University of Waterloo

Technical Report
CS-2009-17

Taking Advantage of the Interplay among Software Product Lines, Service-oriented Architectures and Multi-agent Systems

Ingrid Oliveira de Nunes
Carlos José Pereira de Lucena
Paulo Alencar
Donald D. Cowan

Faculty of Mathematics

DAVID R. CHERITON SCHOOL OF COMPUTER SCIENCE
UNIVERSITY OF WATERLOO
WATERLOO, ONTARIO, CANADA N2L 3G1

Technical Report CS-2009-17 May, 2009
David R. Cheriton School of Computer Science - University of Waterloo

Taking Advantage of the Interplay among Software Product Lines, Service-oriented Architectures and Multi-agent Systems

Ingrid Oliveira de Nunes[1], **Carlos José Pereira de Lucena**[1], **Paulo Alencar**[2], **Donald D. Cowan**[2]

[1] Pontifical Catholic University of Rio de Janeiro (PUC-Rio) - Rio de Janeiro, Brazil
[2] University of Waterloo - Waterloo, Canada

{ionunes,lucena}@inf.puc-rio.br, {palencar,dcowan}@cs.uwaterloo.ca

Abstract. Multi-agent Systems (MASs) are often being applied in a wide range of industrial applications, showing the effectiveness of the agent abstraction to develop open, highly interactive, autonomous and context-aware systems. MASs have been combined with Service-oriented Architectures (SOAs) in order to provide customization and flexibility in these systems. This combination calls for new approaches that support personalized user services through autonomous and pro-active components in dynamic environments. Existing approaches fail to provide reusable multi-agent service components as well as suitable representations and processes that support automated software generation based on common and variable features within a domain. In this paper we present a domain engineering process-oriented approach to build service-oriented user agents using the Software Product Line (SPL) engineering paradigm. The approach comprises activities and models to support the development of service-oriented customized agents that automate user tasks based on service orchestration involving multiple agents in open environments, and takes advantage of the synergy of SOA, MAS and SPL. The domain-based process involves extended domain analysis with goals and variability, domain design with the specification of agent services and plans, and domain implementation.

Keywords: Software Product Lines, Multi-agent Systems, Service-oriented Architectures.

In charge of publications:
Helen Jardine
David R. Cheriton School of Computer Science
University of Waterloo
Waterloo, Ontario, Canada N2L 3G1
Tel: +1 519 888-4567 x33293 Fax: +1 519 885-1208
E-mail: hjardine@uwaterloo.ca
Web site: http://www.cs.uwaterloo.ca/research/tr/

1 Introduction

An agent-based method is often the approach of choice in many of the existing software tools that support applications such as web-based supply-network management, auction staging, medical-record processing, mission scheduling, and e-commerce [8]. These systems are typically open, highly interactive, autonomous and context-aware, and need to support customized and flexible user services. Approaches to Multi-agent Systems (MASs) involve metaphors such as autonomous agents, agent goals and agent societies [27]. In contrast, Service-oriented Architectures (SOAs) [5] and related approaches are often used to deliver application functionality as reusable services to end-user applications or to build other supporting services. However, the development of large-scale service-oriented MASs calls for new software engineering processes, models and methods that support personalized user services through pro-active, autonomous, reactive, and heterogeneous components in dynamic and uncertain environments. Existing approaches do not address the problem of generating these user component services in a Software Product Line (SPL) fashion, and fail to provide reusable software multi-agent service components as well as suitable representations and processes that support automated software generation based on common and variable features within a domain. In this paper we present a domain engineering process-oriented approach to build service-oriented user agents using the Software Product Line Engineering (SPLE) paradigm. The approach comprises activities and models to support the development of service-oriented customized agents that automate user tasks based on service orchestration involving multiple agents in open environments such as the Web. The domain-based process involves extended domain analysis with goals and variability, domain design with the specification of agent services and plans, and domain implementation.

Our approach takes advantage of the synergy of SOA, MAS and SPL. Service-oriented systems follow many of the ideas from research conducted in MASs but there are several challenges that still need to be faced in terms of their combination [7]. The integration of these two approaches is usually referred to as Service-oriented Multi-agent Systems or Agent-based Service-oriented Architectures. This integration has been used in domains such as electronic commerce, in which users have agents act on their behalf to automate their tasks. However, given that agents represent individuals in these scenarios, there remains a need to personalize an agent to meet specific needs of the users and to support their implementation in an automated way. This can be achieved by developing families of agents, which have variable parts but at the same time present common features. In this context, SPLs [4] is a software engineering trend that promotes reduced development costs, shorter time-to-market and higher quality, when developing families of systems by the exploitation of the common features among family members.

Besides the research work that addressed the integration of SOA and MAS, these two approaches have been individually studied in combination with SPLs [11, 15, 16, 21]. The integration between SPL and SOA is currently receiving significant attention both in research and in practice. While SOA provides flexible architectures promoting large-scale reuse of software developed in different organizations, SPL helps in identifying, analyzing and modeling services and their configurations to meet different users' needs. The main rationale for integrating MAS and SPL is that MAS methodologies have not addressed the need to develop large scale customized systems and little effort has been expended on software reuse techniques. As a result, although these approaches have been studied independently and in pairs, to the best of our knowledge, there is no research aimed at taking advantage of the the integration of the three approaches (i.e., MAS, SOA, and SPL).

In this paper, we first present the general results of an exploratory study into the integration

of SOA, MAS and SPL, by providing an overview and comparison of these three approaches, emphasizing the benefits they bring to each other. Second, we propose an approach for building customized user agents using a SPL approach. These agents are characterized as service-oriented MASs and they provide personal services to users. Although, the idea of providing agents to act on behalf of users has been introduced in previous work [10], there is a clear need to provide personalized agents in large numbers given that they represent individuals. This need to produce large numbers of agents leads us to consider automated generation using a SPL approach.

The remainder of this paper is organized as follows. In Section 2 we briefly describe SOA, MAS and SPL, and compare these approaches to provide some rationale for and benefits of their integration. Section 3 gives an overview of our domain engineering process-oriented approach and presents an illustrative example. In Sections 4 to 6 we describe in detail the domain analysis, design and implementation phases of the approach. Section 7 details the derivation process of customized user agents. Section 8 contains a discussion about some relevant issues that emerged from our study, and in Section 9 we describe related work. Finally, in Section 10 we present our conclusions.

2 Integration of SOA, MAS and SPL

In this section, we focus on the rationale and benefits for integrating SOA, MAS and SPL. We briefly present each approach and then compare them in order to capture their similarities and differences. Next, we give an overview of each approach. We do not provide an extensive description but point the interested reader to [4, 5, 27].

Service-oriented Architecture. SOA [5] is a paradigm for developing distributed applications based on the their decomposition into a set of services, which can be used either by users and client applications or by other services. These services may be under the control of different ownership domains. Moreover, SOA provides a uniform means to offer, discover, interact with and use services to produce desired effects consistent with measurable preconditions and expectations. Service-oriented systems follow some key principles, which are: (i) loose coupling; (ii) service contract; (iii) autonomy; (iv) underlying logic abstraction; (v) reusability; (vi) composability; (vii) statelessness; and (viii) discoverability.

Multi-agent Systems. MASs [27] synthesize contributions from different areas, including artificial intelligence, software engineering and distributed computing. In the context of software engineering, it is viewed as a paradigm, which addresses developing systems that contain many dynamically interacting components, each with their own thread of control while engaging in complex, coordinated protocols. The main idea of Agent-oriented Software Engineering (AOSE) [25] is to decompose complex and distributed systems into autonomous, pro-active and reactive entities with social ability, namely agents. 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. Hence, agents are developed with cognitive abilities usually modeled as goals to be achieved, plans to achieve these goals and beliefs (mental state) necessary to execute plans.

Besides agents, MASs provide other abstractions that are inspired by their analogy with human societies. It has concepts such as roles and organizations, identifying not only the responsibilities of agents as individuals, but also their global responsibilities to the MAS as a whole, i.e. social tasks. This leads to agents committed to playing roles in organizations and the need to identify social laws, which must be respected and enforced. This is particularly interesting when avoiding

possible chaos in large-scale open environments, such as the Internet.

Software Product Line. SPLE [4, 18] aims at improving the development of system families by exploiting common features of applications. SPLE allows a systematic derivation of products of the same family from a flexible architecture that supports variability. SPLE is typically composed of two key processes [18]: domain engineering, in which SPL common features and variability features are identified, defined and realized, and application engineering, in which applications of the SPL are built by reusing domain artifacts and exploiting the SPL variability.

Table 1: Similarities and Differences among SOA, MAS and SPL

Approaches	**Similarities**	**Differences**	
SOA and SPL [11, 21]	Development of flexible, cost-effective systems Efficient Reuse Applications development from existing pieces of software Concerns modeling at architectural level Compositional structure of entities and connection among their interfaces	**SOA**	**SPL**
		Composition of dynamic computational elements	Reusing and resolving variability in static elements (typically)
		Opportunistic reuse	Systematic reuse
		Combines artifacts into larger entities	Decomposes artifacts into fine grained artifacts
		Development of systems consisting of loosely coupled services or company-wide infrastructures	One producer alone developing a set of systems
		Compliance to standards	Different modeling initiatives
SOA and MAS [7]	Systems consist of loosely coupled and autonomous entities Focus on distributed systems Dynamic composition	**SOA**	**MAS**
		Services passively waiting for discovery	Agents pro-actively contribute to an application
		Strives for reuse for an effective cost and time systems development	Most approaches have not addressed reuse
		Well-defined interfaces	Dynamic selection of protocols
		Services modeled and understood at a coarse granularity	Different granularity levels (intra-agent and inter-agent view point)
MAS and SPL [15, 16]	Development of flexible architectures	**MAS**	**SPL**
		Flexibility obtained by loosely coupled and autonomous agents	Architecture decomposed into flexible components to address variability
		Development of single systems	Focus on system families
		Most approaches have not addressed reuse	Systematic reuse

As stated previously, the pairwise combination of these approaches has been target of research. The approaches have some similarities and differences, which are summarized in Table 1. In two of these combinations (SOA and MAS; SOA and SPL), the integrated approaches have several similarities and common goals, so the main purpose of studying these approaches together is to capture how they can contribute to each other based on their differences. For instance, the comparison between SOA and SPL reported in [21] concludes that variability modeling in SPLs should take a lesson from behavior modeling and analysis of services and business process in SOA. On the other hand, [7] states that SOA represents an emerging class of approaches with MAS-like characteristics for developing systems in large-scale open environments; and that key MAS concepts such as ontologies, choreography and directories are reflected directly in those of SOA. MAS and SPL, the third combination, have been integrated not because they present similarities, but to address the growing need for development of families of complex distributed systems, while taking advantage of MAS abstractions. AOSE methodologies have failed to capture the reuse

potential adequately since many of the developed methodologies focus on the development of specific software applications.

Based on this comparison, it can be seen that it is feasible to integrate MAS, SOA and SPL. However, this integration results in some challenges, such as how to develop service-oriented MASs and, at the same time, support variability. In the next section, we present our three-way complementary approach, which extracts the major benefits of the three approaches and integrates them.

3 Approach Overview

Our approach aims at defining activities and models to address the development of customized agents, which are deployed in a MAS. These agents achieve their goals by the execution of plans that may use other agents' services and they can provide services to other agents or users. This scenario is currently illustrated by Internet-based applications, on which there are several autonomous organizations providing services with interacting users. The main goal of our approach is to automate these users' interactions, but given that agents represent users (individuals), we must provide a way to deploy personalized agents. The proposed approach incorporates principles and concepts of SOA, MAS and SPL, which are described in the next three paragraphs.

Problem Decomposition into MAS Concepts. Based on the introduction to MAS in the previous section, it can be seen that MAS provides several concepts for understanding and modeling a complex and distributed system. Each agent of a MAS may be classified from two different perspectives [26]: (i) internally as a software system with its own purpose (*intra-agent*); (ii) externally as part of a society interacting with other individuals (*inter-agent*). This classification is illustrated in Figure 2. Our approach focuses on developing a single agent to be part of an existing MAS, detailing its internal structure and interaction with other agents. This agent is structured according to the belief-desire-intention (BDI) model [22], which supportts modeling cognitive agents and whose advantages include: relative maturity having been used successfully in large scale systems; supported by several agent platforms, including Jason and Jadex; based on solid philosophical foundations. In addition, MAS helps with the development of open systems by assigning duties and rights to agents that play roles and providing enforcement of society norms. This last issue is not addressed in this paper.

Service Analysis and Orchestration. Even though AOSE is based on a powerful abstraction for modeling complex systems, most of AOSE methodologies, such as GAIA [28], focus on the development of closed systems, in which agents are known at design time. A key advantage of SOA is that they enable services to be selected and integrated dynamically at runtime, thus enabling system flexibility and adaptation. In our approach, we define specific activities for identifying and specifying services provided by agents.

Analysis and Implementation Support to Variability. Given that there are agents representing users in the MAS, there is the requirement of representing their specific needs. Our approach aims at addressing a SPL of agents through which we can systematically derive customized agents. Our approach contemplates variability analysis in order to capture variations and different possible configurations of the user agents and provide implementation support to build reusable assets with the aim of allowing automatic derivation of agents.

Figure 1 depicts the activities performed in our approach, the sequence in which they must be performed and their output artifacts. The development of the agent product line starts with an analysis of the domain using a goal-oriented approach. The domain variability analysis is performed

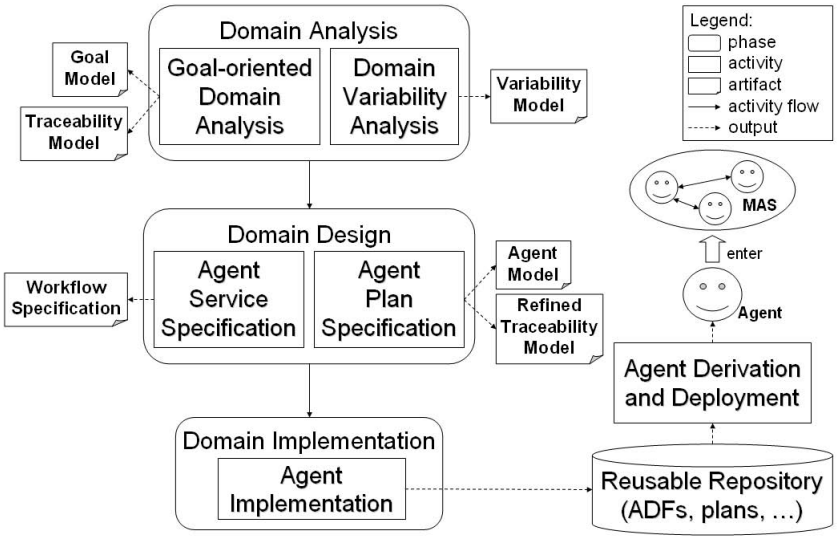

Figure 1: Approach Overview.

with this activity in order to capture variability within the domain that will be addressed by the product line. These two activities define the scope of the SPL. Later, two activities (Agent Service Specification and Agent Plan Specification) are performed in order to model the SPL in terms of services and plans provided by the agent goals, considering the previously identified variability. Based on the models generated in these activities, a set of code artifacts are implemented comprising a reusable infrastructure, which is in turn used to derive different agent instances based on the configuration provided by users.

Five activities presented in Figure 1 are categorized into three different phases: domain analysis, domain design and domain implementation. These phases are the typical ones of domain engineering processes, and they result in reusable artifacts that support the identified variability. The Agent Derivation and Deployment activity is part of the application engineering process, which enables a systematic assembly of a selected collection of artifacts for building a customized application, which in this case is an agent.

3.1 e-Marketplace Case Study

This section presents an e-Marketplace case study, which is used to illustrate our approach. Providing applications to automate commerce is one of the application domains of MAS [8]. Currently, commerce is almost entirely driven by human interactions; humans decide when to buy goods, how much they are willing to pay, and so on. However, some commercial decision-making can be placed in the hands of agents.

Figure 2 depicts the overall structure of the e-Marketplace case study. It is a MAS, on which there are: (i) agents/organizations representing stores that sell products and provide support services; (ii) agents/organizations that support the buying process, such as credit card companies and PayPal; and (iii) user agents that automate the activities performed by users to buy products.

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. Furthermore, this existing MAS already provides an ontology giving a

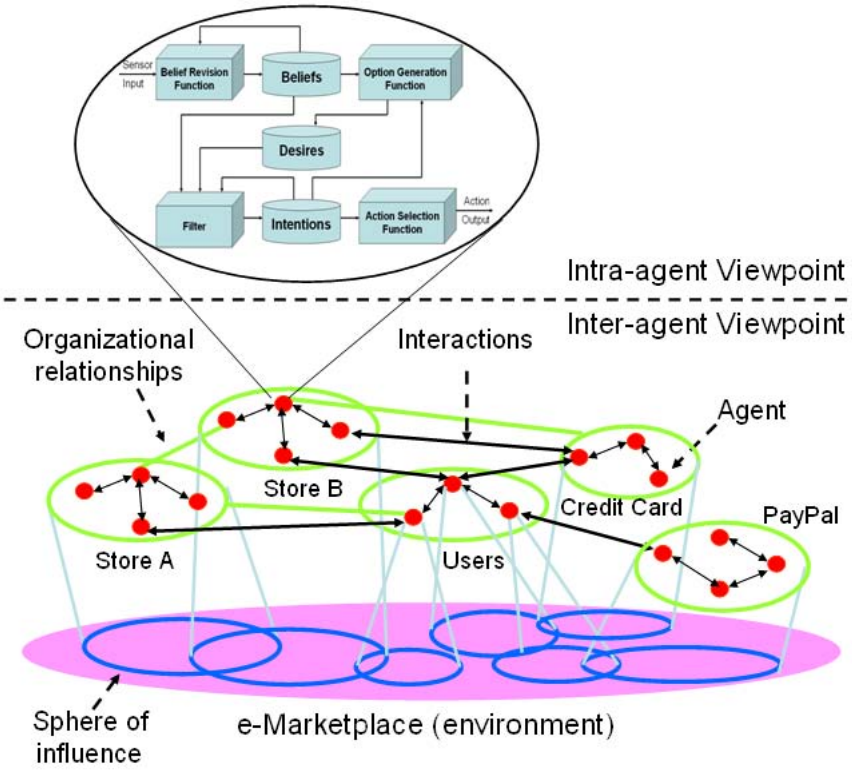

Figure 2: Canonical view of a complex distributed system: e-Marketplace.

formal representation for a set of concepts, which includes the messages exchanged by agents within the domain and protocols defining how messages are exchanged. Our goal is to develop a customizable user agent, focusing on its internal structure, which enters in the MAS and interacts with other agents to achieve its goals.

4 Domain Analysis

Given that agents are proactive and have goals (according to the BDI architecture), it is natural to consider using goals to describe requirements. Goals capture, at different levels of abstraction, the various objectives the system should achieve. Other advantages of using goal-oriented requirements engineering can be found at [23]. However, SPLE is distinguished from single-system engineering by its focus on the systematic analysis and management of the common and variable characteristics of a set of software systems [13]. Thus, we focus on both the activity to analyze the domain in terms of goals, and the activity whose aim is to identify the variation points in the domain.

In SPL approaches, there are typically two ways of modeling variability. One of them is based on feature modeling, which was first proposed in FODA [9]. It identifies common and variable features inside the domain and organizes them into a tree notation. A feature is a system property that is relevant to some stakeholder and is used to capture common traits or discriminate among products in an SPL. The other approach, used in [13, 18] proposes the use of a separate model to express variability. We have adopted this second method primarily for two reasons: (i) modeling

common and variable features in the same model can lead to huge and complex models; (ii) given that our target is to develop agents that follows the BDI model, it is more natural to make a goal-oriented domain analysis.

Thus, in our domain analysis, the two activities are performed in parallel and are complementary to each other. One of them is the Goal-oriented Domain Analysis, which is responsible for capturing the SPL goals. This is performed at different levels of abstraction. Therefore, goals can be decomposed into sub-goals. Figure 3 illustrates the goal model of the user agent SPL to buy products in the e-Marketplace case study. At certain points of the goal analysis, some variable traits can be identified in the domain. For instance, the *Verify if Product in Stock* goal has two different meanings: (i) verify if the desired product is available online; and (ii) verify if the desired product is available in a certain store, in case the user wants to pick up the product in a store.

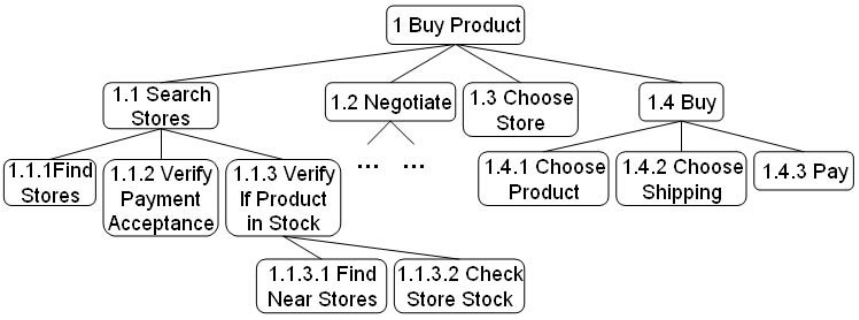

Figure 3: Goal Model.

The activity responsible for analyzing and documenting variability within the domain is the Domain Variability Analysis. In order to document variability, we adopted the notation proposed in [18]. It describes variation points (what varies) and variants (how it varies). In addition, it also contains constraints. These constraints ensure that only valid combinations of the variations are selected. Examples of constraints are when one variant depends on another or two variants are mutually exclusive.

The variability model for the user agent in the e-Marketplace case study is presented in Figure 4. In the model, there are two kinds of variation points – optional and alternative: (i) the optional variation point is one that may or may not be present in the agent. For example, the *Negotiate* variation point is optional, which means that if the variation point is selected the agent will negotiate the price of the target product; (ii) the alternative variation point is associated with a set of variants, some of which must be chosen. In Figure 4, there are five alternative variation points. One is the *Payment Type*, which has three different variants (*Credit Card*, *Pay upon Pick up* and *PayPal*). There is also a variation point named *Services*, because we intend to extend the user agent in the future to incorporate new services for users, but for the moment the only service provided is *Buy Product*.

The variability identified and documented in the Domain Variability Analysis is used for two purposes: (i) to indicate optional and alternative parts that have to be supported among all the subsequent models and code assets; and (ii) to provide a way for users to choose the appropriate configuration of their agent.

This variability information is not present in the goal model. Besides the two reasons presented for documenting variability in a separate model, this practice helps to keep consistency among the models. For example, the *Negotiate* variation point is optional, therefore the *Negotiate* goal is

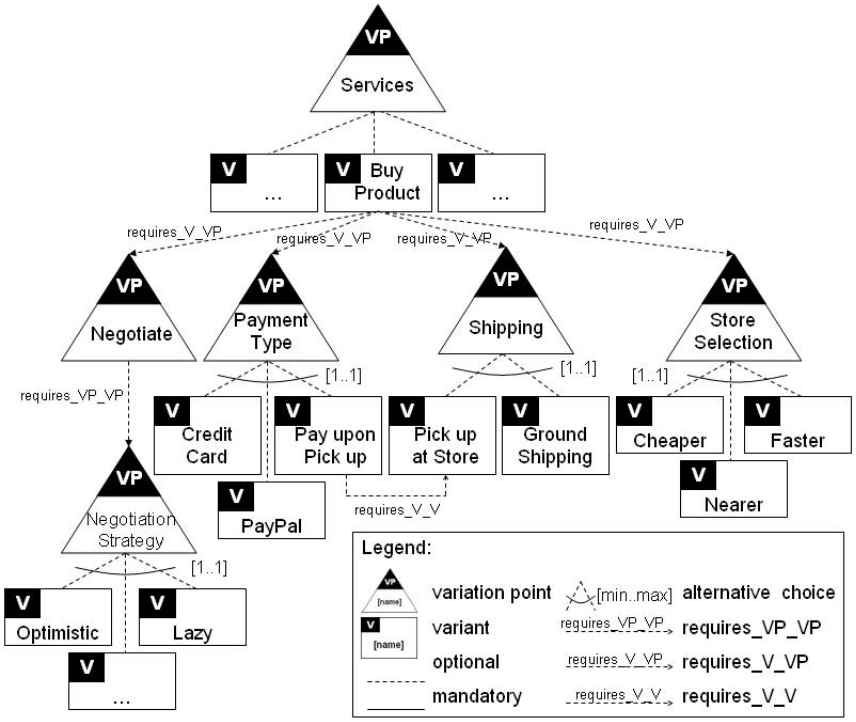

Figure 4: Variability Model.

also optional. So, if one decides to make the *Negotiate* variation point mandatory, the goal model does not need to be updated. However, even though this information is not explicitly shown in goal model, it is important to know what is variable in this model. As a consequence, there is another model that provide variability traceability links, which ensure the consistent definition of the common and variable traits of the SPL throughout all artifacts.

Our traceability model consists of a mapping between goals and variability expressions. One goal may be related to one or more variation points/variants or even a combination of them. For instance, the goal *Pay* is optional, given that if the *Payment Type* is *Pay upon Pick up* the agent must not pay for the product because the user is responsible for paying when the product is collected at the store. So the valid variability expression for this goal is either `!Pay upon Pick up` or `Credit Card | PayPal`.

5 Domain Design

After understanding and modeling the domain in terms of goals and identifying variation points, agents are designed by the identification and specification of their services and plans to achieve goals.

The definition of agents that offer business services has several advantages. First, agent services can be discovered and used by other agents. Agents that are in charge of several responsibilities have a set of goals, beliefs and plans to accomplish each of these responsibilities. Nevertheless, no technique is used to provide modularization of these sets of goals, beliefs and plans in order to provide a better understanding of their purpose. Besides providing modularization, encapsulating these concepts into a service improves the reuse in MASs. Moreover, using this service-oriented

loosely-coupled approach brings to business process models a structure that can significantly improve the flexibility and agility with which processes can be remodeled, thus helping to deal with variability. In addition, services can be dynamically discovered and used when an agent needs to adapt itself because of changes in the environment. This last advantage has not yet been explored in our work.

The Directory Facilitator and Agent Discovery Service, which provide the service of yellow pages in MASs, are part of the standard agent architecture proposed by FIPA. However, most MAS methodologies address the development of closed MAS. In this case, discovering agents and their services is not a concern. In order to deal with open MAS, in which agents are not known *a priori*, our approach is to define the Agent Service Specification activity. Their specification may require the use of other "black-box" services provided by other agents.

Based on the goal model generated in the previous activity, we have identified the services provided by the user agent. Figure 6 highlights these services. It can be seen that there are four different atomic services (*Search Stores*, *Negotiate*, *Choose Store* and *Buy*), which are lower level services and do not use other internal services. On the other hand, the *Buy Product* service, is a composite service, given that it composes atomic services. The *Negotiate* is an optional variation point, therefore the *Negotiate* service is also optional, consequently the *Buy Product* can be composed in two different ways: with or without negotiation. Services are considered atomic from an agent internal viewpoint, however these services can use services provided by other agents. For instance, the *Buy* service doest not use any other service provided by the user agent, but it communicates with the Store and PayPal agents in order to use their services. After identifying services, their workflows are specified with UML 2.0 activity diagrams. The specification of the buy service workflow is depicted in Figure 5. The workflow contains different paths, which are based on the variability of the SPL.

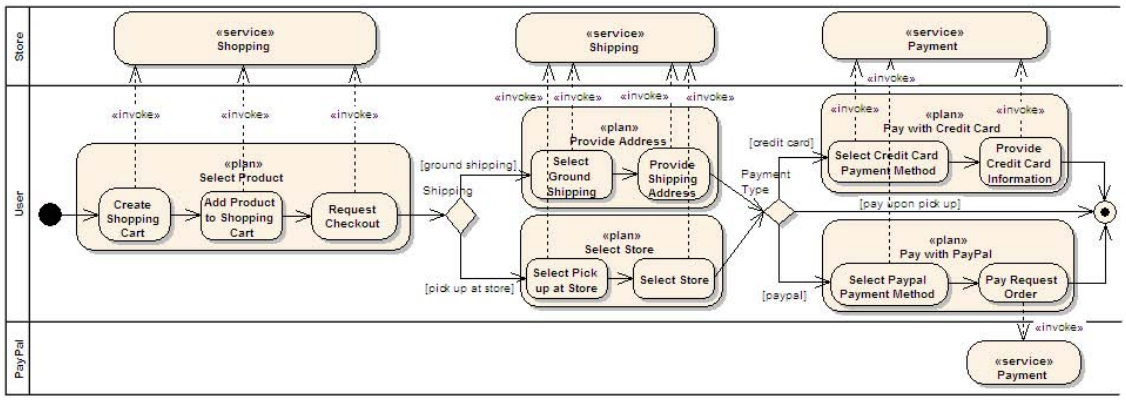

Figure 5: Buy Service Workflow Specification.

The workflow specification shows actions to be performed to achieve service goals and invocations of other agents' services. In addition, it supports the identification of plans to achieve lower-level goals. While goals provide the agent's motivation and are the driving force for its actions, plans represent the agent's means to act within its environment.

The artifact that describes which plans are used to achieve which goals is the Agent model (Figure 6). It is divided into two layers: (i) the Goal Layer – it shows agent's goals and their decomposition into sub-goals. It is structured according to the Goal model; and (ii) the Plan Layer – it shows agent's plans. There are links between goals and plans indicating that a certain plan

achieves a certain goal. For some goals, different plans can be used. In addition, there is one case (*Check Stock* goal) that it can either be achieved by the *Check Online Stock* plan or by its decomposition into the *Look for Near Stores* and *Check Store Stock* goals.

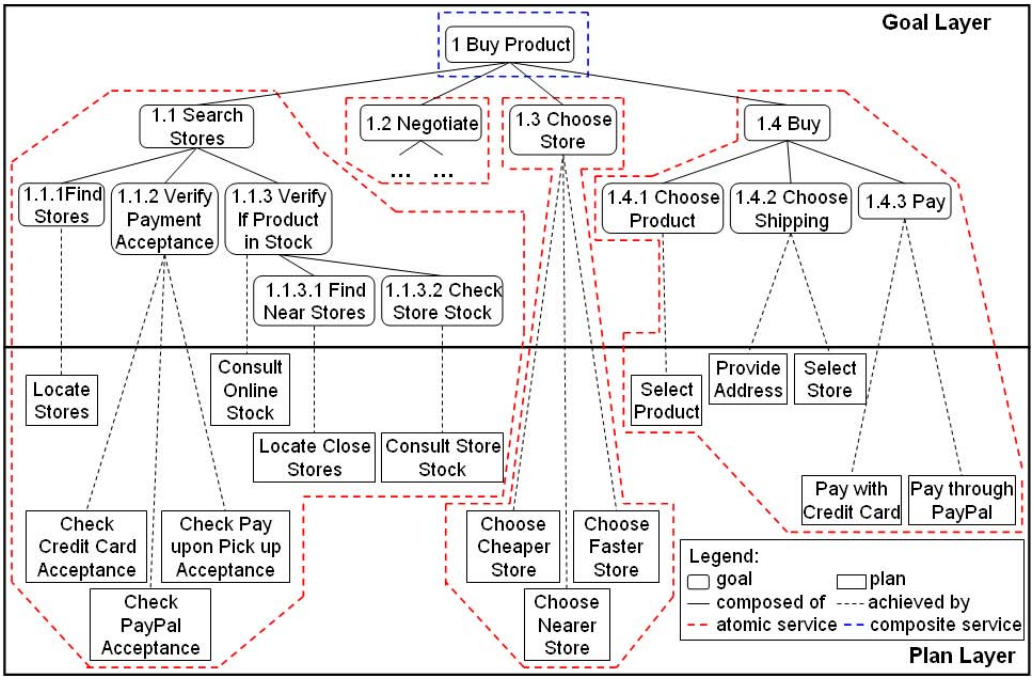

Figure 6: Agent Model.

After specifying an agent's services and plans, traceability links must be provided between these concepts and the variation points, similar to the way it was done with goals. This is to ensure that, for instance, if the *Payment Type* chosen is *Credit Card*, the selected plan for the *Check Payment* goal is *Check Credit Card Acceptance* and for the *Pay* goal is *Pay with Credit Card*. So the Traceability model is refined in order to incorporate these new mappings. This Traceability model facilitates systematic and consistent reuse, and allows the application engineering to be performed efficiently.

6 Domain Implementation

Based on the models generated in previous activities, code assets are implemented. Techniques are used to implement them in order to support the defined variability and allow the derivation of customized agents. We have identified some types of variability and we adopted guidelines to implement them. These guidelines are specific for the Jadex [19], an agent platform based on the BDI model. 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, in particular messages that can be sent and received. Plans are declared in the ADF, but their body is implemented in Java classes, which extend the `Plan` class. In addition, Jadex provides the capability concept, which is an encapsulated agent module composed of beliefs, goals, and plans that can be reused wherever it is needed. Next, we detail the

proposed implementation guidelines.

All goals, whether they are top level goals or sub-goals, need to be declared in the ADF. However, some of these goals are optional and alternative, such as the *Pay* goal. The condition for this goal to be present in a derived agent is according to the variability expression related to it (`Credit Card | PayPal`). So, we adopted a translation from variability expressions to tags, which are put in the code in order to make conditional compilation possible. Figure 7 illustrates the *Pay* goal declaration surrounded by the appropriate tag.

```
<!-- BEGIN CREDIT_CARD_OR_PAYPAL -->
<achievegoal name="pay">
    <parameter name="store" class="Store" />
    <parameter name="shoppingCart" class="ShoppingCart" />
</achievegoal>
<!-- END CREDIT_CARD_OR_PAYPAL -->
```

Figure 7: Optional goal.

Goals in the agent model (Section 4) can be achieved by either a plan or through its sub-goals. In the first case, similar to goals, plans can be optional or alternative. As a consequence, we used the same strategy to support the variability – we introduced tags into the code according to variability expressions. For instance, the agent had two different plans to achieve the *Pay* goal, and one of them is the *Pay with Credit Card* plan, which is associated with the `Credit Card` variability expression. Figure 8 presents the code fragment related to this plan.

```
<!-- BEGIN CREDIT_CARD -->
<plan name="pay_with_credit_card">
    <parameter name="store" class="Store" >
        <goalmapping ref="pay.store" />
    </parameter>
    <parameter name="shoppingCart" class="ShoppingCart" >
        <goalmapping ref="pay.shoppingCart" />
    </parameter>
    <body class="PayWithCreditCardPlan" />
    <trigger>
        <goal ref="pay" />
    </trigger>
</plan>
<!-- END CREDIT_CARD -->
```

Figure 8: Alternative plan.

In the second case – when goals are achieved by decomposition into sub-goals – a plan is created for dispatching the appropriate sub-goals. The plan is declared in the ADF file, and a Java class is created, which extends the `Plan` class provided by Jadex. Into the overridden `body` method of the plan, the sub-goals are dispatched (Figure 7). These goals can be executed in an appropriate order by calling the method `dispatchSubgoalAndWait()`, or just be dispatched and executed in any order determined by Jadex reasoning engine, by calling the method `dispatchSubgoal()`. Some of the sub-goals are mandatory; while others can be optional or alternative. In the same way, we adopt tags to delimit variable sub-goals.

The identified agent services with their respective agent concepts (e.g. goals and plans) are encapsulated into capabilities. Figure 10 shows how a capability is introduced into the ADF. In addition, Jadex allows the specification of messages to be sent and received in the ADFs, to be

```
public void body() {
    IGoal search_stores = createGoal("search_stores");
    dispatchSubgoalAndWait(search_stores);
    // BEGIN NEGOTIATE
    IGoal negotiate = createGoal("negotiate");
    dispatchSubgoalAndWait(negotiate);
    // END NEGOTIATE
    IGoal choose_store = createGoal("choose_store");
    dispatchSubgoalAndWait(choose_store);
    IGoal buy = createGoal("buy");
    dispatchSubgoalAndWait(buy);
}
```

Figure 9: Optional sub-goal.

later used in plans. By defining a service in a capability, the goals of the services and messages that it can send and receive are explicitly defined. Moreover, this capability can be easily reused either by agents or other capabilities, and it can be (un)plugged easily as well.

```
<capabilities>
    <capability name="search_stores"
                file="br.pucrio.inf.service.SearchStore" />
    <!--  BEGIN SERVICE NEGOTIATE  -->
    <capability name="negotiate"
                file="br.pucrio.inf.service.Negotiate" />
    <!--  END SERVICE NEGOTIATE -->
    <capability name="choose_store"
                file="br.pucrio.inf.service.ChooseStore" />
    <capability name="buy"
                file="br.pucrio.inf.service.Buy" />
</capabilities>
```

Figure 10: Optional service.

7 Application Engineering

The activities presented in previous sections are part of the domain engineering process, resulting in a reusable infrastructure that supports the identified variability. The Agent Derivation and Deployment activity is part of the application engineering process, and its purpose is to derive and deploy customized agents based on this reusable infrastructure. Based on the models and code assets generated in previous activities, it is possible to select and customize the code assets manually according to a configuration of the variability model and thus deploy the derived agent. However, we are working toward developing a tool to automate this process. The first version of the tool is specific to the e-Marketplace case study.

Basically, our tool selects and provides conditional compilation for the code assets. Traditional conditional compilation could not be used because the Jadex platform defines agents based on XML files. In addition, the tool has a web-based interface through which the user can configure an agent and deploy it into the e-Marketplace MAS. The agent derivation process and deployment comprise the following steps: (i) the user describes through the web application interface the product that he wishes to buy and the agent configuration. The product information provided is based on the ontology defined for the e-Marketplace MAS. The agent configuration is made

through the selection of optional and alternative variation points and variants; (ii) with the data in step (i) and the traceability model, our tool selects the appropriate code assets; (iii) the tool manipulates the ADFs and Java classes in order to derive customized code for the agent. The tool translates the selected variation points and variants to tags used in the code, and removes optional and alternative fragments of the code that were not selected; (iv) the Java classes are compiled; and (v) the agent is instantiated and deployed into the e-Marketplace MAS using the Jadex platform.

8 Discussions

The work reported in this paper is the result of an exploratory study of the integration of SOA, MAS and SPL in order to support the development of customized agents to be part of service-oriented MASs. In this section, we discuss some relevant issues that arose during our study, which gives directions for future work.

Granularity Levels. Granularity in SPLs refers to the degree of detail and precision of a variability as produced by a design or implementation element. SPL variability may exist at different levels of granularity ranging from entire components to single lines of code. In our approach, we have considered variability at three different levels of granularity: (i) agent goals, (ii) agent services; and (iii) agent plans. Services correspond to coarse-grained variability. They can be orchestrated in different ways in order to provide a composite service. In addition, services can present some fine-grained variability as realized by alternative or optional goals and plans. Basically, the implementation technique used to implement variability was conditional compilation into XML files, given that the agent platform used for implementing agents was Jadex. As an evolution of the approach presented here, we aim at exploring other granularity levels, such as agents and roles. For instance, stores usually present similar organizational structures, consisting of a set of roles, in which some of them can be optional or alternative, such as the promotion manager role.

Dynamic Agent Adaptation. Because of the dynamic nature of the Internet, the availability of any resource cannot be guaranteed at any single moment. One of the main advantages of SOAs is to promote the automation of flexible and highly adaptive business processes through the orchestration of loosely coupled services. In addition, some research addressed the development of self-adaptive MAS to deal with highly dynamic environments [17]. Our approach allows the derivation of agents customized according to specific user needs. In the e-Marketplace, flexibility is provided in the sense that the user agent discovers store agents to achieve their *Buy Product* goal; but adaptation in this process is not yet explored. Furthermore, we aim at providing a dynamic adaptation of the user agent in order to incorporate new services and change existing ones.

Tool Support. The derivation process of user agents in the e-Marketplace case study is accomplished by our application-specific tool. The tool deploys a customized version of the user agent into the e-Marketplace MAS by selecting the appropriate artifacts and manipulating XML and Java source files in order to remove the code related to unselected variations points and variants of the product line. However, there are model-based derivation tools [2, 20] that automate the derivation process of SPLs. One of them is GenArch [2], which was developed in our Software Engineering laboratory, and was recently extended to incorporate a new domain-specific model that addressed Jadex [3]. This GenArch extension is not fully compatible with our approach, however we are aiming to combine the two approaches.

9 Related Work

The main contribution of this paper is a proposal for an integrated approach that exploits the major benefits of SOA, MAS and SPL in order to build customized service-oriented user agents and deploy them into existing MASs. We have not seen any research that addresses this problem, but in this section we describe approaches to personalization of systems that use agent-based solutions, and the combination these three technologies in a pairwise fashion.

A personalized recommendation system with multi-agents based on web intelligence is proposed in [6]. The approach provides an intelligent user agent, which is in charge of interacting with the users and receiving the feedback. This agent communicates to other agents that are responsible for learning and recommending products. The development of a personalized time manager, represented by an agent, is addressed in [1]. The agent is designed to assist a human user in managing her time commitments, a large part of which involves managing the communication and negotiation intrinsic in scheduling meetings. Both approaches provide personalized assistance for users by means of agents that use artificial intelligence techniques. The idea is to use cognition to provide customized information, but the agent configuration and behavior are not adjusted to user needs, as our approach proposes.

There is research in the context of MASs that aims at solving some issues in the service-oriented development. The approach proposed in [12] addresses dynamic service selection via an agent framework coupled with a QoS ontology. With this approach, participants can collaborate to determine each other's service quality and trustworthiness. Other approaches, such as [24], deal with trustworthiness when selecting agents to use their services. This is an important issue in open systems. In [11], SOA and SPL are integrated through a method, whose goal is to achieve flexibility in network based systems though service orientation, but still managing product variations thought SPLE techniques. Finally, a domain analysis approach for the development of Multi-agent Systems Product Liness (MAS-PLs) is presented in [14]. It integrates notations and activities of SPL and MAS in order to address agent variability and tracing.

10 Conclusion

The growing popularity of agent-based approaches for developing open, complex, context-aware and highly interactive systems has motivated the proposal of new methods and processes using the agent abstraction. In addition, agent-orientation is being combined with SOAs in order to build applications that provide flexible services to users, while taking advantage of the agent-oriented paradigm.

Inspired by human societies, several service-oriented multi-agent applications, such as e-commerce systems, are developed by representing users as agents, which act on the users' behalf and automate their tasks. The approach presented in this paper advances the development of these service-oriented user agents by providing a systematic method to derive customized versions of the agents. The main goal is to tailor service provision to the preferences and circumstances of the user requesting the service. The domain-based process proposed involves extended domain analysis with goals and variability, domain design with the specification of agent services and plans, and domain implementation. The approach takes advantage of the interplay of SOA, MAS and SPL. Their comparison presented in this paper showed that they are not mutually exclusive, but complementary. Moreover, they can contribute to the problem we are addressing.

This paper addresses ongoing research on the customization of service-oriented MASs and the

integration of SOA, MAS and SPL. We aim to extend our approach in several directions, which includes the provision of dynamic adaptation of agents and the integration of the GenArch tool to support automation of the derivation process.

References

[1] Pauline Berry, Bart Peintner, Ken Conley, Melinda Gervasio, Tomás Uribe, and Neil Yorke Smith. Deploying a personalized time management agent. In *AAMAS 2006*, pages 1564–1571, 2006.

[2] Elder Cirilo, Uirá Kulesza, and Carlos Lucena. A Product Derivation Tool Based on Model-Driven Techniques and Annotations. *JUCS*, 14:1344–1367, 2008.

[3] Elder Cirilo, Ingrid Nunes, Uirá Kuleszaand Camila Nunes, and Carlos Lucena. Automatic product derivation of multi-agent systems product lines (to appear). In *SAC 2009*, 2009.

[4] Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002.

[5] Thomas Erl. *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall, 2005.

[6] Longjun Huang, Liping Dai, Yuanwang Wei, and Minghe Huang. A personalized recommendation system based on multi-agent. In *WGEC '08*, pages 223–226, 2008.

[7] Michael N. Huhns, Munindar P. Singh, Mark Burstein, Keith Decker, Ed Durfee, Tim Finin, Les Gasser, Hrishikesh Goradia, Nick Jennings, Kiran Lakkaraju, Hideyuki Nakashima, Van Parunak, Jeffrey S. Rosenschein, Alicia Ruvinsky, Gita Sukthankar, Samarth Swarup, Katia Sycara, Milind Tambe, Tom Wagner, and Laura Zavala. Research directions for service-oriented multiagent systems. *IEEE Internet Computing*, 9(6):65–70, 2005.

[8] N. R. Jennings and M. Wooldridge. Applications of intelligent agents. In *Agent technology: foundations, applications, and markets*, pages 3–28. Springer-Verlag, 1998.

[9] K. Kang, S. Cohen, J. Hess, W. Novak, and Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-021, SEI, 1990.

[10] Bruce Krulwich. Automating the internet: Agents as user surrogates. *IEEE Internet Computing*, 1(4):34–38, 1997.

[11] Jaejoon Lee, Dirk Muthig, and Matthias Naab. An approach for developing service oriented product lines. In *SPLC '08*, pages 275–284, 2008.

[12] E. Michael Maximilien and Munindar P. Singh. A framework and ontology for dynamic web services selection. *IEEE Internet Computing*, 8(5):84–93, 2004.

[13] Dirk Muthig and Colin Atkinson. Model-driven product line architectures. In *SPLC 2*, pages 110–129, 2002.

[14] Ingrid Nunes, Uirá Kulesza, Camila Nunes, Elder Cirilo, and Carlos Lucena. A domain analysis approach for multi-agent systems product lines (to appear). In *ICEIS 2009*, 2009.

[15] Ingrid Nunes, Camila Nunes, Uirá Kulesza, and Carlos Lucena. Developing and evolving a multi-agent system product line: An exploratory study (to appear). In *Agent-Oriented Software Engineering IX*, volume 5386 of *LNCS*, pages 228–242. 2009.

[16] Joaquin Pena, Michael G. Hinchey, and Antonio Ruiz-Cortés. Multi-agent system product lines: challenges and benefits. *Communications of the ACM*, 49(12):82–84, 2006.

[17] G. Picard and M. P. Gleizes. *The ADELFE Methodology – Designing Adaptive Cooperative Multi-Agent Systems*, chapter 8, pages 157–176. 2004.

[18] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. 2005.

[19] Alexander Pokahr, Lars Braubach, and Winfried Lamersdorf. Jadex: A bdi reasoning engine. In *Multi-Agent Programming*, pages 149–174, 2005.

[20] Pure::Variants. Url: http://www.pure-systems.com/, 2008.

[21] Mikko Raatikainen, Varvana Myllärniemi, and Tomi Männistö. Comparison of service and software product family modeling. In *SOAPL'07*, 2007.

[22] A. S. Rao and M. P. Georgeff. BDI-agents: from theory to practice. In *ICMAS 1995*, 1995.

[23] Axel van Lamsweerde. Goal-oriented requirements engineering: A guided tour. In *RE '01*, page 249, 2001.

[24] Yonghong Wang and Munindar P. Singh. Trust representation and aggregation in a distributed agent system. In *AAAI*, 2006.

[25] Mike Wooldridge and P. Ciancarini. Agent-Oriented Software Engineering: The State of the Art. In *AOSE*, volume 1957, pages 1–28. 2000.

[26] F. Zambonelli, N.R. Jennings, A. Omicini, and M. Wooldridge. *Agent-Oriented Software Engineering for Internet Applications*, pages 326–346. 2000.

[27] Franco Zambonelli, Nicholas R. Jennings, and Michael Wooldridge. Organizational abstractions for the analysis and design of multi-agent systems. In *AOSE*, pages 235–251, 2000.

[28] Franco Zambonelli, Nicholas R. Jennings, and Michael Wooldridge. Developing multiagent systems: The gaia methodology. *ACM Trans. Softw. Eng. Methodol.*, 12(3):317–370, 2003.