2011 IEEE/WIC/ACM International Conferences on Web Intelligence and Intelligent Agent Technology

Programming Role Enactment through Reflection

M. Birna van Riemsdijk
TU Delft
Delft, The Netherlands
m.b.vanriemsdijk@tudelft.nl

Virginia Dignum
TU Delft
Delft, The Netherlands
m.v.dignum@tudelft.nl

Catholijn M. Jonker
TU Delft
Delft, The Netherlands
c.m.jonker@tudelft.nl

Huib Aldewereld
Utrecht University
Utrecht, The Netherlands
huib@cs.uu.nl

***Abstract*—*Organizational modeling languages* are used to specify an agent organization in terms of its roles, organizational structure, norms, etc. Agents take part in organizations by playing one or more of the specified roles. Using such an organizational specification to organize a multi-agent system can support agents' effectiveness in attaining their purpose, or prevent certain undesired behavior from occurring. In this paper, we investigate the process of role enactment in organizations that have a so-called gatekeeper that is responsible for admitting agents to the organization, like the well-known OperA organizational modelling language. We propose an interaction protocol between gatekeeper and agents that want to play roles, resulting in admittance of agents to the organization (or rejection). We analyze which kinds of reasoning are needed for agents to participate in this protocol. In particular, agents need to be able to reason about whether they have the necessary *capabilities* to play a role in an organization. We make precise what it means to have a capability and propose to integrate reasoning about capabilities in agent programming languages using *reflection*. We show how this kind of reflection about capabilities can be used to program role enactment in the GOAL agent programming language.**

***Keywords*- agent programming; organizational modelling; role enactment**

I. INTRODUCTION

An *organizational modeling language* can be used to specify an agent organization in terms of its roles, organizational structure, norms, etc. (see, e.g., [12], [15]). Such an organizational specification abstracts from the individual agents that will eventually play the roles in the organization. Using an organizational specification is a sine qua non for creating open multi-agent organizations that allow agents to join or leave the organization.

Agents who want to enter and play roles in an organization are expected to understand and reason about the organizational specification, if they are to operate effectively and flexibly in the organization. Agents that are capable of such organizational reasoning and decision making are called *organization-aware agents* [21]. Our broader aim is the development of languages and techniques for programming organization-aware agents.

An important aspect that organization-aware agents should be able to reason about is *role enactment*. In this paper we consider organizations in which a dedicated agent (a *gatekeeper*) is responsible for admitting agents to the organization. An example of an organizational modelling language in which such a gatekeeper is present, is OperA [12].

The idea is that the gatekeeper asks agents who want to join whether they have the necessary *capabilities* for playing the desired role in the organization, and assigns roles to agents on the basis of this. Although the idea of the gatekeeper has been proposed in [12], it has not been investigated which interactions should take place between gatekeeper and an agent who wants to join the organization (the *applying* agent), and what kind of reasoning is needed on the part of the applying agent to participate in this interaction.

We propose an interaction protocol between gatekeeper and agents that want to play roles and analyze which kinds of reasoning are needed for agents to participate in this protocol (Section IV). In particular, agents need to be able to reason about whether they have the necessary capabilities to play a role in an organization. We make precise what it means to have a capability and propose to integrate reasoning about capabilities in agent programming languages using *reflection* (Section V). We show how this kind of reflection about capabilities can be used to program role enactment in the GOAL agent programming language according to the developed interaction protocol (Section VI). We illustrate our framework using teamwork domain Blocks World for Teams (Section II) and introduce GOAL in Section III. These investigations contribute to the development of languages and techniques for programming organization-aware agents. We discuss related work and conclude the paper in Sections VII and VIII.

II. BLOCKS WORLD FOR TEAMS

The Blocks World For Teams (BW4T) simulated environment [16] has been developed as a testbed for human-agent/robot teamwork. The environment consists of nine rooms that are connected through halls. Colored blocks are placed inside the rooms. Simulated robots should work together to pick up blocks from the rooms, bring them to the so-called drop zone and put them down there, in the specified color sequence. Blocks only become visible once a robot enters the room where these blocks are. Robots cannot see each other but they can exchange messages. Once a robot enters a room (including the drop zone), no other robots can enter. Blocks disappear from the environment when dropped in the hall or in the drop zone. Robots can be controlled by agents or humans, thereby providing the possibility to investigate human-agent robot teamwork. Here we consider agent-only teams since human-agent interaction is not the focus of this paper.

An interface that allows a GOAL agent to control a simulated robot has been developed using the Environment Interface Standard (EIS) [2]. Broadly speaking, this standard specifies that agents can control entities in the environment through actions, and agents can observe the environment through percepts that are sent from the environment to the agents. The actions made available to agents are `goTo(<Place>)` to move to the specified place (a room, the drop zone or a hall), `goTo(<Block>)` to move to the specified block, `pickUp` to pick up a block (the robot has to be close to the block) and `putDown` to put a block down (if the robot is not holding a block, the action has no effect). Percepts made available to agents are, for example, `at(<Me>,<Place>)` which specifies in which place the robot currently is, and `color(<Block>,<Color>)` which is sent once an agent enters the room where `<Block>` is located. The color sequence in which agents should put down blocks at the drop zone is sent to agents initially as the percept `sequence([<Color>])` which has a list of colors as parameter, and the color that should be delivered next is made available to agents using the percept `sequenceIndex(<N>)`, where `<N>` is an integer referring to the N-th element in the color sequence.

We chose the BW4T domain for our studies as it is a relatively simple domain in which nevertheless many issues that arise in more complex organizations can be studied. For example, the environment has *limited visibility*; *actions take time*, which means that effective cooperation, which can prevent unnecessary actions from being executed, can significantly reduce the time needed to deliver all the required blocks; *communication* can support effective teamwork, for example by letting other agents know which blocks one has discovered and which block one is carrying.

III. AGENT PROGRAMMING

GOAL is a high-level agent programming language for programming cognitive agents, i.e., agents endowed with high-level mental attitudes such as beliefs and goals. We sketch the main programming constructs as far as they are relevant for the remainder of the paper, using the BW4T example. More details and formal semantics of GOAL can be found in [14].

GOAL agents have a mental state consisting of *knowledge*, *beliefs* and *goals*. The knowledge base represents static, general domain knowledge and the belief base represents information about the current state of the world and other agents. Goals are *declarative achievement goals*, i.e., they represent states of the world that the agent wants to achieve. Goals are automatically removed from the goal base once they are believed to be achieved. In principle several knowledge representation languages can be used for representing the mental state. In the current implementation, the knowledge base and belief base are Prolog programs and the goal base is a set of conjunctions of Prolog atoms. GOAL agents use so-called *mental state conditions* to inspect the beliefs and goals of an agent's mental state. A mental state condition **bel**(ϕ) holds if ϕ follows from the union of the knowledge and belief base. Similarly, **a-goal**(ϕ) holds if ϕ follows from the union of knowledge and one of the goals in the goal base, and ϕ does not follow from the union of belief base and knowledge base. Logical combinations of these mental state atoms can also be used.

Agents pursue their goals by executing actions. Actions are selected in GOAL by so-called *action rules* of the form **`if`** `<cond>` **`then`** `<action>` where `<cond>` is a mental state condition. For example, the action rule **`if a-goal`**`(holding(Block)),` **`bel`**`(at(Block))` **`then`** `pickUp` specifies that the agent should pick up a block if it wants to hold the block and is standing at the location of the block. These rules provide agents with the capability to react flexibly and reactively to environment changes but also allow a programmer to define more complicated strategies for pursuing goals. *External* actions correspond to actions offered by the EIS interface for a certain environment, and have an effect in this environment, while *internal* actions are only used for updating beliefs. GOAL offers built-in actions for adopting and dropping goals, inserting and deleting beliefs, and for sending messages to other agents (**`send`**`(<Receiver>,<Content>)` and **`sendonce`**`(<Receiver>,<Content>)`, the latter for sending a message only once instead of each time the condition of the rule of which it is a consequent holds). We use ρ to denote action rules and *ant*(ρ) and *cons*(ρ) to refer to the antecedent (the mental state condition) and consequent (the action) of ρ. The consequent of an action rule may consist of combinations of one internal or external action and several built-in actions.

Actions have to be specified in the *action specification* section using preconditions which have to hold in order to execute the action and postconditions which specify updates to the belief base. For example, the following gives a specification of the `goTo(Place)` action for the case where the agent is not already going somewhere. Another similar specification has to be added to represent the case where the agent is already going somewhere. The action can be executed if the agent is currently not at `Place`, and the postcondition records where the agent is going. The latter is useful to prevent continuous selection of the `goTo(Place)` action while the agent is already traveling to `Place`.

```
goTo(Place) {
  pre { not(at(Place)), not(traveling(_)) }
  post { traveling(Place) } }
```

Modules in GOAL provide a means to structure action rules into clusters of such rules and corresponding knowledge to define different strategies for different situations. Modules have a context condition which is a mental state condition that specifies when the agent can enter the module, in which case its action rules can be executed. Our BW4T agents have the following modules:

```
module searchBlock {
   context { a-goal(allRoomsChecked) }
     <action rules for checking rooms>
}

module deliverBlock {
   context { a-goal(allBlocksDelivered) }
      <action rules for delivering blocks>
}
```

A module with a goal context condition can be entered when the agent has the goal, and will be exited when the agent no longer has the goal. In the knowledge base it is specified what `allRoomsChecked` and `allBlocksDelivered` mean. For example, the latter holds when the sequence index (representing the index of the next color to be delivered) equals the length of the required color sequence plus one. We refer to the context condition of a module m as *context*(m).

Percept rules are similar to action rules and are mainly used to process percepts received from the environment and messages received from other agents. These rules allow (pre)processing of percepts and allow a programmer to flexibly decide what to do with received percepts (updating by inserting or deleting beliefs, adopting or dropping goals, or sending messages to other agents). For example, the percept rule **if bel**(percept(place(Place))) **then insert**(place(Place)) inserts beliefs about which places there are upon receipt of a corresponding percept.

IV. Interaction for Role Enactment

In this section we propose a basic *interaction protocol* between gatekeeper and applying agent (Section IV-A), and we discuss the kinds of reasoning needed for the applying agent (Section IV-B).

A. Interaction Protocol

We use the notation of [13] to distinguish different kinds of messages: the prefix "`!`" for imperative messages (requests), "`?`" for interrogative (questions), and "`:`" for declarative (information). Figure 1 shows the basic interaction protocol, where the applying agent `agt` sends a message to the gatekeeper that it wants to play a certain role, i.e., that it wants to become a *role-enacting agent* or *rea* for short [12] (`!rea(agt,Role)`). The gatekeeper replies by asking the agent whether it has the capabilities to play this role (`?cap(agt,Cap)`). It does this for each required capability

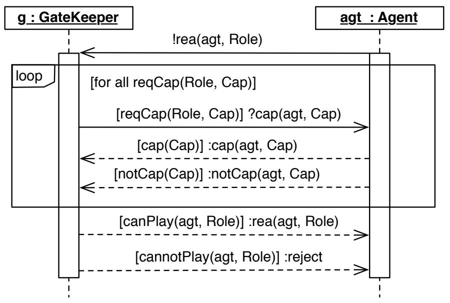
g : GateKeeper
agt : Agent
!rea(agt, Role)
loop
[for all reqCap(Role, Cap)]
[reqCap(Role, Cap)] ?cap(agt, Cap)
[cap(Cap)] :cap(agt, Cap)
[notCap(Cap)] :notCap(agt, Cap)
[canPlay(agt, Role)] :rea(agt, Role)
[cannotPlay(agt, Role)] :reject

Figure 1. Role Enactment Interaction Protocol in UML.

(`reqCap(Role, Cap)`). The agent replies by informing the gatekeeper of the capabilities it has (`:cap(agt,Cap)` and `:notCap(agt,Cap)`). If the agent has all required capabilities, it can play the role (`canPlay(agt, Role)`) and the gatekeeper informs the agent that it is now playing the role (`:rea(agt,Role)`). If the agent does not have all required capabilities, its request to play the role is rejected.

In this approach, the gatekeeper assumes the applying agent is not lying about its capabilities. Telling the truth may be enforced by imposing sanctions in case the agent fails to exhibit behavior required by its role.

The enactment protocol can be extended in various ways. In particular, the gatekeeper and the applying agent could negotiate about whether the latter is allowed to play the role even if it does not have all required capabilities. The agent might then have to do some additional things in return. Also, the gatekeeper could propose other roles that the agent would be able to play, given the capabilities that it has. Moreover, the gatekeeper may request the applying agent to perform certain tasks to demonstrate that it has required capabilities, rather than only asking the agent whether it has them. Exploring these extensions is left for future work.

B. Agent Reasoning

To take part in the interaction protocol with a gatekeeper, an applying agent has to take the decision of requesting to play a certain role in the organization. For this, the agent has to reason about whether it *wants* to play the role, given its own goals [10]. Moreover, it has to reason about whether it *can* play the role, i.e., about whether it has the required *capabilities*. In this paper we focus on the latter. Reasoning about whether the agent can play the role can be done already before the agent decides to request to play the role (such that the agent can decide not to request to play the role if it does not have the capabilities to do it), but it becomes essential when the gatekeeper asks the agent about its capabilities.

In order to develop general techniques that allow agents to answer a gatekeeper's questions about their capabilities, we need to make precise what kind of capabilities the gatekeeper can ask about. We propose to distinguish four *capability types*: capabilities to execute *actions*, to *perceive* aspects of the environment in which the agents operate, to *communicate* information, questions or requests, and to achieve *goals*. We believe this to be a suitable distinction since these four types of capabilities correspond to the commonly adopted notion of intelligent agents as being reactive (able to perceive and react to changes in the environment), proactive (act towards achieving goals) and social (communicate with other agents).

For example, in the BW4T domain we distinguish a searcher role (responsible for checking all rooms for the blocks and providing the information about block locations and colors to other agents) and a deliverer role (responsible for picking up the blocks of the correct color and dropping them at the drop zone). For the searcher role, the following are examples of capabilities: the capability to execute the action of going to a place (`ableToDo(goTo(<Place>))`), the capability to perceive blocks and their colors (`ableToPerceive(color(<Block>, <Color>))`), the capability to send information about blocks to other agents (`ableToSay(send(<Agent>, at(<Block>,<Color>,<Room>)))`), and the capability to achieve the goal of having checked all rooms (`ableToAchieve(allRoomsChecked)`).

Which and how many capabilities are required for agents to enact a role is a modeling choice and reflects the balance between regulation and autonomy demanded by the application [20]. Basically, the less demands are placed on role enactment, the more autonomy is allowed to specific role-enacting agents, but the less guarantees can be made concerning the overall organisational behaviour.

In the following sections we make precise how cognitive agents can be programmed to reason about whether they have the capabilities to play a role.

V. Reflection on Capabilities

In order to program agents that reason about whether they have the necessary capabilities, we need to make precise what it *means* that an agent has certain capabilities (Sections V-A and V-B). This is important to establish a proper grounding for the development of a mechanism that allows an agent to derive whether it has the necessary capabilities, as we do in Section V-C.

A. Meaning of Capability

Our interpretation of capability is based on the philosophical idea that "can" implies *ability* and *opportunity* [8]. We then take capability to mean the former. This implies that having a capability is a necessary but not sufficient prerequisit for exhibiting behavior that demonstrates the capability.

This interpretation of capability as ability follows [19], in which the notion of capability is investigated in the context of BDI logic. In agent programming, capabilities have been introduced as a modularization construct [6], [4] comparable to modules in GOAL. This construct encapsulates basically a set of plans and exposes an interface that expresses which goals can be achieved using these plans. That is, capability is taken as the capability to achieve goals, which is also the capability type considered in [19].

In contrast with the approaches discussed above, we propose to consider not only the capability to achieve goals but also the capability to execute actions, to receive percepts and to communicate (Section IV-B).

For this (and in line with suggestions in [19]), we interpret capability as the agent source code fulfilling "minimal" requirements that should allow the agent to use the capability. For example, considering the capability to pick up a block, if the software steering a robot never calls the gripper, one can be certain that the robot will never pick up a block. Another (weaker) interpretation is a notion of physical capability where one could for example say that a robot is capable of performing an action to pick up a block if it has a gripper. Another (stronger) interpretation is a notion of behavioral capability which refers to the execution of the agent program, where an agent is said to have the capability if there are circumstances, like the initial state of the environment and the contributions of other agents, in which the use of the capability will be demonstrated by the agent. We consider physical capability too weak a notion. Behavioral capability would require an agent to reason about its own execution in combination with the execution of other agents in the MAS and the environment in which the MAS executes, which goes beyond what is possible with current agent reasoning techniques.

B. Formalizing Capability

We start by formalizing the capability to execute actions, to perceive and to communicate as they are closely related. Then we formalize the capability to achieve goals.

1) Actions, Percepts and Communication: First, we note that actions concern *environment actions* rather than internal actions, since role specifications refer only to the former. Internal actions are implementation details from the perspective of the organization, and their execution is not observable from outside the agent. In order to make precise what we mean by having an ability, we introduce Act^{env} and $\mathsf{Percepts}^{env}$ as the set of actions and percepts processable by the sensors as offered by the (physical representative of the agent in the) environment, such as a simulated robot as in the case of the BW4T domain.[1] Actions in Act^{env} and percepts in $\mathsf{Percepts}^{env}$ are of the form $a(x_0, \ldots, x_n)$ and $p(x_0, \ldots, x_n)$, respectively, where a and p are the action

[1]We assume that all agents are connected to physical representatives offering the same actions and percepts. This framework could be extended by having different physical capabilities for the different kinds of physical representatives, and making explicit which agent is connected to which representative. We omit this here for reasons of simplicity.

and percept names, and $x_0, \ldots, x_n$ are variables which can be instantiated with ground terms at run-time.[2]

In order to formalize capability, we need to define the minimal source code requirements without which the agent can never use the capability.

A minimal source code requirement for using a capability is that it "occurs" in the source code. For actions this means that they occur in the consequent of *action rules* $\mathsf{ActRules}_i$ of a GOAL agent i. The GOAL syntax requires that these actions are also included in the action specification section $\mathsf{ActSpec}_i$, i.e., occurrence in action rules implies occurrence in the action specification. For the processing of percepts it is necessary that corresponding *percept rules* $\mathsf{PerceptRules}_i$ exist. For communication of certain information, requests or questions the source code needs to contain action rules or percept rules that have these kinds of communication in their consequent.

To make this precise, we introduce the following definitions. Let $\mathit{varx}(e)$ be the expression e in which variables are substituted by the variables $x_0, x_1, \ldots$ in order of appearance, i.e., the first variable is substituted by x_0, the second by x_1, etc. This is introduced for technical convenience, to ensure that we do not distinguish, e.g., actions with the same action name but with different variable names. We use $\alpha \in \mathit{cons}(\rho)$ to denote that action α occurs in the consequent of rule ρ, and $\phi \in \mathit{ant}(\rho)$ to denote that mental state formula ϕ occurs in the antecedent of ρ. Let $\mathsf{Act}_i = \{\mathit{varx}(\alpha) \mid \exists \rho \in \mathsf{ActRules}_i : \alpha \in \mathit{cons}(\rho)\} \cap \{\mathit{varx}(\alpha) \mid \alpha \in \mathsf{ActSpec}_i\}$ be the set of (internal and environment) actions of agent i that occur in the action rules.[3] Then we define the set of environment actions occurring in the source code of GOAL agent i as $\mathsf{Act}_i^{\mathit{env}} = \mathsf{Act}^{\mathit{env}} \cap \mathsf{Act}_i$. Let $\mathsf{Percepts}_i = \{\mathit{varx}(p) \mid \exists \rho \in \mathsf{PerceptRules}_i : \mathbf{bel}(\mathit{percept}(p)) \in \mathit{ant}(\rho)\}$ be the set of percepts occurring in the antecedent of percept rules of agent i. Since percepts always come from the environment, there exist no "internal" percepts in analogy of internal actions, so $\mathsf{Percepts}_i \subseteq \mathsf{Percepts}^{\mathit{env}}$ should hold. However, this is not enforced by the GOAL language, which means that the programmer could in principle write percept rules for percepts that are never generated by the environment. We do not want to include those in the percepts an agent is able to perceive, since the corresponding rules will never fire because their percepts will never be received. Therefore we define the set of percepts that a GOAL agent i is able to perceive as $\mathsf{Percepts}_i^{\mathit{env}} = \mathsf{Percepts}^{\mathit{env}} \cap \mathsf{Percepts}_i$. We define the set of communication actions in the source code of a GOAL agent as $\mathsf{Comms}_i = \{\mathit{varx}(c) \mid \exists \rho \in \mathsf{ActRules}_i \cup \mathsf{PerceptRules}_i : c \in \mathit{cons}(\rho)\}$, where c is a built-in send action.

[2]One could extend the framework by making the domain of the variables precise, for example, to allow to specify that only blocks can be picked up and no other elements of the environment. We omit this for reasons of simplicity.

[3]We take the intersection of actions in action rules and in the action specification to extract internal and environment actions from all actions that may occur in action rules, such as communication and goal modification actions.

The capability to execute environment actions, to process percepts, and to communicate, are then defined, respectively, as follows.

Definition 1: (capability)

$$\begin{array}{rcl} \mathit{ableToDo}_i(\alpha) & \Leftrightarrow & \alpha \in \mathsf{Act}_i^{\mathit{env}} \\ \mathit{ableToPerceive}_i(p) & \Leftrightarrow & p \in \mathsf{Percepts}_i^{\mathit{env}} \\ \mathit{ableToSay}_i(c) & \Leftrightarrow & p \in \mathsf{Comms}_i \end{array}$$

2) Goals: Defining what it means to be able to achieve goals depends to some extent on what notion of goal one uses. In this paper, we consider a common notion of goals, namely declarative achievement goals which express a state that agents want to reach.

We see two ways of defining the capability to achieve goals. Since goals express desired states that are to be reached, one could say that an agent is able to achieve a goal ϕ if the source code enables the agent to come to *believe that ϕ is the case*. This reflects the usual notion of goal achievement as also built into the semantics of GOAL. Defining this for GOAL involves extracting from the source code the kinds of formulas that may be added to the belief base as a result of action execution, explicit insertion using the predefined **insert** action, and sending messages to oneself (which also inserts the content of the message into the belief base).[4]

A second way of defining the capability of achieving goal ϕ corresponds to the interpretation proposed in [19]: "We understand having a capability (for) ϕ as meaning that the agent has at least one plan that has as its trigger the goal ϕ." The capability to achieve a goal is thus understood as the capability to *proactively* pursue the goal once it is adopted (using the plans that have been defined for them), rather than as the capability to come to believe, possibly "by accident", that the goal has been achieved. We argue that for organization-aware agents it is important that they can proactively pursue goals in order to be able to act towards achieving the objectives of the roles they are playing. Consequently, we follow the interpretation of [19] and define that an agent has the capability to achieve a goal if it occurs in the agent program.[5]

In GOAL, this interpretation of capability to achieve a goal ϕ translates to the existance of goal conditions in action rules and context conditions of modules corresponding to ϕ. Let $\mathsf{Goals}_i = \{\mathit{varx}(\phi) \mid \exists \rho \in \mathsf{ActRules}_i : \textbf{a-goal}(\phi) \in \mathit{ant}(\rho)$ or $\exists m \in \mathsf{Modules}_i : \textbf{a-goal}(\phi) \in \mathit{context}(m)\}$ be

[4]Note that in GOAL, percepts are not inserted directly into the belief base but the programmer has to define percept rules that explicitly do this.

[5]Typically, one would expect that if a goal occurs in an agent program, the source code enables the agent to come to believe that the goal is achieved. One could argue that if this does not hold, the programmer did not make proper use of goals.

the set of goals occurring in the antecedent of action rules or in context conditions of modules of agent i. Then the capability to achieve a goal is defined as follows.

Definition 2: (capability)

$$\mathit{ableToAchieve}_i(\phi) \Leftrightarrow \phi \in \mathsf{Goals}_i$$

C. Derivation of Beliefs about Capabilities

In Section V-B, we have defined what it means that an agent has certain capabilities. These definitions, however, do not specify how an agent should determine whether it has these capabilities, i.e., they do not specify how an agent can *reflect on its own capabilities*.

Reflection is not uncommon in programming languages (see, e.g., Java and Maude [7]). It allows a program to refer to itself at run-time, which facilitates a modification of its run-time behavior based on these reflections. In Java one can use reflection to determine at run-time, for example, which methods are contained in a class (of a certain object), whether the class is public or abstract, and what its super-class is. Also, the agent modelling language DESIRE [5] offers functionality to reason at an arbitrary set of meta-levels in which epistemic information about level n is reflected upwards to level $n+1$ and vice versa. However, DESIRE is a modelling language and not an agent programming language, and meta-level reasoning in that framework has not focused on reasoning about capabilities for the purpose of role enactment.

In Java, reflection is incorporated in the language by means of the `Class` class. One can invoke the method `getClass()` on an object, which returns a `Class` object. On the latter one can then invoke methods like `getSuperclass()` to get its super-class. When incorporating reflection into agent programming languages, it should be integrated into the language's existing agent-oriented constructs. One approach is to add actions to the language by means of which the programmer can obtain information about the agent's program. An example of this is a construct in Jason [3] for inspecting an agent's plans. In this paper, we propose the use of *beliefs* for expressing reflection in cognitive agent programming languages. An agent might, e.g., believe that it has the capability of executing a certain action. Using beliefs, we obtain a single basic mechanism for an agent's reflective capabilities.

We specify that a GOAL agent should be able to derive the following beliefs about capabilities. In Section VI we address how this can be implemented in GOAL and how it can be used for programming role enactment.
Let $\forall : e$ represent the universal quantification over all variables in the expression e.

$$\begin{array}{lll} \mathit{ableToDo}_i(\alpha) & \Rightarrow & \forall \mathbf{bel}_i(\mathit{ableToDo}(\alpha)) \\ \mathit{ableToPerceive}_i(p) & \Rightarrow & \forall \mathbf{bel}_i(\mathit{ableToPerceive}(p)) \\ \mathit{ableToSay}_i(c) & \Rightarrow & \forall \mathbf{bel}_i(\mathit{ableToSay}(c)) \\ \mathit{ableToAchieve}_i(\phi) & \Rightarrow & \forall \mathbf{bel}_i(\mathit{ableToAchieve}(\phi)) \end{array}$$

The framework could be extended to allow the derivation of capabilities to achieve logical combinations of goals from the capability to achieve basic goals, but we omit this for reasons of space.

In the logic of [19] the interplay between beliefs and capabilities is also investigated. Their basic logical system has an axiom that expresses that having a capability implies believing to have the capability. This corresponds to the above specification. A variant of the logic where the bi-implication is an axiom is mentioned, which has the effect of collapsing mixed nested modalities to their simplest form. For agent programming we cannot guarantee the bi-implication without imposing further constraints on the agent program, since actions could add false beliefs about capabilities. Such constraints would disallow the use of actions that result in the addition of these false beliefs.

VI. PROGRAMMING ROLE ENACTMENT

The above lays the foundations for allowing GOAL agents to reflect on their capabilities. In this section, we propose *programming patterns* that show how reflection can be used in a GOAL program for programming role enactment, according to the interaction protocol proposed in Section IV.

The first step in the protocol is to send a message to the gatekeeper that the agent wants to play a certain role. The condition for the corresponding action rule is thus that the agent has the goal of playing the role. The notation `imp` for imperative is used in GOAL in place of `!`.

```
if bel(me(Agt)), a-goal(rea(Agt,Role))
      then sendonce(gatekeeper, imp(rea(Agt,Role))).
```

Then the gatekeeper asks the agent whether it has the necessary capabilities to play the role. The agent answers according to the protocol that it has or does not have these capabilities. This is where the agent needs reflection on its own capabilities. We represent an agent's capabilities in the belief base as `cap(<Cap>)` to allow this reflection, in accordance with the specification in Section V-C. We wrap the capabilities in the predicate `cap` to allow the specification of rules for any type of capability. To make it easier for programmers to use reflection about capabilities, a corresponding *program transformation* could add these beliefs automatically. Alternatively, an extension of GOAL could adapt the semantics of beliefs such that these beliefs can be derived without adding them to the belief base explicitly. The latter would be more flexible as it would also support the agent acquiring new capabilities at run-time, e.g., through exchange of plans.

The following action rule is used to reply that the agent has the necessary capability. A rule for replying that the agent does not have the capability can be programmed analogously. The notation `int` for interrogative is used in GOAL in place of `?`.

```
if bel(me(Agt), received(gatekeeper, int(cap(Agt,Cap))),
      cap(Cap))
   then sendonce(gatekeeper,cap(Agt,Cap)).
```

If the agent has all necessary capabilities to play the role, the gatekeeper sends a message to the agent that it is now playing the role. This message can be handled by inserting its content into the belief base of the agent, to record which roles it is playing.

```
if bel(received(gatekeeper, rea(Agt,Role)))
   then insert(rea(Agt,Role)).
```

Once a role has been enacted, the organization expects the agent to pursue the role's objectives. This may involve reasoning about conflicts between the agent's own goals and the role's objectives, reasoning about prioritization of goals, etc. (cf. [10]). In future work we will investigate how this kind of reasoning can be programmed in GOAL.

VII. Related Work

Our interaction protocol is similar to the protocol for role enactment proposed in [1]. The main difference is that that protocol assumes that the applying agent rejects to play the role if it does not have the required capabilities. In our case, the applying agent informs the gatekeeper about its capabilities and the decision to accept or reject is made by the gatekeeper. Our approach is easier to extend to a setting where applying agent and gatekeeper negotiate about the conditions for role enactment if the agent does not have all required capabilities: as the gatekeeper takes the final decision on whether to let the agent play the role, it can also modify the conditions under which it lets the agent play the role. It does not seem natural to let the applying agent decide this. Moreover, if the applying agent tells the gatekeeper which capabilities it has, the gatekeeper might also propose the agent to play a different role that better matches its capabilities.

In [10], [11] role enactment is also studied in the context of agent programming languages. These approaches focus on defining the result of role enactment (e.g., the adoption of objectives of the role as goals) rather than on the role enactment process itself. In [10], compatibility between the goals of the agent and those of the role is investigated and taken as a prerequisit for enacting the role. These approaches do not address how an agent can reason about whether it has the required capabilities to play a role.

Our notion of capability is different from [6], [4] in that it does not introduce capability as a construct in the agent programming language, primaly used for modularization. Rather, our notion of capability is a derived notion defined on the basis of an agent's source code. This notion of capability considers not only the capability to achieve goals (as done in [6], [4], [19]) but also the capability to execute actions, to receive percepts and to communicate. It differs from [19] in that the latter investigates the notion of capability in the context of BDI logic, while we make precise what these capabilities mean in the context of cognitive agent programming.

Comparing reflection as we use it with reflection in Java, we can see similarities. In both cases, reflection is used to inspect a program's structure. A difference is that this inspection is relatively direct in the case of Java (e.g., getting the methods of a class), while our definitions for reflection in agent programming are somewhat more involved due to the intuitive meaning of capability that we aim to express through reflection. For example, we define the cability to execute an action as the agent having a corresponding definition in the action specification as well as having an action rule that has this action as its consequent. Moreover, the agent should have the physical ability to execute this action. An interpretation of reflection that would be closer to the way it is used in Java, is to introduce mechanisms that allow direct reference to the agent's source code. For example, allowing to retrieve all actions for which the agent has action specifications, or all action rules that have a certain action in their consequent. While we consider this an interesting direction to explore, it would "only" provide the means to inspect an agent's source code, but would not give guidance with respect to appropriate notions of capability.

Our formalization of the notion of capability and the patterns for programming role enactment are developed for the language GOAL. However, since other cognitive agent programming languages (e.g., [3], [9]) are built on similar basic notions as GOAL like beliefs, goals and action selection rules, we expect that similar definitions can be given for other languages. In particular, since the four capability types that we distinguish are general and correspond to the commonly adopted notion of intelligent agents (see Section IV-B), it can be expected that these notions can naturally be transferred to other languages. Moreover, other cognitive agent programming languages all have a notion of belief, and so the idea of incorporating reflection using beliefs can be transferred to other languages.

Finally, we note that reflection on capabilities can be useful in other situations than role enactment, for example when one agent asks another agent to do something. The latter can reflect on its capabilities and reply whether it would be able to do it. To the best of our knowledge, a general framework for reflection on capabilities as we propose here has not been introduced in such other contexts.

VIII. Conclusion and Future Work

If we want to design agents that are able to join different organisations on behalf of their owners, then these agents should be capable of reasoning about role enactment. In this paper we have proposed an interaction protocol between the gatekeeper (responsible for admitting agents to organizations), and agents that want to play roles. We have analyzed which kinds of reasoning are needed for agents to participate in this protocol: in particular reasoning about whether they have the necessary capabilities to play a role in an organization. We have made precise what it means to have a capability and we have proposed the use of beliefs for reflection on capabilities. We have proposed programming patterns to show how this kind of reflection about capabilities can be used to program role enactment in the GOAL agent programming language according to the developed interaction protocol.

In future work, it would be interesting to investigate the relation between an approach that uses a gatekeeper for matching agents and roles, and work on semantic matchmaking in service-oriented systems (e.g., [17]). The latter facilitates a more semantic definition of capability and the application of existing flexible matchmaking algorithms.

we aim to investigate how other kinds of organizational reasoning such as reasoning about whether the agent wants to play a role, and reasoning about how to comply with norms governing agent behavior, can be programmed. In [18] an approach for the latter is proposed in the context of AgentSpeak(L). It will have to be investigated to what extent norms of organizational modelling languages such as OperA can be reasoned about using this mechanism. Moreover, we believe it is interesting to investigate the balance between organizational regulation and agent autonomy, and what the implications are for agent reasoning. For example, if the agent program contains elaborate reasoning strategies on how to solve a particular teamwork problem, this may conflict with the strategies imposed by the organization if these are too regulative. It needs to be investigated to what extent the agent can adapt to the strategies imposed by the organization.

In general, we aim to investigate to what extent existing agent programming languages support organizational reasoning and decision making, and which general reasoning mechanisms should be supported. We conjecture that the more advanced the reasoning, the more reflective mechanisms such as the one proposed in this paper are needed.

REFERENCES

[1] M. Baldoni, G. Boella, V. Genovese, R. Grenna, and L. van der Torre. How to program organizations and roles in the JADE framework. In *MATES'08*, volume 5244 of *LNCS*, pages 25–36. Springer, 2008.

[2] T. Behrens, K. Hindriks, J. Dix, M. Dastani, R. Bordini, J. Hübner, L. Braubach, and A. Pokahr. An interface for agent-environment interaction. In *ProMAS'10*, 2010. To appear in LNAI.

[3] R. H. Bordini, J. F. Hübner, and M. Wooldridge. *Programming Multi-agent Systems in AgentSpeak using Jason*. Wiley, 2007.

[4] L. Braubach, A. Pokahr, and W. Lamersdorf. Extending the capability concept for flexible BDI agent modularization. In *ProMAS'05*, volume 3862 of *LNCS*, pages 139–155. Springer, 2006.

[5] F. Brazier, C. M. Jonker, J. Treur, and N. Wijngaards. Deliberative evolution in multi-agent systems. *International Journal of Software Engineering and Knowledge Engineering*, 11:1–23, 2001.

[6] P. Busetta, N. Howden, R. Rönnquist, and A. Hodgson. Structuring BDI agents in functional clusters. In *ATAL'99*, pages 277–289, London, UK, 2000. Springer-Verlag.

[7] M. Clavel and J. Meseguer. Reflection and strategies in rewriting logic. *Electronic Notes in Theoretical Computer Science*, 4:125–147, 1996.

[8] C. Cross. 'can' and the logic of ability. *Philosophical Studies*, 50:53–64, 1986.

[9] M. Dastani. 2APL: a practical agent programming language. *Autonomous Agents and Multi-Agent Systems*, 16(3):214–248, 2008.

[10] M. Dastani, V. Dignum, and F. Dignum. Role-assignment in open agent societies. In *AAMAS'03*, pages 489–496, Melbourne, 2003.

[11] M. Dastani, M. B. van Riemsdijk, J. Hulstijn, F. Dignum, and J.-J. Ch. Meyer. Enacting and deacting roles in agent programming. In *AOSE'04*, volume 3382 of *LNCS*, pages 189–204. Springer-Verlag, 2005.

[12] V. Dignum. *A Model for Organizational Interaction: based on Agents, founded in Logic*. SIKS Dissertation Series 2004-1. Utrecht University, 2004. PhD Thesis.

[13] K. Hindriks and M. B. van Riemsdijk. A computational semantics for communicating rational agents based on mental models. In *ProMAS'09*, volume 5919 of *LNAI*, pages 31–48. Springer, 2010.

[14] K. V. Hindriks. Programming rational agents in GOAL. In R. H. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni, editors, *Multi-Agent Programming: Languages, Tools and Applications*. Springer, Berlin, 2009.

[15] J. F. Hübner, J. S. Sichman, and O. Boissier. Developing organised multiagent systems using the MOISE+ model: programming issues at the system and agent levels. *International Journal of AOSE*, 1(3/4):370–395, 2007.

[16] M. Johnson, C. M. Jonker, M. B. van Riemsdijk, P. J. Feltovich, and J. M. Bradshaw. Joint activity testbed: Blocks world for teams (BW4T). In *ESAW'09*, volume 5881 of *LNAI*, pages 254–256. Springer, 2009.

[17] L. Li and I. Horrocks. A software framework for matchmaking based on semantic web technology. In *WWW'03: Proceedings of the 12th international conference on World Wide Web*, pages 331–339. ACM Press, 2003.

[18] F. Meneguzzi and M. Luck. Norm-based behaviour modification in BDI agents. In *AAMAS'09*, pages 177–184, Budapest, 2009.

[19] L. Padgham and P. Lambrix. Formalisations of capabilities for BDI-agents. *Autonomous Agents and Multi-Agent Systems*, 10(3):249–271, 2005.

[20] L. Penserini, V. Dignum, A. Staikopoulos, H. Aldewereld, and F. Dignum. Balancing organizational regulation and agent autonomy: An MDE-based approach. In *ESAW'09*, pages 197–212. Springer-Verlag, 2009.

[21] M. B. van Riemsdijk, K. V. Hindriks, and C. M. Jonker. Programming organization-aware agents: A research agenda. In *ESAW'09*, volume 5881 of *LNAI*, pages 98–112. Springer, 2009.