Artif Intell Rev
DOI 10.1007/s10462-016-9521-7

A metamodeling approach for the identification of organizational smells in multi-agent systems: application to ASPECS

Pedro Araujo[1] · **Sebastián Rodríguez**[1] · **Vincent Hilaire**[2]

Abstract Software Quality is one of the most important subjects in the Process Development Software, especially in large and complex systems. Much effort has been devoted to the development of techniques and concepts to improve software quality over the years. We are especially interested on *smells*, which represent anomalies or flaws in the design/code that can have serious consequences in maintenance or future development of the systems. These techniques have a strong development in the Object Oriented paradigm, however, very few studies were conducted in the agent oriented paradigm. In this paper we focus on the detection of design smells applied to multi-agent systems models based on the organizational approach, named *Organizational Design Smells* (ODS). Early and automatic detection of these ODS allows reducing the costs and development times, while increasing the final product's quality. To achieve this objective, validation rules were defined based in the EVL language. The approach is illustrated with two examples, their validation rules, and the refactoring solutions proposed.

Keywords Agent Oriented Software Engineering · Design smells · Validation rules · Organization approach

✉ Pedro Araujo
pedro.araujo@gitia.org

Sebastián Rodríguez
sebastian.rodriguez@gitia.org

Vincent Hilaire
vincent.hilaire@utbm.fr

[1] GITIA, Universidad Tecnológica Nacional - Facultad Regional Tucumán, Rivadavia 1050, San Miguel de Tucumán, Tucumán, Argentina

[2] IRTES-SeT, Université de Technologie Belfort-Montbéliard, Rue Ernest Thierry-Mieg, 90010 Belfort cedex, France

Published online: 20 October 2016

1 Introduction

As stated in Cossentino et al. (2014), "Designing a multi-agent system (MAS) is not an easy task: creating agents, environments, norms, organizations, and making them cooperate in order to solve a collective task is both an art and a science". Indeed, the underlying features of MAS, namely openness, autonomy, reactivity, interactions with other agents and pro-activity, give way to the absence of centralized control on the whole system. This issue has given birth to the prolific field of Agent Oriented Software Engineering (AOSE). There were many contributions of AOSE to the analysis and design of MAS problems. Among these contributions, one can cite definitions of specific MAS concepts, methodologies and software tools that support analysis and design called Computer Assisted Software Engineering (CASE) in classical Software Engineering. The idea underlying these tools is to support the analyst/designer by providing automatic or semi-automatic services during the analysis/design phases. The contribution of this paper consists in defining an approach and a software tool for the identification of the so-called *smells* within MAS analysis and design.

Among the AOSE community research works, many efforts have been done in order to propose concepts and methodology for MAS engineering. Among these methodologies, one can cite, ADELFE (Picard and Gleizes 2004), ASPECS (Cossentino et al. 2010), Ingenias (Pavón and Gómez-Sanz 2003), GAIA (Zambonelli et al. 2003), ROMAS (Garcia et al. 2015), O-MaSE (DeLoach and Garcia-Ojeda 2010), PASSI (Cossentino and Potts 2002), Prometheus (Padgham and Winikoff 2003), ROADMAP (Juan et al. 2002) and TROPOS (Bresciani et al. 2004). As a result of a standardization effort, these methodologies are based on different metamodels, and each one of these metamodels lists its corresponding concepts and their relationships. One of the advantages of a metamodeling approach is that it allows using the numerous results from the Model Driven Engineering field.

The approach proposed in this paper is based on Model Driven techniques for a specific methodology, namely ASPECS (Cossentino et al. 2010). We use a companion CASE tool, named Janeiro Studio (Araujo and Rodriguez 2013), which was conceived as multiplatform, free distributed, and open-source, and that aims at supporting analysts and designers using ASPECS. It is an EMF-Based tool that offers a modeling framework and a validation module and which is implemented using different Eclipse frameworks such as: (i) RCP (McAffer et al. 2010), to be used in a wide range of end-user applications facilitating the development of these through the intensive use of components' reusing; (ii) EMF (Budinsky et al. 2003), that allows defining a model of structured data called Ecore; and finally (iii) GMF,[1] used to build editors that allow manipulating the instances of Ecore (or part of them). That is, it permits defining the graphic metaphor of each of the concepts defined in a metamodel.

Janeiro Studio provides two kinds of support. Firstly, it eases the graphical representation of the diagrams defined by the ASPECS notation and used within ASPECS activities. The idea is to help the modeler to produce syntactically correct diagrams with respect to the restrictions defined by ASPECS. Model Driven techniques allow the deployment of graphical editors, and the manipulation and transformation facilities due to the use of model elements as first class entities.

Secondly, Janeiro Studio provides guidance and advice concerning the produced diagrams. Indeed, even if a diagram is syntactically correct, the analyst/designer cannot be sure of its inherent quality, for example, in terms of coherence, efficiency, robustness, among others. The presented approach for this second point is inspired by the works that took place some decades ago for the object-oriented paradigm in order to ease project development and quality

[1] Graphical Modeling Framework, http://www.eclipse.org/modeling/gmp.

management. Different techniques and concepts have been proposed to tackle these issues. Among them, we find design patterns, micro-patterns, design and codification standards, good practices, and finally code smells. All the techniques mentioned above are widely used for the object-oriented paradigm. However, if someone is interested in agents and multi-agent systems, these techniques and their tools cannot be reused directly since MAS are built on different abstractions and technologies. So, it is imperative to refine or re-define many of these techniques for MAS development.

The starting point of the presented work is the ASPECS metamodel, named CRIO (Rodriguez et al. 2007), and more specifically the part that concerns the analysis in terms of organizational concepts. The contribution of this work is twofold. On the one hand, based on the ASPECS metamodel an EMF model is proposed, which is the main core of Janeiro. The goal of the latter metamodel is to enable the use of the Eclipse modeling suite that allows deploying graphical editors and manipulating operations on ASPECS models. On the other hand, models' instances of this metamodel are studied to define, with the help of the EVL[2] language, rules that detect potential situations that lead to organizational smells.

This paper is structured as follows: Section 2 introduces the concept of smell, the model driven engineering background used in the rest of the paper and the organizational approach (CRIO) used for this paper; Sect. 3 defines the corresponding metamodels; In Sect. 4 the proposed assistance approach is detailed; Sect. 5 illustrates the previously described approach with examples; Sect. 6 presents some related works; and Sect. 7 presents lines of future works and the conclusions.

2 Background

2.1 Smells

Generally speaking, a *smell* represents a potential problem in the system under study. The implementation level (or *code smell*) has traditionally taken a great deal of attention. However, what we propose is to abstract the concept or principles and use them at the analysis and design stages. Indeed, many *code smells* are inherited from design issues (Khomh et al. 2009; Moha et al. 2010). In this sense, a design smell represents a poor or bad design that does not fulfill the standards and/or good practices of the field.

Code Smells were firstly introduced by Fowler et al. (1999). The authors presented a list of 22 low-level code smells with a description about how to identify them. This novel technique was widely accepted by the software industry and it was object of further research in the years that followed.

A code smell indicates a poor codification structure or a bad design that does not fulfill the modeling or codification standards and that can have negative impacts in the future, particularly in the maintainability stage. One of such impacts causes a slow and ineffective development environment thus increasing the effort and the costs to add new functionalities or make a corrective maintenance. In addition to this, the identification of several code smells could be an indicator of the need to start a refactoring process. Refactoring means to introduce modifications in the internal structure without altering the behavior exhibited by the system, making it easier to understand and extend (Fowler et al. 1999).

God Class and *Feature envy* are two classical smells defined by Fowler and Beck. The first term "refers to those classes that tend to centralize the intelligence of the system. An

[2] Epsilon Validation Language, www.eclipse.org/epsilon/doc/evl/.

instance of a God Class performs most of the work, delegating only minor details to a set of trivial classes and using the data from other classes" (Lanza and Marinescu 2006). This type of smells tend to be very large blocks of code with a high degree of responsibilities in the system, which may make the system more difficult to understand and maintain.

The *Feature Envy* term is to indicate if we are in the presence of a method in a class that uses primarily data and methods from another class to perform its work. This is due to either a lack of a clear separation of concerns or to an abstraction fault.

The mentioned smells can be related to the design of the system itself. Therefore, it would be better to detect them at the early stages of the software process. It is important to mention that not all *code smells* can be derived from design smells. However, we propose to tackle those that are related to analysis and design earlier in the process.

There are two main types of techniques that can be used to detect code smells:

- In the first one proposed by Fagan (1976), the verification of the code is performed manually by a process called *code inspection*. This process carries issues related to planning, measurement, functions control, etc. People in charge of this task must be meticulous and able to spend hours at the computer reading the code line by line in search for defects. Besides this, they must have thorough knowledge of the norms and their flexibility to adapt to different aspects of the projects. The process involves certain difficulties as it is error-prone, time-expensive, non-repeatable and non-scalable (Mantyla et al. 2004).
- The second one is based on automatic tools (Carneiro et al. 2010) and is the most used one nowadays. It consists on formalizing each smell in semi-formal or formal rules using a validation language. Generally, automatic tools are modules or extensions of applications that not only allow managing the rules, but also allow automatically executing a list of rules on the models in a sequential manner. As (Emden and Moonen 2002) points out, this process is not error-free since the list of rules that must be applied to the system of interest is never complete. Every new project requires a new set of rules. Besides, the definition of code smells is a subjective process based on the expertise and opinions of the developers involved.

2.2 Validation language

While defining smells for agent design is a significant step forward, our intention is to provide the means to automatically detect these issues during the process of design. To this end, a language should be used to continuously apply the validations rules and notify the designer of possible problems.

The language selected to validate models is the Epsilon Validation Language (EVL from now on) (Rose et al. 2008). We chose EVL instead of OCL (Object Constraint Language) (OCL 2012), which is the facto and most used language for model validation, since it has several practical advantages, some of which are:

- It allows defining constraints that can be dependent on other ones defined by users.
- It supports statements sequence. Thus, it permits a decomposition of complex queries into simpler ones, which promotes the modularization increasing readability and maintainability.
- It provides programming constructs (*while* and *for* loops, statement sequencing, variables etc.) as well as support for handy first-order logic OCL functions (select, reject, collect etc.).
- It provides support for user interaction.

- It allows creating and calling methods of Java objects.
- It supports the users' input and output operations.
- It enables to dynamically repair the inconsistencies found.

Besides the ones already mentioned, there are three mechanisms we consider of paramount importance for our work and which were determining factors in our preference for this language. The first deals with the possibility of personalizing the messages when an invariant was not satisfied, and it is a most significant characteristic since the feedback to the users in OCL is limited to show only the name of the invariant. Another important characteristic is that it allows the differentiation of feedback types. Two types are possible. The first is `Error`; which, as OCL, indicates an invariant violation stopping the validation's execution and indicating that this critical problem has to be immediately tackled. This is, according to us, a rather extreme characteristic for the purpose of our work. Yet, in EVL there is another type of feedback, `Warning`, which allows continuing with the system validation's execution despite the fact that the constraint is no longer true. We use this mechanism simply to send warnings that there is a problem in Janeiro's diagrams, thus letting the designer choose whether to solve it or not.

Third, as we want to look for structural defects in the diagrams taking more than just one perspective, EVL allows us to express inter-model restrictions through which rules that cover multiple models can be defined.

Finally, the combination of EVL, EMF Models, and the Eclipse RCP framework provide a high expressiveness which enables the designer to formulate additional properties (validations) that cannot be expressed through the graphic notation.

2.3 Organizational metamodel

The organizational approach promotes a new way to deal with complex problems. This approach consists in focusing on the organizational structure of a system-to-be. This organizational structure is defined in terms of abstract behaviors, positions, norms and abstract interactions between these behaviors. These organizational structures allow decomposing a system into smaller parts, offering several interaction contexts specific to a set of goals or objectives the system has to fulfill.

As stated in Ferber et al. (2004), there are two levels: organizational and agent level. The organizational or social level (called "what"), where the dynamic and structural aspects of a MAS organization can be observed. This level describes the expected relation and the activity pattern that should occur at the agent level. Furthermore, it is common to find concepts such as roles, groups, communities, tasks, and interactions. The agent level (named "how") describes the agent's behavior. In other words, it details an agent's internal architecture, its mental states, describing beliefs, desires, intentions and goals, and if it is reactive or intentional.

Furthermore, adopting the organizational approach permits designers to deal with problems through two possible strategies: vertical and horizontal decompositions. The vertical decomposition allows the behavior that represents the organization to be decomposed in a set of sub-organizations of lower abstraction level. The horizontal one models the existing interaction between the entities present at the same level of abstraction, which is necessary to reach the required objectives.

Although different Organizational models have been proposed over the years, the present work aims at providing a modeling approach that will assist MAS analysts while applying the ASPECS Methodology, which was considered in Isern et al. (2011) to be one of the methodologies with the best organizational structure support.

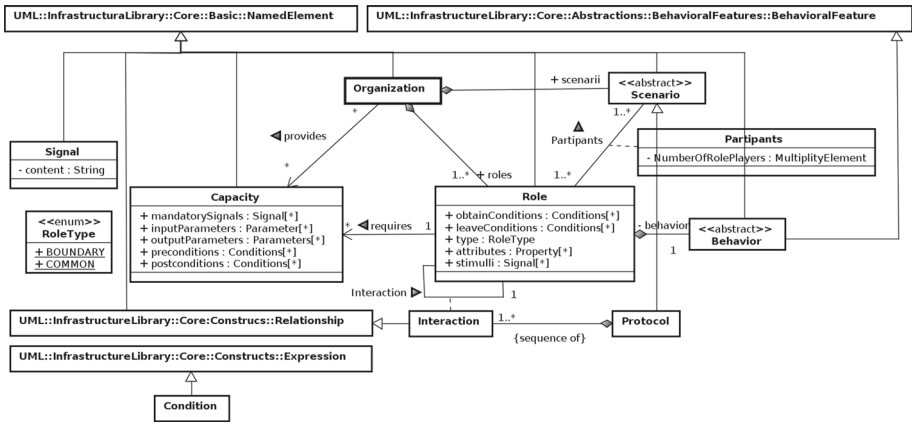

Fig. 1 CRIO metamodel

ASPECS uses the CRIO Metamodel as support for describing the systems model. Therefore, this work is based on this metamodel for the analysis and validation of MAS Models.

Multiple methodologies for MAS have been proposed by the community over the years, each one with its own advantages and features. Many of them have embraced an organizational approach, like O-MaSE or ASPECS. We base this work on the ASPECS methodology. A detailed description of this methodology can be found in Cossentino et al. (2010), and a comparison of different methodologies can be found in Cossentino et al. (2014).

As mentioned before, ASPECS is based on the CRIO metamodel in order to define the models of the system. Every activity of the methodology extends, transforms or generates new elements expressed into the model of the system based on this metamodel. Therefore, a validation approach should concentrate on defining it in a way that can be clearly interpreted and analyzed by automated tools. CRIO is an extension of the UML metamodel. This choice was made for two reasons. First, the initial notation proposed within ASPECS was based on UML profiles. Second, the extension of the UML metamodel allows the reuse of existing frameworks and tools developed within the Model Driven Engineering community.

In this section, we briefly introduce CRIO, whose name is an acronym formed by its four principal concepts *Capacity*, *Role*, *Interaction* and *Organization*, see Fig. 1.

A *Capacity* is a description of a know-how/service. In other words, it is the specification of a transformation of a part of the designed system or its environment. It is a high-level abstraction that promotes reusability and modularity and in this sense can be considered as a basic design component. In addition to this, a *Capacity* allows the definition of a role without making assumptions on the architecture of the agent that may play it.

Role is an expected behavior (a set of role tasks ordered by a plan) and a set of rights and obligations in the organization context. The goal of each Role is to contribute to the fulfillment of (a part of) the requirements of the organization within which it is defined. There are two types of roles: *Common* and *Boundary* roles. *Common* means that it is a role located inside the designed system and interacting with either *Common* or *Boundary* roles. *Boundary* represents a role located at the boundary between the system and its outside and it is responsible for the interaction happening at this border.

Interaction is a dynamic, not a priori known sequence of events (a specification of some occurrence that may potentially trigger effects on the system) exchanged among roles, or between roles and entities outside the agent system to be designed.

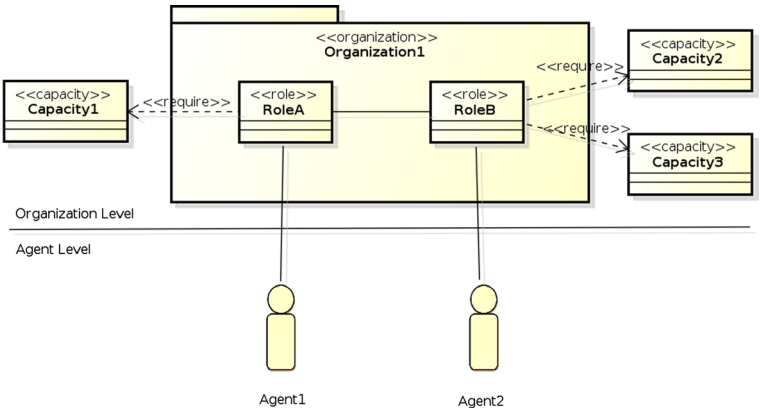

Fig. 2 Simple organization example

Organization is defined by a collection of roles that take part in systematic institutionalized patterns of interactions with other roles in a common context.

This metamodel design is composed of the graphical representations of all the different components that this metamodel holds. The formalism used is strongly inspired by the UML graphical notation. The organization is defined as a package stereotyped with ≪organization≫. Just under this stereotype, a label corresponding to the name of this organization is also written. To be conceptually valid, an organization needs to contain at least one role that interacts with itself; this means that the organization's package needs to contain a role that is linked with itself (source and target connected to that same role). A role is represented by a class box with the ≪role≫ stereotype, drawn inside an organization. The type of the role is written under the stereotype in a label within chevron (≪≫) and a label corresponding to the name of the role is written under the type. A role that communicates with another role (or itself) is linked with a UML association, assuming that the association can be n-ary as defined in the UML Infrastructure Specification (uml 2011). Each association is the graphical representation of a protocol, and the roles are the participants to this protocol. Each end of a protocol association may specify the number of players who are involved in the protocol for the connected role. In Fig. 2 we define an organization, namely *Organization1*, which has two roles. One of them named *RoleA* and the other *RoleB*. *RoleA* requires *Capacity1* and *RoleB* requires both *Capacity2* and *Capacity3*. Thus, in order to play, *RoleA* requires an agent to have the capacity implementation *Capacity1*. In the same way, *RoleB* requires both capacities implementations *Capacity2* and *Capacity3*. A *Capacity* is also represented by a class box with the ≪capacity≫ stereotype and is drawn outside but near the organization box. A label corresponding to the name of the *Capacity* is written under the stereotype. A capacity may be linked to one or more role boxes inside the organization package. A role that requires a capacity is linked to the graphical representation of that capacity (described later) by a straight dashed arrow which has a sterotype ≪require≫ on it.

3 EMF metamodeling of CRIO

Eclipse Modeling Framework (EMF) is an important part of a large family of projects called Eclipse Modeling Project (EMP).[3] It is a library that enables experts to build their own

[3] Eclipse Modeling Project, www.eclipse.org/modeling/.

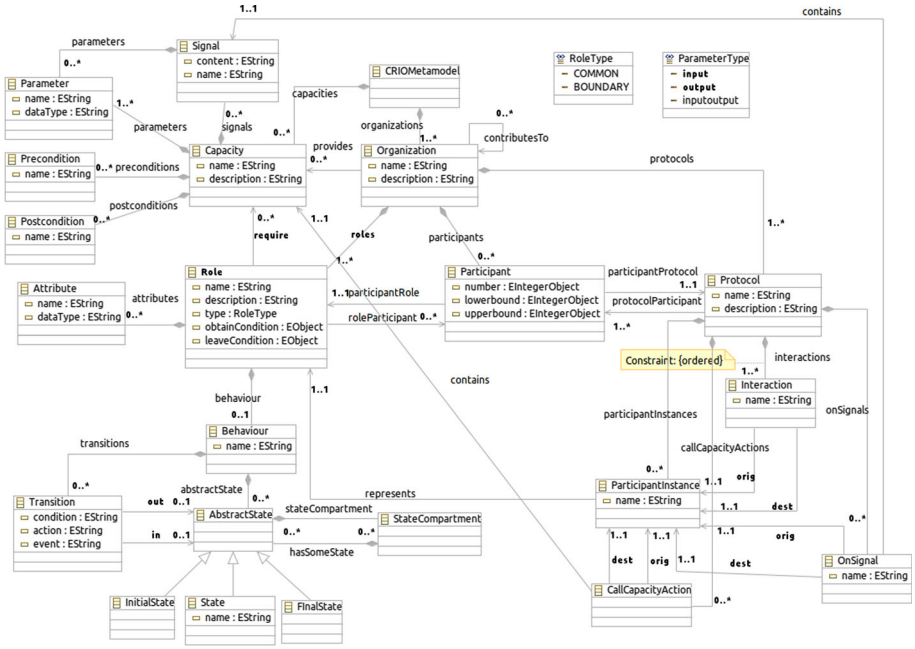

Fig. 3 CRIO metamodel adaptation using EMF

tools based on structural data metamodels named Ecore. An Ecore simplifies the definition of a metamodel facilitating the required programming for the implementation of a Domain Specific Modeling. Moreover, the use of EMF permits modeling hierarchies where a model in a higher level is the metamodel of the model beneath it.

The concepts used in this framework are simpler than those defined by the Object Management Group (OMG).[4] This is due to the fact that the intention is to rapidly obtain an executable code. The principal concepts are *EPackage* (Package), *Eclass* (Class), *EDataType* (Data type), *EEnum* (Enumeration), and *EObject* (Object). There is also the possibility of representing, through the concept of *EReference*, association and aggregation relations. In Fig. 3, the adaptation of CRIO on EMF is shown. This metamodel is named *problemMetamodel*. It is the metamodel for the graphical representation of the abstract elements of the organizational approach. Based on this metamodel, three different views were created. The first is the organizational view, which serves to capture the static and structural aspects of the model. The other two, the interaction view and behavior view, represent the dynamic aspects, like the interaction between roles as the defined behavior for each role. With these three views and the metamodel one can support the three "classical" aspects present in MAS: structural, interactional, and behavioral.

The remaining of this section will be devoted to explain each view, including their goals and the metamodel elements that compose them.

3.1 Organization view

The main goal of the organizational view is to identify, for each requirement, a global behavior embodied in an organization. With the organization, it is possible to capture the static and

[4] Object Management Group, www.omg.org.

structural aspects of the organizational approach. This view takes as input what has been done in the Domain Requirements Description activity, which is defined as one of the steps of ASPECS. From the analysis of the use cases and the ontology structure, the view can identify a group of organizations, each of which is associated with at least one use case.

In our adaptation of the metamodel, *CRIOMetamodel* is the root element of the organization diagram. It contains *Organization* and *Capacity*. *Organization* has the following attributes: (i) *name*, a must for every organization to have, and optionally (ii) a *description*, which is a brief description of the purpose of the organization. Besides, the *Organization* concept is composed by *Role*, *Protocol*, and *Participant*.

In *Role*, both name and description attributes have the same meaning as in *Organization*. Apart from this, *Role* has an attribute of type *RoleType*, an enumeration that can be either *Common* or *Boundary*, concepts which have already been mentioned. Furthermore, the *Role* contains two additional concepts (*obtainCondition* and *leaveCondition*) that will be represented as sections in the graphical editor. For each role, it is possible to define several *Attributes* (specifying name and datatype) and just one behavior, which will be explained later in this article.

A *Protocol* describes how these roles are interacting by following a sequence of interactions. The organization diagram illustrates these high-level interactions by drawing a circle which represents an association between two or more roles. *Participant* is used to link role and protocols and also permits to specify the number of roleplayers that can play the role.

The *Capacity* concept embraces the idea of the agent's possibility of completing a task or reaching a goal. In other words, it represents the competences or skills that an agent must have in order to play the role. It was created to promote reusability and modularity. It is composed by four elements, each of which represent a section in the graphical editor. In the first place, *Parameter* represents the capacity's input values necessary for the services that implement the capacity. Secondly, *Signal* represents an event fired by the roleplayer notifying the finalization of a task. These signals have a name and, possibly, a collection of return values. Lastly, *pre-* and *post- conditions* are logical constraints defined for input and output values, respectively.

3.2 Interaction view

The interaction view makes it possible to capture one of the dynamic aspects of the organizational approach, particularly that of the interaction between roles. This view's challenge is to describe the information exchange sequence among roles involved in each scenario. These scenarios are deduced from a group of textual descriptions: of the system's scenario, of the outputs of the organization's identification activities, and of the roles and their interactions.

Back to our metamodel, it is important to mention the *Protocol* concept is the root element of the Interaction view. The protocol itself is composed of *Interaction*, *ParticipantInstance*, *CallCapacityAction* and *onSignal*. As regards Interaction, there must be at least one interaction between roles for each diagram. All interactions must have a name and two other attributes. These attributes, *orig* and *dest*, represent respectively the source and target of the current interaction. The *ParticipantInstance* concept represents the role in the interaction view. It has two attributes. The first is *name* of the EString type. The second is the name of the role it stands for in the organizational view, which is done through the concept of *represents* shown in the metamodel. *ParticipantInstance* exists because no concept in the metamodel, in this case Role, can be used in two different views, even if they have different graphical metaphors in each diagram.

CallCapacityAction represents an asynchronous self-message. It has an attribute of the *Capacity* type named *contains* and it is the role's possibility to invoke and interact with its player. The last concept, *onSignal*, represents an event fired by the roleplayer notifying the finalization of a task. One of its attributes is *contains* of the *Signal* type. Both *CallCapacityAction* and *onSignal* represent the role's self-messages. For this reason, there is a condition that verifies that the source and target belong to the same role.

3.3 Behavior view

As it has been already mentioned, the role concept allows meeting the requirements of the organization within which it is defined. It is for this reason that the behavior view allows representing the dynamics of the instances. The role's behavior is thus described in terms of states and transitions.

All roles present in the model must have a behavior diagram attached to them. The authors of the CRIO metamodel recommend the use of a state-transition diagram. As Fig. 3 shows, the root element of the diagram is the *Behavior* concept, which consists of *AbstractState* and *Transition*. The first concept is a generalization of the pseudo-states: *Initial* and *Final* as of the regular states. The concept of *StateCompartment*, related with *AbstractState*, permits the diagramming of nested states. That is, the view allows the designer to expand a state by adding more details to depict complex states.

4 Modelling assistance approach

Nowadays, models validation is a mandatory characteristic in the process of software development since it allows designers to verify that the model is conceptually valid. Two levels of validation are possible: syntactically and semantically. The detection of these problems force restructuring the system to improve it and this approach supports that decision. The remainder of this section shows the mentioned levels.

4.1 Syntax validation

Syntax validation rules are critical to achieve the user model specifications at an early stage in the software development process. This mechanism enables designers to tackle the problems immediately.

Currently, Janeiro Studio provides a set of syntax validation rules. For example, there are rules that establish that capacity, organization, protocol, and role concepts must have a name. In the same way, other rules do not allow name duplication in concepts of the same type. There are other defined rules, but due to space concerns, they are not presented in this article. In fact, only two simple validation rules are presented as example.

The first one (Listing 1) detects the presence of at least one reference in all and each of the capacities.

Listing 1 EVL Constraint for Isolated Capacity.

```
1 context Capacity {
2   constraint IsolatedCapacity {
3     check {
4             var capacityList : Set( Capacity );
5             for ( r in Role.allInstances ) {
6               capacityList.addAll( r.require );
7             }
```

```
        return capacityList.includes( self );
      }
      message : 'All capacities must be referenced'
    }
}
```

The second one (Listing 2) detects duplicated parameters in the definition of the capacity concept.

Listing 2 EVL Constraint for Duplicated Parameters.

```
context Capacity {
  constraint HasADuplicatedParameter {
    check {
      var parameterList : Set(Parameter);
        parameterList.addAll(self.parameters);
        var duplicatedParameter : Boolean = false;
        while ( (not parameterList.isEmpty()) and duplicatedParameter = false ) {
          var parameter : Parameter = parameterList.first();
          parameterList.remove( parameter );
          if (parameterList->exists(n | n.name = parameter.name and n.dataType = parameter.dataType)){
            duplicatedParameter = true;
          }
       }
       return not duplicatedParameter;
    }
    message : 'The Capacity ' + self.name + ' has a duplicated parameter'
  }
}
```

4.2 Semantic validation: organizational design smells

Smells theory for the object oriented paradigm has a wide variety of approaches including the study of techniques for the detection of Fowler's proposals, the identification of the most recurring smell, the hardest smell to detect, the determination of the maintainability techniques, the refactoring techniques and/or strategies, among others. However, we consider that many of the works can be applied to the MAS paradigm (with certain redefinitions due to the concepts underlying both paradigms) since most of them apply their proposed techniques on the final product of a traditional process of software development, which is the system's code (or part of it). This conclusion arises after analyzing the most outstanding multi-agent system frameworks in the literature, like JADE (Bellifemine et al. 2001), MadKit (Gutknecht and Ferber 2000a, b), Jack (Busetta et al. 1999) or Janus (version 1.0) (Galland et al. 2010). These are APIs that extend from an object oriented language, usually Java, and that do not provide first-class abstractions at language level.

Besides our interest in the social/organizational approach and its advantages (presented in Sect. 2.3), our work is focused on a particular type of systems: the complex systems. On these, a large number of components are involved, which are mostly autonomous entities that can interact between themselves to achieve global objectives. Because of this, it is not worth considering using techniques and/or concepts of agile methodologies, in which person-to-person communication is prioritized over generating the documentation associated with the system. In other words, the complex systems reinforce the need to do a good modeling before producing any code, so as to have a clearer understanding of the problem to be dealt with. Considering that, in some cases, design smells are symptoms of code smells, we believe that certain defects can be found in the analysis and design stages, thus avoiding transferring them into the code.

In this work, we propose to analyze the organizational design of an agent-based system to find structural and dynamic indicators of ill-designed parts of the system that we called *Organization Design Smells* (ODS from now on). They may indicate poor or bad design that does not fulfill the modeling standards or/and good practices. This situation can have a negative impact in the future either in the maintainability stage or in the system's execution performance.

As Fowler and Beck's code smells, ODS are intended to analyze the structure and relations between different modeling concepts as independently as possible from the final goal (eg. task distribution, memento, resource management, etc). Most organizational metamodels are composed of four basic concepts, which are named differently according to the metamodel in which they are used, but the initial idea is the same. Although, the notion of agent is represented and used in all the metamodels, it will not be discussed here as we are focused on the organization at dimension part. The basic concepts are:

- *Organization/Group*: The concept that represents any set of agents that interact among each other towards an objective. They act in order to reach a specific goal that can be global (common) or local (private). This concept is used under different names in several metamodels, for example: *Groups* in AGR (Ferber et al. 2004); *Society* in SODA (Omicini 2000); *Organizational Unit* in VOM (Criado et al. 2010) and ROMAS (Garcia et al. 2015); *Organization* in MOCA (Ferber and Gutknecht 1998), CRIO (Rodriguez et al. 2007), GAIA (Zambonelli et al. 2003), O-MaSE (Garcia-Ojeda et al. 2008), MOISE+ (Hübner et al. 2002) and Ingenias (Pavón and Gómez-Sanz 2003).
- *Role/Behavior*: The abstract representation of a desired or expected behavior for a specific agent. It is named *Role* in CRIO, AGR, GAIA, VOM, O-MaSE, PASSI MMM, ROMAS, Ingenias, SODA, MOISE+, and ELDA MMM (Fortino and Russo 2012), *Behaviour* in ADELFE (Picard and Gleizes 2004).
- *Interaction*: It represents the communication mechanisms used by agents to share specific information in order to achieve their goals. This information can be transmitted via different forms: events, signals, structured messages, or ACL messages as defined in FIPA. This concept is known as *Interaction* in ROMAS, SODA, and Ingenias; *Event* in VOM and ELDA MMM; *Protocol* in CRIO, GAIA, and O-MaSE; *Communication* in PASSI MMM; *Dependency* in TROPOS; *Role Interaction* in MOISE+.
- *Skill/capacity*: The concept that embraces the idea of what an agent is capable to do or what services it can provide, to complete a task or reach a goal. It can be found under different names: *Skills* in ADELFE, *Capacity* in CRIO; *Service* in GAIA and PASSI MMM; *Service Profile* in VOM and ROMAS, *Capability* in O-MaSE and TROPOS.

Moreover, we propose the concepts, aspects, and/or characteristics of the model on which designers must focus because these are where some indicators of the presence of problems can be found. They should also evaluate the possibility of formalizing them as smells. A subset of these smells can be grouped into building blocks and considered as a category of ODS. We define four categories, namely *Organization Smells*, *Interaction Smells*, *Behavior Smells*, and finally *Skill Smells*.

4.2.1 Organization smells

The organizational approach provides a way of decomposing a system into groups in which all agents cooperate in order to achieve a goal/task. It also permits a description of the structure and the interaction that takes place in MAS. Each organizational group constitutes a common context, which may consist of shared knowledge, common language, social rules, etc.

An organization should have well-defined objectives. Yet, when defining an organization it is possible to find flawed designs, which can lead to different problems:

- Multiple contexts. It can be observed that in the same organization there are several contexts. In other words, the organization can be represented by different ontologies, divergent communication mechanisms, etc. This means that, with a detailed analysis, multiple organizations can be identified within a single one.
- Divergent/conflicting objectives. Roles that form an organization must focus on common or coherent objectives. Nonetheless, in some cases there are antagonistic objectives whose presence can cause problems, making the introduction of modifications in the organization a difficult task.

4.2.2 Interaction smells

In order to achieve a collective task in an organization, the interaction among agents is mandatory. Although every member is free to interact with each other, there exists communication patterns in the organization that represent the most frequent interactions among the agents. These interactions are sequences of events or actions whose consequences affect role's behaviors. Also, the interactions' context is provided by the organization.

The second category is *Interaction Smells* and it focuses on everything related to the information exchanges among agents.

- Throughput issues: An organization that has a high message exchange among agents.
- Bottleneck: If the capability to process the messages an agent has is lower than the number of messages it receives, it can lead to a bottleneck situation. This causes the other agents to be blocked, waiting either for a service or to resend the message.
- Message complexity: An exceedingly large message can affect the performance of the implementation.

4.2.3 Behavior smells

As we mentioned before, a role is an expected behavior whose objective is to help meet goals of the organization to which it belongs. A role has a set of skills / capacities associated to it that can be regarded as a notion of competence that an agent must have in order to play it. In this category what we define as *Behavior* represents the problems associated to the definition of the role's behavior. In some cases, the behavior can be excessively complex turning the role into an interactions centralizer which entails a high degree of interaction with the other roles that conform the organization. To avoid this, complex behaviors must be fragmented into simpler ones until reaching a role design with a unique function behavior.

4.2.4 Skills smells

The last category of this taxonomy is related to the *skill*, *service* or *competence* of the agents. It can be seen in the modeling that some agents/roles have skills that are too complex or too difficult to implement. We can also mention the misuse of the concept of skill as an abstraction. By this we mean that there is a tendency to turn some functions into skills or to inaccurately distinguish the skills associated to the agent from those linked with the abstract behavior.

5 Examples

In the following, we present a list of nine smells with a brief description of each of them (Araujo et al. 2015). These have been detected following some of the criteria defined in the previous section, and are applied in multi-agent models based on the CRIO metamodel. They were obtained from various studies of the models that arose from diverse projects and that represent the most common mistakes made by designers.

Bureaucracy role: At times a role receives messages that are not used by it or by any of its associated capacities. It then acts as a centralizer or hub that later distributes information to its final recipients. This has a negative effect on the performance of the system, it not only introduces an unnecessary latency, in which the message recipient has to wait for that unused message, but also burdens the role with the responsibility of resending the message to the appropriate addressee.

Promiscuous role: It describes a role that sends the same message to different roles without using the broadcast communication type. That is, a role which knows all other roles in the organization. This may lead to the emergence of complex behavior components (a statechart plagued with functions for the sending of messages) and a costly and tedious implementation of the system, as well as, a reduction in the role s performance. This last is due to the fact that the role dedicates most of its processing to the sending of the message instead of doing the tasks that were delegated to it within the organization.

Bottleneck situation: It describes a role that receives multiple messages from different roles. This causes complex representations of both the receiver role's behavior and the interaction diagrams between the receiver role and the sender's, making both diagrams hard to read. Besides, this smell could reduce overall system's performance, because when a role receives more messages than it can process, the messages queue could become too long causing other roles' blocking waiting for its services.

Blocked-role situation: It identifies a role with a capacity that needs different messages from different roles. These messages must be present at the same time for the capacity's execution to avoid a possible block of the role.

Selfish-role behavior: It is a role that has a number of capacities that exceed a criterion established by the software architect, for example by having a larger number of capacities than the average per role. This leads to an overload in the representation of the behavior, making it difficult to understand and extend. Besides this, the role can become a service centralizer, making it necessary for most of the organization roles to communicate with it to obtain any service. This could increase the requirements an agent must have in order to play that role (all the capacities required by the role must be present in the agent). In other words, this role acts as a Single Point of Failure, which means that if this part of the organization fails, it will compromise the organization s objectives.

Different context: It can be observed that in the same organization there is not a unique context, it may have different defined ontologies, different communication mechanisms, etc. In other words, it is possible to detect subgroups with strong interactions among the roles within them but little interaction among subgroups. It would be ideal to have organizations with a unique context since it gives a clear separation of concerns.

Conflicting objective: Similar to *Different Context* yet with roles with antagonistic objectives. Detecting these differences is important since in the organizational theory the organizations are generally formed by roles that collaborate with each other to achieve goals that are common to all. It is important to note that the mentioned objectives are not of the agent and are not its subgroups, but the organization's general objectives.

Table 1 Proposed ODS classification

Category	Design smell
Organization smells	*Different Context*, *Conflicting Objective*.
Interaction smells	*Bottleneck Situation*, *Blocked-Role Situation*.
Behavior smells	*Bureaucracy Role*, *Promiscuous Role*, *Selfish-role Behavior*.
Skills smells	*Capacity Abuse*, *Capacity Chain*.

Capacity Abuse: The capacity may have different implementations associated to it. This smell describes those capacities which have only one implementation since the problems they solve have a single solution. There is a tendency of declaring any function or operation, however simple, as a capacity. In the codification there are more lines for the calls to capacities and signals than for the body of such operations.

Capacity chain: This smell tries to identify capacities that are related by their input, as well as by the signals parameters. It is important to note that, in a model, there can be a set of capacities making a chain, in which outputs of one are used as inputs in another. In this situation, the set of capacities can be replaced by only one capacity where the inputs of the first and the outputs of the last are the respective inputs and outputs of the new capacity. In addition, another condition is that these capacities must not be required by any other role. This smell is also related to *Capacity Abuse* in that capacities offering trivial services can be replaced by a single capacity, saving code lines in the calls to capacities and signals. The implementation is thus simplified avoiding an unnecessary modularization of the system.

Table 1 shows a proposed ODS classification, where each category clusters smells with similar characteristics, based on the criteria defined in Sect. 4.2. This type of taxonomy is useful to gain a better understanding of different ODS and the relationships between them, as expressed by Mantyla in Mantyla et al. (2003). This proposed ODS classification is neither exhaustive nor definitive, and it could be enhanced with new ODS or a new categories.

This section presents a set of simple examples to illustrate the proposed concepts. Due to space concerns, the rest of this section focuses on only two smells (*Selfish-Role Behavior* and *Bottleneck Situation*) and provides a validation rule and a possible solution for each example.

5.1 Selfish-role behavior example

This design smell describes how to identify a role in the model that has too many capacities associated with it (as shown in Figs. 4 and 5), which could represent an anomaly in the model. In some cases, this unnecessarily increases the complexity of the role's behavior (modeled as a statechart in CRIO), turning it into a services centralizer within the organization. For the reason we mentioned above this causes that too many roles must communicate with it to obtain any service. This could increase the requirements that an agent must have in order to play that role (all the capacities required by the role must be present in the agent). In other words, this role acts as a Single Point of Failure, which means that if this part of the organization fails, it will compromise the organization's objectives. Therefore an important point is to detect when a role is a *selfish-role*. Based on our experience, we propose different subjective criteria that are useful for detecting a *selfish-role*:

- Drawing an average calculation of the capacities associated with each role and then considering as *selfish-roles* those that widely surpass this average.

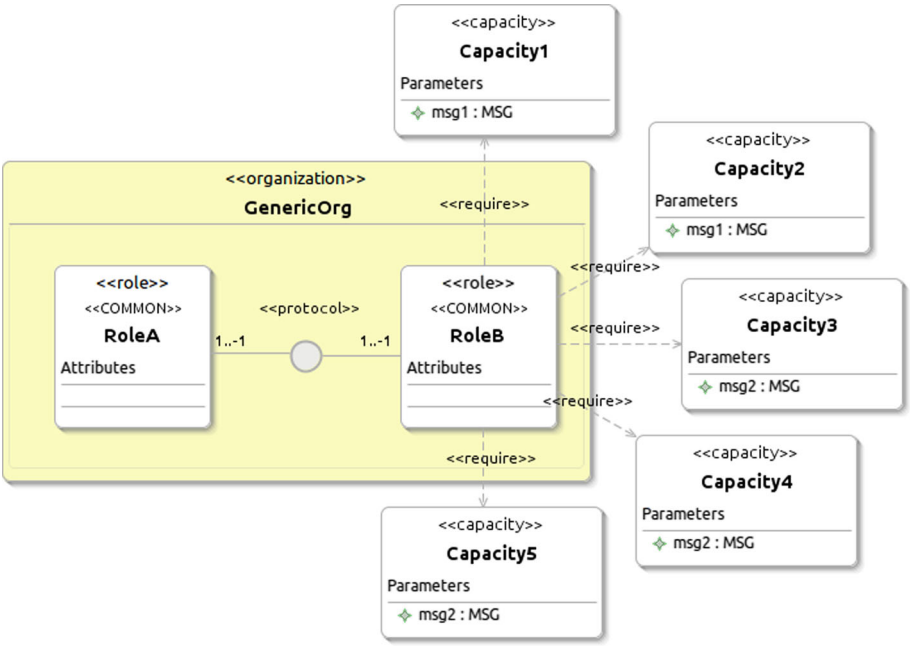

Fig. 4 Behavior smells: *Selfish-Role* requires an excessive amount of capacities

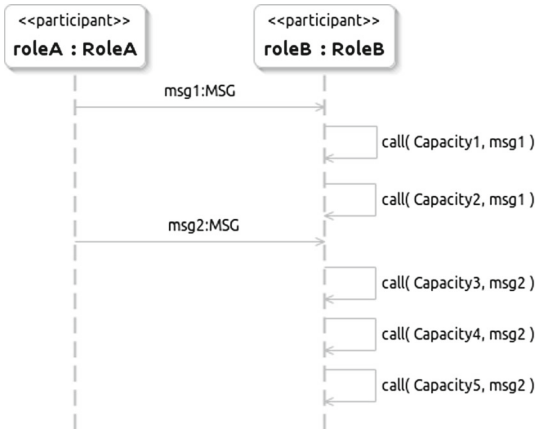

Fig. 5 Behavior smells: *Selfish-Role*— interaction diagram

- Detecting a role that has capacities that can represent disjoint subsets. This conclusion can be reached because there is no communication among disjoint capacities.
- Finding Capacities that require a lot of processing.

How to identify it?. Methodological steps.

Step 1. Determine the average number of role capacities. (Another criteria could be used).
Step 2. Consider as *Selfish-role* those roles that widely surpass the average.

Listing 3 EVL Constraint for Selfish-Role.

```
1  operation Organization numberOfRoles() : Integer {
2      return self.roles.size();
3  }
4
5  operation Role numberOfCapacities() : Integer {
6      return self.require.size();
7  }
8
9  operation Organization averageCapacitiesPerRole() : Real {
10     var nbCapacities : Real = 0.0;
11     for (role in self.roles) {
12         nbCapacities = nbCapacities + role.numberOfCapacities();
13     }
14     return nbCapacities / self.numberOfRoles();
15
16 }
17
18 operation CRIOMetamodel averageCapacitiesPerOrganization() : Real {
19     var average : Real = 0.0;
20     for (organization in self.organizations){
21         average = average + organization.averageCapacitiesPerRole();
22     }
23     return average / self.organizations.size();
24 }
25
26 context Role {
27     critique selfishRole {
28         check : self.require.size() < self.eContainer().eContainer().averageCapacitiesPerOrganization()
29         message : "This role could potentially be a Selfish Role"
30     }
31 }
```

Listing 3 shows the proposed validation rule that serves to detect the smells described in the previous paragraph. This validation rule has four operations defined. The operations, *numberOfRole()* (see in line 1) and *numberOfCapacities()* (line 5), both are quite similar. The first one returns the total number of roles associated to an organization, meanwhile the second one returns the total number of capacities associated to each role.

Line 9 begins the definition of *averageCapacitiesPerRole()* operation. The purpose of this operation is to calculate for each organization the number of capacities per role. The `for` block between lines 11 and 13 iterates through all the defined roles in the organization accumulating them in the *nbCapacities* variable. Finally, the *nbCapacities* variable is averaged with the number of roles calculated for that organization.

Line 18 begins with the definition of the *averageCapacitiesPerOrganization()* operation. The main objective is to compute the average of capacities present in all the organizations involved in the model.

In line 26 the rule's general context is defined, in which, through the `check` function, it is evaluated if the number of capacities related to each role is less than the average number of capacities per role. For this operation, role instances uses *eContainer()* function, which is provided by EMF framework and return the object's parent element. In this line, two chained *eContainer()* functions are used: first one retrieves the organization (whom the role belongs), and the second one retrieves CRIO metamodel (organization's parent element and also metamodel's root element).

The solution presented in Figs. 6,7 and 8 consists in forming subgroups identifying related capacities and associating them to different roles. It would be ideal to have roles only with related capacities. This proposal provides a simplification of role behaviors, it reduces the complexity of the statechart representation, making it easier to read and understand. Besides, in order to play a role, an agent requires fewer services or skills.

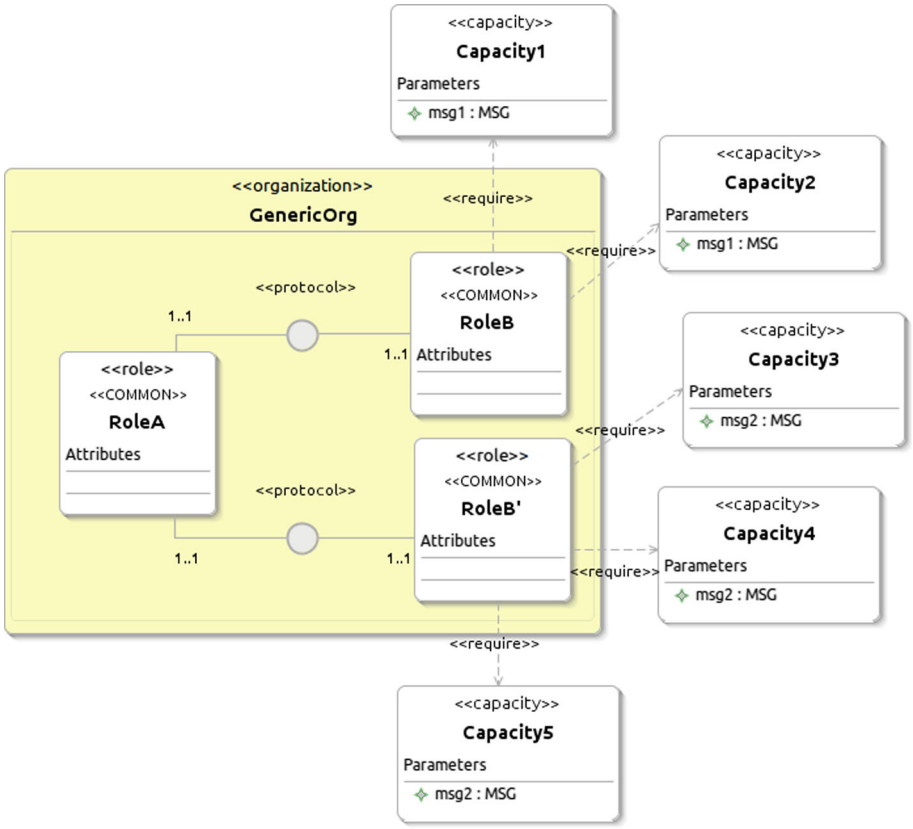

Fig. 6 Proposed solution for the *Behavior Skill* smell

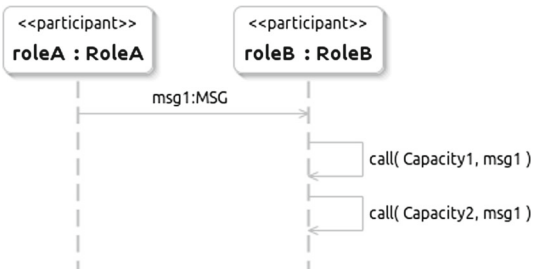

Fig. 7 Protocol between *RoleA* and *RoleB*

How to fix it?. Refactoring steps.

Step 1. In the selfish role, identify related capacities (the capacity outputs are inputs to another capacity and so on) and form subgroups.
Step 2. Create a number of roles that are at least equal to the number of detected subgroups and name them.
Step 3. Take each subgroup of the capacities and relate it with one of the new roles.

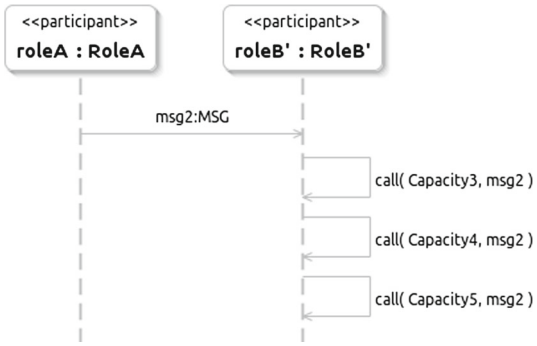

Fig. 8 Protocol between *RoleA* and *RoleB'*

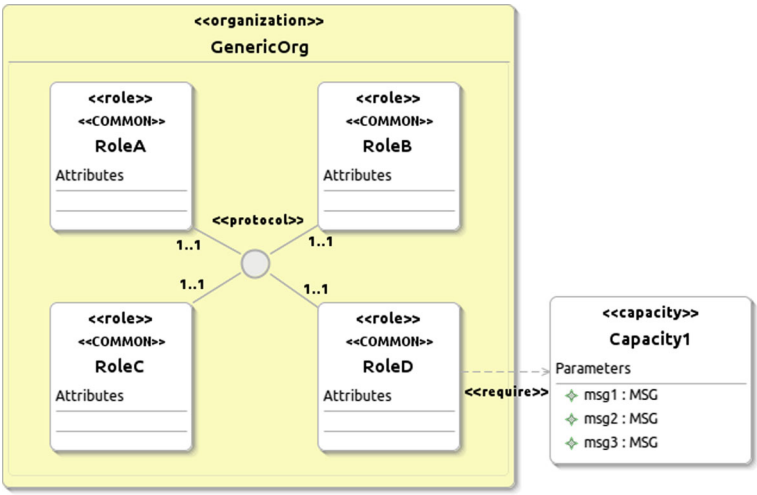

Fig. 9 Interaction smells: *Bottleneck Situation*—it is a role that receives more messages than it can process

5.2 Bottleneck situation example

This design smell, shown in Figs. 9 and 10, describes a role which receives different messages from different role emitters. These messages are required by only one capacity. Furthermore, given the asynchronous nature of the message exchange among agents, sometimes it can happen that a capacity cannot start its execution because it is waiting for all the input parameters necessary to carry on with the task. In other words, all the roleplayers are blocked in a state of the role waiting for all the messages.

How to identify it?. Methodological steps.

Step 1. In the receiving role, identify all the messages (link) that come from different role emitters.
Step 2. Compare messages with the capacities' input parameters associated with the receiver role.

If we want to define a rule to identify the *Bottleneck Situation* with EVL, it can be done as shown in Listing 4. The `context` of this rule is provided by the concept of Role (see line 29).

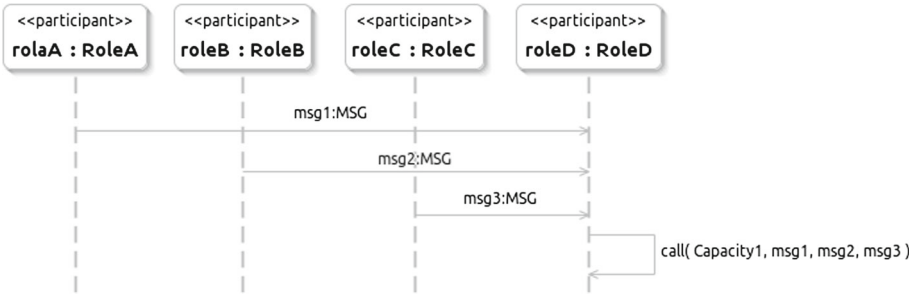

Fig. 10 Interaction smells: *Bottleneck Situation*—interaction diagram

Within the `check` body of this rule all instances of *participantInstance* correspondent to the role are ran through. With *findMatches()*, which will be explained later, the presence of this design smells is determined.

The operation *getParameterNames()*, defined in line 1, has the purpose of recovering a list with the names of the capacity's parameters.

The operation *getInteractionNames()* (see line 9) has the purpose of returning a list with the names of the interactions. This list will be conformed only by the messages that have the *participantInstance* as recipient, passing as parameters in the operation.

The purpose of the function defined in line 17, *findMatches()*, is to verify the existence of a coincidence between the names of the capacity's parameters and the names of the messages. Finding the first coincidence will suffice to consider link `for` as finished. This is due to the fact that if this ill design is identified, one must resolve it first and then execute the validation rule again.

Listing 4 EVL Constraint for Bottleneck Situation.

```
1  operation Capacity getParameterNames() : Set(String) {
2      var parameterNames : Set(String);
3      for ( n in self.parameters.select( m | m.type.asString() = 'input' ) ) {
4          parameterNames.add( n.name );
5      }
6      return parameterNames;
7  }
8
9  operation Protocol getInteractionNames(rs : ParticipantInstance) : Set(String) {
10     var interactionMessages : Set(String);
11     for ( n in self.interactions.select( n | n.dest = rs ) ) {
12         interactionMessages.add(n.name);
13     }
14     return interactionMessages;
15 }
16
17 operation ParticipantInstance findMatches() : Boolean {
18     var isMatchFound : Boolean = false;
19     var interactionNames : Set(String);
20     interactionNames = self.eContainer().getInteractionNames( self );
21     for ( n in self.represents.require) {
22         if ( interactionNames.includesAll( n.getParameterNames() ) ) {
23         isMatchFound = true;
24         }
25     }
26     return isMatchFound;
27 }
28
29 context Role {
30     critique WaitingForAllMessage {
```

```
31          check {
32          var isBottleneck : Boolean = false;
33          var participantInstance : ParticipantInstance;
34          for ( p in ParticipantInstance.allInstances()->select(rs | rs.represents = self ) ) {
35              if ( p.findMatches() ) {
36                  isBottleneck = true;
37                  participantInstance = p;
38              }
39          }
40          return not isBottleneck;
41      }
42      message : 'Multiple interactions which can be combined in a single one. Role name: ' +
43                  participantInstance.name
44      }
45  }
```

The solution proposed on Figs. 11, 12 and 13 is about creating a new role that acts as intermediary between the message sender roles and the recipient. This new role will receive the messages and each time it gets a set (equal in number and type of entry to those of the Capacity), it will resend it to the consumer role. One important advantage to note is the fact that the consumer role receives, always and once and for all, the messages its capacity needs, and at the same time avoids getting blocked in waiting for some messages while new messages arrive.

How to fix it?. Refactoring steps.

Step 1. Create a new role and name it.
Step 2. Redirect all the links related to the design smells to the new role.
Step 3. Add a new link between the new role a role receiver of messages. This new role will be responsible for collecting all the messages and sending them together.

6 Related work

There have been many researches concerning bad-smells/code-smells in the object-oriented literature. Emden and Moonen (2002) presents an empirical approach for the automatic detection of smells. These (smells) are characterized and identified through their aspects. This author proposes a tool called jCosmo that indicates which smells were found, which parts of the system are affected, and their concentration. Mantyla et al. (2003) noted that there

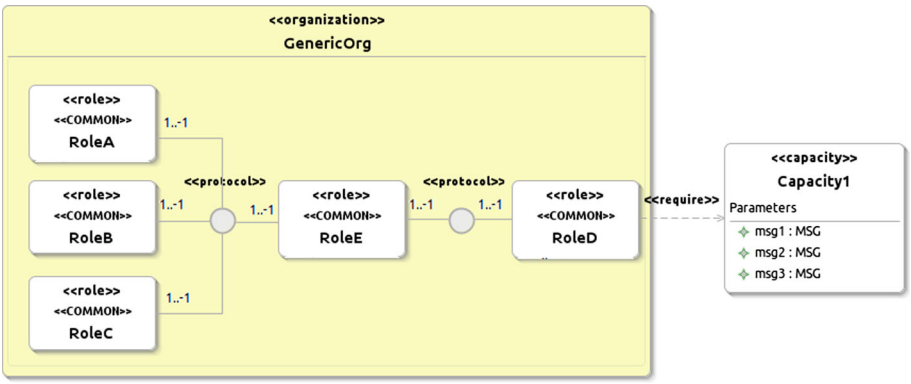

Fig. 11 Proposed solution for the *Bottleneck Situation* smell

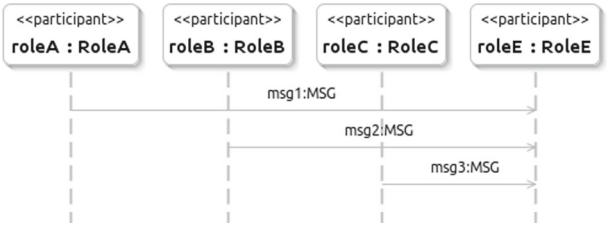

Fig. 12 Protocol among *RoleA*, *RoleB*, *RoleC*, and *RoleE*

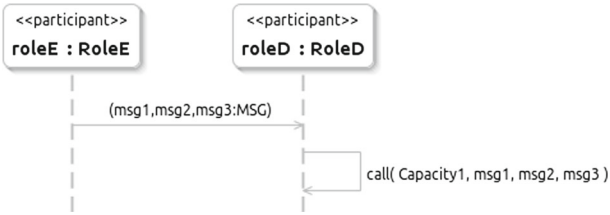

Fig. 13 Protocol between *RoleE* and *RoleD*

are relations between the smells and thus proposed grouping the smells in a six-category taxonomy (*Bloaters*, *Object-Orientation Abusers*, *Change Preventers*, *Dispensables*, *Couplers*, and *Others*). The author also carried out studies of the existing correlations between the smells, something done by Fontana and Zanoni (2011) too, but while looking for a direct or indirect relationship among smells. Counsell et al. (2010) conducts a study trying to determine to which extent the developers understand the smells themselves and to determine which smells are harder to eradicate. For smells eradication, a certain amount of refactoring is necessary. Yamashita and Moonen (2012) claims that smells identification is useful for doing a more precise evaluation of the maintainability factors, such as understandability and changeability. He also notes that some of these factors cannot be identified by the smells and that it is necessary to look for other ways to evaluate them. The smells are an alternative to the software metrics since they are always accompanied by the necessary refactoring strategies to eradicate them. Guo et al. (2010) takes into account the specific characteristics of a particular domain and states that smells definitions should be more precise, so they can adapt themselves to include information of such domains (for example, the GUI usually have too many public members and thus they should not be considered God Class).

As regards the agent oriented paradigm, Tiryaki et al. (2008) presents an approach to detect a series of smells and a set of refactoring patterns for their eradication. His work is based on an agile, iterative and incremental methodology with a testing framework support, called AOTDD[5] (Tiryaki et al. 2007). From our point of view, our proposal has three important differences from his approach. The first is the methodology used as basis. Here, Tyriaky adopts a process oriented to the agile development of multi-agent systems, which, as most of this kind of methodologies, focuses on two of its main pillars: Test-driven Development (TDD) and the refactoring techniques. We, on the other hand, adopt ASPECS, which is a software process for the engineering of open, complex, and distributed systems. Such methodology is strongly structured and provides step by step guidelines from the requirements to the generation of code for a specific platform. It also makes it possible to create models with different levels of abstraction (Hierarchical Models), thus providing support to one of the

[5] Agent Oriented Test Driven Development.

most promising approaches of software engineering. The second difference lies in the phase–-within the development process- where the smells are found. In AOTDD the smells can only be found at the end of the iteration of the development cycle. However, we consider that it is possible to detect smells in the early stages, such as phases analysis and design, so they can be addressed immediately in the cases this is feasible.

The third is the type of smells to be analyzed. Tyriaky has a generic approach; in fact, the authors propose a metamodel with the most recurring abstractions in the MAS development methodologies. For this reason, a three leveled classification of smells is done : (i) role level, (ii) plan task, and (iii) action level. The smells proposed by Tyriaky are a redefinition of those done by Fowler and only focus on matters related to the role, particularly in the final details like finding duplication of actions, parameters, plan and codes structures, input and output long lists, and bad allocation of responsibilities, among others. Our approach, however, aims at the structural aspects of organizational modelling and its functioning (except from *Selfish-role* that could be regarded as an *Overloaded Role Bad Smells*).

In the agent oriented literature there is a large number of proposed development tools. The list comprises tools for academic projects as for commercial applications, for example the authors of Nunes et al. (2009) study tools for code generation. Unlike the object oriented paradigm, which reached a high level of maturity, this number of tools reflects the evolution as well as the number of concepts and techniques that have been proposed by the agent technology.

Nowadays, there are must-have features for every tool, such as project management, creation and edition of files, simplification of the refactoring stage, build and run processes, code generation, testing and so on. Likewise, tools must also allow to transmit, as fast as possible, the stakeholders' intentions through a combination of graphic and textual notations that represent a specific language shared by the development team. Generally, and according to the maturity and correct use of the tools, it is possible to increase the development teams' productivity, reducing time and costs. Moreover, they improve the system documentation, the maintainability parameters, the precision in the requirements and, at the same time, they improve the quality of the delivered software.

Given the vast diversity of the metaphors used for the modeling of multi-agent systems, it is impossible to compare them all in this paper. For this reason, we focused on the tools that are based on the organizational approach (Organizations Centered Multi-Agent Systems, OCMAS). The main goal is to highlight the similarities and differences between Janeiro and some of the most well-known tools for the organizational approach. The results of this comparative analysis are summarized in Table 2, which focuses on the primary objective of this article: models validation. What follows is a brief explanation of the columns:

Tool Name and Methodology. Here, the name of the tool and the methodology it supports are specified. It is important to note that, in the literature, there is at least one tool for each methodology.

Supported Diagram. Tools like aT^3 (also know as AgentTool III) (Garcia-Ojeda and DeLoach 2009), GAIA4E (Cernuzzi and Zambonelli 2009), PTK (Passi ToolKit) (Chella et al. 2004), OpenTool (Picard and Gleizes 2004), TAOM4E (Tool for Agent Oriented visual Modeling for the Eclipse platforms) (Morandini et al. 2011) and Romas modeling tool (Garcia et al. 2009) provide a complete coverage over the defined diagrams of their respective methodologies. Whereas applications like IDK (Ingenias Development Kit) (Gomez-Sanz et al. 2008), PDT (Prometheus Design Tool) (Thangarajah et al. 2005), Rebel (Roadmap Editor Built for Easy Development) (Al-Hashel et al. 2007) and Janeiro only have a partial coverage of the methodologies (denoted by P in Table 2); that is, not all of their diagrams have been developed. In the case of Janeiro, this is due to the fact that the tool is a prototype

Table 2 Organizational CASE tools

	Methodology	All supported diagrams	Cross-checking	Syntax validation	Semantic validation	Code generation	Active
aT^3	O-MaSE	✓	✓	✓	✗	✓	✓
Gaia4E	GAIA	✓	–	–	–	✗	–
IDK	Ingenias	P	✓	✓	✗	✓	✓
PDT	Prometheus	P	✓	✓	✗	✓	–
PTK	PASSI	✓	P	✓	✗	✓	✗
OpenTool	Adelfe	✓	–	–	✗	–	✗
Rebel	ROADMAP	P	✗	✗	✗	✗	✗
TAOM4E	Tropos	✓	–	–	✗	✓	✓
Romas	Romas	✓	✓	✓	✓	✓	✓
Janeiro Studio	Aspecs	P	✗	✓	✓	✗	✓

Legend: ✓: fully supported; ✗: the tool does not have this feature; P: Partial coverage of the feature

that is still in its developmental stages, thus only covers the principal diagrams defined for the first phase of the ASPECS methodology, *Problem Domain*.

The following three criteria indicate whether the tools implement processes for the verification of the completeness, correctness and coherence of the designed model.

Cross-checking. In the Model Driven Development (MDD) approach, all the users' requirements are translated in a set of artifacts that represent the problem and its solution. This is why it is better for the concepts to be consistent with the different artifacts that supply the different perspectives of the model. This criteria highlights those applications that can find inconsistencies that may appear in the same concept in the different artifacts that compose the model. In our analysis, this characteristic is offered by aT^3, IDK, PDT, and Romas.

Syntax Validation. The concept of syntactic validation refers to the possibility of looking for certain expression errors in the diagrams. For example, concepts that have not been named, name duplication, or the bad use of some concepts. This is one of the most common characteristics of development tools, with the exception of Rebel that does not do these types of validations. Cross-Checking and Syntax Validation are considered as mandatory characteristics in tools nowadays.

Semantic Validation. The criterion presents those tools that can interpret the model for the detection of certain properties that conceptually invalidate it. This situation is only contemplated by two tools: Romas and Janeiro Studio. Roma looks for potential conflicts related to the templates of the designed contracts and the norms of the organization. Janeiro Studio, on the other hand, looks for situations that can be detrimental for the model and cause a misuse of the resources or a rise in the maintaining efforts.

Code Generation. It is expected for all tools to allow generating code for a specific language. This generated code is, in most cases, a skeleton on which designers can start to work, adding the most desired features by users. It is expected since most methodologies have an implementation platform and/or a specific language. Almost all tools described provide a code generating module for a specific framework; such as aT^3 for JADE platform (Bellifemine et al. 2001; TAOM4E, also for JADE and JADEX (Pokahr et al. 2005); Romas for THOMAS (Criado et al. 2011) and Electronic Institutions (Sierra et al. 2004) platforms.

The case of IDK, thanks to its integrated plugin Ingenias Agent Framework (Gómez-Sanz et al. 2010), not only generates code for JADE, but also has a series of templates based on a proprietary mechanism that allows automatically generating code for any language. The rest of the tools do not have an extension for the generation of code, or are being developed, as in the case of Janeiro Studio.

Active. Last but not least, this column indicates if the project is being developed or is updated by its creators. We use a subjective criterion for the comparison; we consider dates of the last update in their official sites and some articles published from the release of the application. There are many which are still being improved, such as Janeiro, Romas and aT^3, while there are others which seem to have been abandoned. The reason behind this abandonment is, in some cases, that there have not been significant conceptual advances in the methodologies used as basis or that the authors are engaged in other activities.

7 Conclusion and future works

This work is an additional contribution to the work currently being done on one of the most complete methodologies of multi-agent systems based on the organizational approach, called ASPECS (Isern et al. 2011). In fact, this paper poses a double contribution. First, it has introduced an adaptation of the theoretical metamodel CRIO using an EMF model. This is a core part of a CASE environment called Janeiro Studio which provides support to the diagrams defined in the ASPECS methodology. The aim is to make, through the graphic editors, the manipulation of the model instances easier for the designers. It is important to mention that Janeiro is available under an open source license at http://repo.gitia.org/janeiro/studio/.

Secondly, it has described the *Organizational Design Smells* as an important technique for the detection of anomalies that may be present in the models during the analysis stage. For such purpose, a set of validation rules was defined, with the help of EVL, which enables organizational smells detection. Additionally, a module which allows the automatic execution and validation of models based on those rules was also developed and integrated as part of Janeiro Studio. The automatic validation mechanism is important as it helps engineers to identify this kind of design smells and tackle them, in some cases, immediately. Moreover, we present two examples, their justification and how to detect them. Early detection of these flaws brings about many advantages: a correct balance of behavior skills, an adequate size of the skill, and a proper definition of interactions for an optimal performance of the systems and a good use of the resources.

In our future work, we will analyze the possibility of having many of the proposed solutions for smells serve as basis for the definitions of design patterns. We will also evaluate the possibility of integrating a design patterns repository in Janeiro Studio to develop a multi-agent system based on patterns. With this, it will be possible to have better quality in the system and a high degree of reuse of tested components, thus reducing time.

Acknowledgments Authors would like to thank Gilda Moreno and Nicolás Majorel Padilla for their reviews and suggestions to this paper.

References

Al-Hashel E, Balachandran BM, Sharma D (2007) A comparison of three agent-oriented software development methodologies: roadmap, prometheus, and mase. In: Apolloni B, Howlett RJ, Jain L (Eds.) Knowledge-based intelligent information and engineering systems, no. 4694. Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2007, pp. 909–916. http://link.springer.com/chapter/10.1007/978-3-540-74829-8_111

Araujo P, Lizondo D, Rodriguez S, Hilaire V (2015) An approach for organizational design smells identification within multi-agent systems. In: International workshop on coordination, organisation, institutions and norms in multi-agent systems

Araujo P, Rodriguez S (2013) Janeiro studio. In: Congreso Nacional de Ingeniería Informática/Sistemas de Información, Córdoba, Argentina, 2013

Bellifemine F, Poggi A, Rimassa G (2001) Jade: a fipa2000 compliant agent development environment. Proceedings of the fifth international conference on Autonomous agents, ACM 2001:216–217

Bresciani P, Perini A, Giorgini P, Giunchiglia F, Mylopoulos J (2004) Tropos: an agent-oriented software development methodology. Auton Agents Multi-Agent Syst 8(3):203–236

Budinsky F, Brodsky SA, Merks E (2003) Eclipse modeling framework. Pearson Education, Upper Saddle River

Busetta P, Rönnquist R, Hodgson A, Lucas A (1999) Jack intelligent agents-components for intelligent agents in java. AgentLink News Lett 2(1):2–5

Carneiro GdF, Silva M, Mara L, Figueiredo E, Sant'Anna C, Garcia A, Mendonca M (2010) Identifying code smells with multiple concern views. In: 2010 Brazilian symposium on software engineering (SBES), 2010, pp. 128–137. doi:10.1109/SBES.2010.21

Cernuzzi L, Zambonelli F (2009) Gaia4e: a tool supporting the design of mas using gaia. In: ICEIS (4), Citeseer, 2009, pp. 82–88

Chella A, Cossentino M, Sabatucci L (2004) Tools and patterns in designing multi-agent systems with passi. WSEAS Trans Commun 3(1):352–358

Cossentino M, Gaud N, Hilaire V, Galland S, Koukam A (2010) Aspecs: an agent-oriented software process for engineering complex systems. Auton Agents Multi-Agent Syst 20(2):260–304. doi:10.1007/s10458-009-9099-4

Cossentino M, Hilaire V, Molesini A, Seidita V (2014) Handbook on agent-oriented design processes. Springer, Berlin

Cossentino M, Potts C (2002) PASSI: a process for specifying and implementing multi-agent systems Using UML

Counsell S, Hamza H, Hierons R (2010) The 'deception' of code smells: an empirical investigation. In: 2010 32nd international conference on information technology interfaces (ITI), 2010, pp. 683–688

Criado N, Argente E, Botti V (2011) Thomas: an agent platform for supporting normative multi-agent systems, J Log Comput 23(11):309–333. doi:10.1093/logcom/exr025

Criado N, Julián V, Botti V, Argente E (2010) A norm-based organization management system. In: Padget J, Artikis A, Vasconcelos W, Stathis K, Silva VTd, Matson E, Polleres A (Eds.) Coordination, organizations, institutions and norms in agent systems V, no. 6069. Lecture notes in computer science, Springer Berlin Heidelberg, 2010, pp. 19–35. http://link.springer.com/chapter/10.1007/978-3-642-14962-7_2

DeLoach SA, Garcia-Ojeda JC (2010) O-mase: a customisable approach to designing and building complex, adaptive multi-agent systems. Int J Agent-Oriented Softw Eng 4(3):244–280

Emden Ev, Moonen L (2002) Java quality assurance by detecting code smells. In: Ninth working conference on reverse engineering, 2002. Proceedings, 2002, pp. 97–106. doi:10.1109/WCRE.2002.1173068

Fagan M (1976) Design and code inspections to reduce errors in program development, IBM Syst 15(3):182–211

Ferber J, Gutknecht O, Michel F (2004) From agents to organizations: an organizational view of multi-agent systems. In: Giorgini P, Müller J, Odell J (eds) Agent-oriented software engineering IV, vol 2935. Lecture notes in computer scienceSpringer, Berlin Heidelberg, pp 214–230

Ferber J (1998) Gutknecht O (1998) A meta-model for the analysis and design of organizations in multi-agent systems. Proceedings of the 3rd international conference on multi agent systems, ICMAS '98. IEEE Computer Society, Washington, DC, USA, p 128

Fontana F, Zanoni M (2011) On investigating code smells correlations. In: 2011 IEEE fourth international conference on software testing, verification and validation workshops (ICSTW), 2011, pp. 474–475. doi:10.1109/ICSTW.2011.14

Fortino G, Russo W (2012) Eldameth: an agent-oriented methodology for simulation-based prototyping of distributed agent systems. Inf Softw Technol 54(6):608–624. doi:10.1016/j.infsof.2011.08.006http://www.sciencedirect.com/science/article/pii/S0950584911001960

Fowler M, Beck K, Brant J, Opdyke W, Roberts D (1999) Refactoring: improving the design of existing code, 1st Edition, Addison-Wesley Professional, Boston. http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/0201485672

Galland S, Gaud N, Rodriguez S, Hilaire V (2010) Janus: another yet general-purpose multiagent platform. In: 7th agent-oriented software engineering technical forum (TFGAOSE-10), Agent Technical Fora, 2010

Garcia E, Argente E, Giret A (2009) A modeling tool for service-oriented open multiagent systems. In: Yang J-J, Yokoo M, Ito T, Jin Z, Scerri P (Eds.) Principles of practice in multi-agent systems, no. 5925. Lecture notes in computer science, Springer Berlin Heidelberg, 2009, pp. 345–360. http://link.springer.com/chapter/10.1007/978-3-642-11161-7_24

Garcia E, Giret A, Botti V (2015) Romas methodology. In: Regulated open multi-agent systems (ROMAS), Springer International Publishing, 2015, pp. 51–95. http://link.springer.com/chapter/10.1007/978-3-319-11572-6_6

Garcia E, Giret A, Botti V (2015) Romas modeling language. In: Regulated open multi-agent systems (ROMAS), Springer International Publishing, 2015, pp. 43–49. http://link.springer.com/chapter/10.1007/978-3-319-11572-6_5

Garcia-Ojeda JC, DeLoach SA (2009) Robby, agenttool iii: From process definition to code generation. In: Proceedings of the 8th international conference on autonomous agents and multiagent systems - Volume 2, AAMAS '09, international foundation for autonomous agents and multiagent systems, Richland, SC, 2009, pp. 1393–1394. http://dl.acm.org/citation.cfm?id=1558109.1558311

Garcia-Ojeda J, DeLoach S, Robby Oyenan W, Valenzuela J (2008) O-mase: a customizable approach to developing multiagent development processes. In: Luck M, Padgham L (eds) Agent-oriented software engineering VIII, vol 4951. Lecture notes in computer scienceSpringer, Berlin Heidelberg, pp 1–15

Gómez-Sanz JJ, Fernóndez CR, Arroyo J (2010) Model driven development and simulations with the ingenias agent framework. Simul Modell Pract Theory 18(10):1468–1482. doi:10.1016/j.simpat.2010.05.012http://www.sciencedirect.com/science/article/pii/S1569190X10001024

Gomez-Sanz JJ, Fuentes R, Pavón J, García-Magariño I (2008) Ingenias development kit: a visual multi-agent system development environment. In: Proceedings of the 7th international joint conference on autonomous agents and multiagent systems: Demo Papers, AAMAS '08. International foundation for autonomous agents and multiagent systems, Richland, SC, 2008, pp. 1675–1676. http://dl.acm.org/citation.cfm?id=1402744.1402760

Guo Y, Seaman C, Zazworka N, Shull F (2010) Domain-specific tailoring of code smells: an empirical study. In: 2010 ACM/IEEE 32nd international conference on software engineering, Vol. 2, 2010, pp. 167–170. doi:10.1145/1810295.1810321

Gutknecht O, Ferber J (2000a) Madkit: a generic multi-agent platform. In: Proceedings of the fourth international conference on autonomous agents, AGENTS '00, ACM, New York, NY, USA, 2000, pp. 78–79. doi:10.1145/336595.337048

Gutknecht O, Ferber J (2000b) The madkit agent platform architecture. In: Wagner T, Rana OF (Eds.), Infrastructure for agents, multi-agent systems, and scalable multi-agent systems, no. 1887. Lecture notes in computer science, Springer Berlin, 2000, pp. 48–55. http://link.springer.com/chapter/10.1007/3-540-47772-1_5

Hübner JF, Sichman JS, Boissier O (2002) Moise+: towards a structural, functional, and deontic model for mas organization. In: Proceedings of the first international joint conference on autonomous agents and multiagent systems: part 1, AAMAS '02, ACM, New York, NY, USA, 2002, pp. 501–502. doi:10.1145/544741.544858

Isern D, Sánchez D, Moreno A (2011) Organizational structures supported by agent-oriented methodologies. J Syst Softw 84(2):169–184. doi:10.1016/j.jss.2010.09.005http://www.sciencedirect.com/science/article/pii/S0164121210002451

Juan T, Pearce AR, Sterling L (2002) ROADMAP: extending the gaia methodology for complexopen systems. In: AAMAS, ACM, 2002, pp. 3–10. http://doi.acm.org/10.1145/544741.544744

Khomh F, Vaucher S, Gueheneuc Y-G, Sahraoui H (2009) A bayesian approach for the detection of code and design smells. In: 9th international conference on quality software, 2009. QSIC '09, 2009, pp. 305–314. doi:10.1109/QSIC.2009.47

Lanza M, Marinescu R (2006) Object-oriented metrics in practice, Springer Berlin Heidelberg, Berlin, Heidelberg, 2006. http://link.springer.com/10.1007/3-540-39538-5

Mantyla MV, Vanhanen J, Lassenius C (2004) Bad smells–humans as code critics. In: 20th IEEE international conference on software maintenance, 2004. Proceedings, 2004, pp. 399–408. doi:10.1109/ICSM.2004.1357825

Mantyla M, Vanhanen J, Lassenius C (2003) A taxonomy and an initial empirical study of bad smells in code. In: International conference on software maintenance, 2003. ICSM 2003. Proceedings, 2003, pp. 381–384. doi:10.1109/ICSM.2003.1235447

McAffer J, Lemieux J-M, Aniszczyk C (2010) Eclipse rich client platform, 2nd edn. Addison-Wesley Professional, Upper Saddle River, NJ

Moha N, Gueheneuc YG, Duchien L, Meur AFL (2010) Decor: a method for the specification and detection of code and design smells. IEEE Trans Softw Eng 36(1):20–36. doi:10.1109/TSE.2009.50

Morandini M, Nguyen DC, Penserini L, Perini A, Susi A (2011) Tropos modeling, code generation and testing with the taom4e tool. In: CEUR proceedings of the 5th international i* workshop (iStar 2011), Citeseer, 2011, pp. 172–174

Nunes I, Cirilo E, de Lucena CJ, Sudeikat J, Gómez-Sanz CHJJ (2009) A survey on the implementation of agent oriented specifications. In: Gleizes MP, Gómez-Sanz JJ (Eds.) AOSE, Vol. 6038 of lecture notes in computer science, Springer, 2009, pp. 169–179. http://dx.doi.org/10.1007/978-3-642-19208-1

Object Constraint Language (OCL) 2.3.1 Specification, version 2.3.1 (ja 2012)

Omicini A (2000) Soda: societies and infrastructures in the analysis and design of agent-based systems. In this volume, Springer-Verlag 2000:185–193

Padgham L, Winikoff M (2003) Prometheus: a methodology for developing intelligent agents. In: Giunchiglia F, Odell J, Weiß G (eds) Agent-oriented software engineering III, vol 2585. Lecture notes in computer scienceSpringer, Berlin Heidelberg, pp 174–185

Pavón J, Gómez-Sanz J (2003) Agent oriented software engineering with ingenias, In: Mařík V, Pěchouček M, Müller J (Eds.) Multi-agent systems and applications III, no. 2691. Lecture notes in computer science, Springer Berlin Heidelberg, 2003, pp. 394–403. http://link.springer.com/chapter/10.1007/3-540-45023-8_38

Picard G, Gleizes M-P (2004) The adelfe methodology, In: Bergenti F, Gleizes M-P, Zambonelli F(Eds.) Methodologies and software engineering for agent systems, no. 11 in multiagent systems, artificial societies, and simulated organizations, Springer US, 2004, pp. 157–175. http://link.springer.com/chapter/10.1007/1-4020-8058-1_11

Pokahr A, Braubach L, Lamersdorf W (2005) Jadex: a bdi reasoning engine. Multi-agent programming, Springer 2005:149–174

Rodriguez S, Gaud N, Hilaire V, Galland S, Koukam A (2007) An analysis and design concept for self-organization in holonic multi-agent systems. In: Proceedings of the 4th international conference on Engineering self-organising systems, ESOA'06, Springer-Verlag, Berlin, Heidelberg, 2007, pp. 15–27

Rose LM, Paige RF, Kolovos DS, Polack FA (2008) The epsilon generation language. Model driven architecture-foundations and applications, Springer 2008:1–16

Sierra C, Rodriguez-Aguilar JA, Noriega P, Esteva M, Arcos JL (2004) Engineering multi-agent systems as electronic institutions. Eur J Inform Prof 4(4):33–39

Thangarajah J, Padgham L, Winikoff M (2005) Prometheus design tool. In: Proceedings of the fourth international joint conference on autonomous agents and multiagent systems, AAMAS '05, ACM, New York, NY, USA, 2005, pp. 127–128

Tiryaki AM, Ekinci EE, Dikenelli O (2008) Refactoring in multi agent system development. In: Bergmann R, Lindemann G, Kirn S, Pěchouček M (Eds.) Multiagent system technologies, no. 5244. Lecture notes in computer science, Springer Berlin Heidelberg, 2008, pp. 183–194. http://link.springer.com/chapter/10.1007/978-3-540-87805-6_17

Tiryaki AM, Öztuna S, Dikenelli O, Erdur RC (2007) Sunit: a unit testing framework for test driven development of multi-agent systems. In: Padgham L, Zambonelli F(Eds.) Agent-oriented software engineering VII, no. 4405. Lecture notes in computer science, Springer Berlin Heidelberg, 2007, pp. 156–173. http://link.springer.com/chapter/10.1007/978-3-540-70945-9_10

Unified modeling language (uml), infrastructure, version 2.4.1 (aug 2011)

Yamashita A, Moonen L (2012) Do code smells reflect important maintainability aspects?. In: 2012 28th IEEE international conference on software maintenance (ICSM), 2012, pp. 306–315. doi:10.1109/ICSM.2012.6405287

Zambonelli F, Jennings NR, Wooldridge M (2003) Developing multiagent systems: the gaia methodology. ACM Trans Softw Eng Methodol (TOSEM) 12(3):317–370