Managing the Evolution of an Enterprise Architecture using a MAS-Product-Line Approach

Joaquin Peña
University of Seville
Spain
joaquinp@us.es

Michael G. Hinchey
NASA Goddard Space Flight Center
USA
Michael.G.Hinchey@nasa.gov

Manuel Resinas
University of Seville
Spain
resinas@us.es

Roy Sterritt
University of Ulster
Northern Ireland
r.sterritt@ulster.ac.uk

James L. Rash
NASA Goddard Space Flight Center
USA
James.L.Rash@nasa.gov

Abstract

We view an evolutionary system as being a software product line. The core architecture is the unchanging part of the system, and each version of the system may be viewed as a product from the product line. Each "product" may be described as the core architecture with some agent-based additions. The result is a multiagent system software product line. We describe an approach to such a Software Product Line-based approach using the MaCMAS Agent-Oriented methodology. The approach scales to enterprise architectures as a multiagent system is an appropriate means of representing a changing enterprise architecture and the interaction between components in it.

Keywords: *Multiagent Systems Product Lines, Enterprise architecture evolution.*

1 Introduction and Motivation

When dealing with complex systems, and in particular systems exhibiting any form of autonomy or autonomic properties, it is unrealistic to assume that the system will be static. Complex systems evolve over time, and the architecture of an evolving system will change even at run time, as the system implements self-configuration, self-adaptation, and meets the challenges of its environment.

An evolving system can be viewed as multiple versions of the same system. That is, as the system evolves it essentially represents multiple instances of the same system, each with its own variations and specific changes. That is to say, an evolving system may be viewed as a product line of systems, where the core architecture of the product line is fixed (i.e., the substantial part of the system that does not change), and each version of the evolving system may be viewed as a particular product from the product line.

Similarly, an enterprise architecture may be viewed as the core architecture that is unchanging, and various specializations of the architecture (as the enterprise evolves) implement various products of the product line.

If we consider the unchanging part of a software system or of an enterprise to be the core architecture, the specialization to various products (versions of the system) can be viewed as agent-based additions. The result is that an evolving system can be viewed as a Software Product Line of multiagent systems (MAS).

Our approach scales to enterprise architectures and software architecture for two reasons. Firstly, a multiagent system (MAS) is a very appropriate means of representing an enterprise and the interactions within it, thanks to the organizational metaphor that architects the system mimicking the real enterprise organization. Secondly, the gap between the enterprise architecture and the software architecture is mitigated through the addition of architectural concepts at the running platform. That is to say, MAS platforms are able to manage architectural evolutions and support architectural concepts at the implementation level.

In this paper, we propose a set of modeling techniques based on an agent-oriented methodology called *Methodology for Analyzing Complex Multiagent Systems* (MaCMAS) that is designed to deal with complex unpredictable systems [11][1]. Specifically, the approach we use is based on an extension of MaCMAS that allows to model MAS Product Lines (MAS-PL) [14, 13]. This allows us to manage the

[1]See www.tdg-seville.info/members/joaquinp/macmas/ for details and case studies using this methodology

modeling of the evolution of the system in a systematic way.

To the best of our knowledge, this is the first approach that deals with architectural changes of MASs based on MAS-PL.

2 Background and Related Work

The software product line paradigm (hereafter, SPL) augurs the potential of developing a core architecture from which customized products can be rapidly generated, reducing time-to-market, costs, etc. [1], while simultaneously improving quality, by making greater effort in design, implementation and test more financially viable, as this effort can be amortized over several products. The feasibility of building MAS product lines is presented in [13]. In [14], we discuss the details of how to build the core architecture.

In a MAS-PL, we can observe the enterprise architecture of the system from two different points of view. This distinction stems from the organizational metaphor [9, 10, 18]. These two views are the following:

Acquaintance point of view: shows the organization as the set of interaction relationships between the roles played by agents in models called *role models*. It focuses on the interactions within the system and also on representing how a functionality designated by a system goal is achieved.

Structural point of view: shows agents as artifacts that belong to sub-organizations, groups, teams. In this view agents are structured into hierarchical constructions showing the social structure of the system. It shows which agents are playing the roles in the Acquaintance Organization, and thus, shows how system goals are achieved by means of agents interacting to fulfill the system goals.

As shown in [7], the acquaintance organization can be modeled orthogonally to its structural organization. This allows us to change the system goals that are enabled in the system by changing the parts of the acquaintance organization present in the structural organization. This in fact, is the basis of MAS-PLs.

The software process of MAS-PLs is divided in two main stages: *Domain Engineering* and *Application Engineering*. The former is responsible for providing the reusable core assets that are exploited during application engineering when assembling or customizing individual applications [4]. Entering into details, we might say that, generally, both stages can be further divided into requirements, analysis, design, and implementation (a typical software development lifecycle).

domain requirements: This phase describes the requirements of the complete family of products, highlighting both the common and variable features across the family. In this phase, commonality analysis is of great importance for aiding in determining which are the commonalities and variabilities. The models used in this phase for specifying features show when a feature is optional, mandatory or alternative in the family. The models used are called *feature models* [2, 13]. A feature is a characteristic of the system that is observable by the end user, which in essence represents the same concept than a system goal as shown previously [6].

domain analysis: This phase produces architecture-independent models, i.e. acquaintance organization models, that define the features of the family and the domain of application. MAS-PLs use role models to represent the interfaces and interactions needed to cover certain functionality independently (a feature or a set of features)[14]. The most representative references in the non-MAS-PL field are [5, 17]. Similar approaches have appeared also in the OO field, for example [3, 16], but all of these approaches use role models with the same purpose, namely, representing features of the system in isolation from the final enterprise architecture.

domain design: In this phase, a core architecture of the family is produced, and is termed the core structural organization of the system. The core architecture is formed as a composition of the role models corresponding to the more stable features in the system [13].

application engineering: This phase has the responsibility of building concrete products. As our purpose in this paper is the runtime evolution of the system, we do not illustrate it here.

3 A NASA case study

The case study we use is a swarm of pico-spacecrafts that are used to prospect the asteroid belt. The enterprise architecture of the system changes at run-time depending on the environment and the state of the swarm. From all the possible evolutions we show only two states of the system: in the first one the swarm is orbiting an asteroid in order to analyze it; in the second, a solar storm occurs in the environment and the system changes its state to protect itself.

We will show the role models for both states and an example of composition of both of them since both features of the system are not completely orthogonal: to protect from a solar storm the spacecraft must take two basic steps: (a) orient its solar sails to minimize the area exposed to the solar storm particles (*trim sails*) and (b) power-off all possible electronic components. Step (a) minimizes the forces from impinging solar-storm particles, which could affect the spacecraft's orbit. Both steps (a) and (b) minimize potential damage from the charged particles in the storm (which can degrade sensors, detectors, electronic circuits, and solar energy collectors).

4 Modeling an Evolutionary MASs

As we have shown, each product in a MAS-PL is defined as a set of features. Given that all the products present a set of features that remain unchanged, the core architecture is defined as the part of all of the products that implement these common features[14]. Thus, a system can evolve by changing, or evolving, the set of non-core features.

A product or a state in our evolutionary system can be defined as a set of features. Let $F = \{f_1..f_n\}$ be the set of all features of a MAS-PL. Let $cF \subset F$ be the set of core features and $ncF = F \setminus CF$ be the set of non-core features. We define a valid state of the system as the set of core features and a set of non-core features, that is to say, $S = cF \cup sF$, where $sF \subset ncF$ is a subset of non-core features.

Given that, the evolution from one state S_{i-1} to another S_i is defined as:

$$S_i = S_{i-1} \cup nF_{i,i-1} \setminus dF_{i,i-1}$$

where $nF_{i,i-1} \subset ncF$ is the set of new features and $dF_{i,i-1} \subset ncF$ is the set of deleted features.

Finally, $\Delta_{i,i-1}$ describes the variation between the product of the state $i-1$ and the product of the state i, that is to say, $nF_{i,i-1} \setminus dF_{i,i-1}$.

In [13], we show that a feature correlates with a role model. Thus, for a system to evolve from one state to another, we must compose or decompose the role models in nF and dF. Specifically, we must compose the role models corresponding to the features in nF with the role models corresponding to the features that remain unchanged from the initial state S_{i-1}, that is to say $S_i \setminus dF_{i,i-1}$. Decomposition is used for role models that must be eliminated.

In the following subsections, we describe role models, and the operations for composition and decomposition.

5 Models

MaCMAS is the AOSE methodology that we use for our approach. It is specially tailored to model complex acquaintance organizations [15]. We use this methodology since it is the only that provides explicit support for MAS-PLs.

For the purposes of this paper, we only need to know a few features of MaCMAS, mainly some of the models it uses. Although a process for building these models is also needed, we do not address this in this paper, and refer the interested reader to the literature on this methodology. From the models it provides, we are interested in the following:

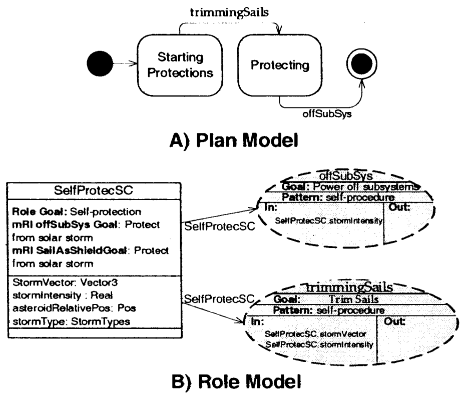

Figure 2. Self-protection from solar storms autonomic property model

a) Static Acquaintance Organization View: This shows the static interaction relationships between roles in the system and the knowledge processed by them. In this category, we can find models for representing the ontology managed by agents, models for representing their dependencies, and role models. For the purposes of this paper we only need to detail role models:

Role Models: show an acquaintance sub-organization as a set of roles collaborating by means of several *multi-Role Interaction* (mRI). mRIs are used to abstract the acquaintance relationships amongst roles in the system. As mRIs allow abstract representation of interactions, we can use these models at whatever level of abstraction we desire.

In Figure 1-B and 2-B, we show the role model that represents how the swarm orbits an asteroid and the one representing the protection from a solar storms. In the figures, interfaces, represented as boxes, represent the static features of roles showing their goals, the knowledge managed, and the services provided. mRIs, represented as dashed ellipses, represent the interactions between the roles linked to them, showing their goal when collaborating, their pattern of collaboration, and the knowledge consumed, used, and obtained from the collaboration.

b) Behavior of Acquaintance Organization View: The behavioral aspect of an organization shows the sequencing of mRIs in a particular role model. It is

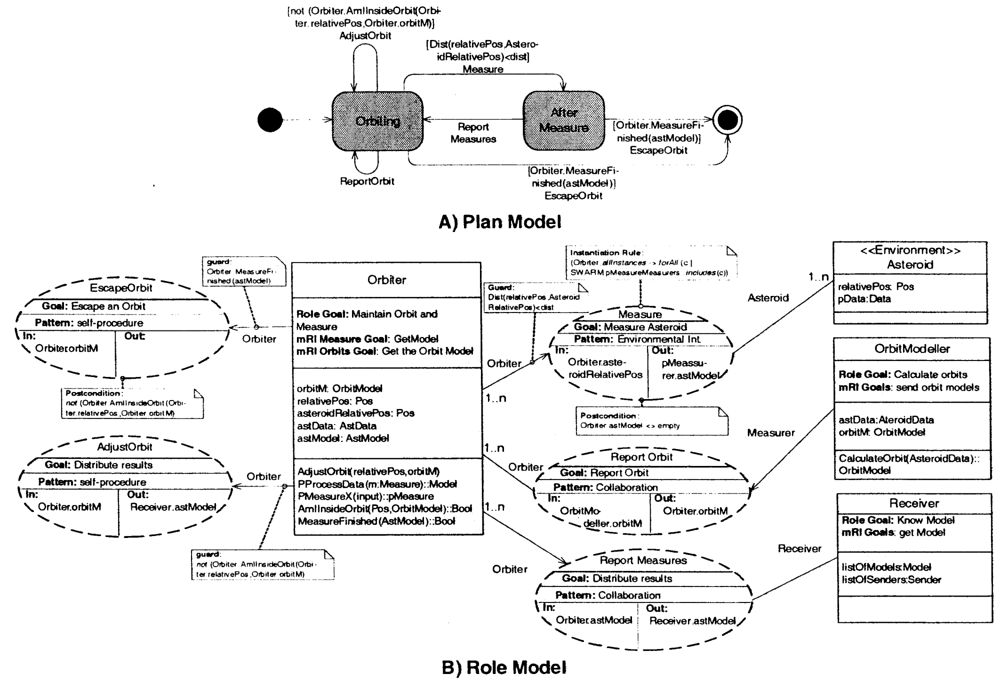

Figure 1. Orbiting and measuring an asteroid autonomous property

represented by two equivalent models:

Plan of a role: separately represents the plan of each role in a role model showing how the mRIs of the role sequence. It is represented using UML 2.0 ProtocolStateMachines. It is used to focus on a certain role, while ignoring others.

Plan of a role model: represents the order of mRIs in a role model with a centralized description. It is represented using UML 2.0 StateMachines. It is used to facilitate easy understanding of the whole behavior of a sub-organization.

In Figure 1-A and 2-A, we show the plan of the role models of our example.

We must add a new model to MaCMAS in order to represent the evolutions of the system. This model is called the evolution plan.

Evolution Plan: Is represented using a UML state machine where each state represents a product, and each transition represents the addition or elimination of a set of features, that is to say, Δ. In addition, the conditions in the transitions represent the properties that must hold in the environment and in the system in order to evolve to the new product.

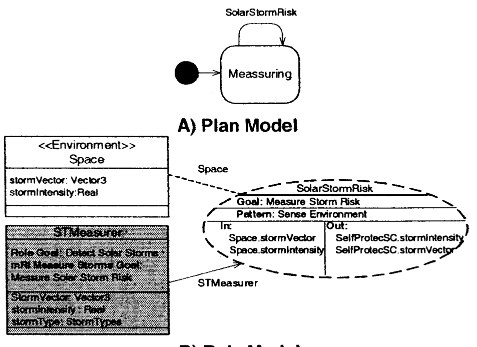

Figure 3. Measure storms model

In Figure 4, we show part of the evolution plan of our case study. There we represent two products, one representing the swarm when orbiting an asteroid, and another representing the swarm when orbiting and protecting from a solar storm. As can be seen, we add or delete the feature corresponding to protect from solar storm depending on whether or not the swarm is under risk of solar storm, which is measured by the feature represented in the role model of Figure 3.

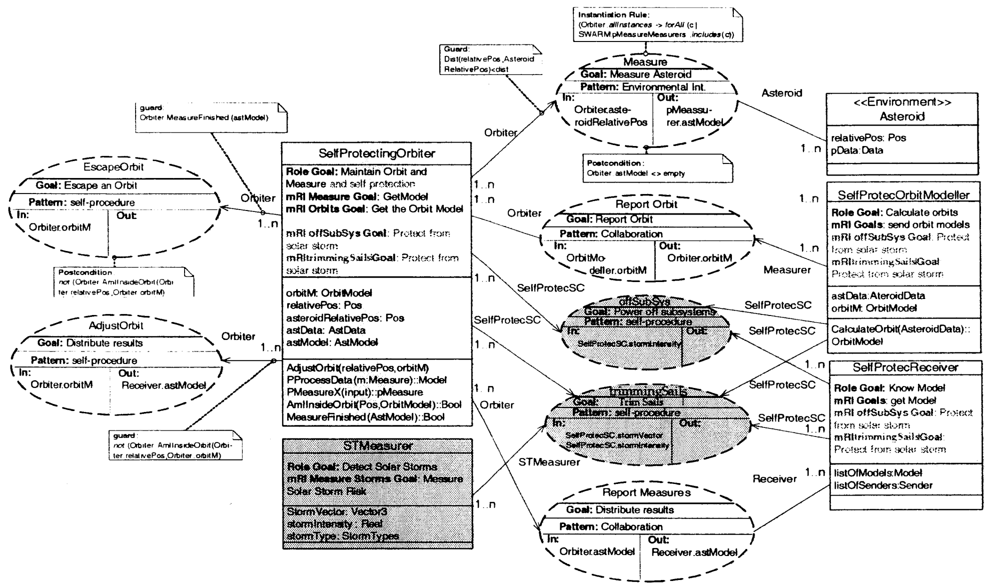
SelfProtectingOrbiter
Role Goal: Maintain Orbit and Measure and self protection
mRI Measure Goal: GetModel
mRI Orbits Goal: Get the Orbit Model
mRI offSubSys Goal: Protect from solar storm
mRItrimmingSailsGoal: Protect from solar storm
orbitM: OrbitModel
relativePos: Pos
asteroidRelativePos: Pos
astData: AstData
astModel: AstModel
AdjustOrbit(relativePos,orbitM)
PProcessData(m:Measure)::Model
PMeasureX(input)::pMeasure
AmIInsideOrbit(Pos,OrbitModel)::Bool
MeasureFinished(AstModel)::Bool
STMeasurer
Role Goal: Detect Solar Storms
mRI Measure Storms Goal: Measure Solar Storm Risk
StormVector: Vector3
stormIntensity: Real
stormType: StormTypes
EscapeOrbit
Goal: Escape an Orbit
Pattern: self-procedure
In: Orbiter.orbitM
Out:
AdjustOrbit
Goal: Distribute results
Pattern: self-procedure
In: Orbiter.orbitM
Out: Receiver.astModel
Measure
Goal: Measure Asteroid
Pattern: Environmental Int.
In: Orbiter.asteroidRelativePos
Out: pMeasurer.astModel
Report Orbit
Goal: Report Orbit
Pattern: Collaboration
In: OrbitModeller.orbitM
Out: Orbiter.orbitM
offSubSys
Goal: Power off subsystems
Pattern: self-procedure
trimmingSails
Goal: Trim Sails
Pattern: self-procedure
Report Measures
Goal: Distribute results
Pattern: Collaboration
In: Orbiter.astModel
Out: Receiver.astModel
<<Environment>> Asteroid
relativePos: Pos
pData: Data
SelfProtecOrbitModeller
Role Goal: Calculate orbits
mRI Goals: send orbit models
mRI offSubSys Goal: Protect from solar storm
mRItrimmingSailsGoal: Protect from solar storm
astData: AteroidData
orbitM: OrbitModel
CalculateOrbit(AsteroidData)::OrbitModel
SelfProtecReceiver
Role Goal: Know Model
mRI Goals: get Model
mRI offSubSys Goal: Protect from solar storm
mRItrimmingSailsGoal: Protect from solar storm
listOfModels:Model
listOfSenders:Sender
Instantiation Rule: (Orbiter allInstances -> forAll (c) SWARM.pMeasureMeasurers includes(c))
Guard: Dist(relativePos,AsteroidRelativePos)<dist
guard: Orbiter.MeasureFinished(astModel)
Postcondition: Orbiter.astModel <> empty
Postcondition: not (Orbiter.AmIInsideOrbit(Orbiter.relativePos,Orbiter.orbitM))
guard: not (Orbiter.AmIInsideOrbit(Orbiter.relativePos,Orbiter.orbitM))
Orbiter
Asteroid
Measurer
SelfProtecSC
STMeasurer
Receiver
1..n

Figure 5. Composed Role Model

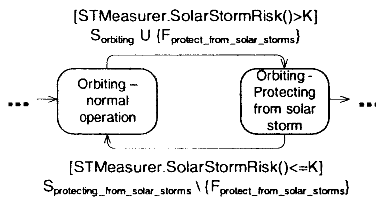
[STMeasurer.SolarStormRisk()>K]
$S_{orbiting} \cup \{F_{protect\_from\_solar\_storms}\}$
Orbiting – normal operation
Orbiting - Protecting from solar storm
...
...
[STMeasurer.SolarStormRisk()<=K]
$S_{protecting\_from\_solar\_storms} \setminus \{F_{protect\_from\_solar\_storms}\}$

Figure 4. Evolution plan of our case study

6 Evolving from one product to another

6.1 Composing role models

It is important to point out that the composition of role models is used to map an acquaintance organization onto a set of agents; that is to say, a structural organization. This mapping is not always orthogonal between all role models—applying two related features to a product may require their integration. The composition of role model is the process required to perform this integration. In the case of having orthogonal features, and thus orthogonal role models, we must only assign the prescribed roles to the corresponding agents.

We have to take into account that when composing several role models that are not independent, we can find: *emergent roles and mRIs*, artifacts that appear in the composition yet they do not belong to any of the initial role models; *composed roles and mRIs*, the roles and mRIs in the resultant models that represent several initial roles or mRIs as a single element; and, *unchanged roles and mRIs*, those that are left unchanged and imported directly from the initial role models.

Once those role models to be used for the core architecture have been determined, we must complete the core architecture by composing role models. In addition, to obtain a certain product we perform the same process. Importing an mRI or a role requires only its addition to the composite role model. The following shows how to compose roles and plans.

When several roles are merged in a composite role model, their elements must be also merged as follows:

Goal of the role: The new goal of the role is a new goal that abstracts all the role goals of the role to be composed. This information can be found in requirements hierarchical goal diagrams or we can add it as the *and* (conjunction) of the goals to be composed. In addition, the role goal for each mRI can be obtained from the goal of the initial roles for that mRI.

Cardinality of the role: It is the same as in the initial role for the corresponding mRI.

Initiator(s) role(s): If mRI composition is not performed, as in our case, this feature does not change.

Interface of a role: All elements in the interfaces of roles to be merged must be added to the composite interface. Notice that there may be common services and knowledge in these interfaces. When this happens, they must be included only once in the composite interface, or renamed, depending on the composition of their ontologies.

Guard of a role/mRI: The new guards are the *and* (conjunction) of the corresponding guards in initial role models if roles composed participate in the same mRI. Otherwise, guards remain unchanged.

In our case study, the evolution from the product *orbiting*, that also have the feature *measure storms*, to the product *protecting from solar storm* requires the addition of the feature to protect from a solar storm. This is due to two reasons: first, the features *orbiting and measure asteroid* and the *measure storms* belongs to the core architecture, and second, the *protection from solar storms* can happen in whichever moment and we must report the last measures of the asteroid before powering-off subsystems. Thus, as these role models are not orthogonal, we must perform a composition of them. This composition, represented in Figure 5, is done following the rule prescribed above. As can be observed, we have imported all the mRIs and most roles. In addition, we have performed a composition of roles *SelfProtecSC* and the rest in the role model *Orbit and measure asteroids*.

The composition of plans consists of setting the order of execution of mRIs in the composite model, using the role model plan or role plans. We provide several algorithms to assist in this task: extraction of a role plan from the role model plan and vice versa, and aggregation of several role plans; see [12] for further details of these algorithms.

Thanks to these algorithms, we can keep both plan views consistent automatically. Depending on the number of roles that have to be merged we can base the composition of the plan of the composite role model on the plan of roles or on the plan of the role model. Several types of plan composition can be used for role plans and for role model plans:

Sequential: The plan is executed atomically in sequence with others. The final state of each state machine is superimposed with the initial state of the state machine that represents the plan that must be executed, except the initial plan that maintains the initial state unchanged and the final plan that maintains the final state unchanged.

Interleaving: To interleave several plans, we must build a new state machine where all mRIs in all plans are taken into account. Notice that we must usually preserve the order of execution of each plan to be composed. We can use algorithms to check behavior inheritance to ensure that this constraint is preserved, since to ensure this property, the composed plan must inherit from all the initial plans [8].

The composition of role model plans has to be performed following one of the plan composition techniques described previously. Later, if we are interested in the plan of one of the composed roles, as it is needed to assign the new plan to the composed roles; we can extract it using the algorithms mentioned previously.

We can also perform a composition of role plans following one of the techniques to compose plans described previously. Later, if we are interested in the plan of the composite role model, for example for testing, we can obtain it using the algorithms mentioned previously.

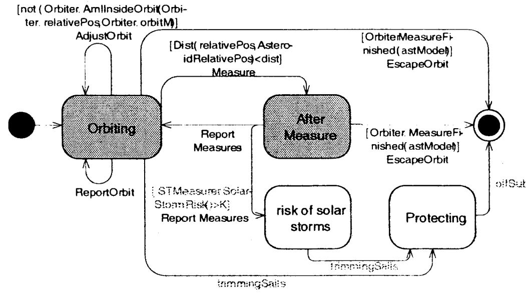
[not (Orbiter. AmIInsideOrbit(Orbiter. relativePos, Orbiter. orbitM)]
AdjustOrbit
[Dist(relativePos, AsteroidRelativePos)<dist]
Measure
[OrbiterMeasureFinished(astModel)]
EscapeOrbit
Orbiting
Report Measures
After Measure
[Orbiter. MeasureFinished(astModel)]
EscapeOrbit
ReportOrbit
[STMeasurer.SolarStormRisk()>K]
Report Measures
risk of solar storms
Protecting
offSubSys
trimmingSails
trimmingSails

Figure 6. Composed plan

In Figure 6, we show the composed plan for our case study. This plan follows an interleaving composition where we include the mRI *report measures* before starting the protection from the solar storm. Notice that when finishing the solar storm, the system will evolve to the other product deleting the feature *solar storm protection*. Then, the plan of the feature *orbiting and measure* will start from its initial state, thus re-starting the exploration of the asteroid.

6.2 Decomposition of role models

The decomposition is simpler than composition. When the role model to be eliminated is orthogonal to the rest, we only have to delete the corresponding roles from the agents that are playing them. In the case that the role model is dependent with others, we have to delete the elements of role models and eliminate all the interactions that refer to them. Given that the software architecture where the system run should support the role concept and its changes at runtime, these changes can be made easily with a lower impact on the system.

However, when dependent, several features may appear whose role models are related. In these cases some roles may have to be decomposed. These roles are those whose mRIs belong to the scope of the role model(s) that have to be eliminated. In these cases, the role has to be decomposed into several roles, in order to isolate the part of the role we want to delete.

In addition, we have to eliminate the mRI(s) of the role model(s) to be eliminated from the role model plan or the role plans. This is done starting from the plan of the initial dependent role models. Each separate role model usually maintains the order of execution of mRIs determined in the initial model but executing only a subset of mRIs of the initial role models. The behavior of the role model to be deleted can be extracted automatically using the algorithms described in [12]. This algorithm allows us to extract the plan of remaining role models from the initial ones constraining this to the set of mRIs that remains in the model.

7 Conclusions and future work

We have described a novel approach to describing, understanding, and analyzing evolving systems. Our approach is based on viewing different instances of a system as it evolves as different "products" in a Software Product Line.

That Software Product Line is in turn developed with an agent-oriented software engineering approach and views the system as a Multiagent System Product Line. The use of such an approach is particularly appropriate as it allows us to scale our view to address enterprise architectures where various entities in the enterprise are modeled as software agents.

The main advantage of the approach resides in the fact that it allows us to derive a formal model of the system and of each state that it may reach. This allows us to clearly specify the differences from one state of the architecture and any subsequent states of that evolving system. This significantly improves our capabilities to understand, analyze and test evolving systems. Additionally, thanks to the use of MaCMAS which allows for the description of the same feature at different levels of abstraction, we can also specify and test the architectural changes at different levels of abstraction.

Finally, such an approach provides support at run time for the addition and deletion of roles in the architecture. It provides reflection mechanisms that enable understanding of the features, roles, and agents in an enterprise architecture at different levels of abstraction, providing capabilities for ensuring quality of service by means of self-organization, self-protection, self-healing and other self-* properties identified by the Autonomic Computing initiative. Furthermore, it decreases the distance between enterprise architectures and software architectures, enabling us to model enterprise architectures as software architectures and exploit all of the advantages of software architectures approaches.

References

[1] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. SEI Series in Software Engineering. Addison–Wesley, Aug. 2001.

[2] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison–Wesley, 2000.

[3] D. D'Souza and A. Wills. *Objects, Components, and Frameworks with UML: The Catalysis Approach*. Addison–Wesley, Reading, Mass., 1999.

[4] M. Harsu. A survey on domain engineering. Technical Report 31, Institute of Software Systems, Tampere University of Technology, December 2002.

[5] A. Jansen, R. Smedinga, J. Gurp, and J. Bosch. First class feature abstractions for product derivation. *IEE Proceedings - Software*, 151(4):187–198, 2004.

[6] 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.

[7] E. A. Kendall. Role modeling for agent system analysis, design, and implementation. *IEEE Concurrency*, 8(2):34–41, Apr./June 2000.

[8] B. Liskov and J. M. Wing. Specifications and their use in defining subtypes. In *Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, pages 16–28. ACM Press, 1993.

[9] J. Odell, H. Parunak, and M. Fleischer. The role of roles in designing effective agent organisations. In A. Garcia and C. L. F. Z. A. O. J. Castro, editors, *Software Engineering for Large-Scale Multi-Agent Systems*, number 2603 in LNCS, pages 27–28, Berlin, 2003. Springer–Verlag.

[10] H. V. D. Parunak and J. Odell. Representing social structures in UML. In J. P. Müller, E. Andre, S. Sen, and C. Frasson, editors, *Proceedings of the Fifth International Conference on Autonomous Agents*, pages 100–101, Montreal, Canada, 2001. ACM Press.

[11] J. Peña. *On Improving The Modelling Of Complex Acquaintance Organisations Of Agents. A Method Fragment For The Analysis Phase*. PhD thesis, University of Seville, 2005.

[12] J. Peña, R. Corchuelo, and J. L. Arjona. Towards Interaction Protocol Operations for Large Multi-agent Systems. In *Proceedings of the Second International Workshop on Formal Approaches to Agent-Based Systems (FAABS 2002)*, volume 2699 of *LNAI*, pages 79–91, NASA-Goddard Space Flight Center, Greenbelt, MD, USA, 2002. Springer–Verlag.

[13] J. Peña and M. G. Hinchey. Multiagent system product lines: Challenges and benefits. *Communications of the ACM*, December 2006. Submitted and pre-accepted.

[14] J. Peña, M. G. Hinchey, and A. Ruiz-Cortés. Building the core architecture of a multiagent system product line: With an example from a future nasa mission. In *7th International Workshop on Agent Oriented Software Engineering 2006*, page to be published, Hakodate, Japan, May, 2006. LNCS.

[15] J. Peña, R. Levy, and R. Corchuelo. Towards clarifying the importance of interactions in agent-oriented software engineering. *International Iberoamerican Journal of AI*, (25):19–28, 2005.

[16] T. Reenskaug. *Working with Objects: The OOram Software Engineering Method*. Manning Publications, 1996.

[17] 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.

[18] F. Zambonelli, N. Jennings, and M. Wooldridge. Developing multiagent systems: the GAIA methodology. *ACM Transactions on Software Engineering and Methodology*, 12(3):317–370, July 2003.