Designing and managing evolving systems using a MAS product line approach[☆]

Joaquin Peña[a,*], Michael G. Hinchey[b,1], Manuel Resinas[a], Roy Sterritt[c], James L. Rash[b]

[a] *University of Seville, E.T.S.I. Informática, Dpto. de Lenguajes y Sistemas Informáticos, Avda. de la Reina Mercedes s/n, Sevilla, 41.012, Spain* [b] *NASA Goddard Space Flight Center, Information Systems Division, Greenbelt, MD 20771, USA*
[c] *School of Computing and Mathematics, Faculty of Engineering, University of Ulster, Jordanstown, Northern Ireland, BT37 0QB, United Kingdom*

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. In addition, we reduce the gap between the enterprise architecture and the software architecture.

Keywords: Multiagent systems product lines; Enterprise architecture evolution; Swarm-based systems

1. Introduction

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

[☆] This work has been partially supported by the European Commission (FEDER) and Spanish Government under CICYT project Web-Factories (TIN2006-00472) and TIC200302737C0201, by the NASA Software Engineering Laboratory, NASA Goddard Space Flight Center, Greenbelt, MD, USA.

[*] Corresponding author.

E-mail addresses: joaquinp@us.es (J. Peña), Michael.G.Hinchey@nasa.gov, mhinchey@loyola.edu (M.G. Hinchey), resinas@us.es (M. Resinas), r.sterritt@ulster.ac.uk (R. Sterritt), James.L.Rash@nasa.gov (J.L. Rash).

[1] Also at: Loyola College, Department of Computer Science, Baltimore, MD 21210, USA.

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.

In this paper, we use multiagent-based modelling techniques to model software systems that evolve over time. The result is that an evolving system can be viewed as a software product line of MultiAgent Systems (MAS). Therefore, 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.

Our approach scales to enterprise architectures and software architecture for two reasons. Firstly, a multiagent system 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 on 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 software engineering 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].[2] Specifically, the approach we use is based on an extension of MaCMAS that allows one to model MAS Product Lines (MAS-PL) [14,13]. This allows us to manage the modeling of the evolution of the system in a systematic way.

The main contributions of this paper are: (i) to the best of our knowledge, this is the first approach that deals with architectural changes of evolving systems based on MAS-PL; (ii) we reduce the gap between the enterprise and software architecture.

2. Motivating our approach with two case studies: A NASA case study and a human organization case study

The NASA case study focuses on showing a complex evolving system and is used to show how this is managed in our approach. The second case study focuses on showing the gap between the enterprise architecture and the software architecture and how it is managed in our approach.

2.1. A NASA case study

There has been significant NASA research on the subject of agent technology, with a view to greater exploitation of such technologies in future missions.

The ANTS (Autonomous Nano-Technology Swarm) concept mission, for example, will be based on a grouping of agents that work jointly and autonomously to achieve mission goals, analogous to a swarm in nature.

The Prospecting Asteroid Mission (PAM) is a concept sub-mission based on the ANTS concepts that will be dedicated to exploring the asteroid belt. A thousand picospacecraft (less than 1 kg each) may be launched from a point in space forming sub-swarms, and deployed to study asteroids of interest in the asteroid belt.

The software architecture of this system changes at run time depending on the environment and the state of the swarm. To simplify our examples, 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.

To orbit and analyze an asteroid the spacecraft in the swarm must calculate and adjust the orbit, perform the measures, report the information retrieved, and escape the orbit when the process is finished.

For protection 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).

As shown, the two situations are quite different requiring completely different architectures in the system. This motivated us to observe these states as different systems, and thus, led us to the product line field.

[2] See james.eii.us.es/MaCMAS/ for details and case studies using this methodology.

2.2. *A human organization case study*

This case study has been proposed by the Modeling TC of FIPA with the purpose of evaluating AOSE methodologies [8]. It is called *The United Nations (UN) Security Council's Procedure to Issue Resolutions*.

The case study is inspired from the procedure used by the UN Security Council to pass a resolution. However, it does NOT necessarily represent reality. 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 is followed:

- At least one member of the UN-SC submits a proposal to the current *Chair*.
- The *Chair* distributes the proposal to all members of UN-SC and sets a date for a vote on the proposal.
- On the date set by the Chair, a vote from the members is made.
- Each member of the Security Council can vote either FOR or AGAINST or SUSTAIN.
- The proposal becomes a UN-SC resolution if the majority of the members voted FOR, and no permanent member voted AGAINST.
- The members vote one at a time.
- The Chair calls the order to vote.
- The vote is open (in other words, when one votes, all the other members know the vote).
- The proposing member(s) can withdraw the proposal before the vote starts and in that case, no vote on the proposal will take place.
- 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 when a proposal is submitted and the time at which it goes into vote; in this case the earlier chair has to forward the proposal to the new one.
- A vote is always finished in one day and no chair change happens on that day. The date of the vote is set by the chair.
- In case of war, the change of chair can be prohibited.

In this case study, we can see that the organization has to change in two situations: when the countries participating in each process change, and when a war happens the process for changing the chair can be prohibited, which changes the enterprise architecture of the system and thus the software architecture.

3. Background information

As a result of combining different fields, we have to contextualize our work in all these areas. In this section, we provide the background needed for formal methods, crucial for many safety-critical applications, MaCMAS, product lines, and how to build the core architecture of a product line.

3.1. *Formal methods*

Formal methods are crucial for dealing with complex systems, as is the case for NASA missions. They provide techniques for analyzing, testing, and proving, etc., properties of the system that help one to deal with the inherent complexity of evolving systems.

However, it has been stated that formal analysis is not feasible for emergent systems due to the complexity and intractability of these systems, and that simulation is the only viable approach for analyzing emergence of systems [1]. For NASA missions, relying on simulations and testing alone is not sufficient even for systems that are much simpler than the ANTS mission, as noted above. The use of formal analysis would complement the simulation and testing of these complex systems and would give additional assurance of their correct operation. Given that one mistake can be catastrophic to a system and result in the loss of hundreds of millions of dollars and years of work, development of a formal analysis tool, even at a great cost, could have huge returns even if only one mission is kept from failing.

The FAST project identified several important attributes needed in a formal approach for verifying swarm-based systems and surveyed a wide range of formal methods and formal techniques for determining whether existing formal methods, or a combination of existing methods, could be suitable for specifying and verifying swarm-based missions and their emergent behavior [17–19].

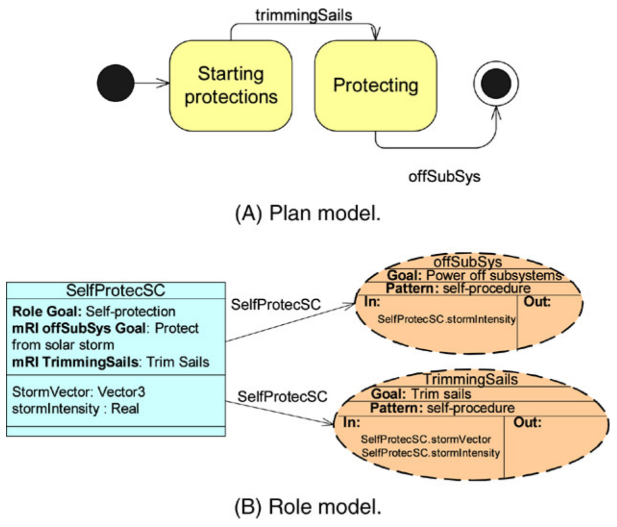


Fig. 1. Self-protection from solar storms autonomic property model.

One of the main drawbacks found in the application of formal methods relates to the size and scope of the specifications. When dealing with complex evolving systems, such as NASA missions based on ANTS, these specifications become unmanageable, needing mechanisms to divide and conquer them. The approach presented in this paper allows us to factorize the systems, easing the decomposition of the specifications and thus their comprehension.

3.2. Overview of MaCMAS models

MaCMAS is the AOSE methodology that we use for our approach. We use this methodology since it is the only one 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 that 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:

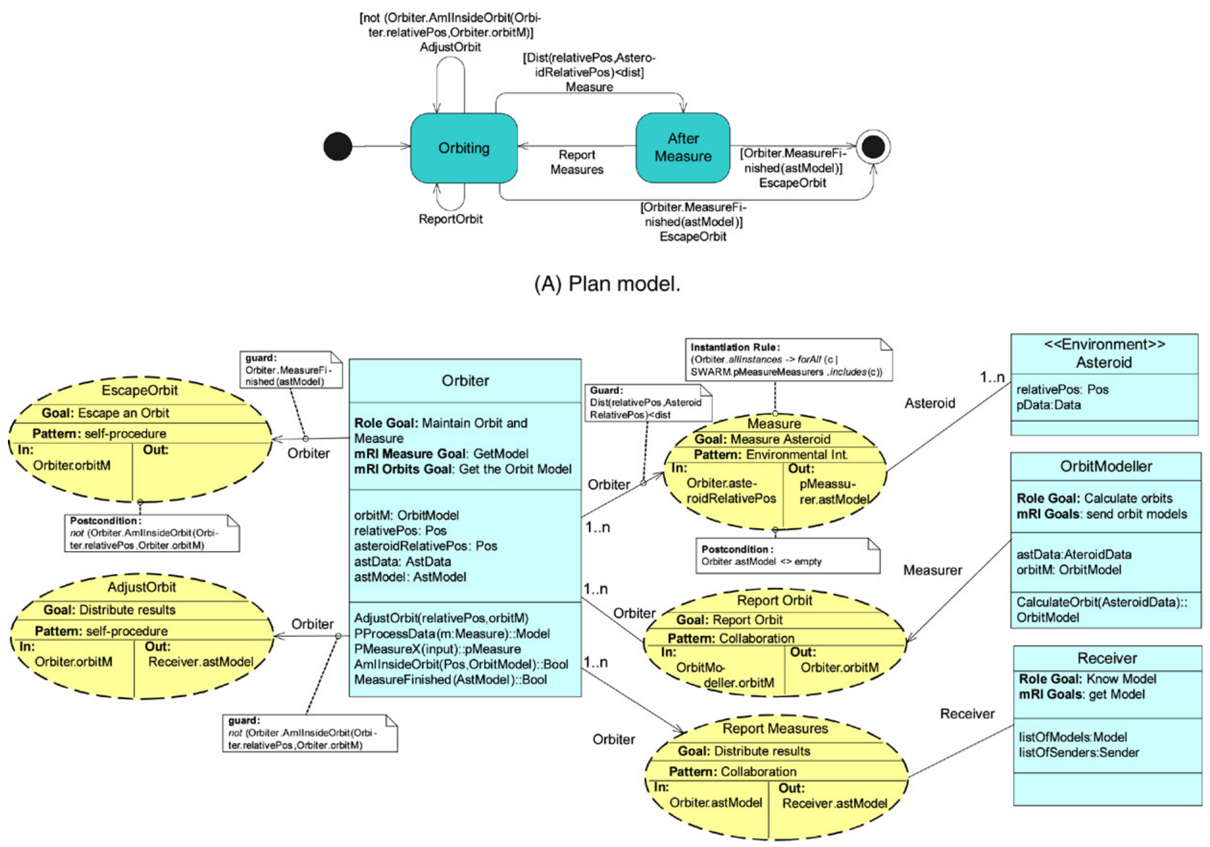
(a) Static acquaintance organization view: This shows the static interaction relationships between roles, parts of the agent involved in an interaction, in the systems and the knowledge processed by them. In this category, we can find models for representing the ontology managed by agents, models for 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 Interactions* (mRIs). 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 Figs. 2B and 1B, we show the role model that represents how a swarm spacecraft orbits an asteroid and the one representing the protection from a solar storm while the swarm spacecraft continues in its orbit. 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 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.



[not (Orbiter.AmIInsideOrbit(Orbiter.relativePos,Orbiter.orbitM)] AdjustOrbit
[Dist(relativePos,AsteroidRelativePos)<dist] Measure
Orbiting
After Measure
Report Measures
[Orbiter.MeasureFinished(astModel)] EscapeOrbit
ReportOrbit
[Orbiter.MeasureFinished(astModel)] EscapeOrbit
(A) Plan model.
EscapeOrbit
Goal: Escape an Orbit
Pattern: self-procedure
In: Orbiter.orbitM
Out:
guard: Orbiter.MeasureFinished(astModel)
Orbiter
Postcondition: not (Orbiter.AmIInsideOrbit(Orbiter.relativePos,Orbiter.orbitM)
AdjustOrbit
Goal: Distribute results
Pattern: self-procedure
In: Orbiter.orbitM
Out: Receiver.astModel
guard: not (Orbiter.AmIInsideOrbit(Orbiter.relativePos,Orbiter.orbitM)
Orbiter
Role Goal: Maintain Orbit and Measure
mRI Measure Goal: GetModel
mRI Orbits Goal: Get the Orbit Model
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
Guard: Dist(relativePos,Asteroid RelativePos)<dist
Instantiation Rule: (Orbiter.allInstances -> forAll (c | SWARM.pMeasureMeasurers.includes(c))
Measure
Goal: Measure Asteroid
Pattern: Environmental Int.
In: Orbiter.asteroidRelativePos
Out: pMeassurer.astModel
Postcondition: Orbiter.astModel <> empty
Report Orbit
Goal: Report Orbit
Pattern: Collaboration
In: OrbitModeller.orbitM
Out: Orbiter.orbitM
Report Measures
Goal: Distribute results
Pattern: Collaboration
In: Orbiter.astModel
Out: Receiver.astModel
Orbiter
1..n
Orbiter
1..n
Orbiter
1..n
Orbiter
Asteroid
1..n
Measurer
Receiver
<<Environment>> Asteroid
relativePos: Pos
pData:Data
OrbitModeller
Role Goal: Calculate orbits
mRI Goals: send orbit models
astData:AteroidData
orbitM: OrbitModel
CalculateOrbit(AsteroidData):: OrbitModel
Receiver
Role Goal: Know Model
mRI Goals: get Model
listOfModels:Model
listOfSenders:Sender
(B) Role model.

Fig. 2. Orbiting and measuring an asteroid autonomous property.

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 Figs. 2A and 1A, we show the plan of the role models of our example.

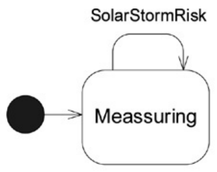
3.3. *Multiagent systems product lines*

The field of software product lines covers the entire software life-cycle needed to develop a family of products where the derivation of concrete products is achieved systematically or even automatically when possible, 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.

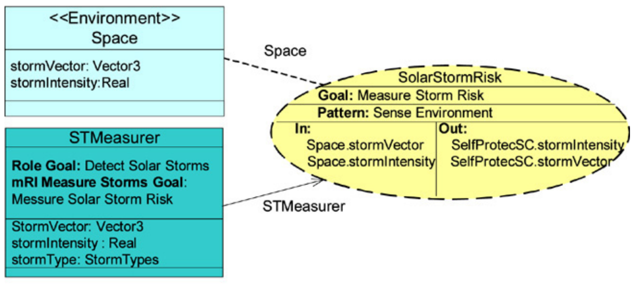
A MAS-PL is a product line of multiagent systems. The feasibility of building MAS product lines is presented in [13]. For the purpose of this paper, it is important to note that in a MAS-PL, we can observe the architecture of the system from two different points of view [13]. This distinction stems from the organizational metaphor [9,10,21]. 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 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



(A) Plan model.



(B) Role model.

Fig. 3. Measure storms model.

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 [6], 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 MAS-PL software process is divided into 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 [15]. Although there are other activities, such as product management, in this section we do not try to be exhaustive, but only discuss those activities directly related to this paper and relevant to our approach.

Thus, following the nomenclature used in [15], the activities, usually performed iteratively and in parallel, of domain engineering are:

Domain requirements engineering: This phase describes the requirements of the complete family of products, highlighting both the common and the 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 as a system goal, as shown previously [5].

Domain design: 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 [4,20]. 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 realization: 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].

Next, we briefly describe an approach, presented in [14], for building the core architecture of a MAS-PL, which is an important part of the domain engineering stage. The remainder of the article builds on the results of that paper and provides techniques that are used during the application engineering stage.

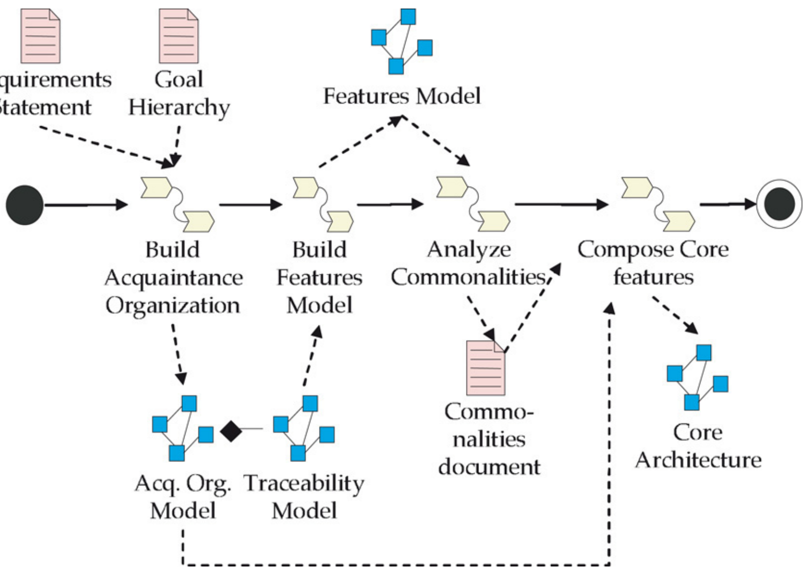


Fig. 4. Overview of building the core architecture of a MAS-PL.

3.4. *Building the core architecture of a MAS-PL*

In [14], we presented an approach to building the core architecture of a MAS-PL at the domain engineering stage. In Fig. 4, we show roughly the process we follow to build the core architecture, whose activities are:

Build acquaintance organization. The first stage to be performed consists of developing a set of models at different layers of abstraction where we obtain a traceability model and a set of role models showing how each goal is materialized. This is achieved by applying the MaCMAS software process.

Build features model. The second activity shown is responsible for adding commonalities and variabilities to the traceability model. This is done by modifying the traceability diagram to add information on variability and commonalities to obtain a feature model of the family. It relies on taking each node of the traceability diagram and determining whether it is mandatory, optional, alternative, or-exclusive, or whether it depends on other(s), as shown in the figure. In Fig. 6, we present the resultant features model for the NASA case study.

In addition, there exists a direct traceability between features and role models.

When a system goal is complex enough to require more than one agent in order to be fulfilled, a group of agents are required to work together. Hence, a role model shows a set of agents, represented by the role they play, that join to achieve a certain system goal (whether by contention or cooperation). MaCMAS uses mRIs to represent all of the joint processes that are required and are carried out amongst roles in order to fulfill the system goal of the role model. These also pursue system sub-goals as shown in Fig. 5, where we can see the correlation between these elements and the feature model of the NASA case study. Note that the role model of this figure can also be seen in Fig. 1.

Analyze commonalities. Later, we perform a commonality analysis to find out which features, called core features, and thus which role models, are more used across products.

To build the core architecture of the system we must include those features that appear in all the products and those whose probability of appearing in a product is high.

Compose core features. Then, given that features models present a direct traceability between system goals and role models, we can use the composition operation of MaCMAS to compose the role models corresponding to the core features to obtain the core architecture.

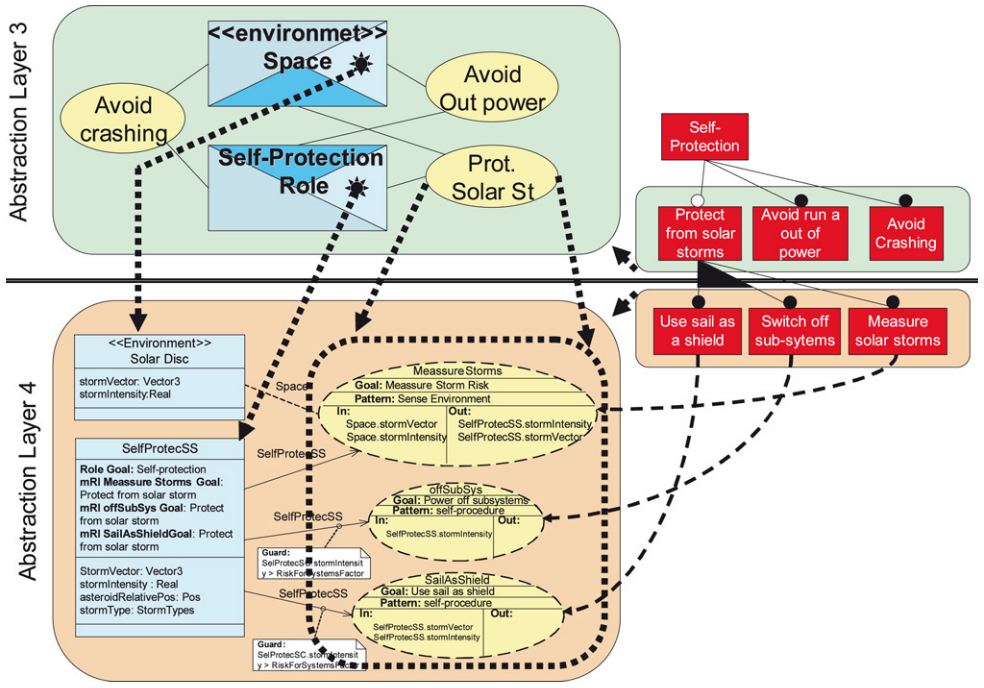

Abstraction Layer 3
<<environment>>
Space
Avoid crashing
Avoid Out power
Self-Protection Role
Prot. Solar St
Self-Protection
Protect from solar storms
Avoid run a out of power
Avoid Crashing
Use sail as a shield
Switch off sub-sytems
Measure solar storms
Abstraction Layer 4
<<Environment>> Solar Disc
stormVector: Vector3
stormIntensity:Real
SelfProtecSS
Role Goal: Self-protection
mRI Meassure Storms Goal: Protect from solar storm
mRI offSubSys Goal: Protect from solar storm
mRI SailAsShieldGoal: Protect from solar storm
StormVector: Vector3
stormIntensity : Real
asteroidRelativePos: Pos
stormType: StormTypes
MeassureStorms
Goal: Measure Storm Risk
Pattern: Sense Environment
In: Space.stormVector Space.stormIntensity
Out: SelfProtecSS.stormIntensity SelfProtecSS.stormVector
offSubSys
Goal: Power off subsystems
Pattern: self-procedure
In: SelfProtecSS.stormIntensity
Out:
SailAsShield
Goal: Use sail as shield
Pattern: self-procedure
In: SelfProtecSS.stormVector SelfProtecSS.stormIntensity
Out:
Guard: SelProtecSC.stormIntensity > RiskForSystemsFactor

Fig. 5. Role model/features relationship.

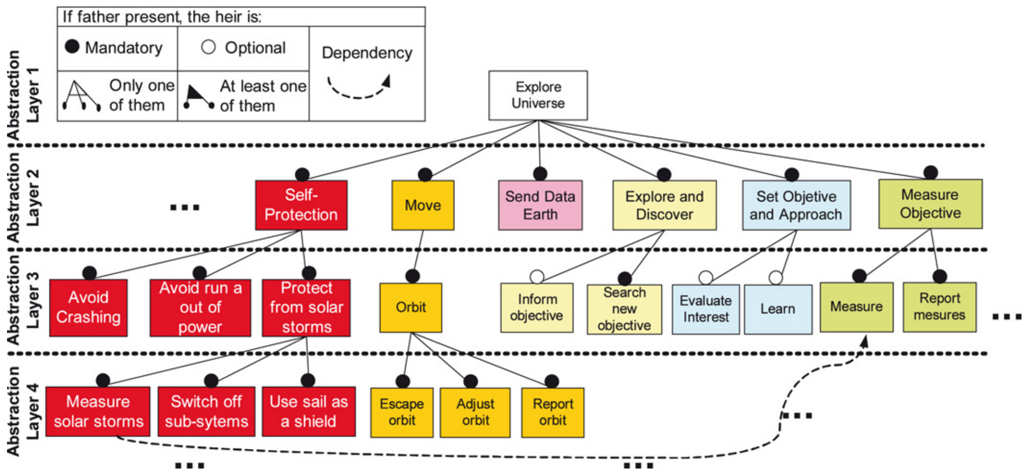


Fig. 6. Features model of the NASA case study.

4. Using MAS-PL to design an evolving system

We define each product in a MAS-PL 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 parts 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 $ncF = F\backslash CF$ 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.

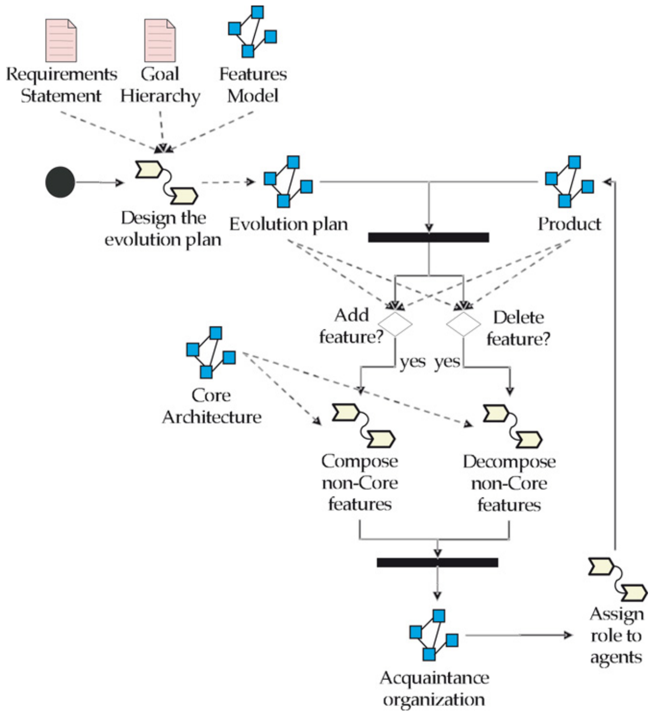


Fig. 7. Overview of our approach.

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

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

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, $nF_{i,i-1} \backslash dF_{i,i-1}$.

As shown previously, every 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 $S_i \backslash dF_{i,i-1}$. Decomposition is used for role models that must be eliminated.

Given this definition, the software process of our approach must start from the core architecture of the system, to later add/delete the needed features for each product.

In Fig. 7, we present the overview of the software process. As shown, the first step is designing the evolution plan that must show all the products in the system and whose features must be added/deleted to go from one product to another. Then, using this information we must compose/decompose the role models and plan models corresponding to each feature over the current product to obtain the acquaintance organization. Finally, we deploy the roles in the acquaintance organization of the product over the agents in the system. Notice that this process is supported by the implementation platform.

The following sections show each of these procedures exemplifying them with the case studies previously presented.

5. Designing the evolution plan

To show the information on which products we are managing and which features are added/deleted from one product to another we must add a new model to MaCMAS. This model is called the evolution plan.

The 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

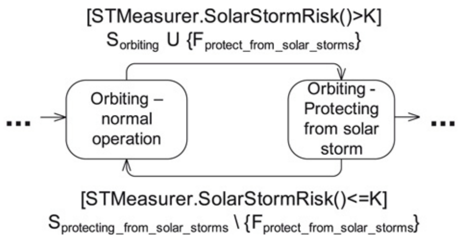

[STMeasurer.SolarStormRisk()>K]
S_orbiting U {F_protect_from_solar_storms}
Orbiting – normal operation
Orbiting - Protecting from solar storm
...
...
[STMeasurer.SolarStormRisk()<=K]
S_protecting_from_solar_storms \ {F_protect_from_solar_storms}

Fig. 8. Part of the evolution plan of the NASA case study.

transitions represent the properties that must hold in the environment and in the system in order to evolve to the new product.

To design this plan, we must use information about requirements to identify which are the conditions that trigger the change of product, and the features that distinguish one product from another.

In Fig. 8, we show part of the evolution plan of the NASA case study. There we present 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 protecting from a solar storm depending on whether or not the swarm is at risk from a solar storm, which is measured by the feature represented in the role model of Fig. 3.

6. Composing role models

The addition of a role model to an existing product is not always orthogonal — applying two related features to a product may require their integration. The composition of role models 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 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 have been determined, we must compose them. 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 abstracts all the 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.

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 the NASA case study, the evolution from the product *orbiting*, that also has the feature *measure storms*, to the product *protecting from solar storm* requires the addition of the feature for protecting from a solar storm. This is for two reasons: first, the features *orbiting and measure asteroid* and *measure storms* belong to the core architecture, and second, the *protection from solar storms* can happen at any 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 Fig. 9, 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*.

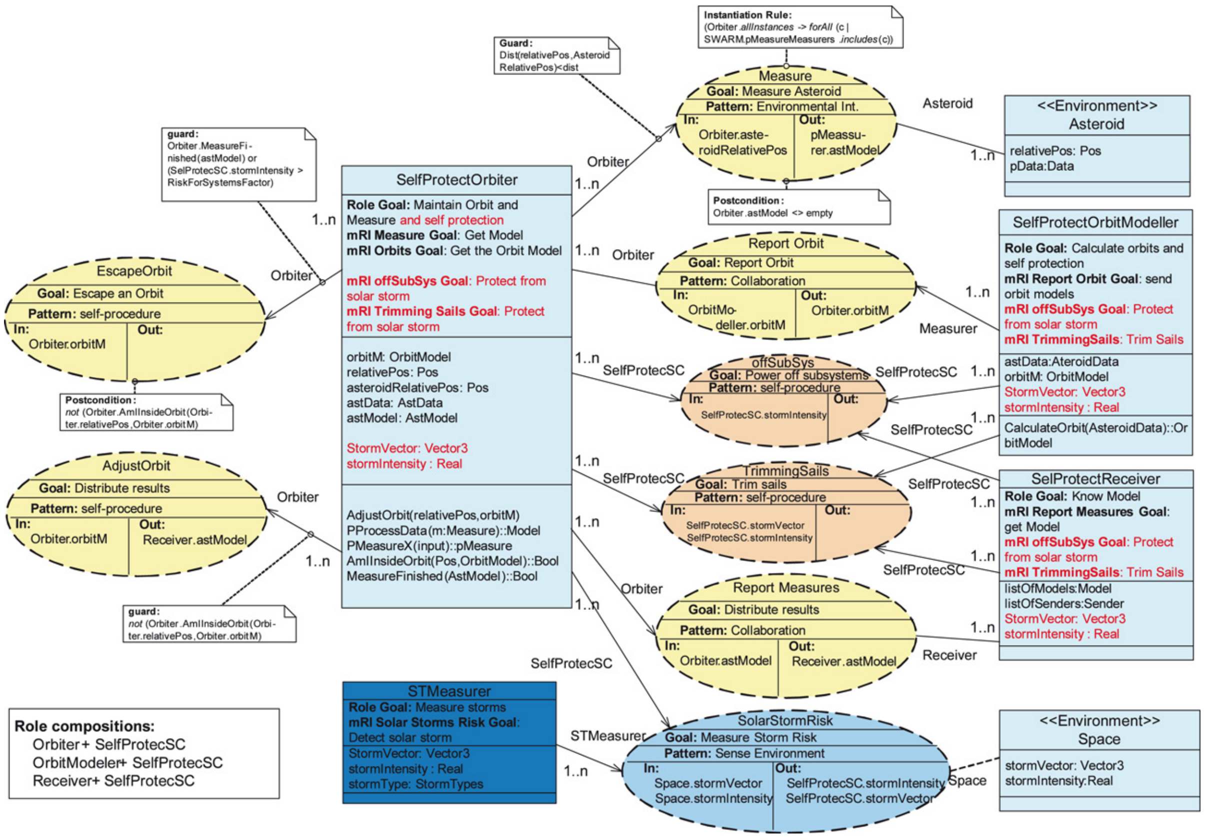
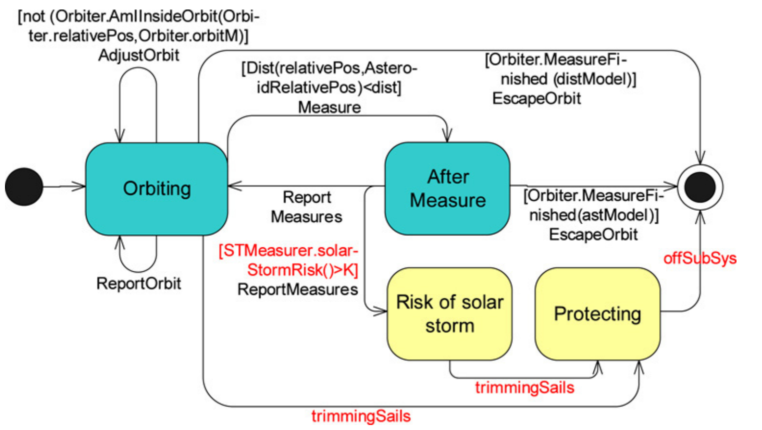


Fig. 9. Composed role model.


[not (Orbiter.AmIInsideOrbit(Orbiter.relativePos,Orbiter.orbitM)]
AdjustOrbit
[Dist(relativePos,AsteroidRelativePos)<dist]
Measure
[Orbiter.MeasureFinished (distModel)]
EscapeOrbit
Orbiting
After Measure
Report Measures
[Orbiter.MeasureFinished(astModel)]
EscapeOrbit
ReportOrbit
[STMeasurer.solarStormRisk()>K]
ReportMeasures
offSubSys
Risk of solar storm
Protecting
trimmingSails
trimmingSails
Fig. 10. Composed plan.

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:

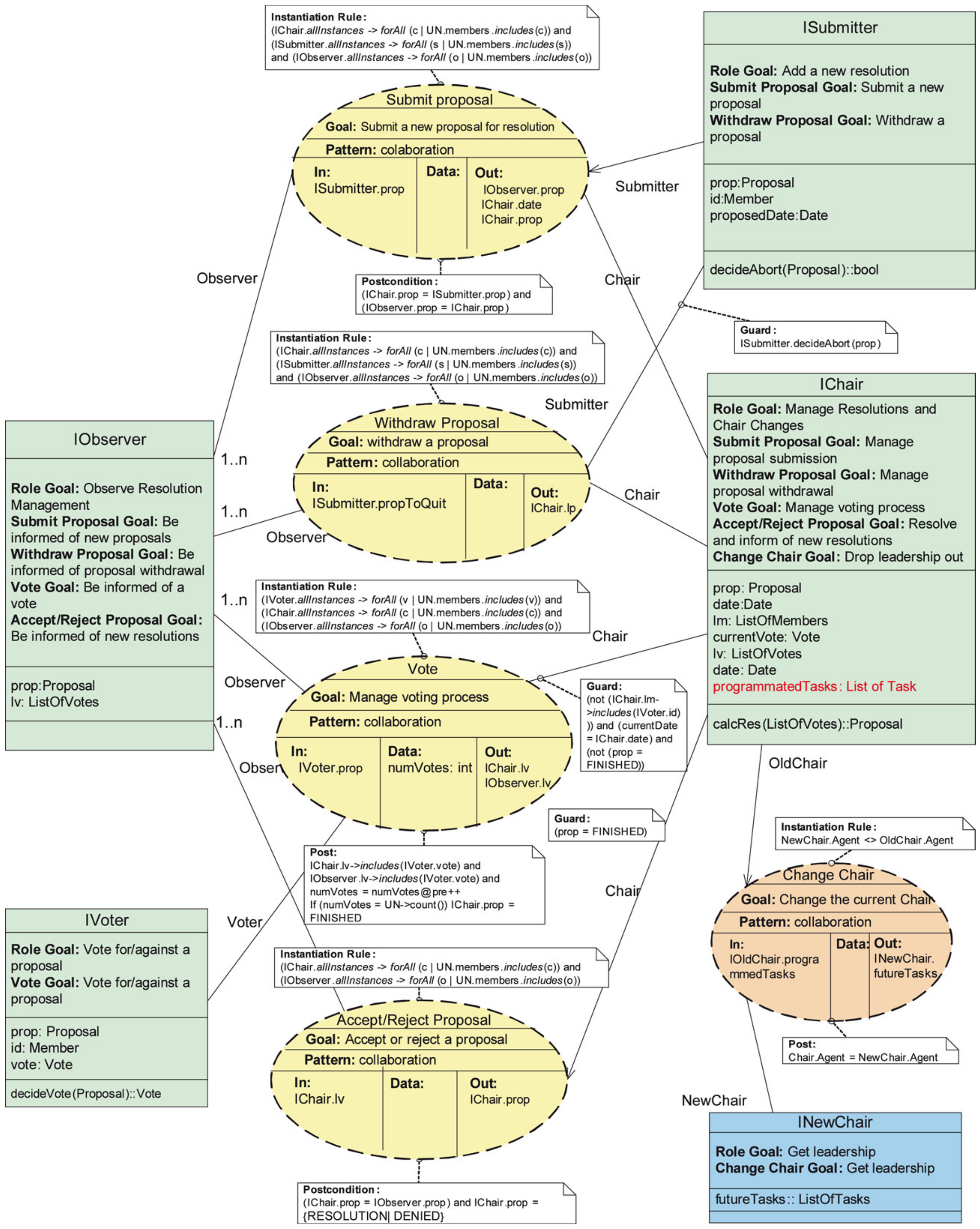

Instantiation Rule: (IChair.allInstances -> forAll (c | UN.members.includes(c)) and (ISubmitter.allInstances -> forAll (s | UN.members.includes(s)) and (IObserver.allInstances -> forAll (o | UN.members.includes(o))
Submit proposal
Goal: Submit a new proposal for resolution
Pattern: colaboration
In: ISubmitter.prop
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
Chair
Observer
Guard: ISubmitter.decideAbort(prop)
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
Chair
Observer
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
1..n
1..n
IChair
Role Goal: Manage Resolutions and Chair Changes
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
Change Chair Goal: Drop leadership out
prop: Proposal
date:Date
lm: ListOfMembers
currentVote: Vote
lv: ListOfVotes
date: Date
programmatedTasks: List of Task
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))
Chair
Observer
Vote
Goal: Manage voting process
Pattern: collaboration
In: IVoter.prop
Data: numVotes: int
Out: IChair.lv IObserver.lv
Observer
Guard: (not (IChair.lm->includes(IVoter.id)) and (currentDate = IChair.date) and (not (prop = FINISHED))
Guard: (prop = FINISHED)
OldChair
Post: IChair.lv->includes(IVoter.vote) and IObserver.lv->includes(IVoter.vote) and numVotes = numVotes@pre++ If (numVotes = UN->count()) IChair.prop = FINISHED
Chair
Instantiation Rule: NewChair.Agent <> OldChair.Agent
Change Chair
Goal: Change the current Chair
Pattern: collaboration
In: IOldChair.programmedTasks
Data:
Out: INewChair.futureTasks
Post: Chair.Agent = NewChair.Agent
IVoter
Role Goal: Vote for/against a proposal
Vote Goal: Vote for/against a proposal
prop: Proposal
id: Member
vote: Vote
decideVote(Proposal)::Vote
Voter
Instantiation Rule: (IChair.allInstances -> forAll (c | UN.members.includes(c)) and (IObserver.allInstances -> forAll (o | UN.members.includes(o))
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}
NewChair
INewChair
Role Goal: Get leadership
Change Chair Goal: Get leadership
futureTasks:: ListOfTasks

Fig. 11. Role model of issue resolution and change chair.

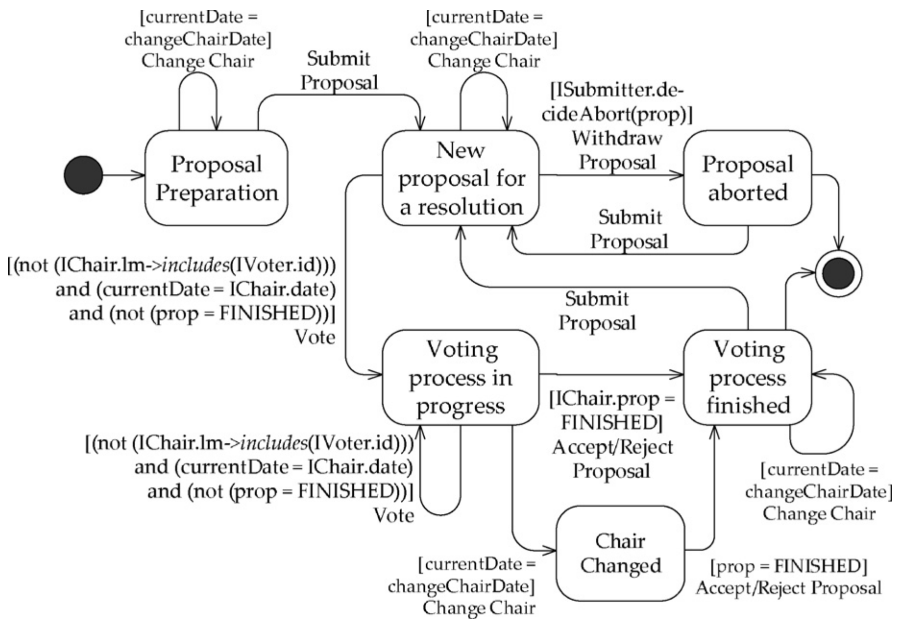

[currentDate = changeChairDate] Change Chair
Submit Proposal
[currentDate = changeChairDate] Change Chair
[ISubmitter.decideAbort(prop)] Withdraw Proposal
Proposal Preparation
New proposal for a resolution
Proposal aborted
Submit Proposal
[(not (IChair.lm->includes(IVoter.id))) and (currentDate = IChair.date) and (not (prop = FINISHED))] Vote
Submit Proposal
Voting process in progress
Voting process finished
[IChair.prop = FINISHED] Accept/Reject Proposal
[(not (IChair.lm->includes(IVoter.id))) and (currentDate = IChair.date) and (not (prop = FINISHED))] Vote
[currentDate = changeChairDate] Change Chair
Chair Changed
[currentDate = changeChairDate] Change Chair
[prop = FINISHED] Accept/Reject Proposal

Fig. 12. Role model plan of issue resolution and change chair.

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 [7].

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.

In Fig. 10, we show the composed plan for the NASA 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 restarting the exploration of the asteroid.

7. Decomposition of role models

In this case, we have to remove some features of the system. Given that the evolution plan gives us the information on the features to be removed and that there is a direct correlation between the features models and the models of the architecture, we can identify which role model must be removed.

Regarding the UN case study, we assume that the current product contains the feature for issuing a resolution and the feature for changing the chair. The role model is presented in Fig. 11, and in Fig. 12 we present the plan.

As can be observed, the roles that we can find in this system are the chair, the observer, the voter, and the submitter. These roles are mapped onto the UN countries forming the structural organization/software architecture. Notice that these countries change over time; thus, the software architecture must also change. However, as our approach separates

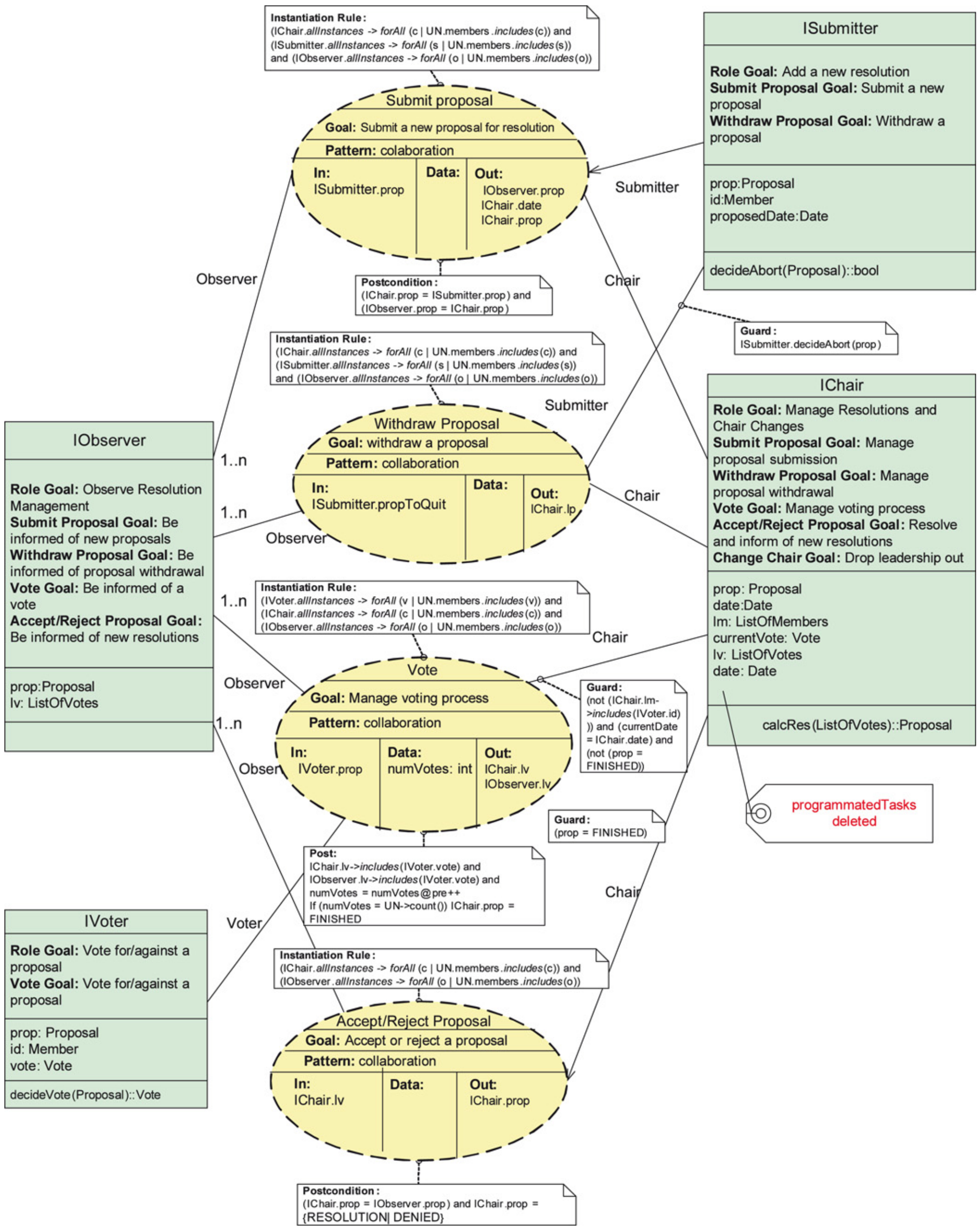

Instantiation Rule:
(IChair.allInstances -> forAll (c | UN.members.includes(c)) and
(ISubmitter.allInstances -> forAll (s | UN.members.includes(s))
and (IObserver.allInstances -> forAll (o | UN.members.includes(o))
Submit proposal
Goal: Submit a new proposal for resolution
Pattern: colaboration
In: ISubmitter.prop
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
Chair
Observer
Guard:
ISubmitter.decideAbort (prop)
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
Chair
Observer
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
1..n
1..n
IChair
Role Goal: Manage Resolutions and Chair Changes
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
Change Chair Goal: Drop leadership out
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))
Chair
Observer
Vote
Goal: Manage voting process
Pattern: collaboration
In: IVoter.prop
Data: numVotes: int
Out: IChair.lv IObserver.lv
Observer
Guard:
(not (IChair.lm->includes(IVoter.id))) and (currentDate = IChair.date) and (not (prop = FINISHED))
Guard:
(prop = FINISHED)
programmatedTasks deleted
Post:
IChair.lv->includes(IVoter.vote) and
IObserver.lv->includes(IVoter.vote) and
numVotes = numVotes@pre++
If (numVotes = UN->count()) IChair.prop = FINISHED
Chair
Voter
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))
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. 13. Role model without change chair.

the roles from the concrete agents that play them, these changes can be easily deployed in the software architecture just changing the assignation of roles to agents, supported by the implementation platform.

In addition, we can observe a set of mRIs that represent the joint processes that are performed by the agents that play these roles. The mRIs we found are *submit*, that represents the submission of a new proposal; *vote*, that shows how an agent playing the role *voter* produces its vote; *withdraw*, that shows the process of quitting a proposal for resolution; *accept/reject*, that represents the process of counting the votes, decide if the resolution is accepted or not, and inform all the *observers*; finally, the *change chair* mRI, that represents how the tasks of an old chair are transferred to a new chair.

The decomposition is simpler than composition. When the features to be eliminated are orthogonal to the rest in the product, we only have to delete the corresponding roles, and their mRIs, from the agents that are playing them.

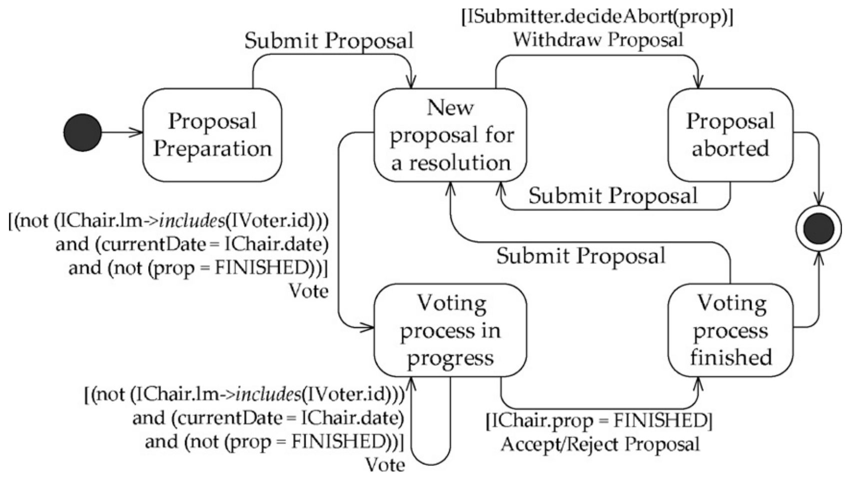

[ISubmitter.decideAbort(prop)]
Withdraw Proposal
Submit Proposal
Proposal Preparation
New proposal for a resolution
Proposal aborted
Submit Proposal
Submit Proposal
[(not (IChair.lm->*includes*(IVoter.id)))
and (currentDate = IChair.date)
and (not (prop = FINISHED))]
Vote
Voting process in progress
Voting process finished
[(not (IChair.lm->*includes*(IVoter.id)))
and (currentDate = IChair.date)
and (not (prop = FINISHED))]
Vote
[IChair.prop = FINISHED]
Accept/Reject Proposal

Fig. 14. Role model plan without change chair.

In the case where the role model of the feature is dependent on others, we have to delete the elements of role models and eliminate all the interactions that refer to them. Given that, in the software architecture we have described, the implementation platform supports the role concept and its changes at run time, the above-mentioned changes can be made easily with a lower impact on the system.

However, features may appear whose role models involve a dependency. 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. This is done by deleting from the interface of these roles the information used by the mRIs to be deleted.

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 executes 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.

Regarding the UN case study, if a war is happening and changing the chair is prohibited we must decompose it from the current product. This leads us to start from the model presented previously to, using the decomposition of role models, remove all the elements related to the change chair feature from both the role model and the plan model.

As can be observed, to delete the *change chair* mRI we have to remove all the roles related to it. The *NewChair* role is not further related to other roles or mRIs, so that they can be deleted.

However, the *Chair* role is involved in the *change chair* feature and in the issue resolution; thus it is dependent on the feature *issue resolution*. Given that, we must decompose it to obtain a new role that does not take into account the change chair feature. The mRI change chair uses the information *programmed tasks* of the *Chair* role as input, and the information *future tasks* of the *NewChair* role as output.

Thus, we have just to delete the information *programmed tasks* from the role *Chair*. The resultant role model is shown in Fig. 13.

Regarding the plan model, we can apply the algorithm to constrain a plan to a set of mRIs, presented in [12], obtaining the role model plan of Fig. 14.

8. 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 as we have seen in the human organization case study.

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 for understanding, analyzing and testing 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, 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 architecture approaches.

References

[1] S. Brueckner, H.V.D. Parunak, Resource-aware exploration of the emergent dynamics of simulated systems, in: Proceedings of Autonomous Agents and Multi Agent Systems, AAMAS, 2003, pp. 781–788.

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

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

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

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

[6] E.A. Kendall, Role modeling for agent system analysis, design, and implementation, IEEE Concurrency 8 (2) (2000) 34–41.

[7] B. Liskov, 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, ACM Press, 1993, pp. 16–28.

[8] J. Odell, FIPA, The FIPA agent UML web site, since 2003. Available at www.auml.org.

[9] J. Odell, H.V.D. Parunak, M. Fleischer, The role of roles in designing effective agent organisations, in: A. Garcia, C.L.F.Z.A.O.J. Castro (Eds.), Software Engineering for Large-Scale Multi-Agent Systems, in: LNCS, vol. 2603, Springer-Verlag, Berlin, 2003, pp. 27–28.

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

[11] J. Peña, On improving the modelling of complex acquaintance organisations of agents. A method fragment for the analysis phase. Ph.D. Thesis, University of Seville, 2005.

[12] J. Peña, R. Corchuelo, J.L. Arjona, Towards interaction protocol operations for large multi-agent systems, in: Proceedings of the 2nd Int. Workshop on Formal Approaches to Agent-Based Systems, FAABS 2002, NASA-GSFC, Greenbelt, MD, USA, in: LNAI, vol. 2699, Springer-Verlag, 2002, pp. 79–91.

[13] J. Peña, M.G. Hinchey, A. Ruiz-Cortes, Multiagent system product lines: Challenges and benefits, Communications of the ACM 49 (12) (2006) 82–84.

[14] J. Peña, M.G. Hinchey, A. Ruiz-Cortes, P. Trinidad, Building the core architecture of a NASA multiagent system product line, in: 7th International Workshop on Agent Oriented Software Engineering 2006, Hakodate, Japan, in: LNCS, May 2006 (in press).

[15] K. Pohl, G. Bockle, F. van der Linden, Software Product Line Engineering: Foundations, Principles and Techniques, Springer, September 2005.

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

[17] C. Rouff, M. Hinchey, W. Truszkowski, J. Rash, Experiences applying formal approaches in the development of swarm-based space exploration systems, International Journal on Software Tools for Technology Transfer 8 (6) (2006) 587–603.

[18] C. Rouff, A. Vanderbilt, M. Hinchey, W. Truszkowski, J. Rash, Formal methods for swarm and autonomic systems, in: Proc. 1st International Symposium on Leveraging Applications of Formal Methods, ISoLA, Oct 30–Nov 2, Cyprus, 2004.

[19] C.A. Rouff, W.F. Truszkowski, J.L. Rash, M.G. Hinchey, A survey of formal methods for intelligent swarms, Technical Report TM-2005-212779, NASA Goddard Space Flight Center, Greenbelt, Maryland, 2005.

[20] Y. Smaragdakis, D. Batory, Mixin layers: an object-oriented implementation technique for refinements and collaboration-based designs, ACM Transactions on Software Engineering and Methodology 11 (2) (2002) 215–255.

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