Issues in Multiagent System Development

Mehdi Dastani, Joris Hulstijn, Frank Dignum, John-Jules Ch. Meyer
ICS, Utrecht University
{mehdi,jorish,dignum,jj}@cs.uu.nl

Abstract

Methodologies for multiagent system development should assist the developer in making decisions about those aspects of the analysis, design and implementation, that are crucial for multiagent systems, namely, social and cognitive concepts (e.g. norms and goals). In this paper, we review existing agent-oriented methodologies. We conclude that there is a big gap between the analysis and design models and the implementation. We identify some open issues for multiagent system development. We introduce our vision of a development methodology for multiagent systems, based on the OperA analysis models and the agent-oriented programming language 3APL.

1. Introduction

Development methodologies for multiagent systems have received a lot of attention recently [15]. These methodologies differ from each other in many respects. They differ on the software development phases they capture (analysis, design, implementation). Some of them focus on inter-agent aspects, while others also provide support for the internal workings of an agent. Finally, some methodologies explicitly deal with the environment, while others do not.

We are interested in developing multiagent systems for applications that are best understood in terms of social and cognitive concepts like norms, organization structures, beliefs and goals. Such applications usually include resources and services that are part of the multiagent environment. Therefore a methodology should account for the environment too. Moreover, the methodologies should provide, besides guidelines for the analysis and design phase, also guidelines for implementation phase, and explain how the design concepts can be mapped onto instructions of an available programming language. This implies that a methodology should indicate in which ways norms can be designed and implemented. For example, a general norm can be hardwired into the environment, but can also be translated into norm-abiding goals of the individual agents. This choice has important consequences for the internal design of an agent. Therefore we believe that a methodology should not only consider inter-agent, but also intra-agent aspects.

With these starting points we selected four dedicated agent-oriented software methodologies and made a review. The methodologies are Gaia [16], AAII [11], SODA [12], and Tropos [1]. These methodologies, except AAII, were selected because they are not based on existing object-oriented development methods, or methods for knowledge-based systems. AAII was selected because it does provide a model for intra-agent aspects.

Given our interest in the development of multiagent systems using social and cognitive concepts, and our concern for implementation, we identified a number of issues that are problematic for these methodologies.

- There is no agreement on how to identify and characterize *roles* in the analysis phase and *agent types* in the design phase.
- The concepts used in the methodologies, like responsibility, permission, goals and tasks do not have a *formal semantics* or *explicit formal properties*. This becomes an important issue when these concepts are implemented; implementation constructs do have exact semantics.
- There is a *gap* between the *design models* of the methodologies and the existing *implementation languages*. It is unreasonable to expect a programmer to implement the proposed complex design models. To bridge the gap, a methodology should either introduce refined design models that can be directly implemented in an available programming language, or use a dedicated agent-oriented programming language which provides constructs to implement the high-level design concepts.
- The methodologies that include an implementation phase, such as Tropos, propose an implementation language in which it is not explained how to implement reasoning about *beliefs*, reasoning about *goals* and *plans*, reasoning about *planning goals*, or reasoning about *communication*.

– It is widely recognized that an agent may enact several roles. None of the methodologies addresses the implementation of agents that need to represent and reason about *playing different roles*.
– The methodologies, with the exception of the organisational rules of [18], ignore *organizational norms* and do not explain how to specify and design them. And even in [18] implementation is not explained.
– *Open systems* are not really supported. The methodologies implicitly suppose that agents are purposely designed to enact roles in a system. But as soon as agents from the outside may enter, the analysis, design and implementation needs to treat agents as given entities.
– In the analysis, methodologies do not consider the *environmental embedding* of a system. The structure of the organization in which a system will be embedded, has a large influence on the type of organizational structure of the system, at least when it interacts with more than one person.

To overcome some of these problems, we propose an alternative multiagent methodology. The analysis phase is based on OperA [4, 6], which captures the notion of norms and organisation structure. The design phase is based on the design models of the methodologies discussed above, extended with design models of individual agent types (intra-agent design). The implementation phase is based on the 3APL programming language [9, 3, 5], which captures the intra-agent issues, social and cognitive concepts, and the multiagent environment. Note that our ideas are not specific to OperA or 3APL; they could well be implemented in a different BDI-based agent programming language, as long as it supports reasoning with and about beliefs and reasoning with and about goals.

The paper is structured as follows. In section 2 we discuss four existing methodologies and their shortcomings. In section 3 we propose an alternative multiagent system methodology. Section 4 contains conclusions and future research.

2. Existing Agent-oriented Methodologies

In this section, we study four dedicated agent-oriented software methodologies. The methodologies are compared along three dimensions: (1) the development phases that they capture (analysis, design, implementation), (2) the inter-agent and intra-agent aspects they capture, and (3) whether they capture the environment explicitly. The comparison is summarized in table 1. The selection of these comparison criteria is motivated by our focus on social and cognitive concepts, and our concern for implementation. In the discussion of each methodology we indicate the most important issues that we felt to be lacking.

	Phases			Int./Ext.		Env.
	A	D	I	Int.	Ext.	Env.
Gaia	y	y	n	n	y	n
AAII	y	n	n	y	y	n
SODA	y	y	n	n	y	y
Tropos	y	y	y	y	y	n

Table 1. Methodologies compared

2.1. Gaia

Gaia [16] comprises an analysis and design phase and explicitly refrains from including an implementation phase. Analysis is driven by a set of requirements and aims at understanding the system and its structure. It provides two models: a role model and an interaction model. The role model specifies the key roles in the system and characterizes them in terms of permissions (the right to exploit a resource) and responsibilities (functionalities). The interaction model captures the dependencies and relations between roles by means of protocol definitions. Gaia is only concerned with the society level; it does not capture the internal aspects of agent design. The design phase provides three models: the agent model, the service model, and the acquaintance model. The agent model identifies so called agent types, which are sets of roles. The service model identifies the services (or functions) associated with a role. Finally, the acquaintance model identifies the communication links between agent types. This model can be used to detect potential communication bottlenecks. The method has been extended with a model of organizational rules and organizational structure [18]. This allows the developer to specify global rules that the organization should respect or enforce. Like norms, such rules are typically formulated at a high conceptual level. Little is said about ways of implementing them. The interaction of agents with the environment is not treated separately.

Discussion Gaia does not support the implementation phase. However, the type of design choices, concepts and their relations are at least partly driven by the type of implementation language one has in mind. In Gaia it is clear that implementing the system in a procedural language that could be easily described using MetateM would fit best. However, Gaia provides very little support for building BDI agents that reason about their different responsibilities, plans and beliefs. Although aspects like *permission* and *responsibility* have a formal description, they do not have a formal semantics. Therefore it is difficult to check whether agents really implement a certain role. Especially when different roles containing several responsibilities are joined into an agent type. Although permissions seem to be norms, it is unclear how they are

actually translated to the system itself. Should the agent itself make sure that it will only perform actions it is permitted to do? Do the resources force the agent to refrain from forbidden actions? Does the agent *know* about its permissions,...? Finally, Gaia cannot support open agent systems, because it does not treat agents as given entities.

2.2. AAII

The AAII methodology proposed by David Kinny [11] makes no distinction between the analysis and design phase. The methodology generates a set of models, based on existing object-oriented models. From an external viewpoint (inter-agent), the system is decomposed into agents, which are modeled as complex objects characterized by their purpose, their responsibilities, the services they perform, the information they require and maintain, and their interaction. This leads to two models: an agent model and an interaction model. From an internal viewpoint (intra-agent) the elements required by a computational BDI architecture are modeled for each agent. This leads to a belief model, a goal model and a plan model. The development of an intra-agent model starts from the services provided by an agent and the associated events and interactions. This leads to an identification of plans. The belief model is constructed by analyzing the conditions that control the execution of actions and decompose them into basic components: beliefs. The environment is not modeled separately.

Discussion David Kinny's AAII is one of the few approaches that takes the intra agent perspective seriously. Roles can be considered as responsibilities, which can in turn be considered as sets of services. Services are activities that are not natural to decompose any further. However, because of this nice hierarchical decomposition, the potential power of the BDI concepts is not realized. It leads to goal models that are simple AND/OR graphs. Hardly any reasoning is required by the agents. The methodology is very practice-oriented which leads to graphical models, but without much semantics of the concepts. It is left to the programmer to fill in the gaps. Like Gaia, AAII does not support open agent systems. The organization of the system is almost completely hierarchical in a truly object-oriented manner. No norms or rules are specified as such.

2.3. SODA

The SODA methodology [12] has a clear distinction between analysis and design. The methodology is only concerned with the inter-agent viewpoint.

The analysis phase provides three models: the role model, the resource model, and the interaction model. The role model defines global application goals in terms of the tasks to be achieved. Tasks can be individual or social. Individual tasks are assigned to roles while social tasks are assigned to groups. A group is an abstract concept that can be analysed as a set of roles. The resource model captures the application environment and identifies the services that are available. The resource model defines abstract access modes (permission), modeling the different ways in which the services associated with a resource can be exploited by agents. The interaction model defines the interaction between roles, groups and resources in terms of protocols.

The design phase refines the abstract models from the analysis phase and provides three models: the agent model, the society model and the environment model. The agent model specifies the mapping from roles onto agent classes. An agent class is characterized by the tasks, permissions and interaction rules associated to a role. It also specifies the cardinality (the number of agents in that class), their location (fixed for static agents and variable for mobile agents) and their origin (inside or outside the system). The society model specifies a mapping from groups onto societies of agents. An agent society is characterized by the social tasks, the set of permissions, the participating social roles, and the interaction protocols. Finally, the environment model specifies a mapping from resources onto infrastructure classes. Infrastructure classes are characterized by the services, the access modes for roles and groups, and the protocols for interacting with the environment.

Discussion SODA is a very usable development methodology. The inter-agent aspect is well developed. The interaction among agents, but also the interaction between agents and the environment is taken seriously. However, SODA does not specify the design of the agents themselves. Therefore it too leaves a gap between the design and implementation of the multiagent system.

Due to the fact that SODA recognizes explicit organizational structures and rules, it becomes applicable for open agent systems. However, in SODA, agents are conceived of as pieces of software designed for one purpose only: to fulfill their roles. The assignment of agents to roles is done at design time, and remains stable. We have a much more dynamic picture of role assignment. Agents are conceived of as entities that are given. Agents may enter a society, and start acting through some API. This means that the roles or tasks of an agent may start to conflict, and that the agent must have the means to resolve such conflicts.

Although many concepts are used for the inter-agent specification, they are not formalised. Therefore it becomes difficult to check whether agents fulfilling a role comply to all the organizational rules. Another worry is that the use of procedural specifications of behavior, like standardized tasks, will bias the design. It suggests traditional imperative programming constructs. Such a simple choice is nice,

when it is enough. However, such a view may limit the potential benefits of multiagent systems, such as flexibility and robustness, because it does not take advantage of the autonomy and possible intelligence of the agents. A declarative specification in terms of goals, objectives or landmarks, would leave the manner in which it is to be achieved up to the individual agent.

2.4. Tropos

The Tropos methodology [1] distinguishes between an early and a late requirements phase, and between architectural design and detailed design. It considers both inter-agent and intra-agent issues.

The early requirements phase, which is based on the i^* organizational modeling framework [17], is concerned with understanding an application by studying its organizational setting. This phase generates two models: a strategic dependency model and a strategic rationale model. These models specify the relevant actors, their respective goals and their inter-dependencies. In particular, the strategic dependency model describes an 'agreement' between two actors: the depender and the dependee. The strategic rationale model determines through a means-ends analysis how an actor's goals (including softgoals) can actually be fulfilled through the contributions of other actors. The late requirements phase results in a list of functional and non-functional requirements for the system.

The architectural design defines the structure of a system in terms of subsystems that are interconnected through data, control and other dependencies. The detailed design defines the behavior of each component. Agent communication languages like FIPA-ACL or KQML, message transportation mechanisms, and other concepts and tools are used to specify these components. Moreover, communication protocols are used to specify communication patterns among actors, as well as constraints on the contents of the messages they exchange. Finally, the internal processes that take place within an actor are specified by plan graphs.

The implementation phase maps the models from the detailed design phase into software by means of Jack Intelligent Agents [10]. Jack extends Java with five language constructs: agents, capabilities, database relations, events, and plans. It is claimed that these constructs implement cognitive notions such as beliefs, desires, and intentions.

Discussion Of all the methodologies considered, Tropos comes closest to a complete development methodology for multiagent systems. It treats most development phases, and it treats both inter-agent and intra-agent perspectives. The use of means-end analysis to determine the goals of an agent and to determine for what goals an agent depends on other agents, leaves enough room for the agents to fill in the plans for the goals, but is specific enough to give some handles on its implementation.

We noted some drawbacks and omissions in Tropos too. First of all the models of Tropos do not have a formal semantics and therefore it is hard to specify an implementation for the design models. It also neglects the environment, and fails to notice that roles affect the access modes or permissions for executing certain actions, or for accessing resources.
Also Tropos is meant to design closed systems, in which the designer has control over the agents that enter. However, if a system would allow external agents to enter and interact, an interface between such external agents and the environment and the other agents is required. Such an interface can be thought of as a specific governor agent, as in Islander [7], a social contract, as in OperA [6], or an agent coordination context (ACC) [13] as developed in the research on coordination languages.

A final drawback is that processes internal to an agent are specified by plan graphs. It shows a bias towards a very procedural implementation of the BDI agents, which are realized by the JACK system. Although JACK agents do have representations for beliefs (database), intentions (plans) and desires (triggers) these implementations are very simple versions of the concepts used in the specification. JACK agents lack the ability to reason with their beliefs and/or about their desires and intentions. Instead, JACK agents have essentially an event-based architecture.

3. An Alternative Multiagent Methodology

In the previous section we discussed some well-known methodologies and how they handle the issues we mentioned in the introduction. In this section we show an alternative methodology that tackles most of the issues. For the analysis phase we use OperA [6], as an example of a methodology that uses organisation structures and provides for open agent systems. For the implementation phase we show how 3APL can be used, as an example of a strong BDI-based agent-oriented programming language.

3.1. The Analysis Phase: OperA + Environment

The OperA approach does not distinguish between analysis and design models. For the purpose of this paper, it is best classified as an analysis model. OperA contains three models.

The *social model* describes roles and their dependencies. Roles have objectives: the goals the organization expects an agent to fulfill when enacting that role. OperA allows for agents to have their own goals which should be combined with those of a role when enacting that role [4]. Therefore OperA caters for open agent systems. A role can be depen-

dent on another role to fulfill (part of) its objective. What is crucial for OperA, is that the form of these dependencies is determined by the organization type. In a hierarchical organization dependencies are often translated to delegation relations, whereas in markets they translate to interactions such as the Contract Net.

The *interaction model* describes the process flow of the system, in terms of scenes and transitions between scenes. This is similar to the Islander approach [7]. The scope of a role is limited to a scene. Each scene contains an abstract and declarative specification of the landmarks to be achieved during interaction. Scenes do not (have to) specify complete protocols; they specify landmarks that can be reached in many different ways. Transitions between scenes are subject to constraints, and to a temporal ordering. E.g. an agent cannot enter a 'conference presentation' scene as a presenter if its paper was never accepted.

The *normative model* contains all the different types of norms that regulate behavior in the system. We distinguish the following types of norms. (1) Norms for roles; e.g. a PC member should not review a paper submitted by another member of the research group he is working in. (2) Norms for scenes; e.g. the reviews have to be returned to the PC chairs before a certain deadline. (3) Norms on scene transitions; e.g. a delegate should pay the registration before coming to the conference.

Norms cannot be translated into a design model directly. They will be distributed over the various models of the design phase. Although normative concepts are found in most of the methodologies discussed in this paper, they are usually immediately associated with roles. They are not formulated in a general way, or associated with activities or scenes. Therefore, norms for roles already bias the design of a system. By contrast, OperA allows one to first formulate norms, and then discuss the various ways of translating them in a society.

The final model of the analysis phase, which is not included in OperA, is the *environment model*. In this model we specify the resources that are available for the agents, like databases, etc. We also specify the available services. These can be like white or yellow pages, but also the Mathematica package that supports all kinds of calculations. In the model we also specify the way the system can interact with the environment. E.g. through the use of SQL queries with a database.

3.2. The Design Phase: Four Models

The design phase builds on the models from the analysis phase to define the inter- and intra-agent architecture of the multiagent system. In this phase, we make use of the design models from the existing approaches mentioned in section 2. In particular, the approach provides four design models: the agent model, the organizational model, the interaction model, and the infrastructure model.

The Agent Model During the design phase, the roles of the social model from the analysis are refined by modifying their responsibilities, capabilities, knowledge, and permissions. For example, if the normative model from the analysis phase specifies that PC members should deliver their reviews before a deadline, then this norm can be incorporated in the specification of a PC member role by refining the goal specification of the PC member role with a deadline specification. Deciding on a specific organizational structure in the design phase may influence the specification of agent roles since in the decided organizational structure the functionalities of agent roles become concrete. For example, according to a certain organizational structure the PC chair may or may not be able to interact with the reviewers.

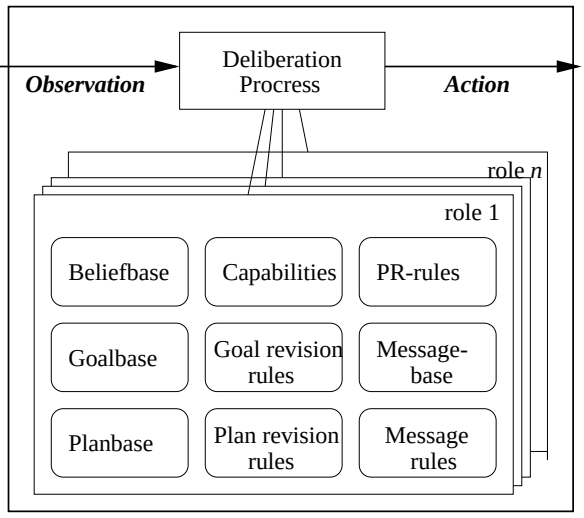

Figure 1. Agent Type Architecture

Based on the refined role models, an agent model is generated to specify the agent types that will be instantiated in the multiagent system. An agent type is generally an aggregation of refined roles that are identified at the analysis phase, together with a specification of the agent's decision making reasoning process. This reasoning scheme is also called the *deliberation process*. It indicates how an agent can reason about the contents of its mental state and about the role it plays. For example, the deliberation process could indicate if an agent should first generate all its (sub)goals before planning them, or first execute all plans before generating new plans. It may also specify when to play a certain role. The set of roles that constitute an agent type is decided by considering several issues such as efficiency, coherence, and reusability [16, 12]. The architecture of an agent type, defined by the agent model, is illustrated in figure 1. An

agent type consists of a deliberation process together with a number of roles each of which is characterized in terms of beliefs, goals, plans, capabilities, messages, and rules to reason about goals, plans, and communication. The deliberation process indicates how and in what order these entities should be processed. Details can be found in [3].

The Organizational Model The organizational model specifies the topology of a system [18]. For example, an organizational model for a certain conference may involve agent types such as PC member and PC chair and indicate that these two agent types are related. The generation of the organizational model is guided by existing organizational patterns indicating a set of agent types, resources, and services within a certain topological structure. Organizational patterns from a library can be refined by adding agent types, resources or services, by modifying its topology, or by adding constraints. In the conference example, constraints on the relation between PC chair and PC members may state that any communication should go through a secure channel. Such constraints are derived from the normative model of the analysis phase. In this way, the norms that a multiagent system should respect or enforce are incorporated in the topology of the organizational structure.

The Interaction Model The interaction model at the design phase refines the interaction model from the analysis phase by specifying the relations that are derived from the organizational model, i.e. it specifies the relations among agent types, between agent types, resources, and services, and between resources and services [16, 12]. These relations are specified in terms of interaction protocols which are in turn defined in terms of various attributes such as the goal of the interaction, the initiator of the protocol (i.e. the agent type that starts the interaction), the responder (i.e. the agent type that accepts to interact with the initiator), and the type information exchanged during the interaction. For example, the PC chair may want the papers to get reviewed, and initiates an interaction with PC members by sending papers to them. After this the PC members must respond with a rejection or an acceptance, indicating whether or not they will review the papers. Specific permissions and constraints may be added, to delimit the scope of the interaction. For example, a PC member can access the reviews of all papers except his own.

The Infrastructure Model The infrastructure model consists of a resource model, a service model, and a model of coordination facilities [16, 12, 13]. Each resource model contains a specification of the resource (e.g. document, database, or library) in terms of various qualities and quantities, access permissions, and admissible actions. The service model defines which services (e.g. any activity that processes information) can be delivered to a certain service request. The services can be defined as abstract operations in terms of input and output functionality. For each service, the model specifies the permissions needed to use the service by an agent playing a certain role. It also specifies the quality of service, such as the maximal delay in providing the service, the format of messages it can process, and the protocols it can support. Finally, coordination facilities are mechanisms that can be used by the agents to coordinate their activities. Examples are synchronization, tuple spaces, subscribe-notify design pattern, or various types of protocols.

3.3. The Implementation Phase: 3APL

The implementation phase is based on the 3APL language and environment. In this section we describe the most important features of 3APL [9, 3, 5] and the various facilities that are part of the 3APL development platform[1]. Together they suffice to implement in a direct way all aspects of the models of the design phase.

3.3.1. 3APL Platform In figure 2 an overview is given of the 3APL platform. The platform provides (access to) the facilities defined in the infrastructure and environment model of the design phase, such as communication and coordination facilities, access to knowledge sources external to an agent, a way of mediating between different agents, and an underlying architecture that supports low level programming facilities, such as arithmetic and a user interface. Many of these facilities are accessed through the *agent management system*, based on the FIPA Agent Management Specification. The platform can be used through a graphical user interface (GUI) which enables the programmer to load agents from a library, implement and execute them, and observe their behavior.

Communication Management The 3APL agent platform provides communication by means of message passing. A message will be delivered by the underlying transport layer, provided the agent management system knows the identifier of the agent being addressed. The agent can be located on a different platform running on a different machine as long as the address is recognized and unique. The messages themselves have the structure of communicative acts, with a sender, receiver and a content, which is compliant with the FIPA standards for agent communication [8].

Environment In the current 3APL platform an agent can only interact with an environment through a Java class called a *plug-in*. Consider for example a Blockworld environment, in which agents can move along a grid, perceive and avoid obstacles or other agents, pick up and drop boxes etc. For each plug-in a detailed API is used to specify which actions an agent playing a certain role can perform. These actions are interpreted as methods in the Java plug-in. In the

1 The software can be found at `http://www.cs.uu.nl/3apl/`.

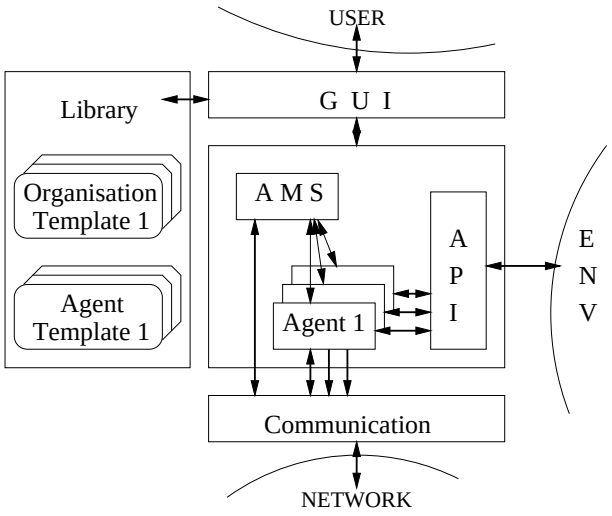

Figure 2. 3APL Development Platform

3APL source code, an external action is identified with the keyword `Java` and a label indicating the particular plug-in.

Service facilitator The platform contains a very simple service directory facility. Agents can register the services that they offer with the AMS. If they are interested in the services offered by other agents, they can query the AMS. This functionality of the agent platform may be extended in the future with more elaborate directory services (yellow pages) that allow more intelligent searching and matching.

Agent library The most important development support, is a library of software templates for common tasks and applications. In this library the templates for the facilitation agents belonging to different organizational structures can be found. E.g. templates for matchmaker agents, notary agents, etc. Thus it implements parts of the organizational model of the design phase. Typically a template will implement a particular kind of behavior that is part of an interaction pattern. A template consists of an initial beliefbase and goalbase, a set of capabilities and a set of practical reasoning rules. As such they are the implementation counterparts of the agent types.

3.3.2. Individual Agents A 3APL agent program closely follows the theoretical BDI paradigm [2, 14]. It contains an initial declarative *goalbase*, representing desired states of affairs and implemented by boolean combinations of basic formulas. The agent program also contains an initial *beliefbase* and a *planbase* representing the beliefs and intentions of the BDI models. These constructs together represent an agent's mental state throughout the execution of the program. Besides these BDI elements, a 3APL program must also specify the internal *capabilities* of the agent that are used to transform the beliefbase. The beliefbase is currently implemented using a Prolog interpreter. It allows one to reason, and match queries to beliefs. Capabilities are of the form {`Pre`}`Capability`{`Post`}, where both `Pre` and `Post` are formulas representing beliefs.

In order to reason with these mental attitudes a number of rules are introduced. These rules generally are of the form `Head` ← `Guard | Body`, where the guard is a test on the beliefbase. The content of the head and body of a rule depend on the kind of rule. *Goal revision rules* are used to reason about the dependencies among goals. In that case, both the head and the body consist of goal formulas. *Plan selection rules* are used to relate declarative goals to plan recipes. In that case, the head is a goal formula and the body is a plan. A plan may consist of capabilities or combinations of these using sequential composition, a while-loop, an if-then-else or a test. Such a test succeeds if a Prolog query to the beliefbase succeeds. Finally, *plan revision rules* are used to reason about plans by indicating how and under which belief conditions plans can be revised. By making either the guard or the body of such rules trivial (true or skip) we get a rich representation format which can express goal adoption, goal reconsideration, and re-planning. Details can be found in [5].

For some applications, an agent needs general background knowledge or skills. Therefore a program can load an external Prolog file. The agent behaves as if the clauses in the Prolog file are part of its personal beliefbase.

Deliberation Cycle The constructs in an agent program are interpreted and manipulated by a meta-level program, called the *deliberation program*. A deliberation program can, for example, be implemeted as a while loop, in which a goal is selected from the goalbase, and matched against the practical reasoning rules. If there is a rule whose guard can be satisfied against the beliefbase, this rule may be applied, which leads to an expansion of the goalbase. Alternatively, an atomic goal from the goalbase can be selected to generate a plan which can subsequently be executed. In this way various deliberation strategies may be implemented. The basic instructions of a deliberation program are, for example, *planning goal*, *plan execution*, *rule selection* and *rule application*. The instructions have been given a formal semantics [3]. Because different kinds of applications (static, dynamic) demand different kinds of agents, the deliberation cycle can be adapted.

4. Conclusion

Given a number of strong assumptions, existing multiagent software development methodologies were found to be wanting. We raised some issues for multiagent system development. Some of these issues are addressed in the OperA analysis models, and the 3APL development platform. The issues can be summarized as follows:

– allow open systems; as a consequence, specify interfaces or contracts that regulate the behaviour of external agents
– specify norms first, and then consider how to implement them: as agent's goals, in the social or interaction structure, or in the environment.
– specify exactly what you mean by a role, and by an organisation structure; different usages lead to different semantics, and therefore different implementations.
– take the use of BDI concepts seriously, by grounding them in design or in a dedicated implementation

Note that these issues do not *have* to be dealt with by OperA or 3APL. We just used these approaches as an illustration of our ideas. These issues could in fact be connected with any sufficiently rich modeling method that supports organizational structures instead of OperA, any agent programming language that really makes use of the BDI concepts, i.e. supports reasoning about goals on the basis of beliefs.

The development of the 3APL programming language and platform is an ongoing activity. In particular, we are extending the 3APL programming language to implement nested beliefs and goals which are crucial for grounded agent communication. In case of open agent systems, where agents are allowed to enter from the outside, we need to manage dynamicity. In our current research, we are extending the deliberation language with explicit programming instructions for reasoning about dynamic role changes. We are experimenting with ways to allow agents to *enact* and '*deact*' specific roles. Enactment means that an agent adds the objectives associated with the role to its own objectives. Deactment, means that an agent is again released from those objectives. Similar to Islander [7], when an agent enters a 'scene' in a particular role, not only does the agent need to start enacting the role, but also the environment or social institution needs to approve of its entering in that role. Note that this is an example of how to implement norms directly in the environment as mentioned above. Finally, the 3APL platform will be extended with more sophisticated directory services (yellow pages) and debugging facilities.

References

[1] J. Castro, M. Kolp, and J. Mylopoulos. Towards requirements-driven information systems engineering: the TROPOS project. *Information Systems*, 27:365–389, 2002.

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

[3] Mehdi Dastani, Frank de Boer, Frank Dignum, and John-Jules Meyer. Programming agent deliberation. In *Second International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS'03)*. 2003.

[4] Mehdi Dastani, Virginia Dignum, and Frank Dignum. Role-assignment in open agent societies. In *Second International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS'03)*. 2003.

[5] Mehdi Dastani, Birna van Riemsdijk, Frank Dignum, and John-Jules Meyer. A programming language for cognitive agents: Goal directed 3APL. In *First Workshop on Programming Multiagent Systems (ProMAS'03)*. 2003.

[6] Virginia Dignum. *A Model for Organizational Interaction, based onAgents, founded in Logic*. PhD thesis, University of Utrecht, 2003.

[7] M. Esteva, D. de la Cruz, and C. Sierra. ISLANDER: an electronic institutions editor. In *First Interantional Joint Conference on Autonoumous Agents and Multiagent Systems (AAMAS'02)*, pages 1045 – 1052. ACM Press, 2002.

[8] FIPA. FIPA ACL message structure specification. Technical Report XC00061, Foundation for Intelligent Physical Agents, 2001.

[9] K.V. Hindriks, F.S. de Boer, W. van der Hoek, and J.-J.Ch. Meyer. Agent programming in 3APL. *Autonomous Agents and Multi-Agent Systems*, 2(4):357–401, 1999.

[10] Nick Howden, Ralph Ronnquist, Andrew Hodgson, and Andrew Lucas. Jack summary of an agent infrastructure. In *5th International Conference on Autonomous Agents*. 2001.

[11] D. Kinny, M. Georgeff, and A. Rao. A methodology and modelling technique for systems of bdi agents. In *Modelling Autonomous Agents in a Multi-Agent World (MAAMAW-96)*, LNCS 1038. Springer Verlag, 1996.

[12] Andrea Omicini. SODA: Societies and infrastructures in the analysis and design of agent-based systems. In *AOSE*, pages 185–193, 2000.

[13] Andrea Omicini. Towards a notion of agent coordination context. In D. Marinescu and C. Lee, editors, *Process Coordination and Ubiquitous Computing*, page 187200. CRC Press, 2002.

[14] Anand S. Rao and Michael P. Georgeff. Modeling rational agents within a BDI-architecture. In J. Allen, R. Fikes, and E. Sandewall, editors, *International Workshop on Knowledge Representation (KR'91)*, pages 473–484. Morgan Kaufmann, 1991.

[15] Gerhard Weiss. Agent orientation in software engineering. *The knowledge engineering review*, 16(4):349–373, 2001.

[16] Michael J. Wooldridge, Nicholas R. Jennings, and David Kinny. The Gaia methodology for agent-oriented analysis and design. *Autonomous Agents and Multi-Agent Systems*, 3(3):285–312, 2000.

[17] E. Yu. Towards modelling and reasoning support for early-phase requirements engineering. In *Proceedings of the 3rd IEEE Interantioanl Symposium on Requirements Engineering (RE'97)*, pages 226–235, 1997.

[18] F. Zambonelli, N. Jennings, and M. Wooldridge. Organizational abstractions in the analysis and design of multi-agent systems. In *First International Workshop on Agent-Oriented Software Engineering at ICSE*. 2000.