Extending the Capability Concept for Flexible BDI Agent Modularization

Lars Braubach, Alexander Pokahr, Winfried Lamersdorf

Distributed Systems and Information Systems
Computer Science Department, University of Hamburg
Vogt-Kölln-Str. 30, 22527 Hamburg, Germany
{braubach | pokahr | lamersd}@informatik.uni-hamburg.de

Abstract. Multi-agent systems are a natural way of decomposing complex systems into more manageable and decentralized units. Nevertheless, as single agents can represent complex subsystems themselves, software engineering principles for the design and implementation of coherent parts of single agents are necessary for producing modular and reusable software artifacts. This paper picks up the formerly proposed capability concept for structuring BDI agents in functional clusters, and generalizes and extends it to support a higher degree of reusability. The resulting mechanism allows for designing and implementing BDI agents as a composition of configurable agent modules (capabilities). It is based on a black-box approach with export interfaces that is in line with object-oriented engineering principles.

1 Introduction

One important traditional software-engineering principle is *modularization* [12], which means that functionality is packaged into delimited units. Thereby, referring to [6] a module is seen as "[...] a well-defined component of a software system that provides a set of services to other modules. Services are computational elements that other modules may use."

For example, in the imperative paradigm modules represent collections of procedures, data types and constants from which only a small subset is made accessible through the module's export interface. For other paradigms such as functional programming or object-orientation adapted forms of modularization have been developed as well. Generally, modularization achieves inter alia the following three advantages: First and most importantly, it enables reuse and extensibility of software artifacts as modules form separate units of functionality. Secondly, modules enhance flexibility through encapsulation, because changes inside a module should not affect other modules. Thirdly, modularization increases the effectiveness of software development and the comprehensibility of the applications as separate modules represent abstractions that can be considered independently for understanding cutouts of the system.

To achieve those advantages in practice, fundamental design principles have to be taken into account for module creation. On the one hand the *coupling*

of different modules (interrelationships) should be minimized, whereas on the other hand the *cohesion* of the elements contained in a module should be maximized [17]. The basic idea of modules is to abstract from implementation details through *information hiding* [12], which means that the internals of a module are encapsulated and can therefore be changed without affecting other modules. Adhering to these principles ensures that modules represent self-contained, reusable entities for some well-defined functionality.

For the agent paradigm modularization is also an important topic. Even though multi-agent systems are a natural technique for decomposing complex scenarios into autonomous actors, the resulting agents can still be fairly complex. Breaking down such complex agents into teams of smaller ones is not always an appropriate solution, because splitting up a self-contained entity requires a connection between those smaller agents to be established at the communication level, leading to possibly inefficient solutions.

Hence, specifically adapted concepts for structuring the internals of an agent are necessary. Such a structuring technique depends in turn on the considered agent architecture determining the high-level elements and processes which may or may not be suitable for modularization. Cognitive architectures such as 3APL[4], BDI [15] or SOAR [11] propose different high-level abstractions leading to possibly different modularization approaches. In this paper Busetta et al.'s capability concept [3] for modularization of BDI agents is taken up and extended to support more flexible agent configuration.

In the next section the original capability concept is shortly sketched and its limitations are described. In section 3 the extended capability concept is proposed and its implementation within the Jadex BDI reasoning engine is described in section 4. The concepts are further clarified by an example application in section 5. A summary and an outlook of future work conclude the paper.

2 Capabilities Revisited

The capability concept for structuring BDI agents in modules and its implementation within the JACK agent framework [8] was first described in [3]. The main idea of the original proposal is to define beliefs, goals and plans that functionally belong together in a common namespace called capability. In addition, scoping rules allow for an exact specification of the parts of a capability that should be visible from the outside. According to [3] a capability is defined as:

1. an identifier (that is, a name);
2. a set of plans;
3. a fragment of the knowledge base concerning the beliefs manipulated by the plans of the capability;
4. the visibility rules for those beliefs, that is, which ones are restricted to the plans of the capability and which ones can be seen from the outside;
5. which types of events, generated as a consequence of the activity of the capability are visible outside its scope, and their processing algorithm;

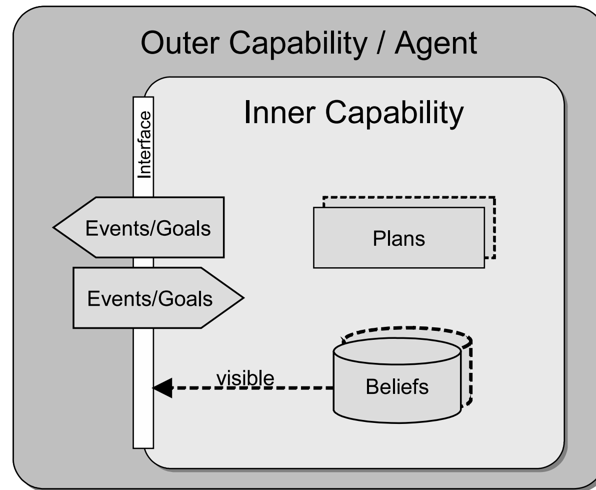

Fig. 1. Original capability concept

6. which types of events, generated outside the capability, are relevant to the capability (that is, to one or more of its plans);
7. finally, recursive inclusion of other capabilities.

Each capability type specified in this way can be included in another capability or in the agent. For this inclusion an additional symbolic name has to be provided to allow multiple usages of one capability type within the same context. This is similar to the usage of a class in an object-oriented language, i.e. the symbolic name identifies a specific instance. In Fig. 1 the main ideas of the original capability concept are also illustrated graphically. The interface of the inner capability is defined by means of the (different kinds of) external events the inner capability can process and also by those exported events that could be handled in the outside capability. In addition beliefs of the inner capability can be made visible for the outer capability, while plans are only visible locally.

As an example, in [3] negotiation functionalities for two types of agents participating in a negotiation are described (initiator and bidder). These functionalities can be encapsulated in two capabilities and reused by all agents which like to take part in a negotiation.

2.1 Limitations

The original capability concept as outlined above allows for grouping mental attitudes according to their functional purpose and therefore is an effective technique for modularization. Nevertheless, this approach exposes some conceptual limitations that are discussed next:

– Concerning the export interface, the approach distinguishes explicitly between mental attitudes and treats them in a different manner. This means

that there is no continuous mechanism for all types of elements. So, for events the propagation (from the outside/to the outside) is relevant, for beliefs the visibility can be defined (local vs. external) and for plans only their usage can be declared. Having specific means for each of the elements to be grouped inside a capability not only renders the mechanism hard to learn and use, it is also not easily possible to adapt the reusability mechanism to other mental notions, be it extensions to the BDI model or alternative mental models.
- Another important limitation of the approach is concerned with parametrization as only the static structure is considered. Besides the static structure the *initial mental state* of a capability respective an agent is of major importance. In the current form, capabilities cannot be configured with some initial mental state, which hinders flexible reuse.
- Only design-time composition has been taken into account. No work has been done so far regarding the possibilities of dynamic agent behavior modification by adding/removing or exchanging capabilities at runtime. In this respect a model would have to be provided, how the addition or removal of a capability influences the functionality of other capabilities.
- The concept does not allow for refinement of parts of a capability specification. E.g. it is not possible to provide an extended context condition for a plan. Elements have to be used in exactly the way they are defined in the inner capability, which hinders flexible reuse.

3 Extending the Notion of Capabilities

This section presents a capability concept, suitable to address the aforementioned shortcomings. It follows the general idea of a capability being "[...] a cluster of plans, beliefs, events and scoping rules over them" [3], but differs from the original capability concept in several important ways.

1. The *locality principle* assures that all elements of a capability are part of exactly one capability.
2. A general *import/export mechanism* is introduced to define which elements are part of the capability's interface and are visible from the outside. Elements contained in the interface can be used in the containing capability by defining local proxy elements.
3. A *creation semantics* determines which element instances of a static structure (composed of a single concrete element type and arbitrarily many proxies) are created at runtime.
4. The explicit specification of *initial configurations* separately to the static structure of a capability is supported.
5. The foundations of *dynamic capability modifications* are layed down.

These extensions are discussed in the following sections.

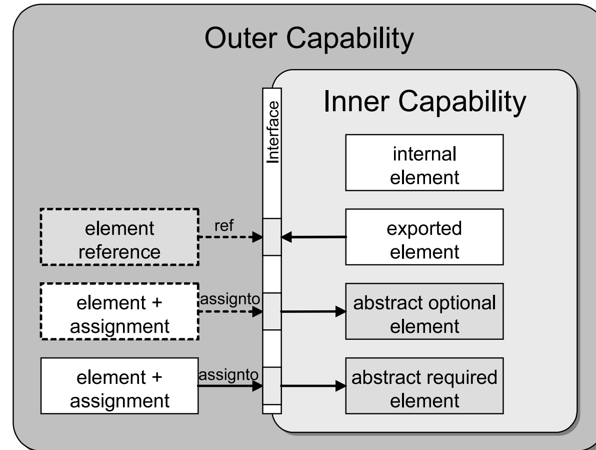

Fig. 2. Reference concept

3.1 Locality Principle

The original capability concept assumes a global repository of mental elements (i.e. event or plan type specifications) which are then just referenced in a capability definition. Therefore the same type of element can be used in different capabilities or even agents (i.e. for sending and receiving message events).

In our model we follow the locality principle of elements which means that a capability itself defines the available types (e.g. of beliefs), and forms the namespace for these types. These element types are not globally available, but only inside the capability. As a consequence e.g. from within plans only locally defined beliefs are accessible, which means that it is not sufficient to know that a belief of a contained capability is exported. To be able to use such a belief a local reference has to be declared.

Following the locality principle has the main advantage of increasing the *transparency*. This is because e.g. a plan only depends on local elements and not on elements defined in a subcapability. Changes with respect to a subcapability are therefore hidden from the plan. Another advantage concerns the *openness* of agent applications. As no global elements such as message events are specified, each agent can interpret a received message in its own respect, not depending on message representation details of other agents.

3.2 Import/Export Mechanism

The locality principle requires a newly designed import/export concept. For usability, all mental notions and their interrelations across capability borders should be defined using the same mechanism. The main idea is to cleanly distinguish relationships between different mental elements (such as a certain goal being handled by a certain plan) from the import and export specifications that relate elements from different capabilities. Relationships between mental

elements follow the locality principle, and therefore are only allowed inside a capability. Import and export specifications permit a single logical element to be present in several connected capabilities using a proxy model (see Fig. 2). The figure shows how a capability (*outer*) can reuse functionality from another (*inner*) capability. Concrete elements (which may be beliefs, goals, etc.) are presented as white rectangles. Proxy elements (i.e. placeholders for beliefs or goals of another capability) are shown as grey rectangles.

Per default, any concrete element is *internal*, meaning that it is only visible locally in its enclosing capability. E.g. internal beliefs can only be accessed by plans of the same capability. If an element should be accessible to the outer capability it can be marked as *exported*. To be used from the outside, a placeholder (called *reference*) for the original element has to be defined in the outer capability. The reference element specification includes the relative name of the inner element it refers to (e.g. in the form "subcapability.elementname"). Note that references to exported inner elements are optional. An outer capability is only required to provide a reference, when the element is accessed from other elements of the outer capability. For example when a plan of the outer capability wants to access a belief defined in the inner capability, the outer capability defines a belief reference, which acts as a proxy for the inner belief. The plan accesses the proxy like it would access a locally defined belief. Therefore, it is transparent to the plan that the belief is actually defined in the inner capability.

A capability may also include *abstract* elements, i.e. proxies which are not assigned to any concrete element by themselves. An abstract *optional* element is an element, which does not require an assignment, what means that the functionality of the capability can also be used, when this element is not present. For example, an abstract belief provides an extension point, where the outer capability can add knowledge to the inner capability. A plan in the inner capability can then check if the belief is available and proceed in different ways according to the information from the belief. An abstract *required* element is an element, which is required for proper operation of the capability.

Both required and optional abstract elements are assigned from the outside, by adding "assignto" specifications to concrete elements of the outer capability. An outer element may be assigned to many different abstract elements, but an abstract element must be assigned from (at most) a single concrete element. In addition, it is also possible to define proxies for proxies, i.e. to define a reference to an element which is itself a reference, or to assign an abstract element to another abstract element of a child capability. This allows building up reference structures through multiple levels of capabilities. E.g. in the figure, the outer capability might choose to re-export elements referenced from the inner capability.

3.3 Creation Semantics

The proxy concept introduced in the last section shows how an element can be visible in a different capability. The definition of proxies only specifies a static structure of references for element visibility. It depends on the creation context, if an instance of a proxy is created for a specific element instance. It

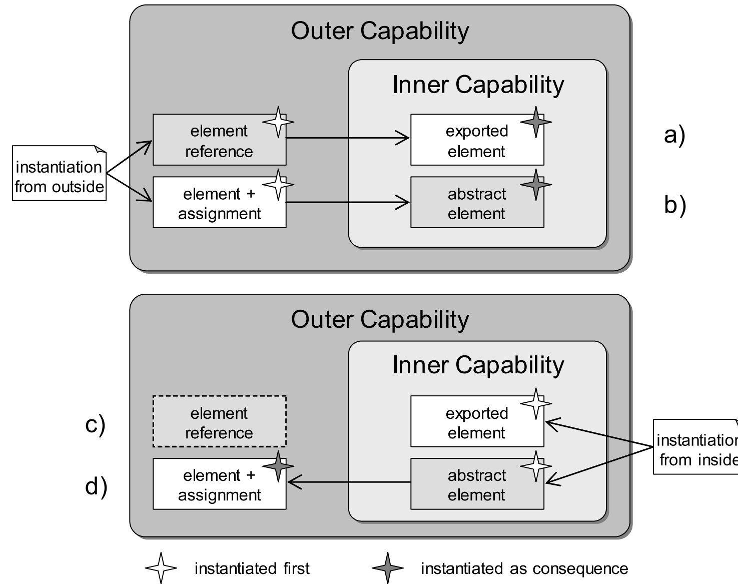

Fig. 3. Element creation cases: a) instantiation of a reference, b) instantiation of an element, when an abstract element exists inside, c) instantiation of an element, when a reference exists outside, d) instantiation of an abstract element assigned from the outside

has to be assured that changes outside of a capability do not affect the internal functionality, as this would violate the information hiding principle.

To solve these issues semantics is associated to the creation process of elements and proxies in all possible cases. On the one hand, a proxy might be in the outer capability (i.e. a reference) or in the inner capability (i.e. an abstract element). On the other hand, creation of an element might be issued on the original element itself, or the proxy. This leads to four different cases (cf. Fig. 3):

In the first two cases (a and b) the creation is triggered inside the outer capability. It is safe to create elements in the inner capability, because they are an explicit part of the interface. In the last case (d) the initially created element in the inner capability is just a proxy. Therefore it is necessary to subsequently create the original element in the outer capability. As can be seen in the figure, it is only in case c, that a proxy element is not instantiated. In this case the original element is part of the inner capability and creation is triggered from the inside. Creating the outer element would lead to problems, because the element (e.g. an event) might inadvertently be handled in the outside capability, thereby breaking functionality of the inner capability. The export / reference concept assures that an element can be used from the outside (e.g. a goal created outside), but the local functionality (e.g. the goal processing) remains unchanged. Abstract elements provide a way for the inner capability to connect to functionality of an outer capability.

3.4 Initial Configurations

One important aspect of reusability is *parametrization* of the reused components, to adapt them to the special requirements imposed by settings in which they get reused [7]. When considering parametrization of capabilities, two questions have to be answered: First, what can be parametrized, i.e. what constitutes a configuration of a capability? Second, how can a capability be parametrized from the outside, when its elements are encapsulated?

As an answer to the first question, we introduce the notion of an *initial mental state*. The initial mental state is a simplified runtime state of a capability, containing the initial values of beliefs (which are singleton instances), as well as zero or more initial instances of the other elements (such as goals, plans, and events) with initial properties (e.g. goal parameter values). In addition, the initial state defines recursively the initial mental state of all included subcapabilities.

To parametrize a capability, its initial mental state has to be adapted to the current needs. Respecting the information hiding principle, it should not be possible to specify all elements of the initial mental state from the outside. E.g. some capability might require an instance of a maintain goal for proper operation, but the maintain goal should not be visible to the outside. Our approach allows parametrization at two levels of granularity: The capability level, and the level of individual elements. A capability itself can provide one or more initial configurations, which can be referenced by a given name. In this way, common use cases can be captured as a whole, allowing easy out-of-the-box reuse. These configurations are part of the capability, and therefore can contain exported and internal elements. Parametrization at the level of individual elements is only possible when these elements are exported. For example for exported beliefs, the outer capability can override the initial value by defining a reference and locally assigning a new value to this referenced belief.

3.5 Dynamic Runtime Composition

The new improved capability concepts is also capable of handling various issues concerning runtime modification of agent behaviour. For that purpose generally two distinct kinds of operations can be performed.

On the one hand complete capabilities could be plugged into or removed from an agent at runtime. The addition of a capability at runtime is conceptually not difficult as it requires only information about the capability type, its initial state, the target capability within the agent and some connection data such as the instance name of the new capability. Given that all information is provided the capability can be linked with the target capability using the connection data and can subsequently be started meaning that its initial state will be executed. The removal of a capability at runtime is far more intricate as the agent's execution state must be considered before a capability can be removed safely. This means the agent e.g. could currently utilize plans or goals from the capability to remove and it needs to be determined if these plans or goals should be executed completely before removal.

On the other hand the modification of a capability at runtime should be possible. Prerequisite for this modifications is that each capability relies on its personal copy of the underlying capability model, so that changes can be performed without affecting other capability instances of the same type. The creation process for a new model element (regardless if it is an element or an element reference) consists basically of two steps. First, the element has to be created in the capability model. In a second step the elements has to be registered at the runtime layer, making the agent aware of its new element. For the deletion of an element at runtime a similar process can be used. The element has to be deregistered at the capability instance and can afterwards be deleted from the model. In this respect it has again to be considered if existing elements of the removed type should be discarded at once.

For the complete freedom of removing or exchanging a capability at runtime it is a necessary prerequisite that a capability instance is a self-contained entity with well-defined connection points. In the proposed capability approach this is supported by the locality principle which is also valid for capabilities at runtime. Elements from other capabilities are not accessed directly but through local proxies. When a capability or element of a capability is removed, proxies in other capabilities can be preserved, and later be reconnected when an alternative capability or element is available. Further elaborating and implementing the details of this mechanism is planned for future work.

3.6 Discussion

Capabilities are a decomposition concept for agents allowing to reuse functionality captured in a self-contained module with clearly defined import and export interface. This form of reuse is termed blackbox-reuse as only the interface and no details about the internals are known. In contrast to (more flexible) whitebox-reuse changing the internals of a blackbox-component does not break existing usages, leading to application designs that are easier to maintain [16].

Furthermore, the capability concept addresses most of the five fundamental criteria of modularization from [12]: decomposability, composability, understandability, continuity and protection. The concept naturally supports decomposability and composability as functional coherent units can be built and connected in flexible ways. The understandability for BDI agents is increased because capabilities represent encapsulated functionalities that normally have few connections to other capabilities. Additionally, the understandability for a single capability is supported by the locality principle which makes them self-contained. The continuity criterion requires that a small change of the problem specification leads to limited changes in only few concerned modules. This is achieved by extensively using the information hiding principle using small and simple interfaces through the general import / export mechanism. Finally, protection is attained when the effect of an abnormal condition occurring at runtime is confined to the originating module. Capabilities do not add a new level of protection to the development of BDI agents. Nevertheless, failure of plans is already covered by the normal BDI mechanism.

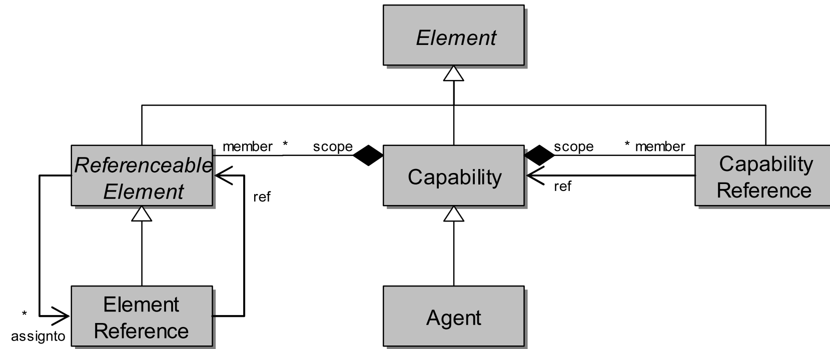

Fig. 4. Capability metamodel

Another decomposition method for BDI agents inspired directly from object-oriented ideas was proposed in [10]. It is based mainly on the inheritance mechanism for agent classes explicitly allowing also multiple inheritance relationships between agent classes. In order to control the exact semantics of inheritance relationships an agent class consists of individual submodels for beliefs, plans and goals respecting the specifics of the individual mentalistic concepts. Similar to capabilities, this approach decomposes agents at the detailed design and implementation level. Other decomposition approaches consider high-level concepts such as roles. E.g. in [9] an experimental system based on the Zeus [13] toolkit is described, which uses roles to group primitive and rule-based tasks as well as external code into a reusable module. Role constraints and a role algebra are introduced to describe how agents can be statically composed of predefined roles. Dastani et al. describe in [5] a formal model of roles composed of beliefs, goals, plans and rules. The approach focuses on an operational semantics for dynamic enacting and deacting of roles. It does not cover interfaces between different roles of the same agent, but assumes that only one role is active at each moment in time.

4 Realization of Capabilities

The extended capability concept as presented in the last section has been implemented within the Jadex BDI reasoning engine [1,14]. In Jadex agents are specified in two different kinds of files. The static structure of an agent or capability including its initial mental state is defined within an XML-file that adheres to the Jadex BDI metamodel specified in XML-schema. The behavior of Jadex agents is encoded within plan bodies that are programmed with plain Java. From within user programmed plans the BDI facilities such as modifying beliefs or creating goals are accessible through an API.

In Fig. 4 the condensed Jadex capability metamodel is depicted. All entities share the same abstract base class “element”. Furthermore an agent is modeled as an extended capability. This reflects the fact that agent specifications are very

similar to capabilities and additionally may support entities composed of agents such as groups sharing e.g. some beliefs or goals.

A capability is composed of two different kinds of elements. On the one hand it is a container for "referenceable elements", which form the abstract base class for mental attitudes such as beliefs, goals, plans and events (not illustrated). Their common property is that they can be referenced by a proxy termed "element reference" from another capability. Element references, which are used to represent abstract elements as well, are themselves also "referenceable elements" as references to references are explicitly allowed in the model. On the other hand capabilities can contain subcapabilities, which is expressed by the relationship to "capability references". This indirection is used, because a capability includes a subcapability under a symbolic name allowing for inclusion of more than one instance of the same capability type.

5 Example Application

To illustrate the aforementioned concepts in this section an example application for a hunter-prey scenario is detailed. Even though the hunter-prey domain is a well-known and extensively studied AI playing field, various different interpretations exist making it necessary to outline our settings. The environment is inhabited by two different species of creatures (hunters and preys) and various obstacles (trees) which hinder them in their movements. The creature's main objective is to survive by looking for food. Hence, hunters are exploring the terrain in search of prey which they will try to chase and eat. Contrarily, preys look for plants growing in the environment and try to flee if chased by some hunter.

5.1 Scenario Design Details

This scenario is designed as (possibly distributed) agent-based simulation in which the creatures as well as the environment are represented as autonomous entities. The environment agent is responsible not only for holding a representation of the environment which is set up as a discrete grid world, but also for controlling the advancement of time. For simplicity reasons a time-driven scheme is employed, which requires the creatures to announce their next intended action within an adjustable timeframe.

Initially, creatures are placed at random locations in the world and are only able to perceive a cutout of the world according to their vision (automatically sent from the environment to the creatures at the beginning of each round). In each round the creatures have to decide which action they would like to perform. Possible actions are moving to an adjacent field (up, down, left or right) or trying to eat some object resp. creature near to it. The actions have to be communicated to the environment as messages following a hunter-prey domain ontology. If a creature fails to provide its intended action within the round time (e.g. because it reasons too slowly or a network error occurred) the simulation proceeds executing no action in that round for the creature.

As all creatures need basic abilities for sensing and acting in their environment it is natural and advantageous to develop a basic module for handling this fundamental aspect of creature behavior in a reusable way.

5.2 Defining a Capability

In Fig. 5 the capability specification for basic sensing and acting in the environment is depicted. It has two main purposes: The first one is to automatically process "inform vision" messages (lines 51-65) which contain the current vision of the creature sent from the environment. Whenever such a message event is received the "update vision" plan (lines 44-47) is triggered. This plan will extract the information contained in the vision and update the creature's belief sets about known hunters, preys, obstacles and food (lines 13-16) accordingly. Note that all of these belief sets are exported to be accessible from an outer capability and the creature agent itself.

The second purpose is to provide high-level abstractions for performing actions in the environment. Therefore, the capability defines exported goal types for moving and eating (lines 24-29). To initiate an action, a creature has to create and dispatch a new move or eat goal. Such goal instances will subsequently be handled within the act/sense capability by triggering corresponding move or eat plans (lines 36-43) which encode the action into a message and communicate with the environment agent (defined as belief in line 17).

To locate the environment agent the act/sense capability itself relies on an included directory facilitator (DF) capability (lines 8-10) which offers goals for (de)registering and searching at a DF. For being able to access the DF functionality the act/sense capability defines a concrete goal reference to the "df search" goal (lines 30-32). Hence, from within plans of the act/sense capability "df search" goals can be created and dispatched.

For the communication with the environment agent it is necessary for the creature to identify itself which is done by including information available in the "my self" belief. As the act/sense capability should be usable by hunters as well as preys the value depends on the concrete usage of the capability. Thus, the belief is specified as abstract and required (which is the default for abstract beliefs) and needs to be assigned from the outer capability respective the agent that uses the act/sense capability.

5.3 Capability Parametrization

To exhibit reasonable behavior it is necessary for creatures to describe their high-level objectives and the means for achieving them. In this section a "basic behavior" capability (Fig. 6) for preys is described, which enables preys to explore their environment, eat food and flee from near hunters. Three goal types are designed for this purpose. An instance of a "keep alone" maintain goal (lines 27-29) has the task to monitor if the prey is currently in danger. It becomes active whenever a hunter is nearby and will trigger plans for fleeing from the hunter. "Eat food" achieve goals (lines 30-35) are created automatically for every piece

```
01 <capability xmlns="http://jadex.sourceforge.net/jadex"
02   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
03   xsi:schemaLocation="http://jadex.sourceforge.net/jadex
04   http://jadex.sourceforge.net/jadex-0.94.xsd"
05   package="jadex.examples.hunterprey.creature.actsense"
06   name="ActSense">
07
08   <capabilities>
09     <capability name="dfcap" file="jadex.planlib.DF"/>
10   </capabilities>
11
12   <beliefs>
13     <beliefset name="hunters" class="Hunter" exported="true"/>
14     <beliefset name="preys" class="Preys" exported="true"/>
15     <beliefset name="obstacles" class="Obstacle" exported="true"/>
16     <beliefset name="food" class="Food" exported="true"/>
17     <belief name="environmentagent" class="jadex.adapter.fipa.AgentIdentifier"/>
18     <beliefref name="my_self" class="Creature" exported="true">
19       <abstract/>
20     </beliefref>
21   </beliefs>
22
23   <goals>
24     <achievegoal name="move" exported="true">
25       <parameter name="direction" class="String"/>
26     </achievegoal>
27     <achievegoal name="eat" exported="true">
28       <parameter name="object" class="WorldObject"/>
29     </achievegoal>
30     <achievegoalref name="df_search">
31       <concrete ref="dfcap.df_search"/>
32     </achievegoalref>
33   </goals>
34
35   <plans>
36     <plan name="move">
37       <body>new MovePlan()</body>
38       <trigger><goal ref="move"/></trigger>
39     </plan>
40     <plan name="eat">
41       <body>new EatPlan()</body>
42       <trigger><goal ref="eat"/></trigger>
43     </plan>
44     <plan name="updatevision">
45       <body>new UpdateVisionPlan()</body>
46       <trigger><messageevent ref="inform_vision"/></trigger>
47     </plan>
48   </plans>
49
50   <events>
51     <messageevent name="inform_vision" type="fipa" direction="receive">
52       <parameter name="performative" class="String" direction="fixed">
53         <value>jadex.adapter.fipa.SFipa.INFORM</value>
54       </parameter>
55       <parameter name="language" class="String" direction="fixed">
56         <value>jadex.adapter.fipa.SFipa.JAVA_XML</value>
57       </parameter>
58       <parameter name="ontology" class="String" direction="fixed">
59         <value>HunterPreyOntology.ONTOLOGY_NAME</value>
60       </parameter>
61       <parameter name="content-class" class="Class" direction="fixed">
62         <value>CurrentVision.class</value>
63       </parameter>
64       <parameter name="content" class="CurrentVision"/>
65     </messageevent>
66   </events>
67 </capability>
```

Fig. 5. Act/sense capability

of food the creature discovers. They will lead to plan executions for reaching the food's location and eating it. The third goal type is called "wander around" (line 36) and initiates random walking on the map. In this paper the details of goal declarations are out of scope, for an extensive description the reader can refer to [2].

For proper operation the basic behavior capability needs access to the environmental beliefs made available by the included act/sense capability (lines 8-10). Therefore concrete belief set references are defined for hunters, obstacles and food (lines 13-16). These belief set references are also exported allowing an outer capability or the creature itself to access these values. An abstract and exported belief reference is assigned for "my self", as the basic behavior capability still does not know in which exact kind of prey it will be used.

To illustrate the parametrization of capabilities it is assumed that two different kinds of preys need to be created from the basic behavior capability. The first labeled "LazyPrey" should only flee from nearby hunters and otherwise just sit and wait where it is. On the contrary a "CleverPrey" should wander around to explore the map, eat food and flee from hunters. For each kind of prey a separate initial mental state has been specified (lines 48-57 and lines 58-63).

For the operation of a LazyPrey it is necessary to have an instance of a "keep alone" goal to flee from hunters. This is achieved by creating an initial goal of that type (line 55). Additionally, it is required to turn off the reactive creation of "eat food" goals which cannot be done directly. Hence, a belief "eating allowed" is introduced as some kind of goal creation switch and used in the creation condition of the "eat food" goal. In the initial state this belief is negated which ensures that no "eat food" goals will be instantiated at runtime.

A CleverPrey needs to exploit the whole functionality of the basic behavior capability. Thus, initial goals for exploring the map (line 60) and for escaping from nearby hunters (line 61) are created. Nothing has to be declared in the initial state for "eat food" goals, because these are created automatically at runtime whenever a new piece of food is discovered as mentioned earlier.

6 Summary and Outlook

This paper revisits the capability concept introduced by Busetta et al. for modularizing BDI agents, in which several conceptual limitations have been identified:

- No generic mechanism for importing / exporting arbitrary mental elements such as beliefs and goals is available leading to a decreased usability.
- The parametrization of capabilities is not supported which hinders flexible reuse of capabilities.
- Only design time composition is supported.
- Refinements of mental elements are not addressed.

In turn a new capability concept based on the main ideas of the original proposal is introduced to address most of these shortcomings. Regarding the usability and generality a new import / export mechanism is presented, allowing to treat all

```
01 <capability xmlns="http://jadex.sourceforge.net/jadex"
02   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
03   xsi:schemaLocation="http://jadex.sourceforge.net/jadex
04   http://jadex.sourceforge.net/jadex-0.94.xsd"
05   package="jadex.examples.hunterprey.creature.preys.basicbehaviour"
06   name="BasicBehaviour">
07
08   <capabilities>
09     <capability name="actsensecap" class="ActSense"/>
10   </capabilities>
11
12   <beliefs>
13     <beliefsetref name="hunters" class="Hunter" exported="true">
14       <concrete ref="actsensecap.hunters" />
15     <beliefsetref>
16     <!-- similar declarations for obstacles and food omitted for brevity. -->
17     <beliefref name="my_self" class="Creature" exported="true">
18       <assignto ref="actsensecap.my_self"/>
19       <abstract/>
20     </beliefref>
21     <belief name="eating_allowed" class="boolean">
22       <fact>true</fact>
23     </belief>
24   </beliefs>
25
26   <goals>
27     <maintaingoal name="keep_alone" exclude="never">
28       <!-- details omitted for brevity. -->
29     </maintaingoal>
30     <achievegoal name="eat_food">
31       <creationcondition>
32         $beliefbase.eating_allowed && $beliefbase.food.length>0
33       </creationcondition>
34       <!-- further details omitted for brevity. -->
35     </achievegoal>
36     <performgoal name="wander_around" retry="true" exclude="never"/>
37     <achievegoalref name="move">
38       <concrete ref="actsensecap.move"/>
39     </achievegoalref>
40     <achievegoalref name="eat">
41       <concrete ref="actsensecap.eat"/>
42     </achievegoalref>
43   </goals>
44
45   <plans><!-- omitted for brevity. --></plans>
46
47   <initialstates>
48     <initialstate name="flee">
49       <beliefs>
50         <initialbelief ref="eating_allowed">
51           <fact>false</fact>
52         </initialbelief>
53       </beliefs>
54       <goals>
55         <initialgoal name="escapegoal" ref="keep_alone"/>
56       </goals>
57     </initialstate>
58     <initialstate name="wander_flee_eat">
59       <goals>
60         <initialgoal name="wandergoal" ref="wander_around"/>
61         <initialgoal name="escapegoal" ref="keep_alone"/>
62       </goals>
63     </initialstate>
64   </initialstates>
65 </capability>
```

Fig. 6. Basic prey behavior capability

elements (e.g. beliefs or goals) in a similar fashion and hence simplifying the way in which a capability interface is defined. Furthermore, parametrization is supported through the definition of an initial mental state, which is defined as a part of the capability itself. This allows for easy capability configuration as only the state names need to be known in the including capability.

In addition to the aforementioned issues the new capability concept is also prepared to handle the dynamic composition of capabilities at runtime to flexibly adopt agent behavior. The extension points to add functionality at runtime are already present in the new capability concept and have been tested in the current implementation. The process of removing capabilities and their elements at any time during the agent execution has only been sketched and is left for future work.

Another area of future work which is also facilitated through the locality principle is the refinement of elements from subcapabilities. Properties of an element might be redefined on a proxy element. In our vision such an element refinement has similarities with the inheritance relationship from object-orientation. E.g. a goal reference type could be used to inherit all the properties of the concrete goal and add new properties such as additional parameters. At runtime the proposed creation semantics could help in deciding in which cases the redefined or the original specification should be used.

Finally, an interesting topic for future research regards bringing together the role-based decomposition approaches with the component-based capability approach. A capability might provide an implementation entity for a role identified in an abstract design. Building agents capable of playing certain roles could then be easily done by composing the agent from capabilities for each role.

References

1. L. Braubach, A. Pokahr, and W. Lamersdorf. Jadex: A BDI Agent System Combining Middleware and Reasoning. In M. Klusch R. Unland, M. Calisti, editor, *Software Agent-Based Applications, Platforms and Development Kits*. Birkhäuser, 2005.
2. L. Braubach, A. Pokahr, D. Moldt, and W. Lamersdorf. Goal Representation for BDI Agent Systems. In *Proc. of the 2nd Workshop on Programming Multiagent Systems: Languages, frameworks, techniques, and tools (ProMAS04)*, 2004.
3. P. Busetta, N. Howden, R. Rönnquist, and A. Hodgson. Structuring BDI Agents in Functional Clusters. In *Proc. of the 6th Int. Workshop, Agent Theories, Architectures, and Languages (ATAL) '99*, pages 277–289. Springer, 2000.
4. M. Dastani, B. van Riemsdijk, F. Dignum, and J.-J. Meyer. A Programming Language for Cognitive Agents: Goal Directed 3APL. In *Proceedings of the first Workshop on Programming Multiagent Systems (ProMAS03)*, 2003.
5. M. Dastani, B. van Riemsdijk, J. Hulstijn, F. Dignum, and J.-J. Meyer. Enacting and deacting roles in agent programming. In *Proceedings of the 5th International Workshop on Agent-Oriented Software Engineering (AOSE'04)*, 2004.
6. C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Fundamentals of software engineering*. Prentice-Hall, Inc., 2003.

7. D. Heimbigner and A. Wolf. Post-deployment configuration management. In *System Configuration Management, ICSE'96 SCM-6 Workshop*, pages 272–276. Springer, 1996.
8. N. Howden, R. Rönnquist, A. Hodgson, and A. Lucas. JACK Intelligent Agents - Summary of an Agent Infrastructure. In *Proceedings of the 5th ACM International Conference on Autonomous Agents*, 2001.
9. A. Karageorgos, S. Thompson, and N. Mehandjiev. Specifying reuse concerns in agent system design using a role algebra. In *Agent Technologies, Infrastructures, Tools, and Applications for e-Services*, 2003.
10. D. Kinny and M. Georgeff. Modelling and Design of Multi-Agent Systems. In *Intelligent Agents III: Third International Workshop on Agent Theories, Architectures, and Languages (ATAL-96)*. Springer-Verlag, 1996.
11. J. F. Lehman, J. E. Laird, and P. S. Rosenbloom. A gentle introduction to Soar, an architecture for human cognition. *Invitation to Cognitive Science*, 4, 1996.
12. B. Meyer. *Object-Oriented Software Construction*. Prentice Hall PTR, 1997.
13. H. Nwana, D. Ndumu, L. Lee, and J. Collis. Zeus: a toolkit and approach for building distributed multi-agent systems. In *Proceedings of the third annual conference on Autonomous Agents*, pages 360–361. ACM Press, 1999.
14. A. Pokahr, L. Braubach, and W. Lamersdorf. Jadex: A BDI Reasoning Engine. In J. Dix R. Bordini, M. Dastani and A. Seghrouchni, editors, *Multi-Agent Programming*. Kluwer, 2005.
15. A. Rao and M. Georgeff. BDI Agents: from theory to practice. In *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS'95)*, pages 312–319. The MIT Press: Cambridge, MA, USA, 1995.
16. C. Szyperski, D. Gruntz, and S. Murer. *Component Software: Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, 2002.
17. E. Yourdon and L. Constantine. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. Prentice-Hall, Inc., 1979.