Capability Relationships in BDI Agents

Ingrid Nunes

Instituto de Informática
Universidade Federal do Rio Grande do Sul (UFRGS)
Porto Alegre Brazil
ingridnunes@inf.ufrgs.br

Abstract. The belief-desire-intention (BDI) architecture has been proposed to support the development of rational agents, integrating theoretical foundations of BDI agents, their implementation, and the building of large-scale multi-agent applications. However, the BDI architecture, as initially proposed, does not provide adequate concepts to produce modular software components. The capability concept emerged to address this issue, but the relationships between capabilities have been insufficiently explored to support the development of BDI agents. We thus, in this paper, introduce and analyse three possible relationships among capabilities in BDI agent development — namely association, composition and generalisation — which are widely used in object-oriented software development, and are fundamental to develop software components with low coupling and high cohesion. Our goal with this paper is to promote the exploitation of these and other mechanisms to develop large-scale modular multi-agent systems and discussion about this important issue of agent-oriented software engineering.

Keywords: Capability, Modularisation, BDI Architecture, Agent-oriented Development.

1 Introduction

The *belief-desire-intention* (BDI) architecture is perhaps the most adopted architecture to modelling and implementing rational agents. It has foundations in a model proposed by Bratman [3], which determines human action based on three mental attitudes: beliefs, desires and intentions. Based in this model, Rao and Georgeff [16] proposed the BDI architecture, integrating: (i) theoretical work on BDI agents; (ii) their implementation; and (iii) the building of *large-scale* applications based on BDI agents. Although their work has been widely used to model and implement BDI agents in theory and practice in academy, there is no real evidence that this approach scales up.

Much work on software engineering aims to deal with the complexity of large-scale enterprise software applications to support their development, and a keyword that drives this research is *modularity.* Software developed with modular software components — i.e. components with high cohesion and low coupling properties — are more flexible and easier to reuse and maintain. Although modularity is highly investigated in the context of mainstream software engineering, it has been poorly addressed not only in work on BDI agents, but also by the agent-oriented software engineering community. Research in this context is limited to few approaches, for example, modularisation of crosscutting concerns in agent architectures with aspects [9, 17] and the use of capabilities in BDI agent architectures [6, 4].

We, in this paper, investigate the concept of *capability*, in order to allow the modular construction of BDI agents, with the aim of supporting the development of large-scale systems based on BDI agents (hereafter, agents). Capabilities are modules that are part of an agent, and they cluster a set of beliefs and plans that together are able to handle events or achieve goals. Therefore, it modularises a particular functional behaviour that can be added to agents. The capability concept is available in some of the BDI agent platforms [10, 12, 15]; however, there is divergence on its implementation, and therefore there is no standard structure for this concept. One communality shared by different capability implementations is the ability to include capabilities to another, but this relationship also varies in the different available implementations, as well as their implications in the agent reasoning cycle at runtime. Moreover, there is a single type of relationship between capabilities in each implementation. This differs from the object-oriented paradigm, which allows to establish many types of relationships between software objects.

We thus present an initial work on the investigation of different types of relationships that may occur between capabilities, introducing three of them, namely *association*, *composition* and *generalisation*. Besides describing each type of relationship, we analyse how a pair of related capabilities work together in the context of the agent reasoning. These relationships may be used in combination to design and implement an agent, and we show examples of this scenario. The presented relationships provide the basis for a discussion with respect to engineering aspects of agents, which support the construction agent-based systems. Our aim is to promote the exploitation of these and other mechanisms to develop large-scale modular multi-agent systems and discussion about this important issue of agent-oriented software engineering.

The remainder of this paper is organised as follows. We first introduce work related to capabilities in Section 2. Then, we describe the different capability relationships in Section 3, and exemplify their combined use in Section 4. We next analyse and compare these relationships in Section 5, also showing how each of the existing BDI platforms that provide the capability concept implement it. Finally, we conclude this paper in Section 6.

2 Related Work

We begin by presenting work that has been done in the context of capabilities. The capability concept was introduced by Busetta et al. [6] and emerged from experiences with multi-agent system development with JACK [10, 1], a BDI

Part	Definition
Identifier	The capability identifier, i.e. a name.
Plans	A set of plans.
Beliefs	A set of beliefs representing a fragment of knowledge base and manipulated by the plans of the capability.
Belief Visibility Rules	Specification of which beliefs are restricted to the plans of the capability and which ones can be seen and manipulated from outside.
Exported Events	Specification of event types, generated as a consequence of the activity of the capability, that are visible outside its scope, and their processing algorithm.
Perceived events	Specification of event types, generated outside the capability, that are relevant to the capability.
Capabilities	Recursive inclusion of other capabilities.

Table 1: Capability Specification.

agent platform. The goal was to build modular structures, which could be reused across different agents. In Table 1, we detail the parts that comprise a capability according to this work. Some of which are specific to the JACK platform, such as the explicit specification of perceived events.

This work is the result of practical experience, so Padgham and Lambrix [13] formalised the capability concept, in order to bridge the gap between theory and practice. This formalisation included an indication of how capabilities can affect agent reasoning about its intentions. In order to integrate capabilities to the agent development process, Penserini et al. [14] proposed a tool-supported methodology, which goes from requirements to code. It identifies agent capabilities at the requirement specification phase, based on the analysis models of Tropos [5], and is able to eventually generate code for Jadex [15], another BDI agent platform.

Among the different available platforms to implement BDI agents, such as Jason[1] [2] and the 3APL Platform[2], three implement the capability concept: JACK[3] [10], Jadex[4] [15, 4], and BDI4JADE[5] [12]. As we already discussed how JACK capabilities are implemented, we next detail the other two implementations, which include a capability identifier.

A Jadex capability is composed of: (i) beliefs; (ii) goals; (iii) plans; (iv) events; (v) expressions; (vi) properties; (vii) configurations; and (viii) capabilities. Some of these parts are platform-specific, such as expressions, which are expressions written in a language that follows a Java-like syntax and are used for different

[1] `http://jason.sourceforge.net/`
[2] `http://www.cs.uu.nl/3apl/`
[3] `http://aosgrp.com/products/jack/`
[4] `http://www.activecomponents.org`
[5] `http://www.inf.ufrgs.br/prosoft/bdi4jade/`

purposes, e.g. goal parameters or belief values. Beliefs can be used only within the scope of the capability, exported to outside the capability scope, or abstract, meaning that a value of a belief outside the capability may be assigned to this abstract belief. The BDI4JADE capability, on the other hand, is composed of: (i) a belief base; (ii) a plan library; and (iii) other capabilities. These are the explicit capability associations. As BDI4JADE is written in pure Java (no XML files), other properties may be obtained by manipulating the capability parts, besides the described components.

Given this analysis of existing work on capabilities, we next introduce three different types of relationships between capabilities. As said before, all the implementations of the capability concept provide limited relationship types, and after introducing our relationship types, we will revisit these capability implementations in Section 5, indicating the meaning of their capability relationship.

3 Relationships between Capabilities

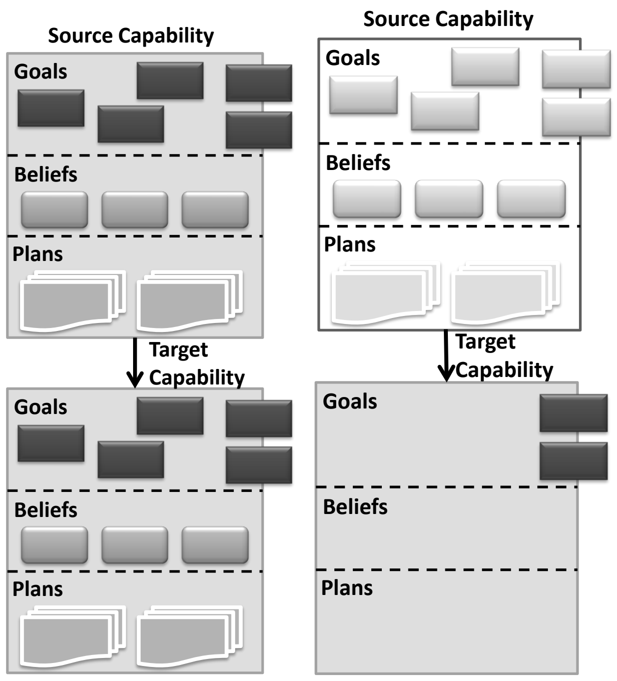
According to the object-oriented paradigm, a system is composed of software objects, which integrate code and data. Such objects are building blocks to construct complex structures, and can be combined using different forms of relationships. In this section, we analyse three of these relationships — association (Section 3.1), composition (Section 3.2), and inheritance (Section 3.3) — in the context of capabilities.

3.1 Association

Software objects encapsulate both state (represented by attributes) and behaviour (represented by methods), and are accessed through its *interface*, which is a collection of method signatures. In order for a system to implement functionality, objects collaborate by invoking methods of other objects with which they are associated.

Similarly, capabilities implement some functionality, and have both state (represented by beliefs) and behaviour (represented by plans). The main difference from the object concept is that, while methods that are part of an object interface can be directly invoked by other objects, plans are dispatched within the context of the agent reasoning cycle, and its execution is triggered by a goal or, in some BDI models, an event. As a consequence, in order for an agent behaviour to be the result of the interaction of more than one capability, an important question arises: *what is a capability interface?*

In a capability, beliefs are a piece of encapsulated knowledge, and are manipulated by the capability's plans. Consequently, following the principle of information hiding, the manipulation of beliefs are restricted to the capability. Plans, which correspond to methods, cannot be explicitly invoked. Therefore, they are accessible only within the context of the capability, and are not part of the capability interface as well. Goals, on the other hand, indicate the objectives that a capability may achieve, and possibly there are different capability plans



(a) Structure. (b) Visibility (Source).

Fig. 1: Association.

that can be used to achieve such goals. Therefore, goals represent services that a capability may provide to another, and thus comprise its interface. This is illustrated in capabilities of Figure 1, in which goals are in the border of the capabilities. Note, however, that there are goals used only internally, and are not part of the capability interface.

Given that we now have an interface for capabilities — specified in terms of a set of goals that a capability may achieve — we are able to associate capabilities so that they can collaborate. An association is a relationship, shown in Figure 1a, where a source capability C_S uses a target capability C_T, by delegating goals to be achieved by C_T. In the context of the agent reasoning cycle, it means that during the execution of plans that belong to C_S, goals that are part of the C_T interface may be dispatched, and only plans that belong to C_T are candidates to be selected to handle such goals. This is similar to the notion of delegating a goal to another agent, but two agents mean two threads of execution, whereas two capabilities of one agent consist of a single thread of execution.

Consider the scenario in which we are developing an intelligent robot, which is responsible for household duties, such as cleaning the floor and washing clothes. For both these duties, the robot has to move around and, while executing plans for cleaning the floor and washing clothes, the robot has to achieve a subgoal

$move(x, y)$, i.e. move from a position x to a position y. In this case, our robot may have three capabilities — `FloorCleaning`, `Laundry`, and `Transportation` capabilities — and there are association relationships from the `FloorCleaning` and `Laundry` capabilities to the `Transportation` capability, which has an external goal $move(x, y)$, part of its interface.

We present in Figure 1b the visibility of components of the target capability by the source capability. In this figure, and others presented throughout the paper, we show what the capability with a white background can access from the capability with the gray background. All components within the scope of the target capability are hidden and inaccessible by the source capability, except the goals that are part of the target capability interface. Such goals may be dispatched by plans of the source capability. The target capability, on the other hand, is not aware of the source capability.

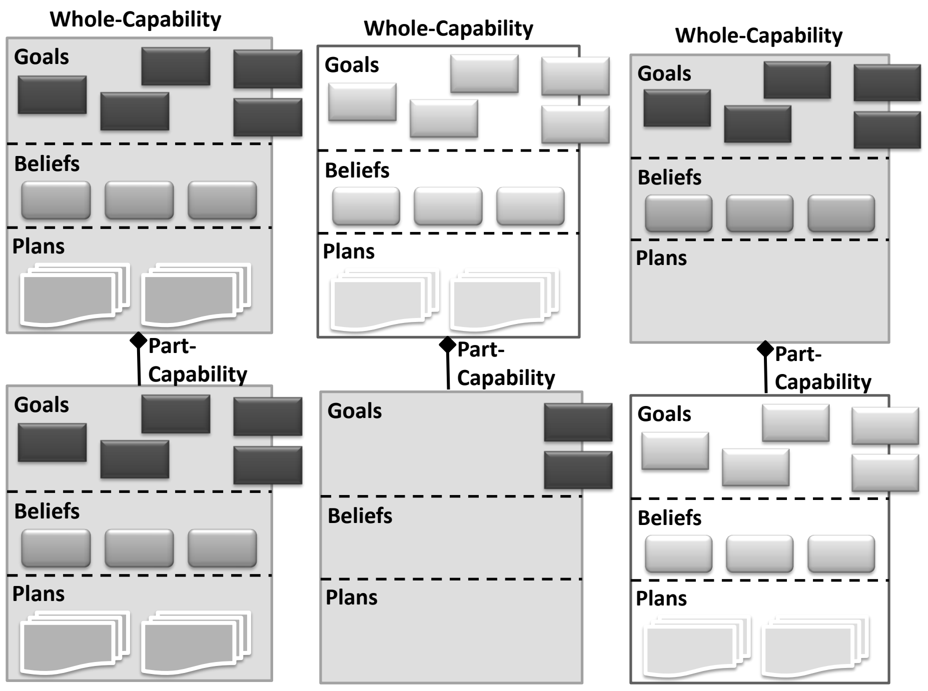
Although the association relationship is directed, it may be bidirectional. In order to better modularise an agent architecture, functionality associated with two different concerns may be split into two capabilities, and they may use each other to achieve their goals.

3.2 Composition

The association relationship allows us to modularise BDI concepts into two capabilities — composed of beliefs, goals, and plans — and each of which should address a different concern, thus having high cohesion. The connection between these capabilities is that the execution of at least one plan of the source capability requires achieving a goal that is part of the target capability. In this case, each capability uses the knowledge captured by their own beliefs to execute their plans.

However, there may be situations in which there should be shared knowledge between capabilities, that is, a capability uses the information stored in other capability's beliefs in the execution of its plans. In this case, the composition relationship is used, which increases the coupling between the two involved capabilities. This kind of relationship expresses the notion of containment, and its structure is presented in Figure 2a.

An agent maybe be built by first developing functionality to achieve lower level goals, and then using it to develop higher level functionality. For example, assume that the `FloorCleaning` capability of the robot agent must have goals, beliefs and plans to both sweep the floor and vacuum the dust, when there are carpets on the floor. As these are two different concerns, they may be modularised into two capabilities, each being composed of the external goals related to their respective duty to be accomplished. The `FloorCleaning` capability, by having a composition relationship with the `Sweeper` and the `VacuumCleaner` capabilities, can thus dispatch external goals of these two capabilities — while executing a plan to clean a room, for instance. This can also be performed using the association relationship, but now there are two differences. First, the `Sweeper` and the `VacuumCleaner` capabilities can have plans to handle `FloorCleaning`'s goals, so if goals are dispatched in plans of this capability, they may be achieved by



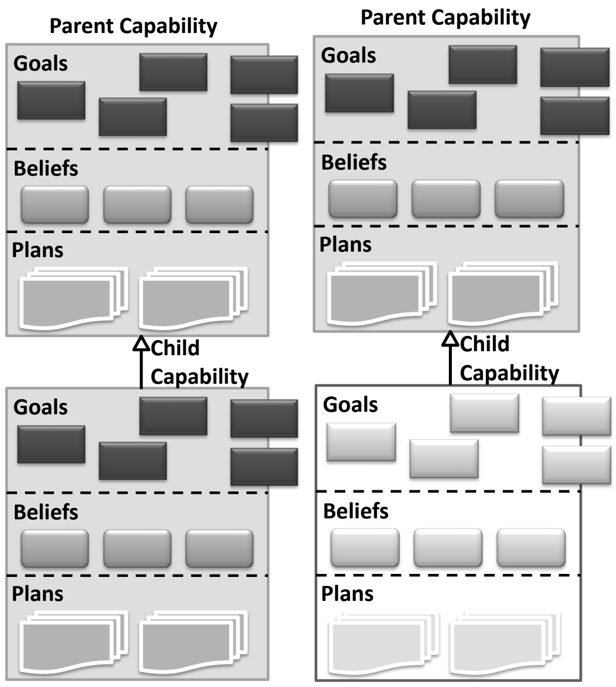
(a) Structure. (b) Visibility (Whole). (c) Visibility (Part).

Fig. 2: Composition.

plans of the composed capabilities. Second, the `FloorCleaning` capability may have knowledge stored in its beliefs, such as those related to the environment, and they need to be used to both sweep the floor and to vacuum the dust. So by composing the `FloorCleaning` capability with the other two, the `Sweeper` and `VacuumCleaner` capabilities may access the `FloorCleaning`'s beliefs in the execution of their plans.

The visibility of the components of the two capabilities involved in a composition relationship, namely the whole and the part, are shown in Figure 2. Figure 2b shows that the whole-capability is able to dispatch external goals of part-capability, but cannot access other components. And Figure 2c details that the part-capability can access both the beliefs and goals of the whole-capability.

This relationship is transitive. Consider a capability C that is part of a capability B, which in turn is part of a capability A. Therefore, C can access beliefs of both B and A in addition to its own beliefs, and A's goals can be handled by plans of both B and C, in addition to its own plans. As a consequence, different compositions may be performed with capabilities that implement low level behaviour.



(a) Structure. (b) Visibility (Child).

Fig. 3: Inheritance.

3.3 Inheritance

While the association and composition relationships focus on collaborating capabilities, the goal of the inheritance relationship — which will now be discussed — is mainly to promote reuse, by generalising common behaviour in a parent capability and specialising it in children capabilities. This relationship increases the coupling between the involved capabilities, with respect to the other two types of relationships. It is also transitive, that is, a child capability inherits from its parent's parent.

The development of a multi-agent system may involve building agents that share a common behaviour, but have some particularities that distinguish one from another. In this case, we may need to design a capability with a set of beliefs, goals, and plans, to which other goals, beliefs and plans must be added to develop particularities. The inheritance relationship thus allows to connect this common behaviour to specialised variable behaviour. This relationship is illustrated in Figure 3a.

When a capability extends another, it inherits all the components of the parent capability. Therefore, the components of a child capability can be seen as the union of its components — beliefs, goals, and plans — with its parent's components. Such parent's components can be accessed within the scope of the child capability, that is, the child capability can: (i) dispatch both external and internal parent's goals; (ii) access and update parent's beliefs while executing its plans; (iii) have a goal handled and achieved by the parent's plans; and (iv) handle and achieve parent's goals. This full access to the parent capability's components by the child capability is shown in Figure 3b. The parent capability, in turn, is not aware that there are capabilities that extend its behaviour.

We will now illustrate a situation where inheritance may be used in the context of the development our intelligent robot. Assume that we have physical robots, which are provided with some basic features, such as walking, moving arms, and so on, so that they are able to perform different household duties, depending on the software deployed on them. We are developing robots for both helping in homes and working on laundries. The `Laundry` capability should have plans to wash clothes in the wash machine and to hand washing, if the robot is for helping at home and, and if it will work on laundries, it should also have components to dry cleaning. Therefore, two capabilities may be designed: `Laundry` and `ProfessionalLaundry`. The latter extends the former, adding new beliefs, goals, and plans needed to provide the dry cleaning functionality.

4 Using Capability Relationships

Given that we presented the three capability relationships, we illustrate their use in this section. We gave examples of their individual use in the previous section within the same context, the intelligent robot example. In Section 4.1, we combine the examples previously given by providing a big picture of the design of our intelligent robot. In Section 4.2, we provide further examples of the use of capability relationships in the context of transportation.

4.1 Intelligent Robots

We provided many examples in the context of robot development, where the capability relationships may be applied to modularise agent concerns. We now present an integration of these different examples to show how relationships can be used together in the development of agents. An overview of the design of the intelligent robots example is presented in Figure 4. This is an overview, and therefore this figure does not correspond to the complete design of a system, many agent components are omitted.

We use a simple notation. Capabilities are represented with rectangles, split into four compartments: (i) capability name; (ii) goals; (iii) beliefs; and (iv) plans. For relating capabilities, we use the notation previously introduced. And we represent agents with ellipses, and an agent is an aggregation of capabilities.

The `Laundry` capability provides the basic functionality for washing clothes, and it is extended by the `ProfessionalLaundryCapability` — an instance of the latter adds the ability of dry washing to the former. The `Laundry` capability is associated with the `Transportation` capability, so that the `Laundry`

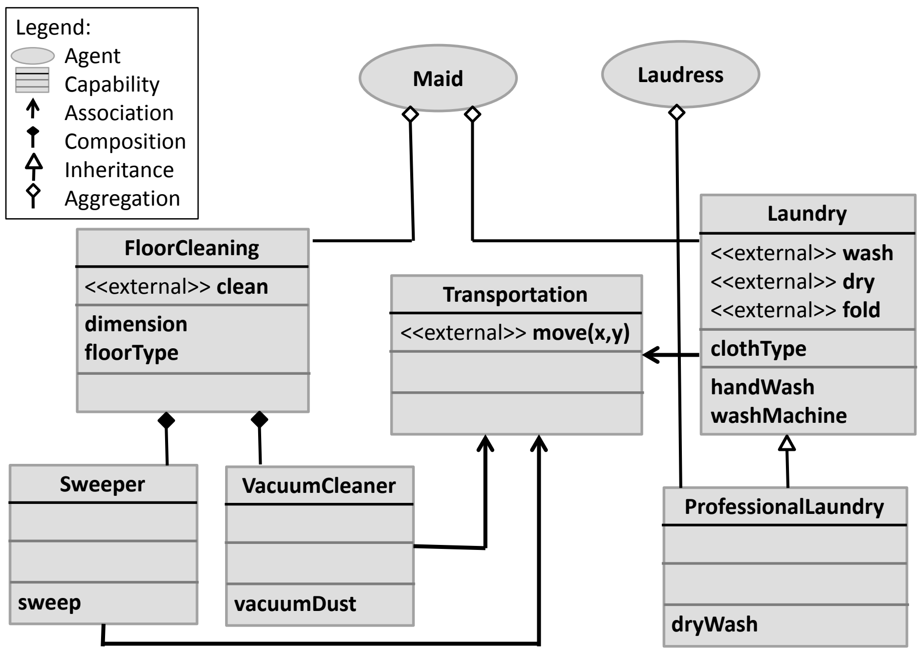


Fig. 4: Example: Intelligent Robots.

capability can dispatch goals related to transportation. Note that, because the `ProfessionalCapability` capability extends the `Laundry` capability, it also inherits the association.

The `FloorCleaning` capability has a goal (*clean*), which is not handled by any plan within this capability. It is, however, composed of two other capabilities, each having a plan that can achieve it, so that they can be selected to achieve the *clean* goal when appropriate (remember that these capabilities have other omitted beliefs, goals and plans). The execution of plans of the `Sweeper` and `VacuumCleaner` capabilities also needs goals related to transportation to be achieved, thus both of them are associated with the `Transportation` capability.

These capabilities are the building blocks to develop agents. A `Maid` agent (that is used to help at home) is an aggregation of both the `FloorCleaning` and `Laundry` capabilities, so that is can perform tasks related to them. The `Laudress` agent (who performs duties at laundries) must be able to perform other tasks related to washing clothes, therefore it is an aggregation of the `ProfessionalLaundry` capability, which in turn inherits the behaviour of its parent capability.

4.2 Driver Agents

We now will introduce a second example, which is in the transportation context. The objective is to design agents able to drive cars and motorcycles. As above,

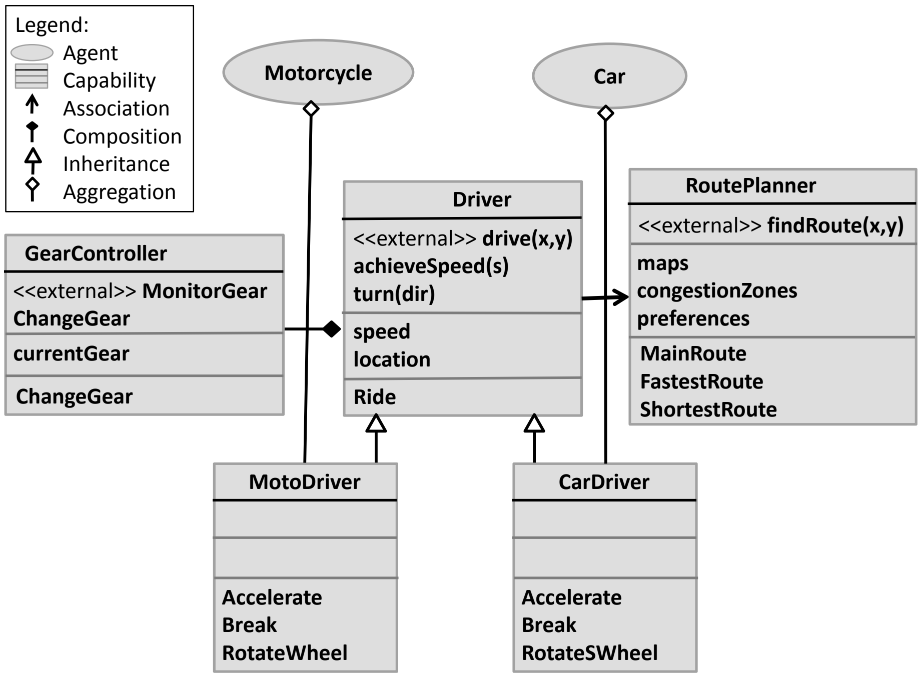


Fig. 5: Example: Intelligent Robots.

we will show an overview of the design, highlighting important parts of it, and omitting details. This example is illustrated in Figure 5.

The key functionalities associated with driving are implemented as part of the `Driver` capability, which has beliefs with respect to the current speed and location, an external goal $drive(x, y)$ that it is successfully achieved when the agent has driven from location x to location y, and internal goals dispatched by plans whose aim is to achieve the $drive(x, y)$ goal. There are two extensions of this capability: `MotoDriver` and `CarDriver`, which specialise the `Drive` capability to add behaviour specific to driving a motorcycle and a car, respectively. Besides other omitted details, each has its own plans to perform similar tasks, such as accelerating.

To drive from a location x to y, the `Driver` capability must first find a route between these two locations. This is modularised into the `RoutePlanner` capability, which has knowledge needed to calculate a route (maps, congestion zones, agent preferences, etc.), and different plans to find a route. To be able to find the route, the `Driver` capability has an association with the `RoutePlanner` capability, and consequently it can dispatch the $findRoute(x, y)$ goal.

Finally, there is a complicated part related to driving, which is the control of gears. This can be modularised in a separate capability, which needs specific beliefs, goals and plans to do so. However, it also needs the knowledge that is part of the `Driver` capability, and consequently there is a composition relationship between the `Driver` and `GearController` capabilities.

In order to build agents able to drive a motorcycle or a car, an agent must aggregate the `MotoDriver` capability or `CarDriver` capability, respectively.

5 Discussion

In this section, we discuss relevant issues with respect to the described capability relationships. We first analyse them, point out their main differences and the impact of choosing one or another in Section 5.1. In Section 5.2, we describe details of how capabilities are implemented in existing BDI platforms, and which kind of capability relationship they provide. We next discuss in Section 5.3 other object-oriented concepts, and how they are related to the presented relationships.

5.1 Relationship Analysis and Comparison

We have presented three different kinds of relationships between capabilities, and understanding their differences in order to be able to choose one to be used in agent design is important. We thus in this section make this discussion.

First, a key difference among these relationships is their purpose. Associations should be used when different *independent* agent parts *collaborate* to achieve a higher level goal. This is similar to collaborations among agents, but capabilities are within the scope of a single agent, i.e. a single thread. Therefore, it is a design choice to develop two agents, each of which with one capability and collaborating through messages, or to develop a single agent with two capabilities, collaborating by dispatching goals to be achieved by the other capability. Composition is adopted when the agent behaviour can be decomposed into modular structures, but parts *depend* on the whole, providing the notion of a *hierarchical* structure. And inheritance is used when there is a need for *reusing* a common set of beliefs, goals and plans, and then specialising it in different ways.

According to software engineering principles, the lower the coupling between capabilities, the better. Additionally, components of each capability should have high cohesion. These presented relationships have different degrees of coupling between the involved capabilities, so consequently relationships that reduce coupling should be preferred, when possible. We summarise this comparison of the relationships — discussed in the previous sections — in Table 2, which also indicates the visibility of components of capabilities involved in the relationships. For example, when there is an association relationship, the whole-capability has access to the part-capability's beliefs, while the part-capability has access to the whole-capability's beliefs, external goals and internal goals. We also emphasise the purpose of each relationship. Therefore, choosing a certain capability relationship is a design choice that not only implies restrictions over the visibility of the capability components, but also expresses the meaning of the relationship.

Now, we will focus on the impact at runtime of choosing different capability relationships. When a capability has access to components of another capability,

			Association	**Composition**	**Inheritance**
Purpose			Collaboration	Decomposition	Extension
Coupling			+	++	+++
Visibility	Source/ Whole/ Parent	Beliefs		X	X
		External Goals		X	X
		Internal Goals		X	X
		Plans			X
	Target/ Part/ Child	Beliefs			
		External Goals	X	X	
		Internal Goals			
		Plans			

Table 2: Relationship Comparison (1).

		Whose goals can be dispatched within the scope of this capability?	**Whose goals can be achieved by this capability's plans?**
Association	Source	Source's goals Target's goals (external only)	Source's goals
	Target	Target's goals	Target's goals
Composition	Whole	Whole's goals Part's goals (external only)	Whole's goals
	Part	Part's goals	Part's goals Whole's goals
Inheritance	Parent	Parent's goals	Parent's goals
	Child	Child's goals Parent's goals	Child's goals Parent's goals

Table 3: Relationship Comparison (2).

it may use these components at runtime. The access to beliefs is already shown in Table 2, and this means that a capability can use and modify knowledge to which it has access. Besides accessing other capability's knowledge, a capability involved in a relationship may: (i) dispatch goals of another capability when one of its plans is executing; and (ii) execute a plan to achieve a goal of another capability. We show when these two possibilities can happen in Table 3, which are associated with goal visibility. For example, if a whole-capability (of a composition relationship) dispatches one of its goals, this goal may be achieved by the execution of a whole-capability's plan or a plan of any the part-capabilities (and their parts).

5.2 Capabilities in Existing BDI Platforms

In Section 2, we introduced three BDI agent platforms that provide the capability concept. We will now discuss how each of these platforms provide capability relationships.

JACK The JACK platform *explicitly* provides a single type of relationship: composition, allowing the construction of a hierarchical structure. Nevertheless, its interpretation is not the same as that adopted in this paper. When this relationship is declared, the visibility of the involved capabilities' components should also be specified. Beliefs may be imported (i.e. shared with its enclosing agent or capability), exported (i.e. accessible from its parent capability), or private (i.e. local to the capability). Events have the role of goals in JACK, and in this platform capabilities should explicitly declare the kinds of events that it is able to handle or post. When declaring this information, an `exports` modifier is used to indicate whether events are to be handled only within the scope of the capability or by any other capability.

Although using these modifiers increases the flexibility of the platform, it goes against the principle of information hiding. When a belief is exported, any other capability can access it, and this increases the possibility of breaking the code. Although in object-orientation sometimes attributes are exposed through getters and setters, this still preserves encapsulation, as a getter hides if the value being returned is the value of an attribute or something else. The semantics of handling exported events is similar to that we adopt with the goal visibility in compositions.

Note that using solely capability compositions results in limiting capabilities to be used as hierarchical structures.

Jadex Jadex extended the capability concept of JACK [4], providing a model in which the connection between an outer and an inner capability is established by a uniform visibility mechanism for contained components. The implemented relationship type is also composition, but it is more flexible by allowing the declaration of abstract and exported components.

In Jadex, any component (beliefs, goals, plans and so on) can be used only internally, if no modifier is specified. They can be exported, and thus accessed outside the capability scope. In addition, they may be declared as abstract, and be set up by an outer capability. This way of modelling capabilities is similar to that discussed above, and have the same issues.

Jadex was recently extended[6] by changing its implementation based on XML files to an implementation based on pure Java, as BDI4JADE, making an extensive use of Java annotations. This makes the implementation of capabilities more flexible, as all object-oriented features can be used.

BDI4JADE BDI4JADE provides a flexible implementation as it is implemented in pure Java. Goals are declared as Java classes, and therefore can be used in different capabilities. Moreover, Java modifiers can be used to limit goal visibility, for instance, by using a package visibility.

As the other two agent platforms discussed, it implements only the composition relationship. However, beliefs are always private to the capability, or

[6] `http://www.activecomponents.org/`

accessible by its included capabilities. A goal is dispatched in a plan with a specification of its scope. There are two possibilities: (i) it can be handled by any plan of any capability; or (ii) it can be handled by the capability whose plan dispatched the goal, or any other included capability. Therefore, this implementation is the closest to the composition relationship described here.

It is also possible to extend capabilities in BDI4JADE as capabilities are Java classes. However, if the belief base or plan library of the parent capability is overridden by the child capability, the inheritance will loose its meaning.

5.3 Further OO Concepts

In this paper, we propose the use of relationships from object orientation to improve the modularity promoted by capabilities. This is just one of the object-oriented mechanisms that support the construction of high-quality software systems from a software engineering point of view. In this section, we discuss other mechanisms that may be adopted.

First, attributes and methods are always associated with an explicitly specified visibility, which can be private, protected, or public. JACK and Jadex, as previously discussed, provide similar concept using the `export` keyword. Here, we do not propose to use of visibility modifiers, except for goals, because exposing capability's beliefs goes against the principles of encapsulation and information hiding. In some situations, it is needed, and we provide mechanisms that explicitly show why there is a need for sharing beliefs, i.e. when there is a whole-part structure, and the parts involved. Nevertheless, visibility may be helpful to restrict the access of part or child capabilities to components of the whole or parent capabilities, respectively.

Associations between objects usually have a cardinality specified. If this is also applied to capabilities, it will allow capabilities to be associated to more than one instance of a capability. However, dispatching a goal of any of these capabilities will produce the same effect, unless their fragments of knowledge have different states. But this is unreasonable. This is also the case of overriding components of extended capabilities, when using inheritance, or using abstract capabilities. We are not stating that any of these mechanisms should not be used, but they should be carefully analysed before being adopted in the context of capabilities, in order to evaluate their usefulness and their meaning.

Finally, configurations of how capabilities are structured can be investigated, so as to form design patterns [8], or anti-patterns that should be avoided, such as object-oriented code-smells [7].

6 Final Considerations

Modularisation plays a key role in software engineering and is crucial for developing high-quality large-scale software. However, it has limited investigation in agent architectures, or more specifically BDI agents. Our previous studies have shown that there is a lack of mechanisms that allow modularising fine-grained variability in BDI agents [11].

Capabilities are one of the most important contributions to allow the construction of modularised BDI agent parts, increasing maintainability and promoting reuse. Nevertheless, this concept could be further explored to provide more sophisticated tools to increase the quality of BDI agents from a software engineering point of view, and supporting the construction of large-scale multi-agent systems. In this paper, we investigated the use of three types of relationships between capabilities, which are association, composition and inheritance. Each of which has a particular purpose, and indicates specific access to its components. We showed examples of their use, and discussed the implications of each relationship at runtime. Although some BDI agent platforms provide mechanisms to emulate these relationships, by means of the exportation of capability's components, they are not in accordance with the principle of information hiding. Furthermore, keeping track of all shared beliefs and capabilities that can handle goals may become an error-prone task, thus making agents susceptible to faults.

The main goal of this paper is to promote the exploitation of capability relationships and other mechanisms to develop large-scale modular multi-agent systems and discussion about this important issue of agent-oriented software engineering. In this context, this work has left many *open issues* to be further discussed, with respect to capabilities and modularisation into agent architectures: (i) does it make sense to add visibility to all BDI agent components? (ii) does it make sense to design and implement abstract capabilities? (iii) is there any situation where there should be cardinality in the association relationship? and (iv) what is the interface of an agent and of a capability?

References

1. Autonomous Decision-Making Software (AOS): Jack intelligent agents: Jack manual. Tech. Rep. 4.1, Agent Oriented Software Pvt. Ltd, Melbourne, Australia (2005)
2. Bordini, R.H., Wooldridge, M., Hübner, J.F.: Programming Multi-Agent Systems in AgentSpeak using Jason. John Wiley & Sons (2007)
3. Bratman, M.E.: Intention, Plans, and Practical Reason. Harvard University Press, Cambridge, MA (1987)
4. Braubach, L., Pokahr, A., Lamersdorf, W.: Extending the capability concept for flexible bdi agent modularization. In: Proceedings of the Third International Conference on Programming Multi-Agent Systems. pp. 139–155. ProMAS'05, Springer-Verlag, Berlin, Heidelberg (2006)
5. Bresciani, P., Perini, A., Giorgini, P., Giunchiglia, F., Mylopoulos, J.: Tropos: An agent-oriented software development methodology. Autonomous Agents and Multi-Agent Systems 8(3), 203–236 (2004)
6. Busetta, P., Howden, N., Rönnquist, R., Hodgson, A.: Structuring bdi agents in functional clusters. In: 6th International Workshop on Intelligent Agents VI, Agent Theories, Architectures, and Languages (ATAL). pp. 277–289. ATAL '99, Springer-Verlag, London, UK, UK (2000)
7. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1999)

8. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-oriented Software. Addison-Wesley (1995)
9. Garcia, A., Lucena, C.: Taming heterogeneous agent architectures. Commun. ACM 51(5), 75–81 (May 2008)
10. Howden, N., Rnnquista, R., Hodgson, A., Lucas, A.: Jack intelligent agents$^{\mathrm{TM}}$: Summary of an agent infrastructure. In: The Fifth International Conference on Autonomous Agents. Montreal, Canada (2001)
11. Nunes, I., Cirilo, E., Cowan, D., Lucena, C.: Fine-grained variability in the development of families of software agents. In: Sabater-Mir, J. (ed.) 7th European Workshop on Multi-Agent Systems (EUMAS 2009). Cyprus (December 2009)
12. Nunes, I., Lucena, C., Luck, M.: Bdi4jade: a bdi layer on top of jade. In: ProMAS 2011. pp. 88–103. Taipei, Taiwan (2011)
13. Padgham, L., Lambrix, P.: Formalisations of capabilities for bdi-agents. Autonomous Agents and Multi-Agent Systems 10(3), 249–271 (May 2005)
14. Penserini, L., Perini, A., Susi, A., Mylopoulos, J.: From capability specifications to code for multi-agent software. In: 21st IEEE/ACM International Conference on Automated Software Engineering. pp. 253–256. ASE '06, IEEE (2006)
15. Pokahr, A., Braubach, L., Lamersdorf, W.: Jadex: A bdi reasoning engine. In: Multi-Agent Programming. pp. 149–174. Springer (9 2005)
16. Rao, A.S., Georgeff, M.P.: BDI-agents: from theory to practice. In: Proceedings of the First Intl. Conference on Multiagent Systems. San Francisco (1995)
17. Sant'Anna, C., Lobato, C., Kulesza, U., Garcia, A., Chavez, C., Lucena, C.: On the modularity assessment of aspect-oriented multiagent architectures: a quantitative study. Int. J. Agent-Oriented Softw. Eng. 2(1), 34–61 (Jan 2008)