2
Background on Multi-agent Systems and Software Product Lines

Multi-agent System Product Lines (MAS-PLs) aim at integrating Software Product Lines and agent-oriented techniques by incorporating their respective benefits and helping the industrial exploitation of agent technology. This Chapter gives an overview of these two technologies, detailing some of their methodologies and processes. Furthermore, a comparative analysis among Software Product Line approaches is presented in Section 2.1.5. The purpose of this investigation is to identify how Software Product Lines and agent-oriented techniques can help in the development of MAS-PLs.

2.1
Software Product Lines

Software Product Lines (SPLs) are a new trend in the context of software reuse that provide a way to design and implement several closely related systems together by changing the form of reuse from an *ad-hoc* to a systematic way. The term family of programs was first introduced by Parnas in (Parnas 1976), defining it as a set of programs with so many common properties that it is an advantage to study these common properties before analyzing individual members. Nowadays, the concept given to a SPL is similar to this definition (Clements 2002): “a set of software intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way”. According to (Czarnecki 2000), a feature is a system property that is relevant to some stakeholder and is used to capture commonalities or discriminate among products in a product line. The features are organized into a coherent model referred to as a feature model (originally proposed in (Kang 1990)), which specifies the features of a product line as a tree, indicating mandatory, optional and alternative features. Mandatory features are part of the SPL core and are present in all products derived from it. Optional features are present just in some members of the SPL and alternative features are the ones that vary from one member to another.

Features are essential abstractions that both customers and developers understand. The main aim of SPLE is to analyze the common and variable features of applications from a specific domain, and to develop a reusable infrastructure that supports the software development. Variability management is the major product-line-unique discipline that must be newly established within non-product-line organizations. It is responsible for systematically managing the scope itself, and ensuring its traceability with genericity of product line artifacts.

There are several motivations for the adoption of a SPL approach, being the three main ones the reduction of developments costs, enhancement of quality and reduction of time-to-market. The costs are reduced by reusing artifacts to derive products from the SPL. This is the same reason for a reduced time-to-market. The enhancement of quality is achieved by reviewing and testing the SPL artifacts in many products. Nevertheless, the development of reusable artifacts requires an up-front investment and a higher time-to-market in the initial phases of the SPL development, but this effort is usually compensated after the third derived product (Pohl 2005). Additional motivations for SPL are: (i) reduction of maintenance effort – whenever an artefact from the SPL is changed, the changes can be propagated to all products in which the artefact is being used; (ii) organized evolution – the introduction of a new artefact into the SPL core gives an opportunity for the evolution of all kinds of products derived from the platform; (iii) complexity reduction – SPLs provide a structure that determines which components can be reused at what places by defining variability at distinct locations.

Migrating from a single-system engineering to a SPL approach has far-reaching consequences. It usually requires a company reorganization in order to establish new modular units compromised with the SPL management or even though to redefine processes, workflows and technologies that are being used. According to (Krueger 2001), SPL adoption strategies are classified in three different categories: proactive, reactive and extractive. The proactive approach motivates the development of product lines considering all the products in the foreseeable horizon. A complete set of artifacts to address the product line is developed from scratch. In the extractive approach, a SPL is developed starting from existing software systems. Common and variable features are extracted from these systems to derive an initial version of the SPL. Finally, the reactive approach advocates the incremental development of SPLs. Initially, SPL artifacts address only a few products. When there is a demand to incorporate new requirements or products, common and variable artifacts are incrementally extended in reaction to them. Commonly, a proactive approach demands a huge up-front investment to develop artifacts that address the whole family of systems – as it was previously stated, the pay-off is around three systems. All the effort is concentrated at the initial phases of the SPL development. In addition, companies might not take the risk to invest in a SPL development, considering that the requirements can change. Nevertheless, the adoption of the reactive and extractive approaches allow the amortization of this investment, distributing it in incremental cycles. These approaches are not mutually exclusive. A combined strategy can be, for instance, to develop a SPL based on legacy software and incrementally evolve the architecture to address new features, enabling the derivation of new products. This means that first an extractive approach is adopted, followed by a reactive approach.

To enable SPLE, a well-accepted convention is to divide the engineering process into two different processes: domain engineering and application engineering. Domain Engineering is the process of SPLE in which the commonality and the variability of the SPL are defined and realized. It is responsible for scoping the SPL, and ensuring that the core assets have the variability that is needed to support the desired scope of products. Domain engineering approaches (Kang 1990, Prieto-Diaz 1999, Almeida 2007) aim at collecting, organizing, and storing past experience in building systems or parts of systems in a particular domain in the form of reusable assets, as well as providing an adequate means for reusing these assets when building new systems (Czarnecki 2000). Domain engineering encompasses three main process components: (i) Domain Analysis – the main concepts and activities in a domain are identified and modeled using adequate modeling techniques. The common and variable parts of a family of systems are identified; (ii) Domain Design – whose purpose is to develop a common system family architecture and production plan for the SPL; and (iii) Domain Implementation (or Realization) – involves implementing the architecture, components, and the production plan using appropriate technologies. The qualifier "domain" emphasizes the multisystem scope of these process components. Application Engineering is the process of SPLE in which applications of the SPL are built by reusing domain artifacts and exploiting the SPL variability. It takes the common assets of the SPL and uses them to create products. Domain engineering and application engineering can be called *engineering-for-reuse* and *engineering-with-reuse*, respectively.

Next sections describe some of the main SPL approaches, detailing their key concepts and method structures. Next, a comparative analysis of these approaches is presented (Section 2.1.5). This study helped to understand these approaches and identify their usefulness and deficiencies while documenting and modeling MAS-PLs.

2.1.1
Feature-Oriented Reuse Method (FORM)

Feature-Oriented Reuse Method (FORM) (Kang 1998) is a systematic method that analyzes applications in a domain in order to identify their commonalties and differences in terms of features. The analysis results are then used to develop domain architectures and components. The model that captures the commonalties and differences is called the feature model and it is used to support both engineering of reusable domain artifacts and development of applications using the domain artifacts. FORM extends Feature-Oriented Domain Analysis (FODA) (Kang 1990) to the software design phase and prescribes how the feature model is used to develop domain architectures and components for reuse.

The FORM method consists of two major engineering processes: domain engineering and application engineering. There are three phases in FORM domain engineering: context analysis, domain (or feature) modeling, and architecture (and component) modeling. FORM proposes the modeling of a SPL architecture using three models: (i) *subsystem model* – presents the overall system structure; (ii) *process model* – details the dynamic behavior of the system; and (iii) *module model* – specifies each reusable component of the architecture. Application engineering proceeds by first analyzing user's requirements and selecting appropriate and valid domain features from the feature model, identifying the corresponding reference model, and completing the application development by reusing software components in a bottom-up fashion.

FORM states some engineering principles for designing reusable architectures and components, which are: (i) separation of concerns and information hiding; (ii) localization of function, data, and control; (iii) parametrization of artifacts using the features; (iv) layering; (v) separation of components from the component connection mechanism; and (vi) synthesis of design components based on feature selection. These engineering principles support the development of architectures that are adaptable and reconfigurable for different applications.

The FORM authors have later extended (Kang 2002) the FORM method to incorporate a marketing and product plan (MPP) to help propel asset development. MPPs identify the information to gather during the marketing and business analysis. It includes a market analysis, a marketing strategy, product features, and product feature delivery methods.

2.1.2
Framework for SPLE by Pohl et al.

Pohl et al. propose in (Pohl 2005) a framework for SPLE that incorporates the central concepts of traditional SPLE, namely the use of platforms – a collection of reusable artifacts – and the ability to provide mass customization. The framework encompasses the two process of the SPLE paradigm: domain engineering and application engineering. Their proposal has its roots in the ITEA projects ESAPS, CAFÉ, and FAMILIES and is based on the differentiation between the domain and application engineering processes. For each one of the sub-processes that compose the framework, the authors provide their inputs and outputs, activities to be performed and emphasize the differences from single-system engineering.

The domain engineering process is composed of five sub-processes: (i) Product Management – deals with the SPL economic aspects and defines the SPL scope; (ii) Domain Requirements Engineering – provides the activities to elicit and document common and variable requirements; (iii) Domain Design – provides the activities to define the reference architecture; (iv) Domain Realization – deals with the detailed design and software components implementation; and (v) Domain Testing – responsible for validating and verifying the reusable components and developing reusable test artifacts. The application engineering encompasses similar sub-processes (Application Requirements Engineering, Application Design, Application Realization and Application Testing). Each of the sub-processes uses domain artifacts and produces application artifacts.

Variability management is an important activity in SPLs. For managing variability, an adequate documentation of variability information is essential. Although most of SPL approaches use feature models to document variability, Pohl et al. propose the Orthogonal Variability Model (OVM), a model that defines the variability of a SPL, relating the variability defined to other software development models such as feature models, use case models, design models, component models, and test models. They claim that with this approach there is a significant reduction of model size and complexity, because only the variable aspects of a product line are documented in the OVM models. Additionally, the OVM approach can be used to document the variability of arbitrary development artifacts, ranging from textual requirements to design models and even test cases. The core concepts of the OVM language are variation point ("what does vary") and variant ("how does it vary"). Each variation point has to offer at least one variant (offers-association). Additionally, the constrains-associations between these elements describe dependencies between variable elements.

2.1.3
Product Line UML-based Software Engineering (PLUS)

Product Line UML-based Software Engineering (PLUS) (Gomaa 2004) is a UML-based software design method for product lines, which provides a set of concepts and techniques to extend UML-based design methods and processes for single systems in a new dimension to address SPLs. It explores how each of the Unified Modeling Language (UML) modeling views – use case, static, state machine, and interaction modeling – can be extended to address software product families.

The method consists of two major processes or life cycles: Software product line engineering and Software application engineering. Each one of them encompasses five phases: Requirements Modeling, Analysis Modeling, Design Modeling, Implementation and Testing. In the SPLE process, a product line use case model, product line analysis model, software product line architecture, and reusable components are developed. All artifacts produced during each phase are stored in the SPL repository for being used during the software application engineering in order to derive individual applications.

Besides the method structure, PLUS provides an UML-based notation to address variability in the requirements, analysis, and design models. It basically uses stereotypes to distinguish among orthogonal characteristics of elements of the models, such as ≪*kernel*≫, ≪*optional*≫ and ≪*alternative*≫ for representing the reuse category; and ≪*entity*≫ and ≪*control*≫ for representing the role of the class in the application. The method also proposes notations, such as the use of packages or tables to represent feature/elements of the models dependencies, and architectural patterns for SPLs.

2.1.4
Komponentenbasierte Anwendungsentwicklung (KobrA)

The Komponentenbasierte Anwendungsentwicklung (KobrA) (Atkinson 2000) method is a component based product line engineering approach with UML, which has been developed at the Fraunhofer Institute for Experimental Software Engineering (IESE). KobrA is an abbreviation of *Komponentbasierte Anwendungsentwicklung* and means a component-based application development method. The authors from KobrA approach aimed at developing a simple, systematic, prescriptive, flexible and scalable method.

KobrA's software life cycle consists of two basic SPLE activities: Framework Engineering and Application Engineering. The purpose of the Framework

Engineering activity is to create, and later maintain, a generic framework that embodies all product variant in a family, including information about their common and disjoint features. This activity applies the Komponent, i.e. KobrA component, modeling and implementation activities, accompanied by additional subactivities for handling variabilities and decision models, to support a family of similar applications. A framework contains a generic Komponent tree that captures common and variable characteristics of a SPL.

The subactivities that are part of the Framework Engineering are the following: (i) Context Realization – produces a set of realization models for the environment of the system; (ii) Component Specification – produces a specification of a component in relationship to a given realization; (iii) Component Realization – produces a realization of a component in relationship to a given specification; (iv) Component Reuse – matches the needs of a given realization to the capabilities of a preexisting component via a mutually agreed specification; and (v) Quality Assurance – ensures the quality of the KobrA models using inspections, testing and quality modeling.

The Application Engineering activity uses the framework developed during Framework Engineering to build particular applications. The variabilities in the framework are removed, and the decisions in the decision model are resolved.

2.1.5
Comparative Analysis

The SPL approaches presented in the previous Sections were compared according to different aspects related to the SPLE. The goals of this comparison were to obtain a clearer overview of the approaches and find out how existing approaches differ for the development of SPLs; however it was not a goal to rate these approaches in order to choose the best one. The comparison was made using an evaluation framework proposed in (Matinlassi 2004), whose purpose is to compare SPL design methods. The framework considers the methods from the points of view of method context, user, structure and validation. The categories and elements that compose the evaluation framework are depicted in Table 2.1.

Tables 2.2 and 2.3 present several approaches for developing SPLs and MAS-PLs described according to the evaluation framework. The approaches are: FORM (Kang 1998), KobrA (Atkinson 2000), Pohl et al.'s Framework (Pohl 2005), PLUS (Gomaa 2004), MaCMAS Extension (Pena 2007) and GAIA-PL (Dehlinger 2007). The first four approaches are SPL methodologies/methods/processes, which were detailed in the previous Sections. The last

Table 2.1: Evaluation Framework – Source: (Matinlassi 2004).

Category	**Element**	**Question**
Context	Specific Goal	What is the *specific* goal of the method?
	Product Line Aspect(s)	What aspects of the product line does the method cover?
	Application Domain(s)	What is/are the application domain(s) the method is focused on?
	Method Inputs	What is the starting point for the method?
	Method Outputs	What are the results of the method?
User	Target Group	Who are the stakeholders addressed by the method?
	Motivation	What are the user's benefits when using the method?
	Needed Skills	What skills does the user need to accomplish the tasks required by the method?
	Guidance	How does the method guide the user while applying the method?
Contents	Method Structure	What are the design steps that are used to accomplish the method's specific goal?
	Architectural Viewpoints	What are the architectural viewpoints the method applies?
	Language	Does the method define a language or notation to represent the models, diagrams and other artifacts it produces?
	Variability	How does the method support variability expression?
	Tool Support	What are the tools supporting the method?
Validation	Method Maturity	Has the method been validated in practical industrial case studies?
	Architecture Quality	How does the method validate the quality of the output it produces?

two are MAS-PL approaches that are detailed in Section 6.1. This investigation did not have the purpose of selecting the best approach for SPL development but of helping to identify useful activities and notations for developing MAS-PLs.

Based upon the results of this study, the following conclusions are offered regarding the development of SPLs:

– Feature model is the typical notation to model variability in SPLs. Proposed in FODA, it is used in FORM, PLUS, MaCMAS Extension, FeatuRSEB (Griss 1998) and in several research works (Lee 2006, Satyananda 2007). Some approaches use alternative notations:

  – Pohl et al. employ OVMs for variability management, providing a general variability model comprising the domain's variation points and variants. With these models, the authors claim that a reduced complexity can be achieved, because only the variable aspects of a SPL are modeled. However, their approach proposes mapping the OVM into other models, such as use case and class diagrams, and this approach can not be scalable with complex diagrams. In addition, managing all the features of a SPL including the mandatory ones can help in evolution scenarios of SPLs, such as when a mandatory feature become optional. Moreover, feature models are widely known in the SPL community and have tool support (Antkiewicz 2004);

Table 2.2: Approaches Comparison (1).

Category	**Element**	**FORM**	**KobrA**	**Pohl et al.'s Framework**
Context	Specific Goal	How to apply domain analysis results (commonality and variability) to the engineering of reusable and adaptable domain components with specific guidelines.	Support a model-driven UML-based representation of components, and a product line approach to their development and evolution. Be as concrete and prescriptive as possible.	Define the key sub-processes of the domain engineering and application engineering processes as well as the artefacts produced and used in these processes.
	Product Line Aspect(s)	Extends FODA to the design and implementation phases and prescribes how the feature model is used to develop domain architectures and components for reuse.	Defines a full SPLE process with activities and artifacts, including implementation, releasing, inspection and testing.	Defines a full SPLE process with activities and artifacts, dividing the process into two key-processes (Domain engineering and Application engineering) and their respective sub-processes.
	Application Domain(s)	Telecommunication domain and information domain.	Information systems domain.	Home Automation case study.
	Method Inputs	The primary input is the information on systems that share a common set of capabilities and data.	Framework engineering: idea of a new framework with two or more applications; Application engineering: elicitation of user requirements within the scope of the framework.	Domain engineering: company goals defined by top management; Application engineering: domain requirements and product roadmap with the major features of the corresponding application.
	Method Outputs	Domain engineering: feature model, reference architecture and reusable components; Application engineering: application software, application architecture and reusable components (selected from feature model/reference architecture/reusable components respectively).	Framework engineering: generic Komponents and decision model; Application engineering: framework instantiated for one particular member of the SPL.	Domain engineering: Reusable software components; Application engineering: running application together with the detailed design artefacts.
User	Target Group	Academic audience.	Software engineers and designers currently working in the industry.	Students, professionals, lecturers, and researchers interested in SPLE.
	Motivation	Customers and engineers often speak of product characteristics in terms of "features the product has and/or deliver".	Simple, systematic, scalable and practical method.	OVM is smaller and less complex than extended UML diagrams or feature models.
	Needed Skills	FORM Notation	-	OVM Notation
	Guidance	Case studies, special guideline	Case studies, extensive manuals	Book with case studies
Contents	Method Structure	First, define the context of the system. After the main two phases are Domain engineering and Application engineering.	Framework engineering and Application engineering.	First, define the product roadmap. After the main two phases are Domain engineering and Application engineering.
	Architectural Viewpoints	Subsystem, process and module.	Specification, realization, containment and context realization.	Logical, development, process and code views.
	Language	-	Adapted UML and manual transformation to code	Own language to design the variability model and UML
	Variability	Define support for variability in requirements elicitation.	Concentrate on capturing variability with graphical language in architectural design.	Captures variability into a variability model.
	Tool Support	ASADAL.	Commercial UML tool; word processing tool and configuration management.	-
Validation	Method Maturity	Validated in practical industrial case studies.	Validated in practical industrial case studies.	Framework defined based on industrial experiences and the results of the European SPLE research projects ESAPS, CAFÉ, and FAMILIES.
	Architecture Quality	Non-architectural evaluation methods, such as model checking, inspections and testing.	Scenario based architecture evaluation (SAAM) for ensuring maintainability.	Testing in Domain and Application engineering.

Table 2.3: Approaches Comparison (2).

Category	Element	PLUS	MaCMAS Extension	GAIA-PL
Context	Specific Goal	Provide a set of concepts and techniques to extend UML-based design methods and processes for single systems to handle software product lines.	Provide explicit support for MAS-PLs and manage the modeling of the evolution of the system in a systematic way.	Capture requirements in such a way that they can be easily and safely reused during system evolution for mission-critical, agent-based distributed software systems.
	Product Line Aspect(s)	Extends the USDP adding: Requirements Modeling, Analysis Modeling, Design Modeling and Application Engineering.	Domain requirements engineering, domain design and domain realization.	Requirements engineering phase.
	Application Domain(s)	Microwave Oven, Electronic Commerce and Factory Automation case studies.	NASA (Prospecting Asteroid Mission) and human organization case studies.	Constellation of context-aware microsatellites case study.
	Method Inputs	SPL engineering: product line requirements; Application engineering: application requirements and SPL repository.	Domain Engineering: requirements statement, goal hierarchy; Application engineering: requirements statement, goal hierarchy, feature model, core architecture.	Requirements.
	Method Outputs	SPL Engineering: SPL repository; Application engineering: executable application.	Domain Engineering: core architecture; Application Engineering: products (MAS seen as a collection od multiple products).	Role Schema and Role Variation Points.
User	Target Group	Professional and academic audience.	-	-
	Motivation	Extends object-oriented methods to model product families and provides explicit modeling of the similarities and variations in a product line.	First approach that deals with architectural changes of evolving systems based on MAS-PL.	Dynamically changing configurations of agents can be captured and reused for future similar systems.
	Needed Skills	-	MacMAS Methodology	GAIA Methodology
	Guidance	Book with case studies	Articles and Case Studies	Articles and Case Studies
Contents	Method Structure	Software product line engineering and Software application engineering.	Domain engineering and Application engineering.	-
	Architectural Viewpoints	Use Case Model, Static Model, Collaboration Model, Statechart Model and Feature Model views.	Static acquaintance and Behavior of acquaintance organization views.	-
	Language	UML (use of stereotypes).	MacMAS	Natural Language
	Variability	Feature Model (with UML).	Feature Model	Role Schema and Role Variation Points
	Tool Support	UML Design Tools and Product Line UML Based Software Engineering Environment (PLUSEE).	Plugin of the ArgoUML CASE tool	Text Editors
Validation	Method Maturity	-	NASA case study	-
	Architecture Quality	Testing in Software product line engineering and Application engineering.	Formal analysis would complement the simulation and testing.	-

  - MaCMAS extension uses a goal-oriented approach to perform the domain analysis, instead of a feature-oriented approach. Goals are represented in a feature model notation in order to provide variability information;
  - Even though PLUS also adopts feature models, it uses a notation based on UML;
  - KobrA, PuLSE (Bayer 1999) and FAST (Weiss 1999) adopt decision models to describe the choices that distinguish distinct members of the family. However, feature models provide an important higher abstraction level.

- Almost all approaches do not address explicitly the modeling of the SPL requirements;

  - PLUS approach defines a customization of the use case model to specify and document the SPL requirements;
  - GAIA-PL proposes a requirements specification template to capture and reuse dynamically changing configurations of agents for future similar systems;

- In the domain design, most of the investigated SPL approaches only provide support to document and detail the SPL architectures in a very high-level manner;

  - PLUS is an object-oriented approach that adopts traditional UML models marked with additional stereotypes to classify the system classes;
  - KobrA is an component-oriented approach that proposes several models (structural, behavioral, functional, non-functional, quality and decision models) to specify its Komponents.

In the context of MAS, the investigated approaches do not provide explicit support to specify and model the SPL architecture and its respective components. However, most of the SPL approaches provide useful notations to model the agent features. Nevertheless, none of them completely covers their specification. Agent technology provides particular characteristics that need to be considered in order to take advantage of this paradigm.

2.2 Multi-agent Systems

Over the past decades, software agents have become a powerful abstraction to support the development of complex and distributed systems (Jennings 2001). They are a natural metaphor to understand systems that present some particular characteristics such as high interactivity and multiple loci of control. These systems can be decomposed in several autonomous and pro-active agents comprising a Multi-agent System (MAS). A software agent is an abstraction that enjoys mainly the following properties (Wooldridge 1995):

Autonomy: agents operate without the direct intervention of humans or others, and have some kind of control over their actions and internal state;

Social ability: agents interact with other agents (and possibly humans) via some kind of agent-communication language;

Reactivity: agents perceive their environment and respond in a timely fashion to changes that occur in it;

Pro-activeness agents do not simply act in response to their environment, they are able to exhibit goal-directed behavior by taking the initiative.

Agent-oriented Software Engineering (AOSE) (Wooldridge 2000b) has emerged as a new software engineering paradigm to help on the development of MASs. In this context, several research works were proposed, such as methodologies and processes (Wooldridge 2000a, Cossentino 2005, Bresciani 2004) and modeling languages (Silva 2007). Some of these approaches are described in the next Sections. Further comparison among MAS can be seen in (Sellers 2005).

A problem of the current MAS methodologies is that most of them do not take into account the adoption of extensive reuse practices that can bring an increased productivity and quality to the software development (Girardi 2002). Software reuse techniques, such as component-based development, object-oriented application frameworks and libraries, software architectures, patterns, have been widely used in the software engineering context to provide reduced time-to-marked, quality improvement and lower development costs. Moreover, none of these MAS methodologies address the development of SPLs and, consequently, do not provide notations to express agent variabilities.

2.2.1 Gaia

Using the analogy of human-based organizations, the Gaia (Wooldridge 2000a, Zambonelli 2003) methodology provides an approach that both a developer and a non-technical domain expert can understand, facilitating their interaction. It exploits organizational abstractions and, taking into account the specific issues associated with the exploitation of such abstractions, defines a coherent design process for MAS development. Gaia prescribes following an ordered sequence of steps, the ordered identification of a set of organizational models, and an indication of the interrelationships, guiding developers towards the development of a MAS.

The organizational abstractions exploited by Gaia are: (i) Environment – considering that a MAS is always situated in some environment, the authors believe this is a primary abstraction during the analysis and design phases; (ii) Roles and Interactions – the role of an agent defines what it is expected to do in the organization, both in concert with other agents and with respect to the organization; (iii) Organizational Rules – explicit identification of constraints is very important for the correct understanding of the characteristics that the organization-to-be must express and for the subsequent definition of the system structure by the designer; and (iv) Organizational Structures – the structure of a MAS is more appropriately derived from the explicit choice of an appropriate organizational structure.

The Gaia process starts with the analysis phase, whose aim is to collect and organize the specification, which is the basis for the design of the computational organization (which implies defining an environmental model, preliminary roles and interaction models, and a set of organizational rules). Then, the process continues with the architectural phase, aimed at defining the system organizational structure in terms of its topology and control regime (possibly exploiting design patterns), which, in turn, helps to identify complete roles and interaction models. Eventually, the detailed design phase can begin. Its aim is to produce a detailed, but technology-neutral, specification of a MAS (in terms of an agent model and a services model) that can be easily implemented using an appropriate agent-programming framework. After the successful completion of the Gaia design process, developers are provided with a well-defined set of agent classes to implement and instantiate, according to the defined agent and services model. Gaia considers the output of the design phase as a specification that can be picked up by using a traditional method or that could be implemented using an appropriate agent-programming framework.

Gaia does not directly deal with implementation issues. Its authors state that it may be the case that specific technology platforms may introduce constraints over design decisions and may require being taken into account during analysis and design. In addition, Gaia does not deal with the activities of requirements capture and modeling and, specifically, of early requirements engineering.

2.2.2 Process for Agent Societies Specification and Implementation (PASSI)

Process for Agent Societies Specification and Implementation (PASSI) (Cossentino 2005) is a step-by-step requirement-to-code methodology for designing and developing multi-agent societies, integrating design models and concepts from both object-oriented software engineering and artificial intelligence approaches using (more properly extending) the UML notation.

The PASSI process encompasses five process components: (i) System Requirements – a model of the system requirements in terms of agency and purpose; (ii) Agent Society – a model of the social interactions and dependencies among the agents involved in the solution; (iii) Agent Implementation – a classical model of the solution architecture in terms of classes and methods; the most important difference with the common object-oriented approach is that we have two different levels of abstraction, the social (multi-agent) level and the single-agent level; (iv) Code – a model of the solution at the code level; and (v) Deployment – a model of the distribution of the parts of the system across hardware processing units and their migration between processing units. The models and phases of the PASSI methodology are illustrated in Figure 2.1.

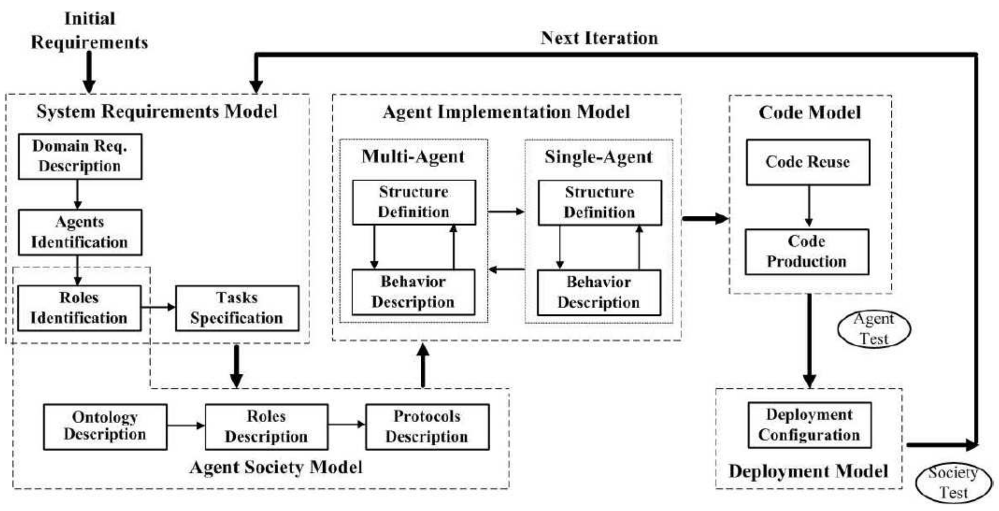

Figure 2.1: PASSI models and phases – Source: (Cossentino 2005).

PASSI differs with some agent-orient methodologies in describing requirements in terms of use case diagrams, instead of making use of goals in requirements engineering. In addition, PASSI promotes pattern reuse and code production during the code reuse phase. Code production is strongly supported by the automatic generation of a large amount of code through the PASSI ToolKit (PTK) used to design the system and a library of reusable patterns of code and pieces of design managed by the AgentFactory application.

PASSI was conceived following one specific guideline: the use of standards whenever possible. This justifies the use of UML as modeling language, the use of the Foundation for Intelligent Physical Agents (FIPA) architecture for the implementation of our agents, and the use of Extensible Markup Language (XML) in order to represent the knowledge exchanged by the agents in their messages. An agile version of PASSI, the Agile PASSI, is proposed in (Chella 2006) in order to have a design process that specifically addresses the needs of developing robotic systems.

2.2.3 Tropos

The Tropos methodology (Bresciani 2004) is intended to support all analysis and design activities in the software development process, from application domain analysis down to the system implementation. In particular, Tropos rests on the idea of building a model of the system-to-be and its environment, which is incrementally refined and extended, providing a common interface to various software development activities, as well as a basis for documentation and evolution of the software.

Tropos is based on two key ideas. First, the notion of agent and all related mentalistic notions (for instance goals and plans) are used in all phases of software development, from early analysis down to the actual implementation. Second, Tropos covers also the very early phases of requirements analysis, thus allowing for a deeper understanding of the environment where the software must operate, and of the kind of interactions that should occur between software and human agents.

Tropos adopts the concepts offered by $i*$ (Yu 1996), a modeling framework proposing concepts such as actor (actors can be agents, positions, or roles), as well as social dependencies among actors, including goal, softgoal, task, and resource dependencies. As a consequence, models in Tropos are acquired as instances of a conceptual metamodel resting on the following concepts/relationships: (i) Actor – models an entity that has strategic goals and intentionality within the system or the organizational setting; (ii) Goal – represents actors' strategic interests; (iii) Plan – represents, at an abstract level, a way of doing something; (iv) Resource – represents a physical or an informational entity; (v) Dependency (between two actors) – indicates that one actor depends, for some reason, on the other in order to attain some goal, execute some plan, or deliver a resource; (vi) Capability – represents the ability of an actor of defining, choosing and executing a plan for the fulfillment of a goal, given certain world conditions and in presence of a specific event; and (vii) Belief – represents actor knowledge of the world.

The proposed methodology spans four phases that can be used sequentially or iteratively: (i) Early requirements – concerned with the understanding of a problem by studying an organizational setting; (ii) Late requirements – where the system-to-be is described within its operational environment, along with relevant functions and qualities; (iii) Architectural design – the system's global architecture is defined in terms of subsystems, interconnected through data, control, and other dependencies; and (iv) Detailed design – where the behavior of each architectural component is further refined.

The Tropos methodology in its current form is not suitable for sophisticated software agents requiring advanced reasoning mechanisms for plans, goals, and negotiations.

2.2.4 MAS-ML

MAS-ML (Silva 2004b, Silva 2004a, Silva 2007, Silva 2008) is an agent-oriented modeling language, which extends the UML meta-model describing new meta-classes and stereotypes, and extending some diagrams and proposing new ones. The MAS-ML meta-models showing the new metaclasses are depicted in Figures 2.2 and 2.3.

MAS-ML is based on the Taming Agents and Objects (TAO) (Silva 2003b) conceptual framework (meta-model). TAO defines the static and dynamic aspects of MASs. The static aspect of TAO captures the system's elements and their properties and relationships. The elements defined in TAO are agents, objects, organizations, environments, agent roles and object roles. The relationships that link these elements are inhabit, ownership, play, control, dependency, associations, aggregation and specialization. The dynamic aspects of TAO are directly related to the relationships between the elements of MASs and they define the domain-independent behaviors associated with the interaction between MAS elements.

Using the MAS-ML meta-model and diagrams, it is possible to represent the elements associated with a MAS and to describe the static relationships and interactions between these elements. The structural diagrams in MAS-ML are the extended UML class diagram and two new diagrams: organization

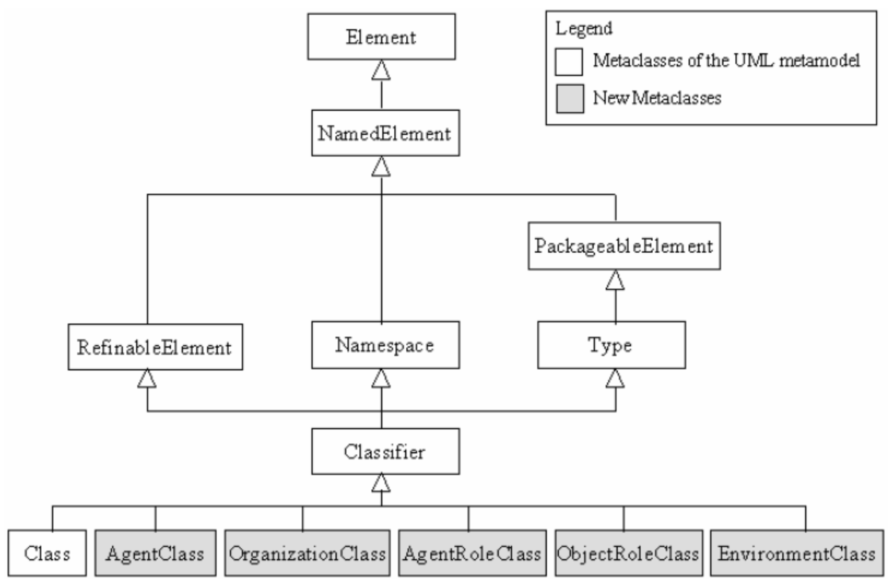

Figure 2.2: MAS-ML metamodel: new metaclasses (part I) – Source: (Silva 2004a).

and role. MAS-ML extends the UML class diagram to represent the structural relationships between agents, agents and classes, organizations, organizations and classes, environments, and environments and classes. The organization diagram models the system organizations and the relationships between them and other system elements. Finally, the role diagram is responsible for modeling the relationships between the roles defined in the organizations. Additionally, MAS-ML extends the sequence diagram to represent the interaction between agents, organizations and environments.

2.3 Final Remarks

This Chapter presented an overview of Software Product Lines and Multi-agent Systems. The former allows the development of families of applications that share common and variable features through a systematic method. The latter are systems whose architecture relies on softwares agents, which are a powerful abstraction to develop complex and distributed systems. Several works have been proposed in both contexts aiming at providing software engineering techniques to help on their development, such as process, methodologies and modeling languages. On the one hand, most of the SPL approaches provide useful notations to model the agent features; however, none of them completely covers their specification. On the other hand, MAS methodologies address the development of single systems, and do not exploit the benefits provided by software reuse, e.g. lower development cost and reduced time-to-market.

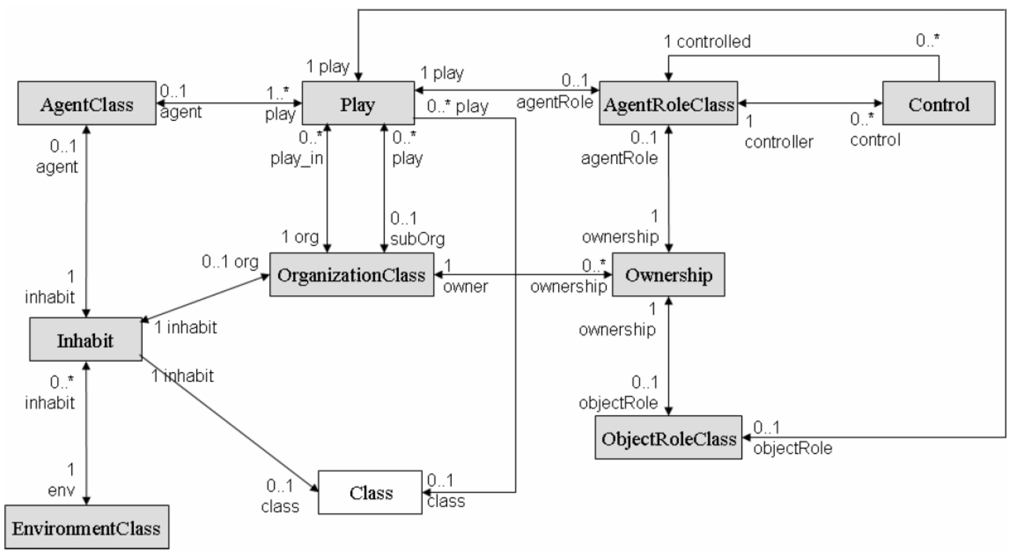

Figure 2.3: MAS-ML metamodel: new metaclasses (part II) – Source: (Silva 2004a).

Even though SPL and MAS approaches do not address the development of MAS-PLs, they provide useful models and notations for that. Therefore, they could contribute to the elaboration of our domain engineering process.