Can Agent Oriented Software Engineering Be Used to Build MASs Product Lines?

Joaquín Peña

Dpto. de Lenguajes y Sistemas Informáticos
Avda. de la Reina Mercedes, s/n. Sevilla 41.012, Spain
joaquinp@us.es
www.tdg-seville.info

Abstract. On the one hand, the Software Product Lines (SPL) field is devoted to build a core architecture for a family of products from which concrete products can be derived rapidly by means of reuse. On the other hand, Agent-Oriented Software Engineering (AOSE) is a software engineering paradigms dedicated to build software applications composed of organizations of agents. Bringing AOSE to the industrial world may prettily benefit from SPL advantages. Using SPL philosophy, a company will be able to define a core MAS from which concrete products will be derived for each customer. This can reduce time-to-market, costs, etcetera. In this paper, we expose the similarities between AOSE and SPL concluding the viability of future research in Multi-Agent Systems Product Lines (MAS-PL).

1 Introduction

Agent-Oriented Software Engineering (AOSE) is a new software engineering paradigm that promises to enable the development of more complex systems than with current Object-Oriented approaches using agents and organizations of agents as the main abstractions [14]. In this field, agents are used as the main implementation artifact and organizations as the main artifact in modelling the system.

A software agent is a piece of software which exhibits the characteristics firstly described by Wooldridge in Ref. [26], namely: autonomy, reactivity, pro-activity and social ability. Autonomy means that an agent operates without the direct intervention of other agents or humans and has control over its actions and its internal state. Reactivity means that an agent perceives its environment and responds in a timely fashion to changes that occur in it. Pro-activity means that an agent does not simply react to changes in the environment, but exhibits goal-directed behaviour and takes the initiative when it considers it appropriate. Social ability means that an agent interacts with other agents (if it is needed) to complete its tasks and helps or contends with others to achieve its goals. Many researchers are in agreement with the vision of software agents as a characterization; and they, depending on their community, attach new attributes to software agents as mobility in distributed systems or adaptability in machine learning.

Since agents are limited to a specific environment and have limited abilities, complex problems are usually solved by a group of agents [4,5,24], that is to say, by a Multi-agent System (MAS hereafter).

A Software Product Line (SPL) is a set of software-intensive systems sharing a common, managed set of features that satisfy 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 the definition used by the Software Engineering Institute (SEI) [6]. This engineering paradigm is focused in bringing reuse until its final outcomes. It is devoted to create software products, which are said to belong to a family of products, which present similar features, from a set of reusable software artifacts and a core architecture.

Both fields present many similarities. In this paper, we expose these similarities and we argue for the viability of integrating both approaches to enable the development of MAS Product Lines.

This paper is structured as follows: Section 2 shows the case study we use; Section 3 shows the features of AOSE that are important for our purpose; Section 4 shows the features of SPLs that are important in the AOSE field; Section 5 shows the main similarities between both and the deficiencies of AOSE to enable MAS-PLs; and finally, Section 6 shows our main conclusions.

2 A Case Study

The case study we have chosen has been proposed in the Modelling TC of FIPA with the purpose of evaluating how AOSE methodologies model the interaction aspect.

This case study is used along this document. Following, we present the original version of the problem[1]. The case study represents the UN Security Council's Procedure to Issue Resolutions. We use some parts of this example to show the similarities between both fields.

In addition, notice that, although it does not represent a product line, we will take some of the requirements as optional to obtain a product line where each product presents a subset of the requirements presented following.

Assumptions

(1) The following procedure is defined for a case study of agent-oriented modelling. It is inspired from the procedure of UN Security Council to pass a resolution. However, it does NOT necessary represents the reality.

(2) There are also some unspecified issues and ambiguity in the specification that models in different notations and languages may resolve by themselves in the process of modelling.

Description

The UN Security Council (UN-SC) consists of a number of members, where some of them are permanent members. Members become the Chair of the Security Council in turn monthly.

To pass a UN-SC resolution, the following procedure would be followed:

1. At least one member of UN-SC submits a proposal to the current *Chair*;
2. The *Chair* distributes the proposal to all members of UN-SC and set a date for a vote on the proposal.

[1] http://www.auml.org/auml/documents/UN-Case-Study-030322.doc

3. At a given date that the Chair set, a vote from the members is made;
4. Each member of the security council can vote either FOR or AGAINST or SUSTAIN;
5. The proposal becomes a UN-SC resolution, if the majority of the members voted FOR, and no permanent member voted AGAINST.
6. The members vote one at a time.
7. The Chair calls the order to vote, and it is always the last one to vote.
8. The vote is open (in other words, when one votes, all the other members know the vote)
9. The proposing member(s) can withdraw the proposal before the vote starts and in that case no vote on the proposal will take place.
10. All representatives vote on the same day, one after another, so the chair cannot change within the vote call; but it is possible for the chair to change between a proposal is submitted until it goes into vote, in this case the earlier chair has to forward the proposal to the new one.
11. A vote is always finished in one day and no chair change happens on that day. The chair sets the date of the vote.

3 Agent Oriented Software Engineering (AOSE)

In addition to the commonly used division in requirements, analysis, design and implementation, the software process of AOSE methodologies can be also divided into two phases, each of them used to describe the organization of the system from a different point of view [27]. In this section, we first show each kind of organization, to later discuss on the software process and models used in AOSE.

An organization represents a group of agents formed in the system in order to obtain benefits from one another in a collaborative or competitive manner [15,20,26]. Therefore, a multi-agent organization emerges when there exists some kind of interaction between its participants, either through direct communication or through the environment. As recognized in the field of economics [19], and other authors have recognized in the agent field, e.g. [3,11,27], an organization can be observed from two different points of views:

The interaction point of view: It describes the organization from the set of interactions between the roles played by agents in the system. The interaction view corresponds to the functional point of view in the field of economics.

The structural point of view: It describes the agents of the system and their distribution over sub-organizations, groups, and teams. In this view, agents are also presented in hierarchical structures showing the social architecture of the system.

In agency, the former is called *Acquaintance Organization*, and the latter is called *Structural Organization* [27]. Both views are intimately related, but they show the organization from radically different points of view.

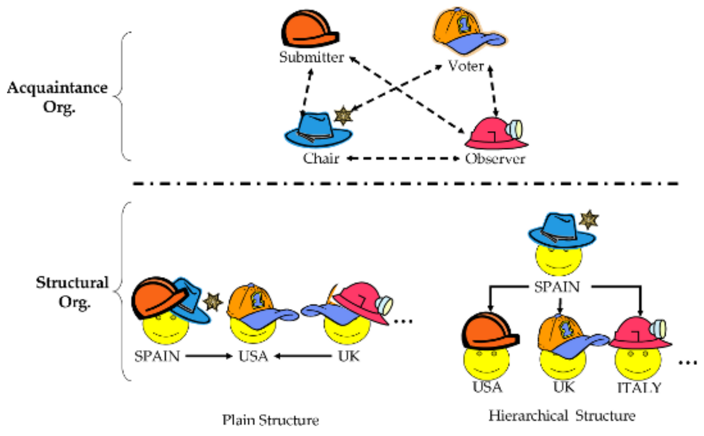

Fig. 1. Acquaintance *vs.* structural organization

Since any structural organization must include interactions between its agents in order to function, it is safe to say that the acquaintance organization is always contained in the structural organization. Therefore, if we determine first the acquaintance organization, and we define the constraints required for the structural organization, a natural map is formed between the acquaintance organization and the correspondent structural organization. This is the process of assigning roles to agents [27]. Thus, we can conclude that any acquaintance organization can be modelled orthogonally to its structural organization [18].

In Figure 1, we present a conceptual model of the case study presented in Section 2. Here we are representing only the procedure required to send, vote, withdraw and accept or reject resolutions not representing the possibility of changing the Chair, that is to say, an acquaintance sub-organization of our problem. Notice that for representing an acquaintance sub-organization we should have represented also the dynamic part of the system, that is to say, the behaviour of the roles in the sub-organization. However, we have not done that in order to simplify.

In the acquaintance organization, it implies a set of roles and interactions between them. In the structural organization, these roles can be mapped onto a certain group of agents to form several organizational structures. For example, as shown in Figure 1, we can map the acquaintance organization into a plain structure, a hierarchical structure, and so on. Thus, this exemplifies the fact that the structural organization of the UN Security Council is different and independent from its structural organization.

Most AOSE methodologies recognize this separation what derives in that some methodologies have proposed a software process where one phase appear for developing each kind of organization: *acquaintance analysis* and *structural analysis*[2]. This fact is ratified by the importance that the role concept, crucial for modelling the acquaintance organization, has reached in this field where most important methodologies

[2] Notice that these phases present a different name in each methodology, for example, in GAIA they are called analysis and architectural design respectively.

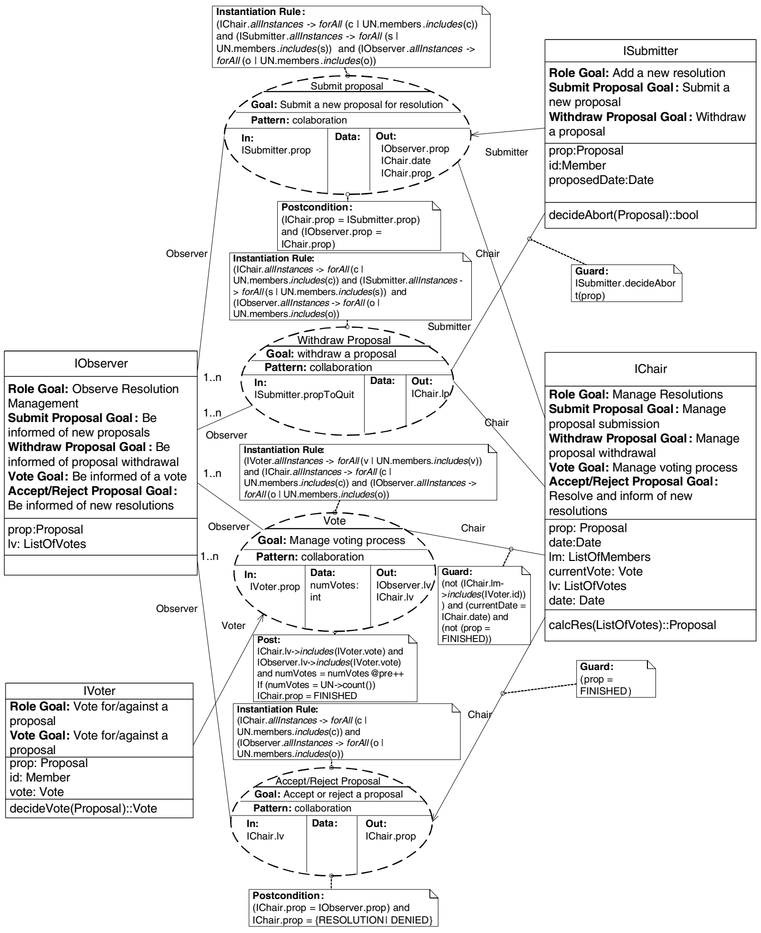
Instantiation Rule: (IChair.allInstances -> forAll (c | UN.members.includes(c)) and (ISubmitter.allInstances -> forAll (s | UN.members.includes(s)) and (IObserver.allInstances -> forAll (o | UN.members.includes(o))
Submit proposal
Goal: Submit a new proposal for resolution
Pattern: colaboration
In: ISubmitter.prop
Data:
Out: IObserver.prop IChair.date IChair.prop
Postcondition: (IChair.prop = ISubmitter.prop) and (IObserver.prop = IChair.prop)
ISubmitter
Role Goal: Add a new resolution
Submit Proposal Goal: Submit a new proposal
Withdraw Proposal Goal: Withdraw a proposal
prop:Proposal
id:Member
proposedDate:Date
decideAbort(Proposal)::bool
Submitter
Guard: ISubmitter.decideAbort(prop)
Observer
Chair
Instantiation Rule: (IChair.allInstances -> forAll (c | UN.members.includes(c)) and (ISubmitter.allInstances -> forAll (s | UN.members.includes(s)) and (IObserver.allInstances -> forAll (o | UN.members.includes(o))
Submitter
Withdraw Proposal
Goal: withdraw a proposal
Pattern: collaboration
In: ISubmitter.propToQuit
Data:
Out: IChair.lp
IObserver
Role Goal: Observe Resolution Management
Submit Proposal Goal: Be informed of new proposals
Withdraw Proposal Goal: Be informed of proposal withdrawal
Vote Goal: Be informed of a vote
Accept/Reject Proposal Goal: Be informed of new resolutions
prop:Proposal
lv: ListOfVotes
1..n
1..n
Observer
1..n
1..n
Chair
IChair
Role Goal: Manage Resolutions
Submit Proposal Goal: Manage proposal submission
Withdraw Proposal Goal: Manage proposal withdrawal
Vote Goal: Manage voting process
Accept/Reject Proposal Goal: Resolve and inform of new resolutions
prop: Proposal
date:Date
lm: ListOfMembers
currentVote: Vote
lv: ListOfVotes
date: Date
calcRes(ListOfVotes)::Proposal
Instantiation Rule: (IVoter.allInstances -> forAll (v | UN.members.includes(v)) and (IChair.allInstances -> forAll (c | UN.members.includes(c)) and (IObserver.allInstances -> forAll (o | UN.members.includes(o))
Observer
Vote
Chair
Goal: Manage voting process
Pattern: collaboration
In: IVoter.prop
Data: numVotes: int
Out: IObserver.lv IChair.lv
Guard: (not (IChair.lm->includes(IVoter.id))) and (currentDate = IChair.date) and (not (prop = FINISHED))
Observer
Voter
Post: IChair.lv->includes(IVoter.vote) and IObserver.lv->includes(IVoter.vote) and numVotes = numVotes @pre++ If (numVotes = UN->count()) IChair.prop = FINISHED
Guard: (prop = FINISHED)
IVoter
Role Goal: Vote for/against a proposal
Vote Goal: Vote for/against a proposal
prop: Proposal
id: Member
vote: Vote
decideVote(Proposal)::Vote
Instantiation Rule: (IChair.allInstances -> forAll (c | UN.members.includes(c)) and (IObserver.allInstances -> forAll (o | UN.members.includes(o))
Chair
Accept/Reject Proposal
Goal: Accept or reject a proposal
Pattern: collaboration
In: IChair.lv
Data:
Out: IChair.prop
Postcondition: (IChair.prop = IObserver.prop) and IChair.prop = {RESOLUTION| DENIED}

Fig. 2. Role Model for the Issue Resolution system goal

use it. Good examples are GAIA [27], INGENIAS [21], MASE [9], PASSI [2] and TROPOS [1].

For example, the MaCMAS methodology fragment [22], represents role models as it is shown in Figure 2. This model uses an extension of UML collaborations to represent the acquaintance organization. We can find two main types of elements in the model: (1) roles, represented as boxes; and (2) interactions, represented using ellipses. Roles show which are their general goals and their particular goals when participating in a certain interaction with other roles or with some part of the environment. Roles also represent the knowledge they manage, middle compartment, and the services they offer, bottom compartment. For example, the goal of the *Chair* role is ”Manage resolutions”, while its goal when participating in the *Withdraw proposal* interaction is to manage the process of withdrawing a proposal. In addition to

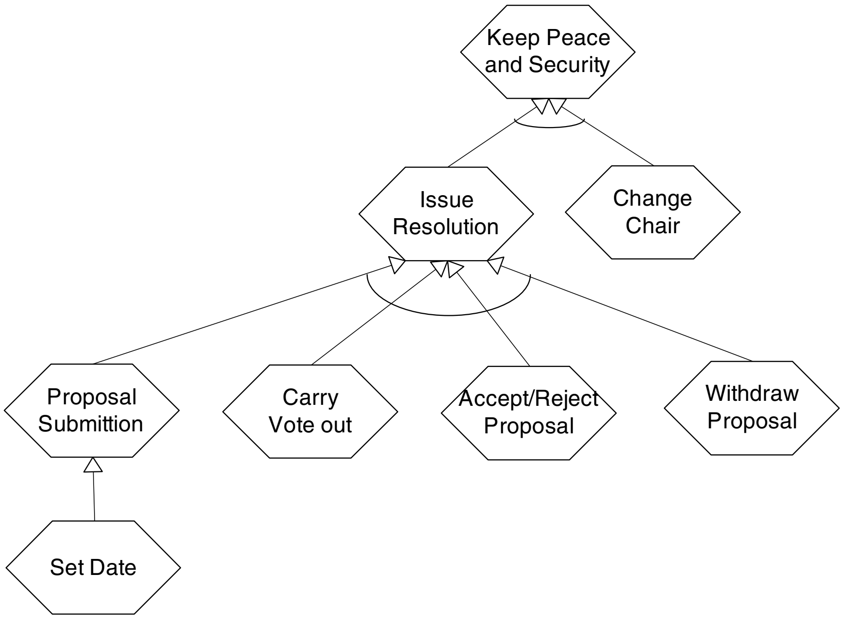

Fig. 3. Example of Goal Hierarchy using TROPOS

roles, interactions show us also some important information. They also must show the system-goal they achieve when executed, the kind of coordination is carried out when executed, the knowledge used as input to achieve the goal and the knowledge produced. For example, the goal of the interaction *Submit Proposal* is to "Submit a new proposal for a resolution". It is done by taking as input the knowledge of the *Submitter* about the proposal and producing as output the date of the proposal in the *Chair*, and the proposal in the *Chair* and in the *Observer*. Notice that as shown in Figure 1, these roles will be mapped onto agents in the phase devoted to build the structural organization.

In addition, many authors recommend the use of goal-oriented requirement approaches for this kind of systems. Goal-oriented requirements techniques analyse system goals producing a hierarchical diagram where each system goal is related with the system goals that decompose it [1,8,9,17,27]. Since agents are designed to fulfill goals, and organizations are designed to fulfill those goals that are sufficiently complex for requiring more than one agent to be satisfied, this information is also used to determine decomposition of the acquaintance organization into sub-organizations by some methodologies, for example, MASE, GAIA and MaCMAS.

In Figure 3, we show the hierarchical goal diagram of our case study using TROPOS. As can be observed, the general system goal of our case study is *Keep Peace and Security*. It shows also that to fulfill this goal the diagram we have to fulfill two finer grain system goals since they are related using an AND decomposition: the system goal *Issue Resolutions* and *Change Chair*. Notice that also OR relationships are allowed.

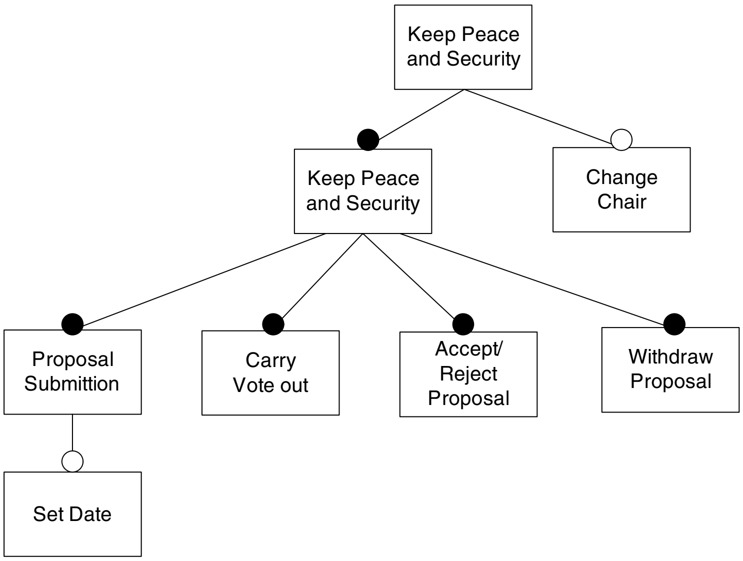

Fig. 4. Example of Feature Model

4 Software Product Lines (SPL)

The software process of SPLs approaches is usually divided in two main stages: *Domain Engineering* and *Application Engineering*. The former is in charge of providing the reusable core assets that are exploited during application engineering when assembling or customizing individual applications [12].

Domain engineering and application engineering phases are usually further divided into analysis, design, and implementation. From all these phases, we only detail those that can be applied in AOSE.

The analysis phase of domain engineering is applied to analyse the problem domain of a family of applications. It main purpose is identifying the common set of features that can be reused across all the applications that can be built under the domain at hand. Following, we show the most relevant stages for our purpose:

Domain scoping is devoted to describe the boundaries of the domain, which is crucial to determine the scope and needs of the product family. Another important stage is the *Commonality analysis*. It specifies the commonalities and points of variation in the domain that represents the first step towards a description of the family and the differences that each product may present.

Domain modelling is dedicated to produce documents where common and variable requirements are showed and the analysis models for these requirements. Thus, these documents specify which set of requirements are valid and may produce a valid application and which do not. One of the most accepted techniques to perform these documents are *feature models* [7]. A feature is a characteristic of the system that is observable by the end user [16].

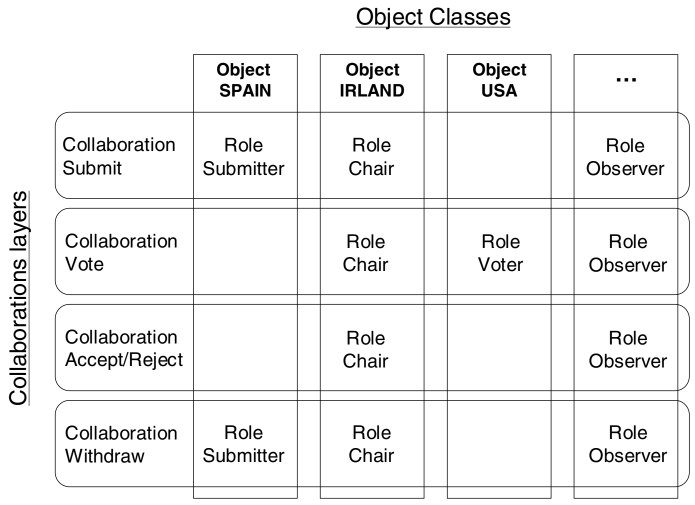

Fig. 5. Conceptual diagram of our case study using MIXIN layers

Feature models represent features hierarchically and its relationships. For example, in FODA, features in the model can be mandatory, optional, or alternative. In figure 4, we show an example of feature model representing our case study with some modifications. Filled circle indicate that the feature is mandatory, that is to say, if the parent feature is present in the product, it must be also present in the product. White circles indicates that feature is optional as it happens with the *Change Chair* feature. This means that we can have a product where the chair is changed and another product where there is no possibility of changing the chair. Although not showed in the example, we can also have filled arc indicating that for the final products we must have at least one of the features linked, and white arcs indicating that we can only have one of the features linked.

Finally, in the domain modelling stage, we can have also analysis models that show the interface(s) required in the system for including each feature. As we discuss below.

The design phase of domain engineering is devoted to model the core architecture. It consists in producing a core architecture by means of combinations of models that represent features that are common. Many approaches have appeared in the literature. From all of them, we are interested on those that are nearer to the AOSE field. In these approaches, role models are used to represent the interfaces and interactions that components of the system should provide to cover certain functionality (a feature in the features model). Later, role models are composed to produce the core architecture, the most representative are [13,25], but similar approaches has appeared in the OO field, for example [10,23].

Figure 5 shows a conceptual diagram of our case study extracted from [13]. There, authors show the mapping of domain models, collaborations, over the core architecture needed to build a certain product using OO programming. In this paper, authors use collaborations to represent features of the system. Collaborations are interaction-centered diagrams that show the interaction in a subsystem and the roles, sub-set of objects, involved in these interactions. Later they propose to map them onto certain objects for building a concrete product, as shown in the figure using our case study.

Finally, notice that we do not discuss on the implementation phase of domain engineering since it does not correlate with agent implementations. We neither detail application engineering due to AOSE methodologies are not prepared to produce a family of MASs.

5 SPL/AOSE Correlation

The main points of correlation between both fields are three: (i) the correlation in the phases; (ii) the correlation in models used at requirements; (iii) and the approach followed to design a concrete MAS and to design a certain product. The main difference is that AOSE does not manage in the requirement phase the existence of several products. Following, we discuss first on similarities, to finish showing the main deficiencies of AOSE to enable a MAS-PL approach.

The first similarity is probably the most significant. This can be found in the phases of both approaches. On the one hand, the *analysis phase of domain engineering* in SPL is dedicated to define an architectural independent document describing the features of the system and the *acquaintance analysis* in AOSE is dedicated to define a set of acquaintance sub-organizations independently from the structural organization. Thus, as shown in previous sections, in both cases role models/collaborations are used to represent these independent models. However, SPL approaches finalize this phases implementing a core architecture, while AOSE approaches do not implement these models.

The second similarity consists in that requirement documents used for this phase in both fields present the same structure. Given that system goals in MASs are functional requirement observable by the end user, system-goals represent the same concept than features. Feature models, used in SPL, and goal-oriented requirement documents, used in AOSE, are both hierarchical where each feature/system-goal is decomposed in lower levels of the hierarchy indicating in both cases when they are mandatory, optional or alternative. Notice only some AOSE methodologies allow having mandatory, alternative or optional goals, but this feature is present in AOSE.

Third, the *design phase of domain engineering* in SPL is dedicated to produce the core architecture for the family of products and *structural analysis* is also devoted to define the structure of the organization of the MAS. In both field, role models/collaborations are used for defining architecture/structure independent models, and in both fields this process consists in composing models. If we concentrate on the SPL approaches that represent features using role models/collaborations, and given that in AOSE this phase consists in composing roles and the also use role models, both fields follows a quite similar procedure and models.

However, there exist some important differences between both fields mainly motivated by that AOSE is concentrated on developing a unique MAS and not a family of them. While in SPL the features that are composed to build the core architecture are those that are common for all products and composition is done guided by the commonality analysis, in AOSE roles are composed when they pursue similar goals or they provide a similar functionality.

As can be observed, current AOSE approaches align with SPL in their phases and models, but lacking from the artillery necessary to develop a general architecture instead of a unique product, namely, commonality analysis and domain scoping.

6 Conclusions

In this paper, we have shown the similarities and differences between SPL and AOSE. Given that AOSE methodologies follows a similar approach to SPL, where phases and models are quite similar, we can conclude that they may be extended to enable the development of MAS Product Lines (MAS-PL).

The main research lines that should be explored for enabling this kind of developments reside in: (i) adding commonality analysis and domain scoping to AOSE methodologies; and (ii); integrating the results of these activities to modify the process of assigning roles to agents and thus building a structural organization able to produce a family of MASs. In addition, the application engineering phase of SPL should also be included in AOSE methodologies to establish the procedures needed to derive a MAS from a MAS-PL.

References

1. P. Bresciani, A. Perini, P. Giorgini, F. Giunchiglia, and J. Mylopoulos. Tropos: an agent-oriented software development methodology. *Journal of Autonomous agents and Multiagent Systems*, 8(3), 2004.
2. P. Burrafato and M. Cossentino. Designing a multi-agent solution for a bookstore with the passi methodology. In *Fourth International Bi-Conference Workshop on Agent-Oriented Information Systems (AOIS-2002). CAiSE'02*, Toronto, Ontario, May 2002.
3. G. Caire, F. Leal, P. Chainho, R. Evans, F. Garijo, J. Gomez, J. Pavon, P. Kearney, J. Stark, and P. Massonet. Agent oriented analysis using MESSAGE/UML. In *Proceedings of Agent-Oriented Software Engineering (AOSE'01)*, pages 101–108, Montreal, 2001.
4. C. Castelfranchi. Founding agent's "autonomy" on dependence theory. In *14th European Conference on Artificial Intelligence*, pages 353–357. IOSPress, 2000.
5. C. Castelfranchi, M. Miceli, and A. Cesta. Dependence relations among autonomous agents. In In Y. Demazeau and E. Werner, editors, *Third European Workshop on Modeling Autonomous Agents in a Multi-Agent World. Decentralized AI 3*. Elsevier, 1992.
6. P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. SEI Series in Software Engineering. Addison–Wesley, August 2001.
7. K. Czarnecki and U Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison–Wesley, 2000.
8. A. Dardenne, A. van Lamsweerde, and S.Fickas. Goal-directed requirements acquisition. *Science of Computer Programming*, 20:3–50, 1993.
9. S. A. DeLoach, M. F. Wood, and C. H. Sparkman. Multiagent systems engineering. *The International Journal of Software Engineering and Knowledge Engineering*, 11(3):231–258, 2001. World Scientific Publishing Company.
10. D.F. D'Souza and A.C. Wills. *Objects, Components, and Frameworks with UML: The Catalysis Approach*. Addison–Wesley, Reading, Mass., 1999.

11. J. Ferber, O. Gutknecht, and F. Michel:. From agents to organizations: An organizational view of multi-agent systems. In Paolo Giorgini, Jörg P. Müller, and James Odell, editors, *IV International Workshop on Agent-Oriented Software Engineering (AOSE'03)*, volume 2935 of *LNCS*, pages 214–230. Springer–Verlag, 2003.
12. M Harsu. A survey on domain engineering. Technical Report 31, Institute of Software Systems, Tampere University of Technology, December 2002.
13. A. Jansen, R. Smedinga, J. Gurp, and J. Bosch. First class feature abstractions for product derivation. *IEE Proceedings - Software*, 151(4):187–198, 2004.
14. N. Jennings. An agent-based approach for building complex software systems. *Communications of the ACM*, 44(4):35–41, 2001.
15. N. R. Jennings. Agent-Oriented Software Engineering. In Francisco J. Garijo and Magnus Boman, editors, *Proceedings of MAAMAW-99*, volume 1647, pages 1–7. Springer-Verlag: Heidelberg, Germany, 30– 2 1999.
16. K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical Report CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie-Mellon University, November 1990.
17. E. Kendall, U. Palanivelan, and S. Kalikivayi. Capturing and structuring goals: Analysis patterns. In *Proceedings of the* 3^{rd} *European Conference on Pattern Languages of Programming and Computing*, Germany, July 1998.
18. E. A. Kendall. Role modeling for agent system analysis, design, and implementation. *IEEE Concurrency*, 8(2):34–41, April/June 2000.
19. H. Mintzberg. *The Structuring of Organizations*. Prentice-Hall, 1978.
20. H. V.n D. Parunak, S. Brueckner, M. Fleischer, and J. Odell. A design taxonomy of multi-agent interactions. In Paolo Giorgini, Jörg P. Müller, and James Odell, editors, *IV International Workshop on Agent-Oriented Software Engineering (AOSE'03)*, volume 2935 of *LNCS*, pages 123–137. Springer–Verlag, 2003.
21. J. Pavón and J. Gómez-Sanz. Agent oriented software engineering with ingenias. In V. Marík, J. Müller, and M. Pechoucek, editors, *Multi-Agent Systems and Applications III, 3rd International Central and Eastern European Conference on Multi-Agent Systems, CEEMAS 2003, Prague, Czech Republic, June 16-18, 2003, Proceedings*, volume 2691 of *Lecture Notes in Computer Science*, pages 394–403. Springer, 2003.
22. J. Pena. *On Improving The Modelling Of Complex Acquaintance Organisations Of Agents. A Method Fragment For The Analysis Phase*. PhD thesis, University of Seville, 2005.
23. T. Reenskaug. *Working with Objects: The OOram Software Engineering Method*. Manning Publications, 1996.
24. J. S. Sichman, Y. Demazeau, R. Conte, and C. Castelfranchi. A social reasoning mechanism based on dependence networks. In Y. Demazeau and E. Werner, editors, *11th European Conference on Artificial Intelligence*, pages 416–420. John Wiley and Sons, 1994.
25. Y. Smaragdakis and D. Batory. Mixin layers: an object–oriented implementation technique for refinements and collaboration-based designs. *ACM Trans. Softw. Eng. Methodol.*, 11(2):215–255, 2002.
26. M. J. Wooldridge and N. R. Jennings. Intelligent Agents: Theory and Practice. *Knowledge Engineering Review*, 10(2):115–152, June 1995.
27. F. Zambonelli, N. Jennings, and M. Wooldridge. Developing multiagent systems: the GAIA methodology. *ACM Transactions on Software Engineering and Methodology*, to be published 2003/2004.