Fine-grained Variability in the Development of Families of Software Agents

Ingrid Nunes[1], Elder Cirilo[1], Donald Cowan[2], and Carlos J.P. de Lucena[1]

[1] PUC-Rio, Computer Science Department, LES - Rio de Janeiro, Brazil
{ionunes,ecirilo,lucena}@inf.puc-rio.br
[2] University of Waterloo - Waterloo, Canada
dcowan@cs.uwaterloo.ca

Abstract. This paper contains a presentation of an exploratory study of the development of a family of software agents, in which we adopted techniques to support the construction of agents using reusable assets that address domain variability. This agent family was built using a Software Product Line (SPL) architecture with fine-grained variability. We describe the development of our case study, the Buyer agent SPL, and report on lessons learned primarily related to: what variable types were handled, how current Multi-agent System (MAS) methodologies are able to document variability and provide mechanisms to enable software reuse, and what implementation techniques were adopted.

Key words: Multi-agent Systems, Software reuse, Software product lines, Fine-grained variability

1 Introduction

Multi-agent Systems (MASs) synthesize contributions from different areas, including artificial intelligence, Software Engineering (SE) and distributed computing. In the context of SE, MASs are viewed as a paradigm, whose main idea is to decompose complex and distributed systems into autonomous, pro-active and reactive agents with social ability. Such properties are particularly appropriate in the development of modern software systems, which tend to be open, distributed and situated in dynamic environments. However, the state-of-the-art of Agent-oriented Software Engineering (AOSE) is insufficiently reflected in the state-of-practice in developing complex distributed systems [1]. One of the reasons for this is the poor connection between agent research and mainstream SE. While the SE community cares about modularity, stability, reusability and maintainability when developing applications, the MAS community has shown little interest in building systems following these SE principles.

In previous work [2], we have proposed building customized service-oriented agents using a Software Product Line (SPL) approach. SPLs [3] are a new software reuse approach that aims at systematically deriving families of applications based on a reusable infrastructure with the intention of achieving both reduced costs and reduced time-to-market. This work has evolved from previous

research [4], which aimed at documenting and modeling multi-agent SPLs with a focus on coarse-grained variability. Current research that aims at integrating MASs and SPLs has not dealt with fine-grained variability, i.e. variability in an agent architecture, such as optional and alternative beliefs, goals and plans. Fine-grained variability is essential when extracting features from legacy applications. Furthermore, some existing SPLs could benefit from fine-grained variability to reduce code replication or improve readability [5].

Given that an SPL architecture must address variability within a domain, it is essential to adopt techniques to modularize variable portions of the architecture, thus enabling the reuse of these assets in different product configurations. This is particularly challenging when dealing with fine-grained variability. It is important to rely on implementation techniques that support modular configuration of the variable parts. Otherwise, the stability of an agent SPL architecture will naturally decay over time and this instability will be perpetuated through all future generations of MAS product architectures. Nevertheless, during the development of our case study of [2], we have identified several issues in the state-of-the-art AOSE to allow building modularized agent SPL architectures with reusable assets.

The focus of this paper is to report and discuss lessons learned during the development of our case study, mainly related to the lack of techniques that support building agent architectures that take into account SE principles. We present an exploratory study of the development of a family of software agents, in which we aim at adopting appropriate techniques to build agents using reusable assets. The main objective of this study is to explore how parts of an agent architecture can be modularized and be made sufficiently generic in order to be reused. In particular, our study focuses on agents that follow the BDI architecture [6], which is widely used for developing cognitive agents. In addition, we focus on web-based systems with some components that are software agents. Many software systems are not operating in isolation, but are in a distributed and dynamic environment like the web, where new problems such as trust and coordination between components become important. Even though MASs have characteristics that may be appropriate to solve these problems, alternative technologies such as Service-oriented Computing (SOC) are being chosen, because of a lack of reusable agent assets [7]. Based on our case study, we discuss issues that arose during its development including (i) variability types we encountered during the SPL development; (ii) expressiveness of existing agent models; and (iii) techniques adopted to cope with SPL variability.

The paper is organized as follows. Section 2 presents related work. Section 3 describes our case study, the family of buyer agents. Section 4 is a presentation of lessons learned with this exploratory study. Section 5 concludes this paper.

2 Related Work

In this section, we present work related to the research reported in this paper. It consists of approaches that have been proposed to promote software reuse in

MASs. However, most of MAS approaches do not adopt extensive reuse practices that provide both reduced time and costs for software development [8].

The first initiatives on exploiting software reuse in MAS were the proposals of pattern reuse [9]. They provide a solid basis for improving the MAS development from a SE perspective. Nevertheless, they are somewhat deficient as: (i) several of the proposed patterns are related to protocol definition, therefore they do not solve the problem of defining part of an agent architecture given a specific context; and (ii) overall organizational structures are presented as patterns in [10], but there is a gap between the textual description of the pattern and its implementation. This final step is very dependent on the experience of the designer.

Recent approaches have focused on the integration of SPLs and MASs, namely Multi-agent Systems Product Liness (MAS-PLs). In [11] an approach is proposed to build the core architecture of a MAS-PL based on the composition of role models. However, the approach deals with model composition and not implementation composition. [12] proposes an extensible agent-oriented requirements specification template for distributed systems that supports safe reuse, This approach only deals with coarse-grained variability, which was discussed in our domain engineering process for MAS-PLs [4] in which we have made empirical studies of MAS-PL implementation techniques [13]. The exploratory study presented in this paper deals with different variability granularity, e.g. belief and plan parameters, and is concerned with adopting new strategies to improve the agent architecture not just strategies using available platforms.

3 The Buyer Agent Family Case Study

In this section, we describe our case study. Our exploratory study consists of the development of a family of agents in the domain of electronic commerce. This family is composed of buyer agents referred to as Buyer agent SPL in the remainder of the paper. The main idea is to develop an agent architecture that supports domain variability. When a user makes a request to buy a product with a specific configuration, a customized buyer agent is derived from this architecture according to the configuration and it buys, or tries to buy, the requested product. A configuration is a selection of a valid set of variation points and variants. For instance, in a car product line, there may be two variation points: optional air conditioning and the alternative between a manual or automatic transmission. In this case, a configuration could be air conditioning and automatic transmission.

3.1 e-Marketplace Overview

The domain of electronic commerce is a typical application domain of MASs. Some commercial decision-making can be placed in the hands of agents. Our exploratory study focuses on the buyer agent, i.e. the internal structure of a customizable buyer agent that enters into the MAS and interacts with other

agents to achieve its goals. In this section, we present the MAS to which this Buyer agent belongs.

Figure 1 depicts the overall structure of the e-Marketplace MAS, in which there are four main organizations. The *e-Marketplace* organization contains buyers and sellers, which interact in order for a seller to sell a product to a buyer. In the *Payment Services* organization, there are two agents, the `PayPal` and `CreditCardCompany`, which provide the necessary services for paying for a product. The *Geographic Services* organization is composed of the `Map` agent that calculates the distance between two locations. The fourth organization is the *Ipagent*. This organization is composed of agents of the *Ipagent* system. The system structure is based on the Web-MAS architectural pattern [14], whose aim is to integrate agents into a web-based system structured with the typical layered pattern. The `User` agent acts on behalf of users. It is composed of services provided for users, including the buy service, which is responsible for deriving customized `Buyer` agents.

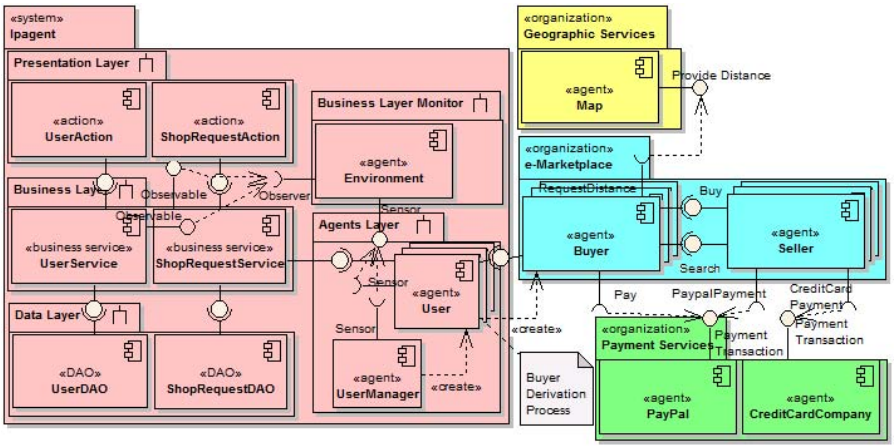

Fig. 1. e-Marketplace MAS.

Our focus is not to address the development of the whole MAS. Organizations representing stores and other companies are already deployed in the system and ready to interact and provide services to other agents. This existing MAS already provides an ontology giving a formal representation for a set of concepts, which includes the messages exchanged by agents within the domain and protocols related to the exchange of messages.

3.2 Buyer Agent SPL Architecture

The buyer agent represents a user in the *e-Marketplace* organization. Given that users are individuals, their preferences should be part of the buyer agent so

this agent can act appropriately. Thus, we have analyzed the domain related to buying products and we have identified the variable aspects of the domain, and the goals and subgoals of the Buyer agent SPL. These points of variability are taken into account while designing and implementing the agent SPL architecture. Consequently, it is possible to derive customized buyer agents based on a configuration provided by the user.

One could say that it is not necessary to derive specific agents to present customized behavior, and introducing parameters into the architecture is enough. This is not a good solution because these parameters are control variables, which is a program variable that is used to regulate the flow of control of the program. They introduce a control coupling in the system and, if we are dealing with large scale software, it is hard to understand and maintain the code. We agree that this case study is not sufficiently large enough to illustrate adequately the problem of introducing control variables, but it provides the necessary variability scenarios to be modularized, which is our main interest.

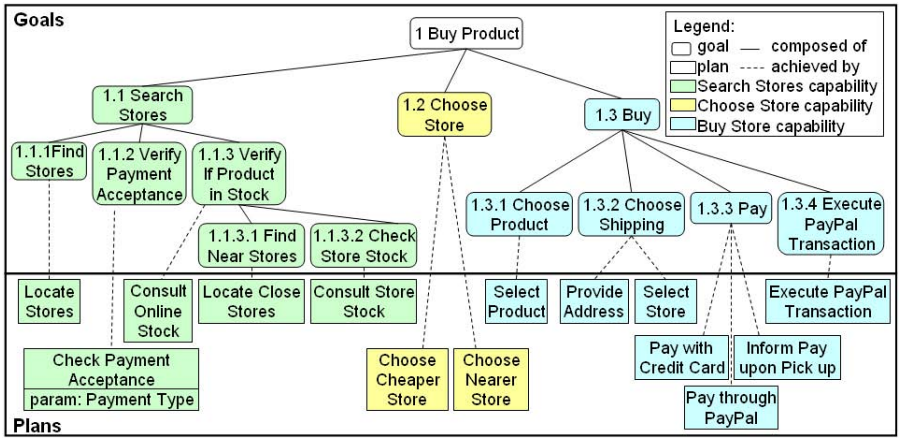

Fig. 2. Buyer Agent SPL – Goals and Plans.

The Buyer agent SPL was structured using the belief-desire-intention (BDI) architecture [6], as the agents are proactive and have goals. In addition, this architecture has been thoroughly analyzed, and implemented by several agent platforms. Based on these considerations, we designed our agents based on the concepts of goals (previously identified), beliefs and plans.

The variation points that we identified and considered in the Buyer agent SPL are: (i) *Payment Type*, with variants: Credit Card, Pay Pal and Pay upon Pick up; (ii) *Shipping Type* with variants: Ground Shipping and Pick up at Store; and (iii) Store Selection Strategy, with variants: cheaper and faster. In addition, constraints were defined in order to allow only the selection of valid variants sets,

e.g. the Pay upon Pick up variant can only be selected if the Pick up at Store variant is selected.

Besides the variation points, we have identified the goals and subgoals of the buyer agent. These goals are depicted at the top of Figure 2. A set of subgoals must be achieved in order to reach the parent goal. Domain variability was also considered while performing this goal decomposition. For instance, the goal *Verify If Product in Stock* can be achieved either by a set of actions that verifies the online stock of a seller (*Consult Online Stock* plan), in the case where "Ground Shipping" is chosen, or by achieving two subgoals (*Find Near Stores* and *Check Store Stock*) to verify the stock of a store of the seller, in the case where "Pick up at Store" is chosen. A (sub)goal may be optional, meaning that it will be part of the agent only if the variant related to the goal is selected for the agent being derived, such as the *Execute PayPal Transaction* goal.

The plans for the Buyer agent that achieve its goals are shown at the bottom of Figure 2. Note that some goals can have different plans. Based on a Buyer agent configuration, the plans related to the selected variants will be chosen to be part of the derived agent. The complete implemented agent architecture is presented in Figure 4.

3.3 Variability Implementation Techniques

The Buyer agent SPL was implemented with the Jadex agent platform, which follows the BDI architecture. Jadex supports programming software agents in XML and the Java programming language. An agent is defined in an XML file, named Agent Definition File (ADF), which specifies the agent's beliefs, goals and plans. An ADF can also contain the definition of other concepts that help with the agent implementation such as messages that can be sent and received. Although plans are declared in the ADF, their body is implemented in Java classes, which extend the `Plan` class. In order to support the variability of the Buyer agent SPL, we have adopted some implementation techniques, which are described next.

Goal Decomposition and Plan Modularization. The variability modularization starts in the analysis phase, while identifying goals and decomposing them into subgoals. When this decomposition is performed, some goals may be alternatives or optional, such as the *Find Near Stores* and *Check Store Stock*, which are related to the Ground Shipping variant. In addition, plans are modularized in such a way that each of them is either mandatory or corresponds to one single variant. The goal decomposition helps with this modularization because the finer-grained the goals, the more specific the plans.

Plans Parametrization. One feature of Jadex is to allow passing of parameters to plans. The goal *Verify Payment Acceptance* could be achieved by three different plans, each corresponding to one payment type. However, the only difference in these plans would be a parameter passed in a message. Therefore, we adopted the technique of passing parameters to plans of

Jadex to reuse the same plan for the different payment types where there were three different parameters.

Capabilities. A capability is essentially a set of plans, a fragment of the knowledge base that is manipulated by those plans and a specification of the interface to the capability. This concept is implemented by JACK and Jadex agent platforms. Capabilities have been introduced into some MASs as a SE mechanism to support modularity and reusability while still allowing meta-level reasoning. We used the capability concept in order to encapsulate beliefs, goals and plans related to a certain concern, such as searching stores. Therefore, we have modularized related concepts into a component, the capability, which can be easily (un)plugged from the agent and reused in other agents.

Conditional Compilation. The last adopted implementation technique is conditional compilation. The Buyer agent SPL architecture has optional and alternative parts that were not modularized in specific code assets, mainly because all beliefs, goals and plans must be declared in ADFs. For instance, even though the *Pay* goal is achieved by three different plans, i.e. Java classes, the plan must be declared in the ADF. Therefore, the three different plans are declared in the ADF with tags surrounding them indicating the variant related to this XML code fragment. With this information, it is possible to remove the fragments that are related to unselected variants before compiling the code. This technique is also adopted in Java class files. When a goal is decomposed into subgoals, a plan is created in order to dispatch the set of subgoals, but some of these subgoals may be optional. In this case, tags surrounding the dispatch of the subgoal are introduced in the code in order to make conditional compilation possible.

3.4 Automatically Deriving Buyer Agents

Buyer agents are automatically and dynamically derived from the Buyer agent SPL during the execution of the *Ipagent* web-system. When a user wants to buy a certain product, that user must configure, through the web interface of the system, a buy request by choosing the variants of the SPL variation points and the desired product. After doing that, the `Environment` agent detects this business operation and propagates it to the `User` agent. This agent is responsible for deriving the customized `Buyer` agent and starting it. Figure 3 depicts the process performed by the `User` agent in order to derive `Buyer` agents. We now describe this process.

The first step of the derivation process is to produce customized source code. Three inputs are necessary for this task: (i) configuration knowledge – it is part of the `User` agent's belief base namely the knowledge of which code assets are related to which variants; (ii) "markup" code – the tags described in the last section indicate which code fragments are related to which variants; and (iii) the user configuration. With these inputs, the `User` agent first loads code assets that have been selected based on the configuration knowledge and the user configuration, then removes the code fragments related to the unselected

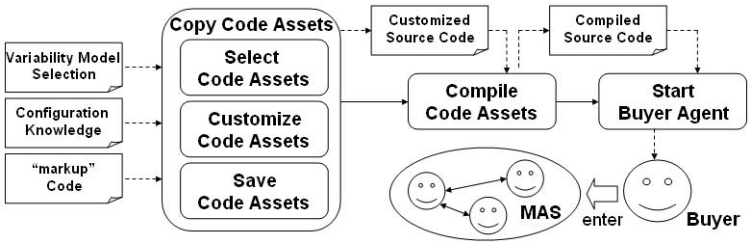

Fig. 3. Buyer Agent Derivation process.

variants and finally saves the customized source code. The code is then compiled and the `Buyer` agent is started. When the derived `Buyer` agent is operational, it sends a message to the `User` agent indicating its operational status. The `User` agent then requests the `Buyer` agent to buy the product that the user wants. After finishing buying the product or realizing that the purchase is not possible, the `Buyer` agent informs the `User` about the success or lack of success while pursuing the buy request, and then dies.

4 Lessons Learned

In this section, we present and discuss lessons learned from our experience of developing the Buyer Agent SPL. In order to build it, we adopted techniques that support domain variability and allow to reuse SPL assets to derive different buyer agent configurations. The lessons learned are related mainly to the following points: variability types that occurred (Section 4.1), how current MAS methodologies are able to document variability types and provide mechanisms to enable software reuse (Section 4.2), and the implementation techniques adopted in our case study (Section 4.3). In addition, we also discuss the challenges that we have faced using the Jadex platform (Section 4.4).

4.1 Variability Types

Each agent of a MAS may be classified based on two different perspectives: (i) internally as a software system with its own purpose (*intra-agent*); and (ii) externally as part of a society interacting with other individuals (*inter-agent*). Figure 1 shows the inter-agent view of our study, presenting the agents and organizations that are part of the MAS and how they interact. Figures 2 and 4 depict the internal structure of the Buyer agent SPL. In this study, our focus is exploring this intra-agent viewpoint and the variable part of its structure, i.e. our focus is to deal with fine-grained variable structure not coarse-grained, as in an optional agent.

While developing the Buyer agent SPL, we have identified the following variable structures:

- *Capabilities:* we used the capability concept provided by Jadex to aggregate beliefs, goals and plans that are related to a specific concern, such as search stores and buy product. Another capability that could be part of the Buyer agent SPL is the negotiate capability, which would aggregate concepts to provide means for the agent to negotiate prices with sellers. Thus, capabilities can be optional or alternative in an agent SPL architecture.
- *Beliefs:* agents' beliefs, in the BDI architecture, influence the two activities of practical reasoning: (i) deliberation – the activity of deciding what goals the agent wants to achieve; and (ii) means-ends reasoning – the activity of deciding how to achieve these goals. Therefore, a belief must be part of the agent knowledge base if it participates in at least one of these activities. In our study, the knowledge about the product store varies according to the shipping type and the choose store strategy. If the product is to be shipped and the strategy is choose cheaper store, the agent must know the sellers that have the product in online stock, while if the user is to pick the product up at a store and the strategy is choose nearer store, the agent must know the different stores that have the product in stock and their location. Consequently, the beliefs of an agent may also vary.
- *Goals:* As our study illustrates, subgoals for achieving a goal may be different when dealing with different variants. An example is the *Verify If Product in Stock* goal, which can be achieved either by a plan or decomposed into two subgoals. Therefore, there are two optional subgoals. In the Buyer agent SPL there are no alternative goals, however we believe that it is certainly a possible point of variation.
- *Plans:* In the same way that a goal may be decomposed into different sets of subgoals, the goal can be achieved by different plans. Thus, these different plans are alternative descriptions. In addition, there are optional plans – if a goal is optional, the plan that achieves the goal is also optional.
- *Plan parameters:* As discussed in Section 3.3, we used the plan parametrization provided by Jadex. In the *Check Payment Acceptance* plan, there are alternative parameters that are given as input of a plan. Thus, Jadex allowed us to reuse the actions of the plan to implement three different variants.

Figure 4 shows how these variable structures are present in the Buyer agent SPL. We have adopted UML notation with stereotypes to represent agents, beliefs, goals, plans and capabilities. Each color in the figure represents a different variant and the white color indicates that the element (or fragment of an element) is present in all agent configurations. It should be noticed that each plan can be related to only one variant because of the goal decomposition. However, other variability types are tangled and spread throughout the capabilities.

4.2 Documenting and Modeling Variabilities

Several MAS approaches have been proposed in order to analyze and design MASs, such as methodologies and modeling languages. However, these approaches provide models based on concepts that do not necessarily have counterparts

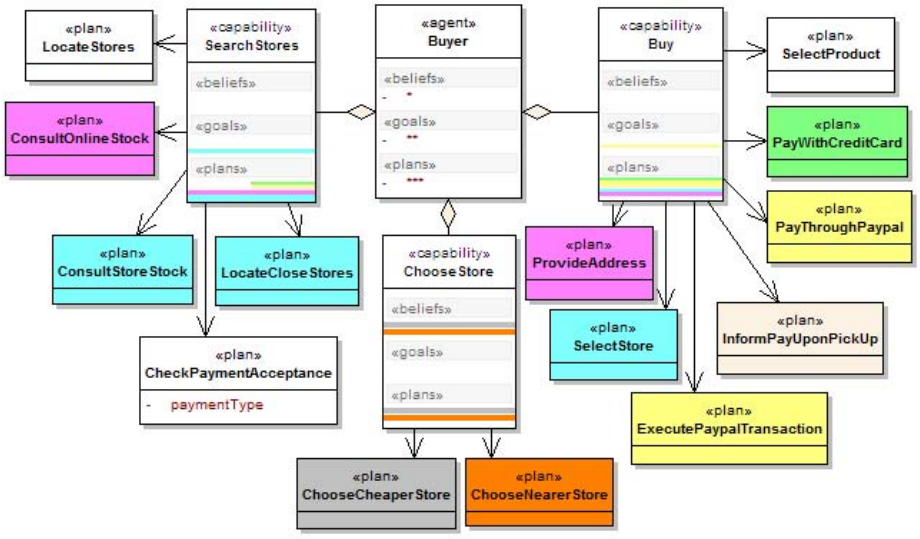

Fig. 4. Buyer Agent SPL Architecture and its Variabilities.

in the implementation platform. As a consequence, the effort spent developing good design models, i.e. taking into account principles such as modularity and reusability, may not be worthwhile, because these principles may not be reflected in the code. This gap between design and implementation models makes code understanding harder because elements in the implementation do not correspond directly to elements in the design. Consequently, it also makes it difficult to maintain and evolve the code.

In the past few years, AOSE community has been proposing approaches that take advantage of Model-driven Architectures (MDA) in order to bridge the gap between the design and implementation of MASs [15]. In a nutshell, the MDA approach defines system functionality using a platform definition model (PIM) using an appropriate domain-specific language. Then, given a platform-independent model (PDM) corresponding to a specific platform, the PIM is translated to one or more platform-specific models (PSMs) that computers can run. Therefore, models designed with abstractions defined in a meta-model of a specific methodology can be automatically translated to a model that describes the same system in terms of platform-specific abstractions.

However, besides this issue, we have identified other deficiencies in MAS methodologies and modeling languages. They are mainly related to the lack of mechanisms to design reusable elements. Examples of such mechanisms are:

Goal reuse. Plenty of MAS approaches adopt the goal concept. Some of them associate agents with a list of goals and others represent goals as a tree, meaning that children of a goal are the subgoals that must be achieved in order to realize the goal. This relationship is represented in Figure 2. How-

ever, subgoals may be necessary to achieve more than one goal. Therefore, goals have an `n:n` relationship, and not `1:n` as expressed in a tree structure. One methodology that allows modeling this `n:n` relationship is Tropos [16], which is based on the i* framework. Even though Tropos can be used to illustrate that a goal is used to achieve more than one goal, we believe that Tropos has scalability issues. Each concept in a Tropos model is represented by a node and each node may have several arrows connecting it to other nodes. As a consequence, in complex system scenarios, these models may become unreadable. In addition, Tropos models are powerful in the sense that they provide lots of information, such as goals and plans that are part of an actor (agent) and relationships among goals and plans. However, when a single model provides so much information, it may also compromise its readability. One solution for this problem is to define modules of the system in separate models, and provide different system views, which capture different aspects of the system (or a product line).

Capabilities. Capabilities are a mechanism that enables the modularization and reuse of a specific agent behavior. They are basically composed of the same concepts as agents, i.e. beliefs, goals and plans, however they must be incorporated into an agent in order to be part of a MAS. This concept was introduced by the JACK platform, but only a few approaches adopted capabilities as a first-class element. In MAS methodologies that do not present the capability concept, the only way of modularizing related concepts is to associate them with two or more different agents. Nevertheless, it may not be a good solution for two reasons: (i) semantically, the concepts must be part of the same agent. For instance, if in a MAS an agent A represents a person and this person plays the role of a mother and a teacher, agent A must aggregate the concepts related to both roles; and (ii) one must not forget that MASs are multi-threaded systems, and each agent has its own thread. Therefore, creating new agents in the system for modularity reasons may cause unnecessary overhead.

Plan parameterization. Jadex allows inputs to plans and this provides for the instantiation of plans in different contexts. This idea can also be considered in human behavior, because people usually have a pre-defined course of actions to accomplish some goals (plans) that are instantiated according to a context. For instance, when going to the movies, a plan can be buying the ticket, enter the theatre, and watch the movie. In this situation, the input parameters may be the cinema and the film. Even though plan parameterization is a mechanism that can be adopted at the implementation level, few MAS design approaches provide such parametrization. In Figure 4, we represent the plan parameter as a class attribute.

MASs are essentially inspired by different aspects of human nature, such as organizational and cognitive functions. This is interesting from a computer science viewpoint only if using approaches inspired by human behavior brings advantages to software development. Therefore, it would be interesting to eval-

uate if techniques that do not strictly follow human models may be introduced in current MAS approaches in order to improve software development.

4.3 Implementation Techniques and Variability Modularization

One major benefit of AOSE is that this paradigm supports decomposing a complex problem into autonomous agents, which communicate with each other by messages. A main difference between an agent and an object is that the former encapsulates not only data (its state), but also the behavior selection process and when such behaviors are necessary. This approach enables the construction of a system composed of components (agents) with lower coupling than objects. In addition, these components have a high cohesion while playing related roles within an organization. Nevertheless, these principles of low coupling and high cohesion are not typically taken into account while modeling and implementing an agent's internal structure.

The BDI architecture states that an agent has a set of beliefs, goals and intentions. An intention is a goal that an agent is committed to achieve, and is typically associated with a plan, which defines the actions necessary to achieve the goal. This structure is based on philosophical foundations inspired by human reasoning. However, if an agent has more than one responsibility, i.e. has goals that are related to different purposes, the concepts related to each of them will be mixed into the agent architecture, leading to code that is harder to understand and maintain. In our case study, for instance, if the `Buyer` agent were a `User` agent with different services, e.g. buy product and search the web, the beliefs, goals and plans associated with both services would be part of the agent, and there is no way of telling which service requires a certain belief. This scenario is also illustrated in Figure 4. Each capability is associated with a set of plans that are related to different variants (each color represents a different variant). However, some of these plans are related to the same variant, but this semantic relationship among them is not represented in the Buyer agent SPL. A first solution to this problem was the capability concept, introduced by the JACK platform and used in our study where all goals, beliefs and plans associated with the service of buying a product are encapsulated into a capability. The `Buy` capability aggregates the concepts needed for buying a product and can be reused in other agents that need to buy a product in the e-Marketplace MAS.

Moreover, with goal decomposition and plan modularization variants in the Buyer agent SPL could be modularized into single plans. This can be seen in Figure 4, in which all plans have only one color. However, given that all beliefs, goals and plans are part of an agent (or capability), and must be defined into ADFs, the code related to variants are tangled and spread throughout the ADFs. Even though conditional compilation solved the problem of managing variable structures, this technique is not a good practice because it leads to code and configuration knowledge that is hard to understand and maintain. Conditional compilation increases the complexity of the code, because a developer has to understand the logic of conditional compilation tags, as well as the logic that is already present in the code. The configuration knowledge, i.e. the relationship

between variants and implementation elements, is buried in the code and there is no way of telling the impact of a variant on the SPL architecture.

However, even though conditional compilation has these drawbacks, we have adopted this technique because the only alternative for modularizing variants is the use of capabilities. In such a solution, each variable part is modularized into a separate capability. Nevertheless, this technique would significantly increase the number of components of the agent architecture, increasing its complexity and the difficulty of managing it [17]. Consequently, we claim that there is a need to explore existing techniques or proposing new ones to provide mechanisms that allow the modularization of agent architectures, thus increasing the systems' reusability and maintainability. One example is the use of Aspect-oriented Programming (AOP) to modularize agent architectures [18]. AOP has been investigated in the context of SE as a technique to modularize cross-cutting concerns, i.e. modularize some stakeholder interests that cannot be modularized with the base paradigm.

In the context of SE, several approaches have been proposed in order to improve software architectures, following principles such as information hiding, encapsulation, reusability, maintainability, high cohesion and low coupling. An example of an approach that could be applied in the Buyer agent SPL is the use of a design pattern, namely the Abstract Factory [19]. The intent of this pattern is to provide an interface for creating families of related or dependent objects without specifying their concrete classes. In our SPL, we have alternative sets of plans for achieving a given set of goals. Alternative plans have the same pre- and post-conditions, i.e. the same interface. This is exactly the problem that the Abstract Factory pattern solves where our goal is to instantiate a family of plans according to a selected variant. By means of the Abstract Factory pattern it is possible to show the semantic relationship between plans and variants explicitly. In addition, this helps to manage the variable structure because using an abstract factory in the client code allows an interchange of concrete factories without impacting the client code. In addition, the inclusion of a new variant, i.e. a new payment method, would be easier because it only requires a new concrete factory and its associated plans. In this Buyer SPL example, there are only two plans associated with the factories, however the pattern would bring more benefits if there were more plans. Therefore, we claim that AOSE could learn from research work that has been done in state-of-the-art SE to design and implement better software architectures.

In addition to approaches that aim at improving software architectures, empirical SE provides techniques to evaluate if proposed approaches indeed bring the benefits they say they provide. This has been done in AOP community in order to identify situations in which aspects are an appropriate technique to solve a problem. Only few empirical studies have been done with agent-oriented systems, for instance [13]. As a consequence, there is little evidence of the real benefits of AOSE. Thus, it is interesting to propose not only new approaches but to evaluate them with empirical studies, and we believe this is a good way of comparing the plethora of existing agent methodologies.

4.4 Use of Jadex agent platform

In Section 3, we have introduced the Jadex platform, which was used to implement the Buyer agent SPL. In this section, we relate our experience using Jadex, and the challenges we faced. The main benefit of Jadex is that it provides the concepts of the BDI architecture for developers, therefore an agent modeled using this architecture may be directly implemented without defining an implementation strategy for the modeled concepts. In addition, the capability concept is very useful in modularizing parts of the agent. As a consequence, capabilities can easily be(un)plugged from agents and reused.

As previously stated, Jadex defines agents through XML files, and this creates problems during the implementation. Finding errors in XML files is a tedious task. Additionally errors are not caught during compilation, because typos may occur even though the document is valid according to its DTD. For instance, if a goal is referenced within the XML file and the name contains a typographical error, an error will occur only during execution, and the message is that the XML file has errors. As a consequence, the developer has to find the error manually. Moreover, even though plans are Java classes, beliefs and parameters are retrieved by methods that return an object of the class `Object`, so there must be type casting while invoking these methods. This leads again to the capture of errors only at runtime. Thus, the use of XML files is not appropriate for the support of modularization techniques. Therefore we are considering other agent platforms to determine if they provide solutions to these Jadex deficiencies.

5 Conclusion

MASs aim at developing complex, distributed systems in terms of high level abstractions in order to reduce the gap between the problem and solution spaces, thus facilitating communication with stakeholders. However, these models will not likely be adopted in the industry if they do not promote reduced time-to-market, lower costs and higher quality. Research work on software reuse has been addressing these issues for several years, nevertheless little research effort has been performed in this direction within the context of MASs.

This paper presented an exploratory study of the development of a family of buyer agents following the BDI model and using a SPL architecture. This architecture allows the derivation of customized agents from an existing MAS which is configured according to a user specification. Within the Buyer agent SPL, we have explored different variability granularity, including capabilities and fine-grained variable structures, such as beliefs, goals, plans and plan parameters. Based on our study, we presented and discussed important issues that arose during the development of the Buyer agent. These issues are mainly related to the lack of techniques, both at the design and implementation levels, to develop MASs based on traditional SE principles, such as modularity, reusability and maintainability. Several SE practices have been proposed over the last few decades to improve software development, and MAS approaches could learn from them.

In the future, we aim to investigate the use of SPL in order to change an agent configuration dynamically. In our study, all variable points are bound at compile time, therefore once the agent is derived, it cannot change its behavior to adopt a configuration with other variants. However, adopting techniques associated with a SPL architecture results in components with low coupling, and easier interchangeability. Consequently, we believe that using SPL architectures improves the development of agents that can change their configuration at runtime based on the current context.

References

1. Weyns, D., Parunak, H.V.D., Shehory, O.: The future of software engineering and multi-agent systems (editorial, special issue). IJAOSE **3**(4) (2009) 369–377
2. Nunes, I., Lucena, C.J., Cowan, D., Alencar, P.: Building service-oriented user agents using a software product line approach. In: ICSR '09. (2009) 236–245
3. Clements, P., Northrop, L.: Software Product Lines: Practices and Patterns. Addison-Wesley (2002)
4. Nunes, I., Lucena, C., Kulesza, U., Nunes, C.: On the development of multi-agent systems product lines: A domain engineering process. In: AOSE'09. (2009) 109–120
5. Kästner, C., Apel, S., Kuhlemann, M.: Granularity in software product lines. In: ICSE '08, USA, ACM (2008) 311–320
6. Rao, A., Georgeff, M.: BDI-agents: from theory to practice. In: ICMAS'95. (1995)
7. Brazier, F.M.T., Kephart, J.O., Parunak, H.V.D., Huhns, M.N.: Agents and service-oriented computing for autonomic computing: A research agenda. IEEE Internet Computing **13**(3) (2009) 82–87
8. Girardi, R.: Reuse in agent-based application development. In: SELMAS'02. (2002)
9. Lind, J.: Patterns in agent-oriented software engineering. In: AOSE'02. (2002) 47–58
10. Gonzalez-Palacios, J., Luck, M.: A framework for patterns in gaia: A case-study with organisations. In: AOSE'04. (2004) 174–188
11. Pena, J., Hinchey, M.G., Ruiz-Corts, A., Trinidad, P.: Building the core architecture of a multiagent system product line: with an example from a future nasa mission. In: AOSE'06. (2006)
12. Dehlinger, J., Lutz, R.R.: Supporting requirements reuse in multi-agent system product line design and evolution. In: ICSM. (2008) 207–216
13. Nunes, C., Kulesza, U., Sant'Anna, C., Nunes, I., Garcia, A., Lucena, C.: Assessment of the design modularity and stability of multi-agent system product lines. J.UCS **15**(11) (2009) 2254–2283
14. Nunes, I., Kulesza, U., Nunes, C., Cirilo, E., de Lucena, C.J.: Extending web-based applications to incorporate autonomous behavior. In: WebMedia'08. (2008)
15. Fischer, K., Hahn, C., Madrigal-Mora, C.: Agent-oriented software engineering: a model-driven approach. IJAOSE **1**(3/4) (2007) 334–369
16. Bresciani, P., Perini, A., Giorgini, P., Giunchiglia, F., Mylopoulos, J.: Tropos: An agent-oriented software development methodology. JAAMAS **8**(3) (2004) 203–236
17. Figueiredo, E. et al.: Evolving software product lines with aspects: an empirical study on design stability. In: ICSE '08. (2008) 261–270
18. Garcia, A., Lucena, C.: Taming heterogeneous agent architectures. Commun. ACM **51**(5) (2008) 75–81
19. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-oriented Software. Addison-Wesley (1995)