Automating the Product Derivation Process of Multi-Agent Systems Product Lines

Elder Cirilo*,Ingrid Nunes*,Uirá Kulesza†,Carlos Lucena*
*Computer Science Department
Pontifical Catholic University of Rio de Janeiro
Rio de Janeiro, Brazil
e-mail: {ecirilo,ionunes,lucena}@inf.puc-rio.br
†Department of Informatics and Applied Mathematics
Federal University of Rio Grande do Norte
Natal, Brazil
e-mail: uira@dimap.ufrn.br

***Abstract*— Agent-oriented software engineering and software product lines are two promising software engineering techniques. Recent research work explores the integration between them to allow reuse and variability management in the context of complex systems. However, the automatic product derivation process is not addressed in the current literature. In this paper, we present our proposed approach to deal with multi-agent systems product lines (MAS-PL) variability management and automatic product derivation. Our approach is implemented as an extension of the GenArch product derivation tool. A case study illustrates how the proposed approach can be used to derive products (instances) from a MAS-PL.**

***Resumo*— Engenharia de software orientada a agentes e linha de produtos são duas técnicas de engenharia de software promissoras. Trabalhos recentes exploram a integração entre essas técnicas para permitir reuso e gerência de variabilidade no contexto de sistemas complexos. No entanto, o processo de derivação automática não é abordado na literatura corrente. Este artigo apresenta uma abordagem para lidar com gerência de variabilidades e derivação automática de produtos em linhas de produto de sistemas multi-agentes (LP-MAS). Ela é implementada como uma extensão de uma ferramenta de derivação de produtos existente. Um estudo de caso ilustra como a abordagem proposta pode ser utilizada para derivar (instanciar) produtos de uma LP-MAS.**

***Keywords*-Multi-agent Systems, Software Product Lines; Application Engineering; Model-driven Development; Product Derivation Tool.**

I. Introduction

A wide range of modern software systems present several common characteristics, e.g. pro-activity, autonomy, context-awareness and high interactivity, which have been challenges for the software engineering discipline. These systems are usually distributed in dynamic and uncertain environments and have multiple loci of control. A not exhaustive list of application domains that illustrate this scenario are robotics, decision-support, personal-assistance, vehicle insurance, simulation, medical-record processing and e-commerce. These complex and distributed systems call for new software engineering methods and techniques to address their particularities. Among several approaches that have aimed at developing this kind of systems [1], [2], agent-based approaches are often the choice [3], which involve metaphors such as autonomous agents, agent goals and agent societies. As a consequence, advances in the area of Agent-oriented Software Engineering (AOSE) have been proposed through novel techniques such as methodologies, modeling languages, processes, and implementation strategies directed to Multi-agent Systems (MASs). AOSE [4] is a prominent software engineering paradigm, which addresses the analysis, design and implementation of software systems based on these higher level abstractions, i.e. agents, roles, organizations, structuring applications with autonomous, pro-active and communicative components.

Even though the contributions of the AOSE have significantly improved the development of distributed and complex systems, current AOSE methodologies have barely taken into account the adoption of extensive reuse practices that may bring an increased productivity and quality to the software development [5]. Software reuse techniques, such as component-based development, object-oriented application frameworks and libraries, patterns, have been widely used in the software engineering context to promote reduced time-to-marked, quality improvement and lower development costs. A new promising trend is Software Product Lines (SPLs), which have become a mainstream reuse practice that addresses the design and implementation of a set of domain related artifacts in order to deliver high quality customized software in a short time-to-marked by the exploitation of applications commonalities.

Only recent research [6], [7], [8] has explored the integration between SPLs and agent-based approaches, which have been denoted by Multi-agent Systems Product Lines (MAS-PLs). The main aim of MAS-PLs is the incorporation of the benefits of both SPL and MAS and allowing reuse and variability management in the context of complex systems. Research work associated with the MAS-PL development has proposed extensions of MAS methodologies [6], [7] and new processes [8] to support the analysis, design and implementation of MAS-PLs. Although these proposed approaches provided substantial advances in the MAS-PL development, they focused in the domain engineering process and little effort has been done in the application engineering process, in which the applications of the SPL are built by reusing domain artifacts and exploiting

the SPL variability. Moreover, they do not provide tool support in order to allow an effective product derivation process. Given that the success of the application engineering process is directly associated with the effectiveness of the SPL [9], there is a clear need of (i) approaches that address this process of the MAS-PL development and (ii) product derivation tools that automate the instantiation process by facilitating the selection, composition, configuration and integration of MAS-PL assets and their respective variabilities. This is particularly essential in the context of MASs, given that they typically encompass several concerns (e.g. trust, coordination, transaction, state persistence) that are implemented by different technologies, application frameworks or platforms. This fact demands managing plenty of core assets and their configuration, which in turn is highly error-prone and time-consuming without the adoption of appropriate techniques and tool support.

Modern software engineering approaches, such as Generative Programming [10] and Software Factories [11], motivate the definition of mechanisms to support automatic product derivations through the use of domain-specific languages (DSLs) and code generators. Several product derivation tools based on feature model [12] or DSLs [10], [11] have already been proposed and are used in the industry. Many of these tools are generic enough to deal with MAS-PLs implemented using agent frameworks and platforms based on object-oriented technology. However, the configuration knowledge associated with agent abstractions are not captured by these tools, which is essential information for tracing features to agent concepts/abstractions and for a better MAS-PL management and evolution.

In this paper, we present an approach that addresses the application engineering process for MAS-PLs. The main goal of our approach is to provide models to capture the configuration knowledge associated with agent abstractions, thus enabling the MAS-PL variability management and automatic product derivation. Our approach extends a model-based approach [13] by the incorporation of domain-specific architecture models and dependency links among models already provided (e.g. feature model) and the new domain-specific architecture model. This approach [13] is implemented by the GenArch tool [13], [14], a model-driven product derivation tool that aims at defining a lightweight process to achieve automatic product derivation. Using the information provided by an agent-specific architecture model, the extended tool allows to automatically deriving specific products of MAS-PLs. This model allows specifying agent abstractions and concepts, e.g. agents, goals and plans. The tool is then extended to incorporate this model and to allow deriving agents implemented with Jadex, a widely adopted agent implementation framework [15]. We evaluate our approach by comparing it with a manual derivation process based on configuration files.

The remainder of this paper is organized as follows. Section II provides a brief background on the main approaches related to this paper, i.e. MASs and SPLs. Section III gives an overview of the OLIS MAS-PL, which is used to illustrate and motivate our approach. The main contributions of this paper are presented in Sections IV, V and VI, which present our application engineering approach to allow the automatic derivation of MAS-PLs, an evaluation of our approach and point out some discussions about the proposed approach, respectively. Section VII presents related work, followed by Section VIII, which concludes this paper and gives directions for future work.

II. Background

MAS-PLs are the integration between MASs and SPLs, whose aim is to take advantage of both approaches and help on the industrial exploitation of MASs. In this section, we introduce multi-agent systems (Section II-A) and software product lines (Section II-B) in order to provide a background for the reader understand the approach presented in this paper. In addition, we present the GenArch (Section II-C), our previous work on the product derivation process of SPLs.

A. *Multi-agent Systems*

In the context of software engineering, MASs are viewed as a paradigm, which addresses the development of systems that contain many dynamically interacting components, each with their own thread of control while engaging in complex, coordinated protocols. The main idea of AOSE [4] is to decompose complex problems into autonomous, pro-active and reactive entities with social ability, namely agents. Besides the agent concept, agent-oriented approaches are based on high level abstractions, such as roles and organizations, which become natural metaphors for developing complex and distributed systems.

A main difference between an agent and an object is that the former encapsulates not only state, but also the behavior selection process and when such behaviors are necessary. Hence, agents are typically developed with cognitive abilities usually modeled as goals to be achieved, plans to achieve these goals and beliefs (mental state) necessary to execute plans.

B. *Software Product Lines*

Software product lines (SPLs) have emerged as a mainstream software development practice to promote improvements in the time-to-market, cost, productivity and software quality. A SPL comprises a set of products (variant), scoped to a specific market segment (domain) and based on inter-product commonality and variability. Software Product Line Engineering (SPLE) is a paradigm to develop software applications using mass customization, and a common and flexible architecture (platform) implemented by a set of reusable assets in order to deliver high quality software in a short time-to-marked with a significant reduced cost. SPLE typically specifies, designs and implements software products in terms of features. A feature [10] is a system property that is relevant to some stakeholder. Features are typically organized into feature models [16] and have been widely used to represent variability in a domain. It provides an ample description of the SPL requirements, capturing commonalities and discriminating among products

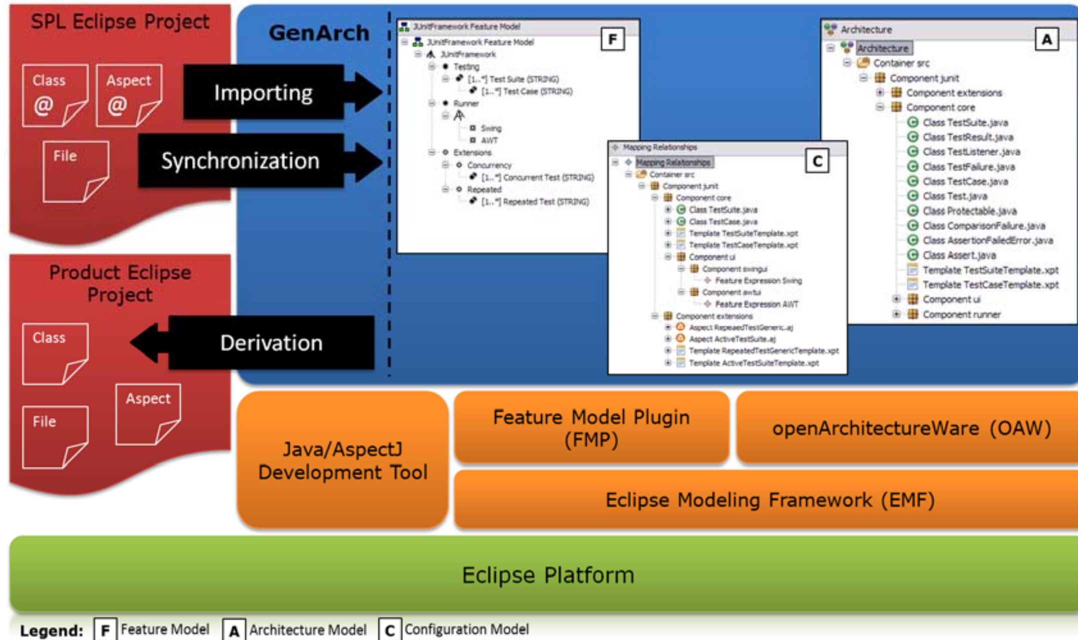

Figure 1. GenArch Approach Overview

in a SPL. In addition, a feature model also models constraints among features. A constraint can be a first-order logic expression that models dependency rules among features. It can describe: (i) illegal combinations of features; or (ii) dependence among features.

C. GenArch: a Model-based Product Derivation Tool

GenArch [13] is a model-based product derivation tool founded on generative programming [10]. The variability management in the GenArch approach is accomplished by means of three models: (i) feature model (problem space); (ii) architecture implementation model (solution space); and (iii) configuration model (configuration knowledge). The approach uses the feature model proposed in [10], which aggregates additional information, such as feature cardinality and attributes, to the one proposed in FODA [16]. The architecture model defines a visual representation of the SPL implementation elements (classes, aspects, templates, folders, files, and fragments) in order to later relate them to domain features. Fragments are used to aggregate pieces of code (or text) that implement a specific feature. They are mainly used to represent variabilities that exist in configuration files, e.g. XML or properties files. Templates are used to define incomplete implementation elements, which are either configuration files or classes and aspects, that contain common and variable code from a SPL. The variable parts of a template are customized based on information provided by the derivation models. Finally, the configuration model is responsible for defining the mapping between features and implementation elements. It represents the configuration knowledge from a generative approach [10], being fundamental to link the problem space (features) to the solution space (implementation elements). These models provide the necessary information to derive products from a SPL.

Based on this approach, the GenArch tool offers a code-oriented variability management, which supports automatic product derivation based on three main steps: (1) automatic models construction; (2) artifacts synchronization; and (3) product derivation. Figure 1 shows an overview of the GenArch approach. The importing module enables the creation of initial versions of the derivation models (Step 1) by parsing the code assets that implement the SPL architecture. It is based on specific annotations to characterize that a particular Java code asset addresses a specific feature (`@Feature`) and/or represents a variation point (`@Variability`), such as a hotspot framework class. The synchronizer module keeps the consistency between GenArch derivation models and SPL code assets (Step 2).

The GenArch product derivation process (Step 3), i.e. the customization and compositions of the SPL architecture, is driven by an instance (also known as configuration) of the feature model. So, the first step of the GenArch product derivation process is the creation, by an application engineer, of this feature model configuration. During the derivation process, GenArch decides, based on both feature model configuration and configuration model, which code assets will be selected and customized. Template technology is used to implement the elements that must be customized in the derivation time. A template is able to collect information from derivation models to customize its respective variable parts of code. More details about GenArch templates can be found in [13]. The GenArch derivation process is concluded with code generation based on the processing of templates and the loading of the selected and generated code assets. Additional details about GenArch tool can be found in [13].

1) GenArch Architecture Overview

GenArch has been developed as an Eclipse [17] plug-in using different generative technologies available for this platform (see Figure 1). Some of them are: (i) Eclipse Modeling Framework (EMF) [18] – used to specify GenArch derivation models; (ii) openArchitectureWare (oAW) [19] – used to deal with templates; and (iii) Java/Aspect Development Tooling (JDT/AJDT) [20] API – used to browse the Abstract Syntax Tree (AST) of Java classes and AspectJ aspects. This allows us to parse code assets (classes, aspects, configuration files), and process Java annotations and metadata in order to enable the automatic creation of GenArch models.

The feature model used in our tool is specified by a separate plugin, called FMP (Feature Modeling Plugin) [21]. It allows modeling the feature model proposed in [10], which supports modeling mandatory, optional, and alternative features, and their respective cardinality.

III. OLIS MAS-PL: THE MOTIVATING EXAMPLE

In this section, we present the OLIS case study, which is a MAS-PL of web systems that provide several personal user services, such as calendar and events announcement. The OLIS product line defines services that can be configured to automate user tasks by means of software agents (optional features). Because of that, it can be considered a MAS-PL, which enables the introduction and customization of some of its agents. The OLIS case study will be used, in next section, to exemplify how the approach for the derivation process of MAS-PLs proposed in this paper is able to automatically derive MAS-PL members.

The four main services that compose OLIS are: (i) User Management – allows users to register themselves and configure their account; (ii) Events Announcement service – allows users to announce events to other system users through an events board; (iii) Calendar service – allows users to schedule events in their calendar. Besides the information of events published in the events board, calendar events have a list of users that participate of it. Additionally, announced events can be imported to the users' calendar; and (iv) Weather service – provides information about the current weather conditions and the forecast of a location. Besides these services, the OLIS MAS-PL provides an alternative feature: the event type. The product line can derive systems for dealing with generic, academic and travel events. Figure 2 shows the feature model of the OLIS MAS-PL, detailing its services and their optional features. Additional details about the OLIS architecture and implementation can be found in [22].

There are different customizations that can be applied to the OLIS services. These customizations represent optional features of the OLIS MAS-PL. Examples of such customizations are: (i) Events Reminder – sends notifications to notify the user about events that are about to begin; (ii) Events Scheduler – checks the event participants' schedule to verify if a new event conflicts with other existing ones. In this case, the system suggests a new date for the calendar event that is appropriate according to the participants' schedule; and (iii) Events Suggestion – automatically recommends events based on user preferences.

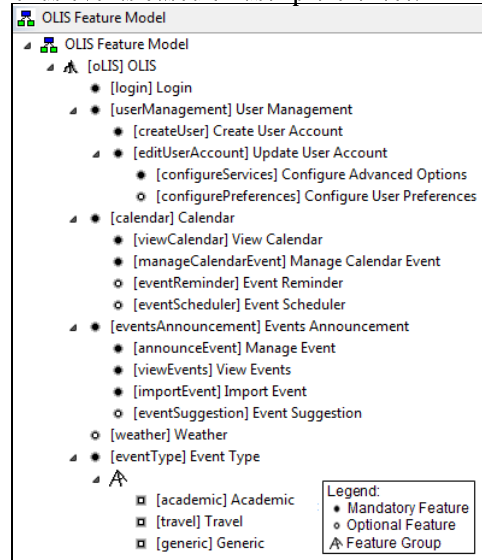

Figure 2. OLIS Feature Model

Most of the optional OLIS features provide a pro-active and autonomous behavior for the system services. Due the suitability of the agent abstraction to model this kind of behavior, we have introduced software agents and agent roles into the architecture to implement these features, which we named agent features. This autonomy property refers to agents able to act without the intervention of humans or other systems: they have control both over their own internal state, and over their behavior [23]. OLIS agent features are realized by the Jadex framework. This framework addresses developing agents that follow the belief-desire-intention (BDI) model [24]. Through a reasoning engine, this framework makes it easier the development of cognitive agents. The Jadex provides agent concepts as first-class elements, which are agents, believes, goals, plans, capabilities, events and expressions. In particular, a Jadex capability allows that beliefs, goals and plans to be placed together in a separated module, in a way that this module can be reused by different agents.

The OLIS MAS-PL was designed in such way that the system can be evolved to incorporate new services without interfering in the existing ones. It was structured according to the Layer architectural pattern. The layers that compose the architecture are GUI, Business, and Data.

A. Challenges on MAS-PL Product Derivation

The application domain that we are currently exploring is MAS-PLs of web-based systems. Applications in this domain typically present a set of different concerns, e.g. transaction, persistence, pro-activeness and autonomy. Usually such concerns are resolved at the implementation level by different application frameworks. For example, in the OLIS case study, the pro-activeness and autonomy concerns are achieved by means of the Jadex framework. The use of these frameworks introduces new concepts in MAS-PL architectures that must be considered in their

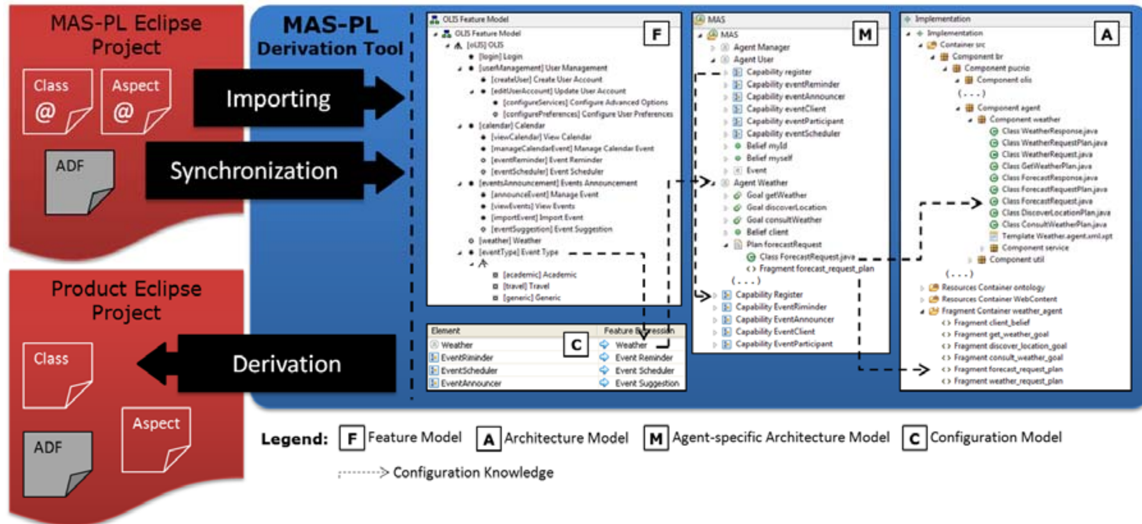

Figure 3. GenArch approach extended to MAS-PL

documentation. In addition, several frameworks are based on configuration files that must be manipulated in the product derivation process of product lines.

The OLIS MAS-PL provides five optional features and one alternative feature with three different options, which lead to 96 different product configurations, not taking into account dependencies between features that restrict some of these product configurations. Even though the OLIS architecture was developed with appropriate techniques to modularize its features, the application engineer still needs to manually configure the artifacts that implement the MAS-PL, including configuration files, during the derivation process, which is a challenging task. First, to ensure the reliability and consistency of the derivation process the right implementation elements must be selected. This selection must be in accordance with the selected features and dependencies among them. For instance, if the *Event Suggestion* feature was selected for a certain product, it must also contain the *Configure User Preferences* feature due to a feature constraint. Consequently, in the solution space, the agent that implements the *Event Suggestion* feature depends, directly, on the code assets that implement the *Configure User Preferences* feature. This process corresponds to the propagation of the feature selection from the problem space to implementation elements (solution space). Second, in the OLIS MAS-PL, for example, to manage variabilities and their dependencies, the application engineer needs to configure eleven Jadex agent definition files, which together add more than one thousand lines of code. Additionally, most of OLIS MAS-PL features have their implementation spread over more than two files, which makes it difficult and increase the complexity of understanding the mapping between different elements from the problem to the solution space.

This scenario shows that without suitable variability management mechanisms to modularize the configuration knowledge and to deal with the large amount of variabilities, dependencies among features and heterogeneous implementation elements, the manual derivation process may become error-prone and time-consuming.

Furthermore, despite some works [25], [26], [27] have stated how to automatically resolve dependencies among features in the feature model, there is still a lack of MAS-PL specific mechanisms that enable the propagation of this decision made in the problem space (feature model) to the solution space. A particularity of MAS-PLs is that they present additional abstractions, such as agents, beliefs and goals. MASs are designed with such abstractions, which are typically implemented based agent platforms that rely on object-oriented languages. Therefore, MAS-PLs require dealing with three different levels of abstractions: (i) feature; (ii) MAS design abstractions; and (iii) implementation elements. Current approaches allow mapping from (i) to (iii), however the configuration knowledge related to MAS abstractions are not captured by them.

In this context, next section details an extension of the model-driven approach used in the GenArch approach that addresses the variability management and automatic product derivation of MAS-PLs.

IV. An Approach for Automating the Derivation Process of MAS-PLs

In Section 2, we presented the GenArch model-driven approach that allows to automatically deriving products from SPL architectures. In this section, we present how this approach was extended to address the variability management and automatic product derivation of MAS-PLs. Our approach was implemented as an extension of the GenArch approach. Figure 3 depicts the approach proposed in this paper, highlighting the differences from the previous version (Section II-C).

Basically, our solution to manage the configuration knowledge associated with MAS-PLs is based on the agent-specific architecture model. It provides modular variability management of agents and their related concepts and underlying implementation elements. This architecture

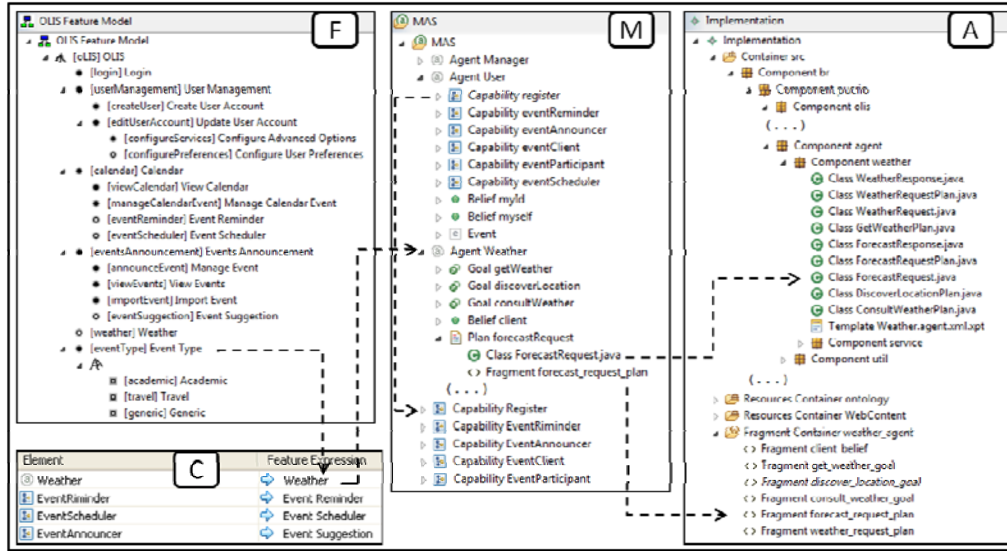

Figure 4. OLIS Agent-architecture Model and Configuration Knowledge

model is based on higher level abstractions, which latter are mapped to lower level ones. Lower level elements in our approach are any object-oriented concept, or configurations defined into Agent Definition Files (ADFs). An ADF is an XML file that captures a complete definition of an agent or a capability in the Jadex framework. It contains all relevant properties of an agent (e.g. beliefs, goals and plans).

In next sections, we detail the approach describing (i) the agent-specific architecture model (Section IV-A); (ii) how to document and model the configuration knowledge (Section IV-B); (iii) how agent-specific implementation code assets (Jadex) are processed to create an initial version of the models (Section IV-C); and (iv) how the new models are used to automatically derive MAS-PL products, i.e. the MAS-PL derivation process (Section IV-D). Along all sections, our approach is illustrated with the OLIS MAS-PL, which was described in previous section.

A. *Agent-specific Architecture Models*

The aim of the agent-specific architecture model is to provide a formal and modular solution to document the configuration knowledge of MAS-PLs based on agent abstractions – it describes the MAS-PL architecture by means of agents, capabilities, goals, believes, and so on.

Figure 4(M) shows the agent-specific architecture model that describes the OLIS implementation architecture using Jadex vocabulary. This model characterizes Jadex agents as an aggregation of capabilities, goals, beliefs, plans, events, and expressions. A capability, similar to an agent, can be associated with goals, beliefs, plans, events, expressions, and others capabilities. Finally, plans, goals, events, beliefs and expressions are simple elements. Figure 4(M) illustrates, for example, that the `Weather` agent encompasses three goals (`getWeather`, `discoverLocation`, `consultWeather`), one belief (`client`), and five plans (`forecastRequest`, `weatherRequest`, `getWeather`, `discoverLocation`, `consultWeather`).

The agent-specific architecture model was specially designed to allow GenArch to deal not only with object-oriented (Java) and aspect-oriented (AspectJ) elements (Section II-C), but also with Jadex ADFs during the derivation process (Figure 3).

B. *Adding New Levels to the Configuration Knowledge*

Besides documenting the SPL architecture by means of agent's concepts, the agent-specific architecture model enables defining mapping relationships between these concepts and their respective implementation elements. For example, Figure 4 illustrates that the `ForecastRequest.java` and a code fragment called `forecast_request_plan` are directly related to the `forecastRequest` plan of the `Weather` agent.

For the derivation process, each reference defines an implication constraint where the presence of the related lower level element depends on the positive evaluation of the presence of the higher level one. In other words, it means that, if the higher level element must be part of one product, then the associated lower level element must also compose this product. In the scenario illustrated in Figure 4, for example, if the `Weather` agent is selected to be part of one product, the `ForecastRequest.java` and `forecast_request_plan` must also be part of this product.

The extended configuration model provided by our approach (Figure 4(C)) allows the domain engineer to define different levels of configuration. Fine-grained configurations can be created by the default mapping relationships of specific implementation elements (classes, aspects, files) to any product line feature [13]. Coarse-grained mapping relationships of Agent elements to product line features can be defined by domain engineers in specific views into the configuration model (Figure 4(C)). A mapping relationship between a feature expression and an architectural element (configuration model) defines an implication constraint

where the presence of the architecture element depends on the positive evaluation of the related feature expression.

As it can be seen in Figure 4(C), the *Weather* optional feature was mapped to the `Weather` Agent. It means that the `Weather` Agent will only be part of the final product if the *Weather* feature is selected in the feature model configuration provided by an application engineer. The `EventReminder`, `EventAnnouncer` and `EventScheduler` variable capabilities are also mapped to specific features (*Event Reminder*, *Event Announcement* and *Event Scheduler*, respectively) and are subjected to the same evaluation.

C. Automating the Generation of Agent-specific Architecture Models

One of the main targets of our approach is to provide functionalities for parsing code assets metadata in order to automatically generate initial versions of the derivation models. Feature, configuration and architecture models can be automatically created by parsing: (i) the Eclipse Java project or directory that contains the implementation elements of a MAS-PL; (ii) the GenArch annotations introduced into the source code of Java classes; and (iii) the Jadex ADF description files. In Section II, we presented the automatic parsing of GenArch annotations. In this section, we show how Jadex-specific artifacts are processed to derive an initial version of their respective architecture model.

The creation of Agents, Capabilities, Goals, Plans, Events and Expression elements in the agent-architecture model is accomplished by means of an automatic parsing of code assets (Jadex ADF and XML tags) that implement these elements (Figure 3). Each Jadex ADF can describe either agents or capabilities. Therefore, an ADF can demand the creation of an Agent or Capability element in the agent-specific architecture model. The Goals, Plans, Events and Expressions are created from respective tags described in the ADF. The source codes that implements these elements are extracted to implementation model fragments. It also demands the creation of a mapping between the created implementation model fragments and the respective agent-specific architecture model element.

The initial version of these models, built automatically, may or must be refined by the domain engineer. This refinement is necessary in order to guarantee that: (i) the feature model represents all domain variability and commonalities; (ii) the architecture model represents the whole MAS-PL code assets; (iii) the agent-specific architecture model expresses the entire design of the agents that compose the MAS-PL; and (iv) additional mapping relationships between variability in the feature model and agent elements in the agent-architecture model.

D. The MAS-PL Automatic Product Derivation Process

As mentioned previously, our approach was implemented by extending the GenArch tool. Additional details of the implemented tool can be seen in [28]. In this section, we describe the derivation process of our approach, with some implementation details. This implementation is specific for the Jadex framework.

The GenArch derivation process for the OLIS MAS-PL (or each MAS-PL based on the Jadex framework) is accomplished by the following steps: (i) selection of the Agents, Capabilities, Plans, Goals, Events, Expression that will compose the architecture of the instance (product) from the MAS-PL; (ii) selection of the code assets (class, aspects, files, components, folders) that will be part of the derived product; and (iii) customization of Jadex ADFs – ADFs are XML files that define agents and capabilities.

The selection of agents, capabilities and other abstractions from the Jadex framework (step 1) is accomplished based on the configuration knowledge provided by the configuration model, which relates features to agent concepts. After that, the GenArch tool uses the information provided by the agent-specific architecture model (step 2) that relates agent concepts with implementation elements; to decide which implementation elements (classes, interfaces, aspects, etc.) will be part of the final product generated. The selection process is supported by the dependency links defined in both agent-specific architecture model and configuration model. To automatically compute these dependency links, we map them into a constraint satisfaction problem (CSP). Consequently, the CSP is evaluated with respect to a valid feature model configuration by a constraint solver. The purpose of a constraint solver is to find a valid value for each variable of the CSP that simultaneously satisfies all constraints in the CSP. Previous work [25], [26], [27] has shown how to transform feature models into a CSP in order to automate, for example, feature selection. In our work, we extend this technique to help the resolution of the implementation elements selection in the solution space.

The customizations of the OLIS Jadex Agent and/or Capabilities ADFs are realized by means of template files (step 3). Figure 5 shows a summarized version of the template that implements the `UserAgent` ADF. This part of the code enables GenArch to customize the capabilities the `UserAgent` will contain. The **`LET`** statement enables the template to get the specified element (`UserAgent`) from the agent-specific architecture model. The **`FOREACH`** statement enables the template to iterate through the collection of capabilities of the `UserAgent`. Thus, for each capability presented in this collection, the template gets the code fragment associated with it, which is also defined in the agent architecture model (see Figure 4(M)), and writes the content in the generated file. It means that, if the application engineer selects the *Event Reminder* and *Event Scheduler* optional features and does not select the *Event Suggestion*, consequently the derived collection of capabilities will not contain the `EventSuggestion` capability. The content of each code fragment related with the selected elements will be written in the generated `UserAgent` ADF, as shown in Figure 5.

V. Evaluation

This section summarizes our experience with the use of our approach to enable the automatic product derivation of MAS-PLs implemented using the Jadex framework. We use

```
01. «IMPORT br::pucrio::inf::les::genarch::models::instance»
02. «EXTENSION br::pucrio::inf::les::genarch::models::Model»
03. «DEFINE Main FOR Instance»
04. «FILE "User.agent.xml"-»
05. (...)
06. <capabilities>
07. «LET domainModelElement("mas","User",domainModels) AS u»
08.      «FOREACH u.capabilities AS c»
09.      «c.fragment.content»
10.      «FOREACH»
11. «ENDLET»
12. </capabilities>
13. (...)
14. «ENDFILE»
15. «ENDDEFINE»
```

```
<capability name="eventremindercap"
file="br.pucrio.olis.agent.eventreminder.EventReminder
```

Figure 5. User Agent ADF Template

the OLIS case study to conduct a preliminary evaluation of the gain and effort of using our approach.

TABLE I. CONFIGURATION COST OF THE OLIS MAS-PL

Feature	Lines of Code/Files
Configure User Preferences	3/1
Event Reminder	114/2
Event Suggestion	237/2
Academic	125/2
Event Scheduler	362/3
Travel	175/2
Generic	0
Weather	224/1
Total	1240/13

Our first step was to quantify the cost of manually deriving products from the MAS-PL in terms of the number of files (and their size) needed to be manipulated in the derivation process, i.e. we have counted how many XML files and their lines of code an application engineer should deal with during the derivation process. Table I summarizes the total of lines of XML code that must be manipulated for each optional and variable feature implemented by the OLIS MAS-PL during the derivation process. Without appropriate mechanisms to modularize and make the configuration knowledge related with Jadex elements explicit, the application engineer would be required to know about the manual configuration of more than one thousand lines of XML code spread over 13 XML configuration files.

By modeling the OLIS MAS-PL using the new agent-specific architecture model we have assessed that it helps to reduce 100% the total amount of lines of XML code that must be manipulated. Once the models proposed in our approach are generated, the derivation process is performed automatically. In addition, this preliminary assessment also states the importance of the agent-specific architecture model to face the challenge associated with configuration knowledge that is usually found spread over different configuration files, as discussed in Section III-A. Table I also shows that most of the features have their configuration knowledge described in more than two files. In this way, our agent-specific architecture model also contributes to reduce the spreading of the configuration knowledge by aggregating information about all the configuration files in a unique model representation.

Besides the benefits related to the amount of lines of XML code and modularization of the configuration knowledge, we believe that the agent-specific architecture model also brings advantages to the product line engineer to understand, localize and modify both the SPL architecture implementation and the associated configuration knowledge, thus contributing directly to the variability management.

Although our approach brings the advantages discussed above, it requires an extra initial investment of building the derivation models. It also requires the addition of extra mapping relationships on the configuration model that maps feature to architectural elements (classes, files, agents, etc). Therefore, the second step of our evaluation was to measure the cost of building the models of our approach. The initial effort needed to build our approach models and its respective configuration knowledge was assessed by counting the number of operations needed to accomplish this task. An operation is (i) create a model; and (ii) include a model element. Table II presents the results for the OLIS MAS-PL.

TABLE II. MODELS CREATION COST OF THE OLIS MAS-PL

Feature	Number of Operations
Feature Model	24
Agent-specific Architecture Model	203
Implementation Model	585
Configuration Model	13
Total	825

Comparing the number of lines of code that must be manipulated and the number of operations of creating models, it can be seen that the latter is lower than the former. So, it indicates that the cost of adopting our approach is lower than manually deriving products. However, this comparison is between lines of code and operations, therefore we are aware that this evaluation may be unfair. As stated previously, this is our preliminary evaluation, and our goal is to perform a more sophisticated evaluation of our approach.

Furthermore, even though there is an initial cost of building the derivation models, it is spent only once, rather than each time of a product derivation. In the literature, it is discussed that the initial investment and the time-to-market of building a SPL is higher than building a single product. However, according to [29] this effort is usually compensated after the third derived product. Finally, the process of building an initial version of the architecture model and of the agent-specific architecture model can be amortized by the GenArch automatic model generation feature (described in Section IV-C). The simple model generation feature implemented by GenArch is able to create a complete version of these models, which are 95% of the effort to build the whole configuration knowledge. We are currently exploring new mechanisms to improve the generation of this additional configuration knowledge from existing code artifacts (configuration files, classes, aspects, etc).

VI. Discussions

Generalizing our Approach to Other Technologies. Despite the focus on the Jadex framework, our approach is generic enough to deal with different agent frameworks or platforms. In fact, GenArch has been evolving to incorporate new component technologies and aspect-oriented programming [13], [14]. This work is a first step in the direction of a generic derivation tool based on multiple domain and platform specific models. Multi-models can also improve the management and traceability of SPL features bringing several benefits to the change impact analysis during the SPL evolution. However, keeping the dependency links between these models and artifacts updated, synchronized and consistent is a difficult task. The configuration knowledge between feature models and the Jadex platform-specific model provided by our GenArch extension is a prominent solution to help the automatic generation of dependency links between models from problem to solution space. In the case of MAS development, the generation of the dependency links is still more important due to large amount of models and abstractions that the paradigm requires to leverage the abstraction level of software development.

Using our approach to provide self-adaptation. Over the last years, the software engineering community has investigated how to support the development of dynamic evolutionary systems. A MAS can be seen as an open and evolutionary system, where agents can enter and leave at any time and dynamically modify its structure. Due to the autonomy property of agents, they also provide a low coupling model in which a change on any entity does not deeply affect the entire system. Thus, it can be adopted as a technique to implement software systems, in which an autonomous and pro-active agent is able to manage itself. On the other hand, Dynamic Software Product Line Engineering (DSLPE) [30] has emerged motivated by the application of SPLs in several dynamic domains, such as ubiquitous computing, context-aware computing, and autonomic computing. DSLPE can promote the reuse of domain-specific adaptations across a family of related self-adaptive products while providing a systematic approach for dynamic variability management [30]. This paper can be seen as an initial attempt toward a SPL based approach to support self-adaptation in MASs. Each different state of the MAS can be seen as a MAS-PL product. Feature model and Multi-level models can be used to provide to the system the capability to describe its properties and to reason about the possible adaptations at different abstraction levels. We are currently investigating how GenArch can also provide a feature derivation process, which enables the achievement of automatic feature deployment and reconfiguration.

VII. Related Work

Only few works that consider the automatic development of MASs have been proposed. Kulesza et al. [31] present a generative approach that addresses the challenges in MAS modeling and development, mainly focusing on crosscutting agent features. The approach proposed by these authors is composed of: (i) a domain-specific language (Agent-DSL) used to model the orthogonal and crosscutting features of software agents; (ii) a code generator; and (iii) an aspect-oriented architecture that encompasses a set of aspectual components that modularize the crosscutting agent features. The code generator is used to map the abstractions described in the Agent-DSL to specific compositions of objects and aspects in the proposed agent architecture. The approach proposed in this paper can be seen as an evolution of this previous approach by providing more extensible mechanisms to derive different domain and technology specific product lines and applications. In this paper, for example, we have demonstrated that our approach is suitable to address the product derivation of Jadex based applications.

Hahn [32] presents a set of platform independent domain specific languages (DSLs), called DSML4MAS, which enable the definition of MASs in a graphical visualized manner. These DSLs are used to automatically generate a MAS in a model-driven architecture (MDA) manner. Similar to our approach, this work enables the MAS development based on the definition of high-level models. However, it is concerned only with the automatic generation of single systems, not addressing the development and automatic derivation of system families or SPLs. Our approach enables the mapping between MAS DSLs and feature models in order to allow the modeling and implementation of MAS-PL architectures with their respective variabilities, as well as to help the process of automatic product derivation of MAS-PLs.

VIII. Conclusion and Future Work

In this paper, we presented an approach that enables the automatic product derivation for Multi-Agent Systems Product Lines. The approach extended a model-driven approach implemented by GenArch, which is our existing product derivation tool. The main idea of our approach is to incorporate the agent-specific architecture model to the SPL specification in order to deal with higher level concepts, including agent concepts such as beliefs, goals and plans. In addition, we have provided means of linking this new model to the existing ones (feature and implementation models) to

have the configuration knowledge that is needed for automating the product derivation process. Our approach was implemented as an extension of the GenArch tool. Besides providing the new model, we improved the tool to deal with files of the Jadex platform in order to generate an initial version of the model. The main benefit of our approach is to allow the automatic product derivation of MAS-PLs, however we point out other advantages: (i) with the domain-specific model, the knowledge of higher level elements are not spread and obfuscated into the code; and (ii) the quality and time of the derivation process are improved because dealing manually with lots of files and configurations is an error-prone and time-consuming task. We illustrated the use of our approach and the MAS-PL GenArch extension with the OLIS case study, which is a MAS-PL of web applications that provide personal services to users. We also presented a preliminary evaluation of our approach to show its effectiveness.

As a future work, we aim at extending our approach to address the composition of different domain-specific architecture models. In the OLIS case study, for example, we are also using a Spring-specific architecture model [14], thus reflecting the composition between two different domain-specific architecture model. We are currently working in a flexible approach that addresses this kind of composition exploring extensible mechanisms at the metamodel level. Finally, we intend to apply the tool in more complex and different MAS-PL case studies with the presence of both coarse and fine-grained variabilities.

ACKNOWLEDGMENTS

This work has been partially supported by CNPq 557.128/2009-9 and FAPERJ E-26/170028/2008. It is related to the following topics: Software technologies for web applications - Model-driven Design and Implementation of Web Applications - G3. Develop methodologies, empirical studies and tools to support the development of software product lines for the Web context.

REFERENCES

[1] G. Booch, *Object-Oriented Analysis and Design with Applications (3rd Edition)*. USA: Addison Wesley, 2004.

[2] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*. USA: Addison-Wesley, 2002.

[3] N. R. Jennings, "An agent-based approach for building complex software systems," *Commun. ACM*, vol. 44, no. 4, pp. 35–41, 2001.

[4] M. Wooldridge and P. Ciancarini, "Agent-oriented software engineering: the state of the art," in *AOSE 2000*, 2001, pp. 1–28.

[5] R. Girardi, "Reuse in agent-based application development," in *SELMAS '02*, 2002.

[6] J. Pena, M. G. Hinchey, A. Ruiz-Corts, and P. Trinidad, "Building the core architecture of a multiagent system product line: with an example from a future nasa mission," in *AOSE'06*, 2006.

[7] J. Dehlinger and R. R. Lutz, "Supporting requirements reuse in multi-agent system product line design and evolution," in *ICSM*, 2008, pp. 207–216.

[8] I. Nunes, C. Lucena, U. Kulesza, and C. Nunes, "On the development of multi-agent systems product lines: A domain engineering process," in *AOSE'09*, 2009, pp. 109–120.

[9] S. Deelstra, M. Sinnema, and J. Bosch, "Product derivation in software product families: a case study," *J. Syst. Softw*., vol. 74, no. 2, pp. 173–194, 2005.

[10] K. Czarnecki and U. W. Eisenecker, *Generative programming: methods, tools, and applications*. USA: Addison-Wesley, 2000.

[11] J. Greenfield, K. Short, S. Cook, and S. Kent, *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. Wiley, 2004.

[12] pure systems, "pure::variants," http://www.puresystems.com/, 2009.

[13] E. Cirilo, U. Kulesza, and C. J. P. de Lucena, "A product derivation tool based on model-driven techniques and annotations." *J.UCS*, vol. 14, no. 8, pp. 1344–1367, 2008.

[14] E. Cirilo, U. Kulesza, R. Coelho, C. J. Lucena, and A. Staa, "Integrating component and product lines technologies," in *ICSR '08*, 2008, pp. 130–141.

[15] L. Braubach and A. Pokahr, "Jadex bdi agent system," http://jadex.informatik.uni-hamburg.de, 2009.

[16] K. Kang, S. Cohen, J. Hess, W. Novak, and Peterson, "Feature-oriented domain analysis (foda) feasibility study," Software Engineering Institute, Carnegie-Mellon University, Tech. Rep. CMU/SEI-90-TR-021, November 1990.

[17] T. E. Foundation, "Eclipse.org home," 2009, http://www.eclipse.org.

[18] F. Budinsky, D. Steinberg, E. Merks, R. Ellersick, and T. J. Grose, *Eclipse Modeling Framework*. Addison-Wesley, 2003.

[19] openArchitectureWare.org Official openArchitectureWare Homepage, "openarchitectureware.org," 2009, http://www.openarchitectureware.org/.

[20] S. Shavor, J. D'Anjou, S. Fairbrother, D. Kehn, J. Kellerman, and P. McCarthy, *The Java(TM) Developer's Guide to Eclipse*. Addison-Wesley, 2003.

[21] M. Antkiewicz and K. Czarnecki, "Featureplugin: feature modeling plug-in for eclipse," in *eclipse '04*, 2004, pp. 67–72.

[22] I. Nunes, "Towards a multi-agent product line development methodology," 2008, http://www.inf.pucrio.br/˜ionunes/maspl/.

[23] M. Wooldridge, *Intelligent Agents*. England: The MIT Press, 1999, ch. 1, pp. 27–78.

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

[25] D. B. Pablo, P. Trinidad, and A. Ruiz-corts, "Automated reasoning on feature models," in *CAiSE 2005*, 2005.

[26] J. White, D. C. Schmidt, D. Benavides, P. Trinidad, and A. Ruiz-Cort´es, "Automated diagnosis of product-line configuration errors in feature models," in *SPLC '08*, 2008, pp. 225–234.

[27] J. White, D. Benavides, B. Dougherty, and D. C. Schmidt, "Automated diagnosis of product-line configuration errors in feature models," in *SPLC '09*, 2009.

[28] E. Cirilo, I. Nunes, U. Kulesza, and C. Lucena, "A multiagent systems product lines derivation tool (to appear)," in *ICSR 2009 – Tool Demos, 2009*.

[29] K. Pohl, G. Bckle, and F. J. van der Linden, *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag, 2005.

[30] S. Hallsteinsen, M. Hinchey, S. Park, and K. Schmid, "Dynamic software product lines," Computer, vol. 41, no. 4, pp. 93–95, 2008.

[31] U. Kulesza, A. F. Garcia, C. J. P. de Lucena, and P. S. C. Alencar, "A generative approach for multi-agent system development," in *SELMAS, 2004*, pp. 52–69.

[32] C. Hahn, "A domain specific modeling language for multiagent systems," in *AAMAS '08*, 2008, pp. 233–240.