Extending Gaia with Agent Design and Iterative Development

Jorge Gonzalez-Palacios[1] and Michael Luck[2]

[1] University of Southampton jlgp02r@ecs.soton.ac.uk
[2] King's College London michael.luck@kcl.ac.uk

Abstract. Agent-oriented methodologies are an important means for constructing agent-based systems in a controlled repeatable form. However, agent-oriented methodologies have not received much acceptance in industrial environments, which can be partially explained by drawbacks in current agent-oriented methodologies, mainly in terms of applicability and comprehensiveness. Specifically, Gaia, one of the most cited methodologies, does not consider agent design, nor exhibits an iterative methodological process. On the other hand, Gaia is based on organisational abstractions (which makes it suitable to cope with the design of complex systems) and possesses a simple methodological process whose neutrality makes it suitable for extensions. In this paper, we extend Gaia in two directions: we incorporate an agent design phase, and we enhance the methodological process with the use of iterations.

1 Introduction

Emergent technologies such as the Grid, peer-to-peer computing and ubiquitous computing, require systems that are open, highly distributed, and whose components exhibit some level of autonomy and pro-activeness. It has been claimed [15, 5] that the combined use of the multi-agent approach and organisational abstractions is a suitable means to model such systems. More specifically, organisational abstractions provide agent-oriented methodologies with the necessary design abstractions to cope with the development of complex systems in a systematic and controlled form.

Among those agent-oriented methodologies based on organisational abstractions [16, 12, 4, 1], Gaia [16] is arguably the most used. The popularity of Gaia can be explained by the characteristics of its methodological process, which is simple to understand, has a good separation of development phases, and is neutral to any specific implementation technique or platform. However, Gaia focuses only on the *organisational* aspect of multi-agent systems (or macro level), leaving the actual design of agents (the micro level) unconsidered. This results in the absence of key development phases in the methodological process, such as agent design and implementation, both essential for the development of real world systems. Additionally, there is a propensity of the methodological process in general to construct systems once and for all, rather than part by part. This constitutes another drawback of Gaia, since it is very difficult to accomplish in a single opportunity the complete and detailed design of a whole complex system. Similarly, other development activities such as implementation and testing are also complicated if no explicit mechanism for decomposing the development is present.

In order to address these drawbacks, in this paper we present two extensions to the Gaia methodology. The first extension consists of incorporating a phase for agent design, based on the use of well-known *agent architectures* [14]. The second provides the Gaia process with a mechanism that decomposes the development of a system into *iterations*, an approach that has been used successfully in mature object-oriented methodologies, for example in the Unified Software Development Process [11].

The rest of this paper is organised in the following way. In Section 2 we briefly describe the main aspects of the Gaia methodology. In Section 3 we present the models and activities of our proposed agent design phase, and show how this phase fits in the Gaia methodology. In Section 4 we describe how we enhance the Gaia process to incorporate the agent design phase, and use iterations to decompose the development of a system. Finally, in Section 5 we present our conclusions.

2 Gaia overview

Gaia [16] is an agent-oriented methodology based on the organisational concepts of roles, interactions, and organisations, and is divided into analysis, *architectural* design, and *detailed* design. A brief description of these concepts and phases is presented below.

Roles Roles in Gaia represent well defined positions in the organisation, and the behaviour expected from them. Roles are characterised by: a *name* that identifies the role; a brief *description*; the *protocols* through which it interacts with other roles; the *activities* that the role performs without interacting with other roles; the *responsibilities* that express the functionality of the role (divided into *liveness properties* and *safety properties*, which relate to states of affairs that a role must bring about, and the conditions whose compliance the role must ensure, respectively); and the *permissions* to access the resources that the role needs for fulfilling its responsibilities. A role is depicted graphically by means of a *role schema*, an example of which is shown in Figure 1. As can be observed in the figure, boxes in the schema correspond to the characterisation of roles, and the names of activities are underlined to distinguish them from names of protocols. Additionally, the responsibilities are expressed in a purpose-built language that includes operators to represent sequence (.), alternatives (|) and indefinite repetition (^{w}).

Interactions Interactions in Gaia are characterised by means of *protocol definitions*, which consist of: a *purpose*, that provides a brief description of the interaction; a list of *initiators*, that enumerates the roles that can start the interaction (usually a single element); a list of *responders* that enumerates the roles involved in the interaction; a list of *inputs* and *outputs* that provides the information required or produced during the interaction; and a brief *description* of the purpose of the interaction. This characterisation is represented graphically using a diagram like that shown in Figure 1.

Organisations An organisation in Gaia is formed of an organisational structure and a set of organisational rules. The former consists of a topology (the set of communication paths between the roles), and a control regime (relationships of authority between the roles). Organisational rules are constraints about how the different elements of the organisation interact, and either express situations that agents try to bring about, or express conditions that must be kept invariable.

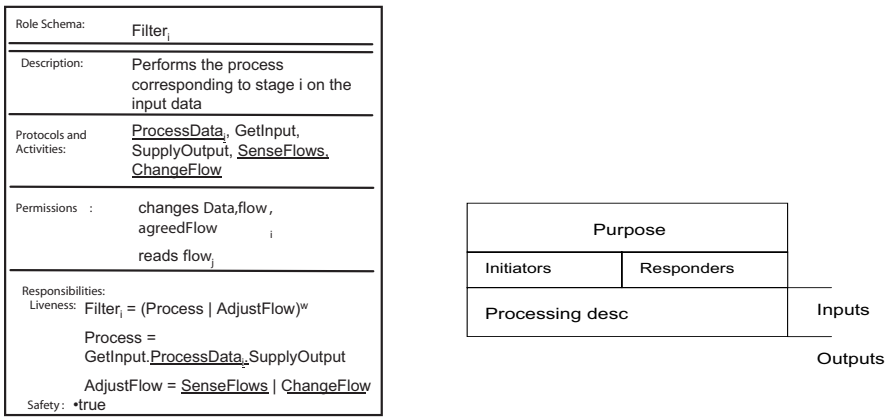


Fig. 1. Example of a role schema (left) and a generic protocol definition (right)

The analysis phase The analysis phase deals with collecting the features needed to understand the system, and consists of five activities: decomposition of the system into sub-organisations, which aims to partition the system into more manageable units; identification of environmental entities, which deals with creating a list of the resources used by agents while carrying out their activities and the rights of access to them (such as *read* or *change*); creation of the preliminary role model, which consists of the construction of all role schemata; creation of the preliminary interaction model, consisting in the creation of protocol definitions; and determining the organisational rules. It must be noted that, for the preliminary models, the emphasis is placed on the identification of roles and protocols, rather than in providing a complete description of their features.
The architectural design phase The next phase, architectural design, consists of two parts: the selection of an organisational structure, and the completion of the role and interaction models. The organisational structure plays an important part in the development of the system because it impacts on the way agents are coordinated, and on how the organisational rules are implemented. The completion of the role and interaction models deals with detailing the roles and protocols with the information obtained once the structure is determined. This activity includes the incorporation of new roles and interactions which may have resulted from the application of the previous step.
The detailed design phase The final phase of Gaia, the design phase, consists of producing the agent model, which involves determining which roles will be played by which agents, based on considerations such as efficiency and physical distribution.

3 Agent design

In addition to what Gaia provides, we also need to consider the design of the internal composition of agents. However, since no agent design phase is included in Gaia, we have constructed one which takes its inputs from the organisational design and its output is a specification of *how* agents fulfil their requirements, which in turn serves as

input to the implementation phase. Our proposed agent design phase complements the organisational design, in which agents are considered to be black boxes, and their detailed composition is ignored. Although, in general agents can play more than one role, in the following we assume that each agent plays exactly one role. At the end of the section, however, we provide guidelines for the general case in which an agent implements more than one role. The design phase consists of models and activities to produce these models, and is applied for each role of the system. These models and activities are described below.

3.1 Models

The agent design considers two models, the *structure model* and the *functionality model*, the former providing a structural decomposition of a role into classes, and the latter specifying how these classes collaborate to achieve the expected behaviour of the role.

The structure model The structure model decomposes a role into classes, thus the structure model is formed of class diagrams [6], one diagram for each role in the system. Although a class diagram is common in object-based techniques, it is used to represent different concepts — depending on the stage of the process in which it is used — so it is worth explaining the way in which we use it here. In the structure model, we use a class diagram to describe the main internal components of a role (as classes), and the static relationships between them, such as *dependence*, *part-of* and *inheritance*. The level of detail in the description must be sufficient to identify the core classes, and for each of these classes, the operations necessary to achieve the functionality of the role, and the internal information required to implement these methods (attributes). However, it is not necessary that the diagram includes all the classes needed to implement the role, nor all the attributes and methods to implement each class. The exact set of classes in the class diagram of a role largely depends on the agent architecture used to model the behaviour of the role, but there are some classes present in almost any architecture nevertheless. Such classes include: those representing the sensing and effecting capabilities of the role (such as message handling); classes for representing and manipulating the state of the agent; and classes for controlling agent behaviour.

The functionality model The functionality model consists of a set of *scenarios*, each of which represents a piece of functionality of the role, and contains a sequence diagram [6] showing how the role executes the functionality. The pieces of functionality are obtained from the role's responsibilities (as stated in its role schema), and together must cover all such responsibilities. This can be achieved by considering each term of the role's responsibilities as a piece of functionality. The classes involved in the sequence diagram are those of the class diagram corresponding to the role. For example, in a market application, a possible scenario for the buyer role would represent the functionality *find the best price seller for a given product* by means of a sequence diagram showing how the classes of the buyer interact to achieve it. Thus, any role in the system has an associated class diagram and a number of scenarios, each referring to a piece of functionality and described by means of a sequence diagram. This association is illustrated in Figure 2 for a generic role *i*, and a generic scenario *j*.

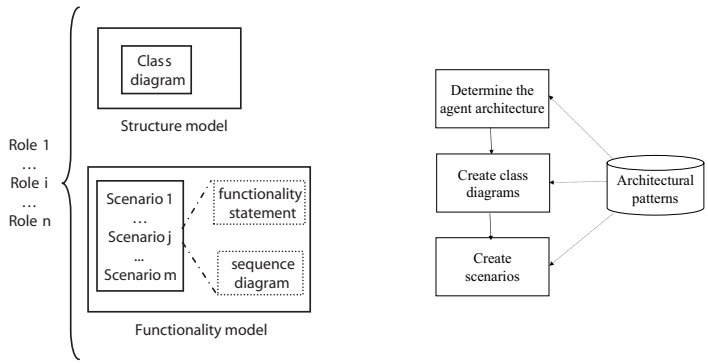


Fig. 2. The models (left) and activities (right) of the agent design

3.2 Activities

The activities involved in agent design consist of selecting an appropriate agent architecture and building the models described above. These activities are illustrated in Figure 2, together with indications of those stages in which the use of architectural patterns is valuable. Architectural patterns [10] are design patterns that represent well established agent architectures, and include a class diagram, scenarios, and situation in which their use is advisable, and methodological guidelines for their use. Developers refine architectural patterns, with application-dependant information, to obtain design models (structure and behaviour) of the agents in the system. According to this figure, for every role in the system the following activities are performed.

Determining the agent architecture In this activity, the architecture for the role is determined. In order to do this, several factors must be considered. The most important factor deals with the complexity of the behaviour expected from the role. For example, a simple behaviour can be more easily implemented through reactive architectures, whereas a complex behaviour may require the use of deliberative or hybrid architectures. A second factor deals with the level of pro-activeness required. Reactive architectures typically produce agents which are not pro-active, but operate on request of other agents, whereas BDI architectures are suitable for constructing pro-active agents. Other factors that affect the decision are the level of familiarity of developers with a specific architecture, and the support that different development tools provide for specific architectures. This activity can be facilitated by a catalogue of architectural patterns showing, for each architecture, its characteristics, advantages, limitations and applicability. The construction of such a catalogue is subject of further work, as well as of a procedure to select the pattern that best suits a given application.

Creating class diagrams In order to create the class diagram, two different methods can be used. The first is to employ an object-based methodology to create a design of the agent, according to the selected agent architecture. However, this can be a difficult

task for a typical software developer, since understanding agent architectures requires familiarity with concepts uncommon in traditional software engineering. The second method consists of using a catalogue of architectural patterns, together with guidelines for selecting an appropriate pattern for a specific role, and adapting the selected pattern to the specific application. Regardless of which method is used to construct the class diagram, the inputs are taken from the organisational design models of Gaia. Specifically, the role model provides three inputs: the liveness responsibilities (which describe the functionality that the role is expected to exhibit), the safety responsibilities (describing the conditions that must hold during the lifetime of the role), and the permissions (which contain the environmental entities employed by the role, together with the rights to access them). Additionally, the interaction model of Gaia provides the inputs and outputs of the protocols in which a role participates, the organisational structure provides the authority relationships involving the role, and the organisational rule model provides the rules that constrain the behaviour of the agent.

Creating scenarios Similarly to the class models, the scenarios can be obtained by following an object-based methodology, or by using a catalogue of architectural patterns together with procedures to select the appropriate pattern. When following an object-based methodology, it is advisable to decompose the functionality of the role by means of *use cases* [6], and then build the corresponding sequence diagram for each of them. The inputs for creating the set of scenarios are the same as for creating the class diagrams, with the addition of the class diagrams themselves.

3.3 Example

In order to illustrate agent design, we consider a system for segmenting users according to common interests, which is inspired in [9]. It deals with a segmentation of users in groups of common interests, and is meant to be used for marketing purposes, such as for offering specific products only to potentially interested users. The system is conceived as a multi-agent system in which each (human) user is represented by a *personal clerk*, which groups with other personal clerks to form a community. Such a community is represented by a *clerk of community*, and relates to one subject. This segmentation of interests helps to control the quality of documents provided to users, as explained below.

A community can be seen as a source of information to which users subscribe to obtain relevant information for their interests. Once subscribed, a user begins to receive information from the community. This information originates from members of the community or from other sources of information such as forums of news and other communities (different communities can exchange information as long as it is authorised by the administrator of the system). The information that the users receive passes through a series of filters to ensure its quality.

When a user suggests information to the community, the community first compares the suggestion with the *community profile*. If the information matches the community profile, the document is evaluated by a set of members of the community. However, before being evaluated by their users, each of their personal clerks decides, on their own, whether the document is interesting to its user. In the affirmative case, the evaluation request is presented to the user, so that he evaluates the document. In the negative case, a vote against the document is produced. The suggested document is approved only if

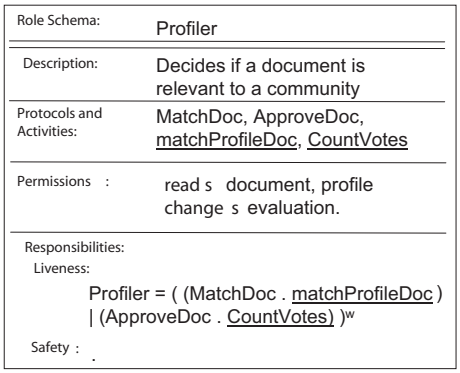


Fig. 3. The *Profiler* role

most of the consulted members vote in favour of the document, and the positive and negative evaluations are registered and used in the acceptance of future suggestions.

The permanence of members in a community is subject to the following restrictions: first, users who have suggested many documents evaluated negatively are expelled, since their interests are not in accordance with those of the community; and second, users who evaluate too many documents negatively are also expelled, since they have not shown interest in the type of information provided by the community.

Community clerks and personal clerks describe their interests by means of a *profile*, which can take the form of a set of documents (the last documents evaluated positively), keywords or categories. The keywords and categories of a clerk can be modified by its user. Users connect with their clerks by means of a Web interface that allows them to: suggest documents, evaluate documents, see documents, and see statistics of operation.

In Gaia, during the analysis, the main roles of the system are obtained from the basic skills of the system organisation. Since in our example a basic skill refers to determining if a document can be of interest to the community, in the following we assume the existence of a role in charge of determining if a proposed document is relevant to a community, hereafter called the *Profiler*, and whose role schema is shown in Figure 3. Additionally, we assume that, according to the organisational structure of the system, the *Profiler* role is completely subordinated to the authority of the community clerk, so that its behaviour can be modelled as a process of receiving orders, performing activities related to accomplishing these orders, and replying with the results produced by the activities. Considering its purely reactive behaviour, we conclude that the *Profiler* role can be modelled by means of the subsumption architecture [2].

The subsumption architecture The subsumption architecture [2, 13] is a reactive architecture developed by Brooks, that bases its function on the existence of *behaviours* and their relationships of *inhibition*. Each behaviour is intended to achieve a specific task and associates perceptual inputs with actions. For example, in the case of a vehicle control application, the behaviour, *changing direction if an obstacle is found in front*, associates the perceptual input, *an obstacle is in front*, with the task, *change direction*. To pursue its aim, each behaviour continually senses the environment until the environ-

mental state matches its associated perceptual input, in which case the associated action is performed. In this example, the environment is continually sensed until an obstacle is detected in front of the vehicle, in which case the action of changing direction is performed. However, since an environment state may match more than one behaviour, an *inhibition relation* is used to specify priorities. According to this inhibition relation, the behaviours are arranged into layers, with lower layers capable of inhibiting upper layers, and the higher the layer the more abstract its behaviour. For example, in the case of vehicle control, the behaviour corresponding to *collision avoidance* occupies a lower layer than that of the behaviour corresponding to *reach the destination*, since avoiding an obstacle has priority over reaching the destination. Therefore, using the subsumption architecture, we construct the structure and functionality models corresponding to the *Profiler* role, as described below.

Structure model The *Profiler* interacts with its environment by means of interaction protocols. As can be observed in its schema (Figure 3), the *Profiler* role participates in two protocols: *MatchDoc* and *ApproveDoc*. According to this, the environment perceived by the *Profiler* can be described as the set of tuples, *(command, content1, content2)*, where: *command* is an identifier of the type of protocol (for example, *Match* for the *MatchDoc* protocol, or *Approve* for the *ApproveDoc* protocol); *content1* is a document; and *content2* is an *evaluation* if *command* is *Match*, or *nil* otherwise (this corresponds to the outputs of these protocols, as stated in the interaction model). Accordingly, there are two behaviours for this role, as described below.

b1 *if (Match, d, e) is perceived then execute MatchProfileDoc(d) and continue the execution of protocol MatchDoc.*

b2 *if (Approve, d, e) is perceived then execute IsApproved(d, e) and continue the execution of protocol ApproveDoc.*

Here, *MatchProfileDoc* and *IsApproved* are activities of the *Profiler* role, dealing with matching a document to the community profile, and approving a document, respectively, as is stated in its role description. Note that, in this particular case, the inhibition relationship is irrelevant, since no perceived state can match both *b1* and *b2*.

The class diagram for the structural model is obtained by enhancing the class diagram of the subsumption pattern, with the particular characteristics of the *Profiler* role, resulting in the diagram shown in Figure 4. The enhancements consist in the elimination of the original *Inhibitor* class (since no inhibition relationship is required), the description of the information perceived (*Percept* class), and the representation of the *Inhibitor* activities as actions of behaviours.

Functionality model The operation of the *Profiler* is so simple that only one scenario is needed to describe its functionality. Such a scenario describes the dynamics followed by the classes to accomplish the functionality of the role, and is expressed by a sequence diagram adapted from the subsumption pattern. This sequence diagram, which is shown in Figure 5, is easier to interpret if we consider that the *Profiler* perceives the environment by receiving messages and interpreting their content, and affects the environment by sending messages.

3.4 Agents that play more than one role

In the previous description of the agent design phase, the agent in question encompasses only one role. However, in general this is not the case, since a given agent may encom-

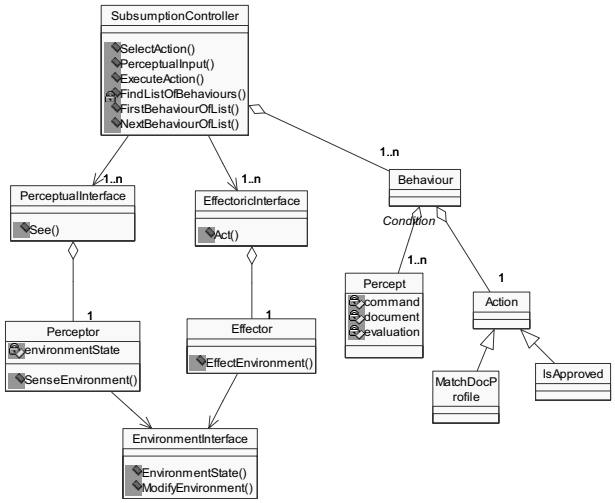


Fig. 4. Class diagram of the structure model

pass more than one role for reasons of efficiency, physical distribution and facility of implementation. For instance, in our example the same agent can perform the *Profiler* and *Community Clerk* roles for incrementing the efficiency of the system. When more than one role is included in one agent, the agent design is applied to each of the roles, and the resulting models analysed to identify common classes. These common classes can then be merged, resulting in a reduction of the number of classes, particularly when the roles are modelled by the same agent architecture. However, an excessive merging of classes increases the coupling between the roles, and can bring some difficult problems. For example, in the case of the subsumption architecture, when merging the behaviours of two different roles, a new inhibition relationship (that considers the behaviours of the two roles) must be determined.

Now that we have described the agent design phase, we also need to consider how it might be incorporated into the methodological process. We do this below.

4 Iterative development

The main idea behind applying an *iterative* approach to the development of a system is to divide the development into simpler, and thus more manageable, units. Each unit is then analysed, designed and implemented to produce an executable deliverable which extends, in functionality, the previous deliverable, in such a way that the final executable

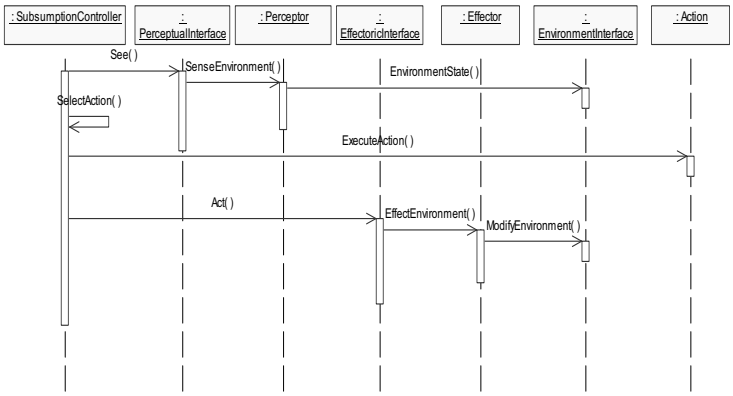


Fig. 5. Sequence diagram of the functionality model

deliverable encompasses all the functionality expected from the system. This reduces the risk of producing the *wrong* system, and of exceeding delivery times.

The iterative approach decomposes the development activities in two dimensions. The first dimension corresponds to the typical ways of developing software, which in our case consists of decomposing the development into analysis, architectural design, detailed design, and agent design. The second dimension relies on the use of *iterations*, which consist of the application, to some degree, of all the phases mentioned above, and several iterations are carried out during the development of the whole system. Early iterations focus on the first phases, analysis and architectural design, while subsequent iterations focus on the last phase, agent design.

While the decomposition into phases is common for all applications, iteration decomposition varies from application to application, in terms of work dedicated to each phase, number of iterations, and, more importantly, purpose. As a general rule, the larger the system, the more iterations are needed. In addition, the actual decomposition of the development cycle into iterations is guided by the functionality of the system. This means that the functionality of the system is divided into parts, one or more of which are assigned to an iteration, whose purpose is to accomplish that part of the functionality. The order in which the iterations must be carried out is important and must be established as part of the iteration decomposition, since the most critical and important parts of functionality must be considered first, to obtain earlier user feedback and foresee possible changes in delivery times.

Some guidelines for iteration decomposition are as follows: the set of iterations must cover all the functionality expected from the system; the early iterations in the de-

Iteration	Parts	Functionalities
1	Part 1	Approve new information
2	Part 5 and Part 6	Register new users and expel users
3	Part 3 and Part 4	Create communities and Eliminate communities
4	Part 2	Exchange information

Table 1. Iteration decomposition of the case study

composition must be occupied by those functionalities that form the core of the system (such as critical processes that are poorly described), or by those functionalities that involve a high risk of creating the wrong system or delaying the delivery of the system (such as the employment of new technology); early iterations must provide insight of most of the system; and it is desirable to achieve a balance in the iterations, so that no iteration is too big nor too small.

To illustrate all this, we apply the iterative process to the problem described in Section 3.3. First, the functionality of the system can be divided into several parts.

Part 1 Approve new information: receiving, filtering, and disseminating documents suggested by users.
Part 2 Exchange information: the part dealing with the exchange of information between different communities or other sources of information.
Part 3 Create communities: the process of creating new communities in the system.
Part 4 Eliminate communities: elimination of unwanted communities from the system.
Part 5 Register new users: the process of accepting new users in the communities.
Part 6 Expel users: the part dealing with expelling unwanted users from communities.

This partition is used as the basis for the iteration decomposition of the system, which also takes into account the following two factors. First, it considers the potential size of each of the parts, and tries to keep a balance in the sizes of the iterations. Second, it prioritises the parts by their importance in the functionality of the system. In particular, it recognises *Part 1* as the core of the system, since it directly supports the accomplishment of the goal of the system, and is also the most complex part, involving several components of the system. The decomposition of the system into iterations is presented in Table 1, and the next section describes the first of these iterations.

4.1 First iteration

For the first iteration, the following are the roles that form the role model: the *Profiler*, which decides if a document is relevant to a community; the *PersonalClerk*, in charge of interacting with a user; the *CommunityClerk*, that act as the representative of a community; and the *Evaluator*, which decides if a document is interesting to a particular user. The role schema for the *Profiler* was presented previously in Figure 3, but lack of space prevents us from including the other schemas.

The interaction model consists of six protocols, which are summarised in Table 2. In this table, note the existence of the environmental entities *Recommender*, *Reader* and *Voter*. The resulting organisational structure is a tree with the CommunityClerk at

Protocol	Initiator	Collaborators	Description
ProposeDoc	Recommender	PersonalClerk	a user suggests a document
DisseminateDoc	CommunityClerk	PersonalClerk, Reader	an approved document is distributed
EvaluateDoc	CommunityClerk	Evaluator, Voter	a user and her clerk evaluate a document
MatchDoc	CommunityClerk	Profiler	a document is checked against a profile
ApproveDoc	CommunityClerk	Profiler	acceptance or rejection of a document
ChangeProfile	Reader	PersonalClerk	a user changes her profile

Table 2. Interactions in the first iteration

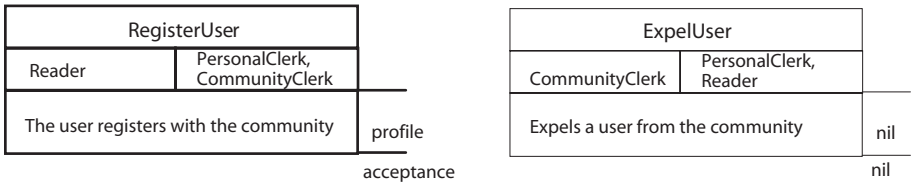
the root, and a peer branch to the PersonalClerk, and authority branches to the Profiler and to the Evaluator. Lastly, and assuming that each role is implemented by a different agent, the first iteration ends with the design of each of the four corresponding agents. In Section 3.3 we presented the design of the *Profiler*, but space constraints prevent us from presenting the design of the other agents.

4.2 Second iteration

The second iteration addresses another part of the functionality of the system, and consists of enhancing the results of the previous iteration in terms of adding elements to the models, extending elements or modifying them. For example, in the case of the preliminary role model, new roles can be added and existing roles can be modified to include interaction with the new roles. In this example, the second iteration deals with the registration of new users and the expulsion of inadequate users, as described below.

In order to incorporate the functionality of the second iteration, no new roles are required, since it can be carried out by the *PersonalClerk* and *CommunityClerk*. However, the incorporation of new protocols to the preliminary interaction model is necessary to cope with these tasks. Such protocol descriptions are shown in Figure 4.2, the first of which refers to the registration of new users and the second to the expulsion of users.

The role model obtained in the first iteration is updated to incorporate these introduced protocols, specifically, the *PersonalClerk* and *CommunityClerk* schemas should be modified to include them. In contrast, the organisational structure needs no modification, since the introduced protocols do not change either the communication paths or the authority relationships between the roles. Finally, since no new roles were introduced in the second iteration, the agent design phase consists only of updating the design of those roles that were affected, namely the *PersonalClerk* and *CommunityClerk*.



The completion of the example would require the accomplishment of the rest of the iterations, but since they are very similar, they are not included in this paper.

5 Conclusions

Gaia has been recognised as a valuable methodology for the development of open complex systems based on the multi-agent approach. However, in order to be used in the development of real world systems, Gaia needs to be extended in several respects. For example, in [7], Garcia Ojeda et al. extend Gaia to consider *implementation* issues, by refining the Gaia models using the AUML modelling language. Similarly, in [3], Cernuzzi and Zambonelli extend Gaia to incorporate *maintainability* issues, particularly for making designs more adaptable to modifications in the organisational structure caused by changes in requirements and environment.

In this paper, we have extended the Gaia methodology in two directions. The first extension deals with the design of the internal composition of agents in a multi-agent system. For this, we have presented an agent design phase that follows the organisational design phase of Gaia and produces an object-based specification from which an implementation can follow. This agent design phase relies on the use of agent architectures as a means to specify the classes that form an agent and the way they interact to fulfil its behaviour. As an example of the application of the design phase, we have presented the design of a reactive agent which is based on the subsumption architecture. Although simple, this example shows how entities acting as service providers can be agentified, which can also be applied to legacy software.

The benefits of this approach are that the resulting design phase does not depend on a specific agent architecture, but developers are free to select the architecture that best models a given agent. This contrasts with other approaches, such as Tropos [1], which consider only one type of architecture in agent design (a BDI architecture). Additionally, the existence of pre-defined solutions (architectural patterns) reduces the work involved in designing a new agent model every time, as with the INGENIAS methodology [8].

Nevertheless, the drawback of this approach is that agent architectures are not based on organisational concepts (like those on which Gaia's organisational design is based), so it is necessary to *adapt* them. However, since this process is essentially independent of the domain, it can be pre-determined and then reused.

The second extension to Gaia presented in this paper provides Gaia with a flexible methodological process that facilitates the development of large systems. This enhancement consists of decomposing the development into iterations, each of which corresponds to a part of the functionality of the system and consists of the analysis, organisational design and agent design phases.

The benefits of this iterative process are multiple. First, it enables the production of executable deliverables from early stages of the development. Second, it explicitly prioritises those parts of the system that are critical, unclear or involve technological risks. Finally, it speeds up the development by incrementing the parallelism in development tasks. The full potential of this iterative process, however, is limited by the lack of an implementation phase, which is subject of further work.

References

1. P. Bresciani, A. Perini, P. Giorgini, F. Giunchiglia, and J. Mylopoulos. Tropos: An agent-oriented software development methodology. *Autonomous Agents and Multi-Agent Systems*, (8):203–236, 2004.
2. R. A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, 2(1):14–23, 1986.
3. L. Cernuzzi and F. Zambonelli. Dealing with adaptive multiagent systems organizations in the gaia methodology. In *6th International Workshop on Agent-Oriented Software Engineering (AOSE 2005)*, 2006.
4. S. A. DeLoach. Modeling organizational rules in the multiagent systems engineering methodology. In R. Cohen and B. Spencer, editors, *15th Canadian Conference on Artificial Intelligence*, volume 2338 of *Lecture Notes in Artificial Intelligence*. Springer, 2002.
5. J. Ferber, O. Gutknecht, and F. Michel. From agents to organisations: An organizational view of multi-agent systems. In P. Giorgini, J. Muller, and J. Odell, editors, *Proceedings of the Fourth International Workshop in Agent-oriented Software Engineering (AOSE–2003)*, volume 2935 of *Lecture Notes in Computer Science*, pages 214–230. Springer-Verlag, 2004.
6. M. Fowler and K. Scott. *UML Destilled: Applying the Standard Object Modeling Language*. Addison-Wesley, 1997.
7. J. Garcia-Ojeda, A. Arenas, and J. Perez-Alcazar. Paving the way for implementing multiagent systems. In *6th International Workshop on Agent-Oriented Software Engineering (AOSE 2005)*, 2006.
8. J. Gomez-Sanz and J. Pavon. Agent oriented software engineering with INGENIAS. In V. Marik, J. Muller, and M. Pechoucek, editors, *Multi-Agent Systems and Applications III*, pages 394–403. Springer-Verlag, 2003.
9. J. Gomez-Sanz, J. Pavon, and A. D. Carrasco. The psi3 agent recommender system. In *International Conference on Web Engineering (ICWE 2003)*, volume 2722 of *Lecture Notes in Computer Science*, pages 30–39. Springer-Verlag, 2003.
10. J. Gonzalez-Palacios. *Increasing Accessibility in Agent-Oriented Methodologies*. PhD thesis, University of Southampton, 2006. To appear.
11. I. Jacobson, J. Rumbaugh, and G. Booch. *The Unified Software Development Process*. Addison-Wesley, 1999.
12. A. Omicini. SODA: Societies and infraestructures in the analysis and design of agent-based systems. In P. Ciancarini and M. J. Wooldridge, editors, *Proceedings of the First International Workshop in Agent-oriented Software Engineering (AOSE–2000)*, volume 1957 of *Lecture Notes in Artificial Intelligence*, pages 185–194. Springer-Verlag, 2001.
13. M. Wooldridge. *Multiagent Systems: a Modern Approach to Distributed Artificial Intelligence*, chapter Intelligent Agents. MIT Press, 1999.
14. M. Wooldridge and N. R. Jennings. Intelligent agents: theory and practice. *The Knowledge Engineering Review*, 10(2):115–152, 1995.
15. F. Zambonelli, N. R. Jennings, and M. Wooldridge. Organisational abstractions for the analysis and design of multi-agent systems. In *First International Workshop on Agent-Oriented Software Engineering*, pages 127–141, 2000.
16. F. Zambonelli, N. R. Jennings, and M. Wooldridge. Developing multiagent systems: The Gaia methodology. *ACM Transactions on Software Engineering and Methodology*, 12(3):317–370, 2003.