teilausführbaren Aktivitäten (in german). In: Proceedings of the D-CSCW'98 Workshop "Flexibilität und Kooperation in Workflow-Management-Systemen", University of Münster, Angewandte Mathematik und Informatik, Technical Report No. 18/98-I, pp. 44-55

[10] Krogh, C. The Rights of Agents. In: Intelligent Agents II. Agent Theories, Architectures, and Languages. (Eds. Wooldridge, Müller, Tambe). ATAL 95

[11] Lesser, V. R.: Reflections on the Nature of Multi-Agent Coordination and Its Implications for an Agent Architecture. In: Autonomous Agents and Multi-Agent Systems, Vol. 1, 1998, Kluwer Academic Publishers, pp. 89-111