, , 1–27 ()

The Gaia Methodology for Agent-Oriented Analysis and Design

MICHAEL WOOLDRIDGE m.j.wooldridge@qmw.ac.uk
Department of Electronic Engineering, Queen Mary and Westfield College, London E1 4NS, UK

NICHOLAS R. JENNINGS n.r.jennings@qmw.ac.uk
Department of Electronic Engineering, Queen Mary and Westfield College, London E1 4NS, UK

DAVID KINNY dnk@cs.mu.oz.au
Department of Computer Science, University of Melbourne, Parkville 3052, Australia

Received

Abstract. This article presents Gaia: a methodology for agent-oriented analysis and design. The Gaia methodology is both general, in that it is applicable to a wide range of multi-agent systems, and comprehensive, in that it deals with both the macro-level (societal) and the micro-level (agent) aspects of systems. Gaia is founded on the view of a multi-agent system as a computational organisation consisting of various interacting roles. We illustrate Gaia through a case study (an agent-based business process management system).

1. Introduction

Progress in software engineering over the past two decades has been made through the development of increasingly powerful and natural high-level abstractions with which to model and develop complex systems. Procedural abstraction, abstract data types, and, most recently, objects and components are all examples of such abstractions. It is our belief that agents represent a similar advance in abstraction: they may be used by software developers to more naturally understand, model, and develop an important class of complex distributed systems.

If agents are to realise their potential as a software engineering paradigm, then it is necessary to develop software engineering techniques that are specifically tailored to them. Existing software development techniques (for example, object-oriented analysis and design [2, 6]) are unsuitable for this task. There is a fundamental mismatch between the concepts used by object-oriented developers (and indeed, by other mainstream software engineering paradigms) and the agent-oriented view [32, 34]. In particular, extant approaches fail to adequately capture an agent's flexible, autonomous problem-solving behaviour, the richness of an agent's interactions, and the complexity of an agent system's organisational structures. For these reasons, this article introduces a methodology called Gaia, which has been specifically tailored to the analysis and design of agent-based systems[1].

The remainder of this article is structured as follows. We begin, in the following subsection, by discussing the characteristics of applications for which we believe Gaia is appropriate. Section 2 gives an overview of the main concepts used in Gaia. Agent-based analysis is discussed in section 3, and design in section 4. The use of Gaia is illustrated by means of a case study in section 5, where we show how it was applied to the design of a real-world agent-based system for business process management [20]. Related work is discussed in section 6, and some conclusions are presented in section 7.

Domain Characteristics

Before proceeding, it is worth commenting on the scope of our work, and in particular, on the characteristics of domains for which we believe Gaia is appropriate. It is intended that Gaia be appropriate for the development of systems such as ADEPT [20] and ARCHON [19]. These are large-scale real-world applications, with the following main characteristics:

- Agents are coarse-grained computational systems, each making use of significant computational resources (think of each agent as having the resources of a UNIX process).
- It is assumed that the goal is to obtain a system that maximises some global quality measure, but which may be sub-optimal from the point of view of the system components. Gaia is *not* intended for systems that admit the possibility of true conflict[2].
- Agents are heterogeneous, in that different agents may be implemented using different programming languages, architectures, and techniques. We make no assumptions about the delivery platform.
- The organisation structure of the system is static, in that inter-agent relationships do not change at run-time.
- The abilities of agents and the services they provide are static, in that they do not change at run-time.
- The overall system contains a comparatively small number of different agent types (less than 100).

Gaia deals with both the macro (societal) level and the micro (agent) level aspects of design. It represents an advance over previous agent-oriented methodologies in that it is neutral with respect to both the target domain and the agent architecture (see section 6 for a more detailed comparison).

2. A Conceptual Framework

Gaia is intended to allow an analyst to go systematically from a statement of requirements to a design that is sufficiently detailed that it can be implemented directly. Note that we view the requirements capture phase as being independent of the paradigm used for analysis and design. In applying Gaia, the analyst moves from abstract to increasingly concrete concepts. Each successive move introduces greater implementation bias, and shrinks the space of possible systems that could be implemented to satisfy the original requirements statement. (See [21, pp216-222] for a discussion of implementation bias.) Analysis and design can be thought of as a process of developing increasingly detailed *models* of the system to be constructed. The main models used in Gaia are summarised in Figure 1

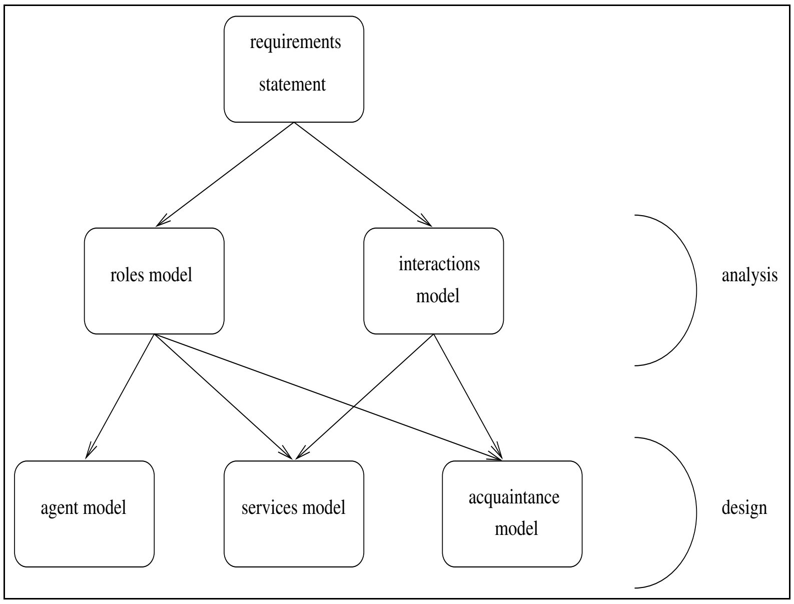

Figure 1. Relationships between Gaia's models

Gaia borrows some terminology and notation from object-oriented analysis and design, (specifically, FUSION [6]). However, it is not simply a naive attempt to apply such methods to agent-oriented development. Rather, it provides an agent-specific set of concepts through which a software engineer can understand and model a complex system. In particular, Gaia encourages a developer to think of building agent-based systems as a process of *organisational design*.

The main Gaian concepts can be divided into two categories: *abstract* and *concrete*; abstract and concrete concepts are summarised in Table 1. Abstract entities are those used during analysis to conceptualise the system, but which do not necessarily have any *direct* realisation within the system. Concrete entities, in contrast, are used within the design process, and will typically have direct counterparts in the run-time system.

3. Analysis

The objective of the analysis stage is to develop an understanding of the system and its structure (without reference to any implementation detail). In our case, this understanding is captured in the system's *organisation*. We view an organisation as a collection of roles, that stand in certain relationships to one another, and that take part in systematic, institutionalised patterns of interactions with other roles — see Figure 2.

The most abstract entity in our concept hierarchy is the *system*. Although the term "system" is used in its standard sense, it also has a related meaning when talking about an

Table 1. Abstract and concrete concepts in Gaia

Abstract Concepts	Concrete Concepts
Roles	Agent Types
Permissions	Services
Responsibilities	Acquaintances
Protocols	
Activities	
Liveness properties	
Safety properties	

agent-based system, to mean “society” or “organisation”. That is, we think of an agent-based system as an artificial society or organisation.

The idea of a system as a society is useful when thinking about the next level in the concept hierarchy: *roles*. It may seem strange to think of a computer system as being defined by a set of roles, but the idea is quite natural when adopting an organisational view of the world. Consider a human organisation such as a typical company. The company has roles such as “president”, “vice president”, and so on. Note that in a concrete *realisation* of a company, these roles will be *instantiated* with actual individuals: there will be an individual who takes on the role of president, an individual who takes on the role of vice president, and so on. However, the instantiation is not necessarily static. Throughout the company’s lifetime, many individuals may take on the role of company president, for example. Also, there is not necessarily a one-to-one mapping between roles and individuals. It is not unusual (particularly in small or informally defined organisations) for one individual to take on many roles. For example, a single individual might take on the role of “tea maker”, “mail fetcher”, and so on. Conversely, there may be many individuals that take on a single role, e.g., “salesman”[3].

A role is defined by four attributes: *responsibilities*, *permissions*, *activities*, and *protocols*. *Responsibilities* determine functionality and, as such, are perhaps the key attribute associated with a role. An example responsibility associated with the role of company president might be calling the shareholders meeting every year. Responsibilities are divided into two types: *liveness properties* and *safety properties* [27][4]. Liveness properties intuitively state that “something good happens”. They describe those states of affairs that an agent must bring about, given certain environmental conditions. In contrast, safety properties are *invariants*. Intuitively, a safety property states that “nothing bad happens” (i.e., that an acceptable state of affairs is maintained across all states of execution). An example might be “ensure the reactor temperature always remains in the range 0-100”.

In order to realise responsibilities, a role has a set of *permissions*. Permissions are the “rights” associated with a role. The permissions of a role thus identify the resources that are available to that role in order to realise its responsibilities. In the kinds of system that we have typically modelled, permissions tend to be *information resources*. For example, a role might have associated with it the ability to read a particular item of information,

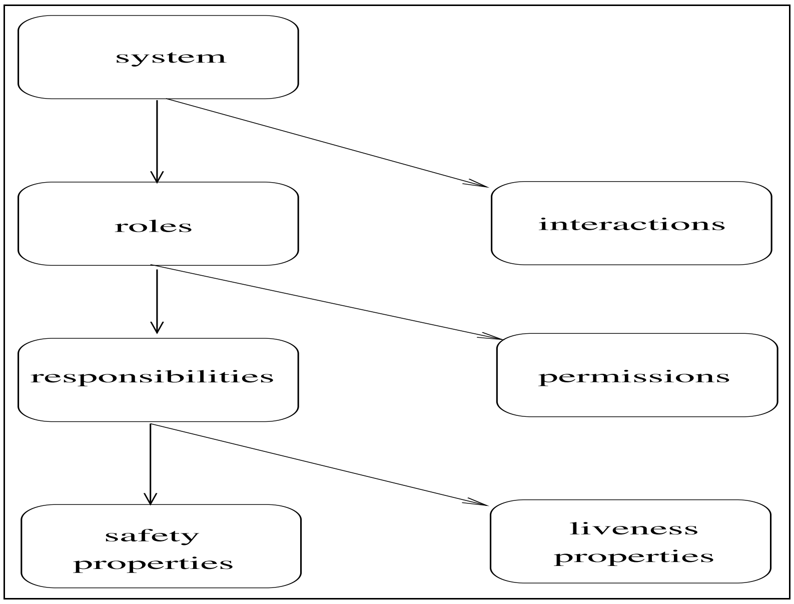

Figure 2. Analysis Concepts

or to modify another piece of information. A role can also have the ability to *generate* information.

The *activities* of a role are computations associated with the role that may be carried out by the agent without interacting with other agents. Activities are thus “private” actions, in the sense of [28].

Finally, a role is also identified with a number of *protocols*, which define the way that it can interact with other roles. For example, a “seller” role might have the protocols “Dutch auction” and “English auction” associated with it; the Contract Net Protocol is associated with the roles “manager” and “contractor” [30].

Thus, the organisation model in Gaia is comprised of two further models: the *roles model* (section 3.1) and the *interaction model* (section 3.2).

3.1. The Roles Model

The roles model identifies the key roles in the system. Here a role can be viewed as an abstract description of an entity’s expected function. In other terms, a role is more or less identical to the notion of an *office* in the sense that “prime minister”, “attorney general of the United States”, or “secretary of state for Education” are all offices. Such roles (or offices) are characterised by two types of attribute [5]:

- *The permissions/rights associated with the role.*

A role will have associated with it certain permissions, relating to the type and the amount of resources that can be exploited when carrying out the role. In our case, these aspects are captured in an attribute known as the role's *permissions*.

- *The responsibilities of the role.*

  A role is created in order to *do* something. That is, a role has a certain functionality. This functionality is represented by an attribute known as the role's *responsibilities*.

Permissions The permissions associated with a role have two aspects:

- they identify the resources that can legitimately be used to carry out the role — intuitively, they say what *can* be spent while carrying out the role;
- they state the resource limits within which the role executor must operate — intuitively, they say what *can't* be spent while carrying out the role.

In general, permissions can relate to any kind of resource. In a human organisation, for example, a role might be given a monetary budget, a certain amount of person effort, and so on. However, in Gaia, we think of resources as relating only to the *information* or *knowledge* the agent has. That is, in order to carry out a role, an agent will typically be able to access certain information. Some roles might generate information; others may need to access a piece of information but not modify it, while yet others may need to modify the information. We recognise that a richer model of resources is required for the future, although for the moment, we restrict our attention simply to information.

Gaia makes use of a formal notation for expressing permissions that is based on the FUSION notation for operation schemata [6, pp26–31]. To introduce our concepts we will use the example of a COFFEEFILLER role (the purpose of this role is to ensure that a coffee pot is kept full of coffee for a group of workers). The following is a simple illustration of the permissions associated with the role COFFEEFILLER:

reads	*coffeeStatus*	*// full or empty*
changes	*coffeeStock*	*// stock level of coffee*

This specification defines two permissions for COFFEEFILLER: it says that the agent carrying out the role has permission to access the value *coffeeStatus*, and has permission to both read and modify the value *coffeeStock*. There is also a third type of permission, generates, which indicates that the role is the producer of a resource (not shown in the example). Note that these permissions relate to *knowledge that the agent has*. That is, *coffeeStatus* is a representation on the part of the agent of some value in the real world.

Some roles are *parameterised* by certain values. For example, we can generalise the COFFEEFILLER role by parameterising it with the coffee machine that is to be kept refilled. This is specified in a permissions definition by the supplied keyword, as follows:

reads	supplied *coffeeMaker*	*// name of coffee maker*
	coffeeStatus	*// full or empty*
changes	*coffeeStock*	*// stock level of coffee*

Table 2. Operators for liveness expressions

Operator	Interpretation
$x.y$	x followed by y
$x \mid y$	x or y occurs
$x*$	x occurs 0 or more times
$x+$	x occurs 1 or more times
x^{ω}	x occurs infinitely often
$[x]$	x is optional
$x \parallel y$	x and y interleaved

Responsibilities The *functionality* of a role is defined by its *responsibilities*. These responsibilities can be divided into two categories: *liveness* and *safety* responsibilities.

Liveness responsibilities are those that, intuitively, state that "something good happens". Liveness responsibilities are so called because they tend to say that "something will be done", and hence that the agent carrying out the role is still alive. Liveness responsibilities tend to follow certain patterns. For example, the *guaranteed response* type of achievement goal has the form "a request is always followed by a response". The *infinite repetition* achievement goal has the form "x will happen infinitely often". Note that these types of requirements have been widely studied in the software engineering literature, where they have proven to be necessary for capturing properties of *reactive* systems [27].

In order to illustrate the various concepts associated with roles, we will continue with our running example of the COFFEEFILLER role. Examples of liveness responsibilities for the COFFEEFILLER role might be:

- whenever the coffee pot is empty, fill it up;
- whenever fresh coffee is brewed, make sure the workers know about it.

In Gaia, liveness properties are specified via a *liveness expression*, which defines the "life-cycle" of the role. Liveness expressions are similar to the *life-cycle* expression of FUSION [6], which are in turn essentially regular expressions. Our liveness expressions have an additional operator, "ω", for *infinite repetition* (see Table 2 for more details). They thus resemble ω-regular expressions, which are known to be suitable for representing the properties of infinite computations [32].

Liveness expressions define the potential execution trajectories through the various activities and interactions (i.e., over the protocols) associated with the role. The general form of a liveness expression is:

$$\text{ROLENAME} = \textit{expression}$$

where ROLENAME is the name of the role whose liveness properties are being defined, and *expression* is the liveness expression defining the liveness properties of ROLENAME.

The atomic components of a liveness expression are either *activities* or *protocols*. An activity is somewhat like a method in object-oriented terms, or a procedure in a PASCAL-like language. It corresponds to a unit of action that the agent may perform, which does not involve interaction with any other agent. Protocols, on the other hand, are activities that *do* require interaction with other agents. To give the reader some visual clues, we write protocol names in a sans serif font (as in xxx), and use a similar font, underlined, for activity names (as in $\underline{\mathsf{yyy}}$).

To illustrate liveness expressions, consider again the above-mentioned responsibilities of the COFFEEFILLER role:

$$\textsc{CoffeeFiller} = (\mathsf{Fill}.\,\mathsf{InformWorkers}.\,\underline{\mathsf{CheckStock}}.\,\mathsf{AwaitEmpty})^{\omega}$$

This expression says that COFFEEFILLER consists of executing the protocol Fill, followed by the protocol InformWorkers, followed by the activity $\underline{\mathsf{CheckStock}}$ and the protocol AwaitEmpty. The sequential execution of these protocols and activities is then repeated infinitely often. For the moment, we shall treat the protocols simply as labels for interactions and shall not worry about how they are actually defined (this matter will be discussed in section 3.2).

Complex liveness expressions can be made easier to read by structuring them. A simple example illustrates how this is done:

$$\begin{aligned} \textsc{CoffeeFiller} &= (\mathsf{All})^{\omega} \\ \textsc{All} &= \mathsf{Fill}.\,\mathsf{InformWorkers}.\,\underline{\mathsf{CheckStock}}.\,\mathsf{AwaitEmpty} \end{aligned}$$

The semantics of such definitions are straightforward textual substitution.

In many cases, it is insufficient simply to specify the liveness responsibilities of a role. This is because an agent, carrying out a role, will be required to maintain certain *invariants* while executing. For example, we might require that a particular agent taking part in an electronic commerce application never spends more money than it has been allocated. These invariants are called *safety* conditions, because they usually relate to the absence of some undesirable condition arising.

Safety requirements in Gaia are specified by means of a list of predicates. These predicates are typically expressed over the variables listed in a role's permissions attribute. Returning to our COFFEEFILLER role, an agent carrying out this role will generally be required to ensure that the coffee stock is never empty. We can do this by means of the following safety expression:

- $\mathit{coffeeStock} > 0$

By convention, we simply list safety expressions as a bulleted list, each item in the list expressing an individual safety responsibility. It is implicitly assumed that these responsibilities apply across *all* states of the system execution. If the role is of infinitely long duration (as in the COFFEEFILLER example), then the invariants must *always* be true.

It is now possible to precisely define the Gaia roles model. A roles model is comprised of a set of *role schemata*, one for each role in the system. A role schema draws together the

Role Schema:	*name of role*
Description	*short English description of the role*
Protocols and Activities	*protocols and activities in which the role plays a part*
Permissions	*"rights" associated with the role*
Responsibilities	
Liveness	*liveness responsibilities*
Safety	*safety responsibilities*

Figure 3. Template for Role Schemata

Role Schema: COFFEEFILLER

Description:
This role involves ensuring that the coffee pot is kept filled, and informing the workers when fresh coffee has been brewed.

Protocols and Activities:
Fill, InformWorkers, <u>CheckStock</u>, AwaitEmpty

Permissions:

reads	supplied *coffeeMaker*	// *name of coffee maker*
	coffeeStatus	// *full or empty*
changes	*coffeeStock*	// *stock level of coffee*

Responsibilities
Liveness:

$$\text{COFFEEFILLER} = (\text{Fill}.\ \text{InformWorkers}.\ \underline{\text{CheckStock}}.\ \text{AwaitEmpty})^{\omega}$$

Safety:
- $\mathit{coffeeStock} > 0$

Figure 4. Schema for role COFFEEFILLER

various attributes discussed above into a single place (Figure 3). An exemplar instantiation is given for the COFFEEFILLER role in Figure 4. This schema indicates that COFFEEFILLER has permission to read the *coffeeMaker* parameter (that indicates which coffee machine the role is intended to keep filled), and the *coffeeStatus* (that indicates whether the machine is full or empty). In addition, the role has permission to change the value *coffeeStock*.

3.2. The Interaction Model

There are inevitably dependencies and relationships between the various roles in a multi-agent organisation. Indeed, such interplay is central to the way in which the system functions. Given this fact, interactions obviously need to be captured and represented in the

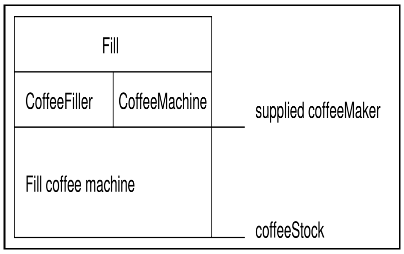

Figure 5. The Fill Protocol Definition

analysis phase. In Gaia, such links between roles are represented in the *interaction model*. This model consists of a set of *protocol definitions*, one for each type of inter-role interaction. Here a protocol can be viewed as an institutionalised pattern of interaction. That is, a pattern of interaction that has been formally defined and abstracted away from any particular sequence of execution steps. Viewing interactions in this way means that attention is focused on the essential nature and purpose of the interaction, rather than on the precise ordering of particular message exchanges (cf. the interaction diagrams of OBJECTORY [6, pp198–203] or the scenarios of FUSION [6]).

This approach means that a single protocol definition will typically give rise to a number of message interchanges in the run time system. For example, consider an English auction protocol. This involves multiple roles (sellers and bidders) and many potential patterns of interchange (specific price announcements and corresponding bids). However at the analysis stage, such precise instantiation details are unnecessary, and too premature.

A protocol definition consists of the following attributes:

- *purpose*: brief textual description of the nature of the interaction (e.g., "information request", "schedule activity" and "assign task");
- *initiator*: the role(s) responsible for starting the interaction;
- *responder*: the role(s) with which the initiator interacts;
- *inputs*: information used by the role initiator while enacting the protocol;
- *outputs*: information supplied by/to the protocol responder during the course of the interaction;
- *processing*: brief textual description of any processing the protocol initiator performs during the course of the interaction.

As an illustration, consider the Fill protocol, which forms part of the COFFEEFILLER role (Figure 5). This states that the protocol Fill is initiated by the role COFFEEFILLER and involves the role COFFEEMACHINE. The protocol involves COFFEEFILLER putting coffee in the machine named coffeeMaker, and results in COFFEEMACHINE being informed about the value of *coffeeStock*. We will see further examples of protocols in section 5.

3.3. The Analysis Process

The analysis stage of Gaia can now be summarised:

1. Identify the *roles* in the system. Roles in a system will typically correspond to:

    - individuals, either within an organisation or acting independently;
    - departments within an organisation; or
    - organisations themselves.

    Output: A prototypical roles model — a list of the key roles that occur in the system, each with an informal, unelaborated description.

2. For each role, identify and document the associated *protocols*. Protocols are the patterns of interaction that occur in the system between the various roles. For example, a protocol may correspond to an agent in the role of BUYER submitting a bid to another agent in the role of SELLER.

    Output: An interaction model, which captures the recurring patterns of inter-role interaction.

3. Using the protocol model as a basis, elaborate the roles model.

    Output: A fully elaborated roles model, which documents the key roles occurring in the system, their permissions and responsibilities, together with the protocols and activities in which they participate.

4. Iterate stages (1)–(3).

4. Design

The aim of a "classical" design process is to transform the abstract models derived during the analysis stage into models at a sufficiently low level of abstraction that they can be easily implemented. This is not the case with agent-oriented design, however. Rather, the aim in Gaia is to transform the analysis models into a sufficiently low level of abstraction that traditional design techniques (including object-oriented techniques) may be applied in order to implement agents. To put it another way, Gaia is concerned with how a society of agents cooperate to realise the system-level goals, and what is required of each individual agent in order to do this. Actually *how* an agent realises its services is beyond the scope of Gaia, and will depend on the particular application domain.

The Gaia design process involves generating three models (see Figure 1). The *agent* model identifies the *agent types* that will make up the system, and the *agent instances* that will be instantiated from these types. The *services model* identifies the main services that are required to realise the agent's role. Finally, the *acquaintance model* documents the lines of communication between the different agents.

Table 3. Instance Qualifiers

Qualifier	Meaning
n	there will be exactly n instances
$m..n$	there will be between m and n instances
$*$	there will be 0 or more instances
$+$	there will be 1 or more instances

4.1. The Agent Model

The purpose of the Gaia agent model is to document the various *agent types* that will be used in the system under development, and the *agent instances* that will realise these agent types at run-time.

An agent type is best thought of as a set of agent roles. There may in fact be a one-to-one correspondence between roles (as identified in the roles model — see section 3.1) and agent types. However, this need not be the case. A designer can choose to package a number of closely related roles in the same agent type for the purposes of convenience. Efficiency will also be a major concern at this stage: a designer will almost certainly want to optimise the design, and one way of doing this is to aggregate a number of agent roles into a single type. An example of where such a decision may be necessary is where the “footprint” of an agent (i.e., its run-time requirements in terms of processor power or memory space) is so large that it is more efficient to deliver a number of roles in a single agent than to deliver a number of agents each performing a single role. There is obviously a trade-off between the *coherence* of an agent type (how easily its functionality can be understood) and the efficiency considerations that come into play when designing agent types. The agent model is defined using a simple *agent type tree*, in which leaf nodes correspond to roles, (as defined in the roles model), and other nodes correspond to agent types. If an agent type t_1 has children t_2 and t_3, then this means that t_1 is composed of the roles that make up t_2 and t_3.

We document the agent instances that will appear in a system by annotating agent types in the agent model (cf. the qualifiers from FUSION [6]). An annotation n means that there will be exactly n agents of this type in the run-time system. An annotation $m..n$ means that there will be no less than m and no more than n instances of this type in a run-time system ($m < n$). An annotation $*$ means that there will be zero or more instances at run-time, and $+$ means that there will be one or more instances at run-time (see Table 3).

Note that inheritance plays no part in Gaia agent models. Our view is that agents are coarse grained computational systems, and an agent system will typically contain only a comparatively small number of roles and types, with often a one-to-one mapping between them. For this reason, we believe that inheritance has no useful part to play in the design of agent types. (Of course, when it comes to actually *implementing* agents, inheritance may be used to great effect, in the normal object-oriented fashion.)

4.2. The Services Model

As its name suggests, the aim of the Gaia services model is to identify the *services* associated with each agent role, and to specify the main properties of these services. By a service, we mean a *function* of the agent. In OO terms, a service would correspond to a method; however, we do not mean that services are available for other agents in the same way that an object's methods are available for another object to invoke. Rather, a service is simply a single, coherent block of activity in which an agent will engage. It should be clear there every activity identified at the analysis stage will correspond to a service, though not every service will correspond to an activity.

For each service that may be performed by an agent, it is necessary to document its properties. Specifically, we must identify the *inputs*, *outputs*, *pre-conditions*, and *post-conditions* of each service. Inputs and outputs to services will be derived in an obvious way from the protocols model. Pre- and post-conditions represent constraints on services. These are derived from the safety properties of a role. Note that by definition, each role will be associated with at least one service.

The services that an agent will perform are derived from the list of protocols, activities, responsibilities and the liveness properties of a role. For example, returning to the coffee example, there are four activities and protocols associated with this role: Fill, InformWorkers, <u>CheckStock</u>, and AwaitEmpty. In general, there will be at least one service associated with each protocol. In the case of <u>CheckStock</u>, for example, the service (which may have the same name), will take as input the stock level and some threshold value, and will simply compare the two. The pre- and post-conditions will both state that the coffee stock level is greater than 0. This is one of the safety properties of the role COFFEEFILLER.

The Gaia services model does *not* prescribe an implementation for the services it documents. The developer is free to realise the services in any implementation framework deemed appropriate. For example, it may be decided to implement services directly as methods in an object-oriented language. Alternatively, a service may be decomposed into a number of methods.

4.3. The Acquaintance Model

The final Gaia design model is probably the simplest: the *acquaintance model*. Acquaintance models simply define the communication links that exist between agent types. They do *not* define what messages are sent or when messages are sent — they simply indicate that communication pathways exist. In particular, the purpose of an acquaintance model is to identify any potential communication bottlenecks, which may cause problems at run-time (see section 5 for an example). It is good practice to ensure that systems are loosely coupled, and the acquaintance model can help in doing this. On the basis of the acquaintance model, it may be found necessary to revisit the analysis stage and rework the system design to remove such problems.

An agent acquaintance model is simply a graph, with nodes in the graph corresponding to agent types and arcs in the graph corresponding to communication pathways. Agent acquaintance models are *directed* graphs, and so an arc $a \rightarrow b$ indicates that a will send messages to b, but not necessarily that b will send messages to a. An acquaintance model may be derived in a straightforward way from the roles, protocols, and agent models.

4.4. The Design Process

The Gaia design stage can now be summarised:

1. Create an *agent model*:
   - aggregate roles into *agent types*, and refine to form an agent type hierarchy;
   - document the instances of each agent type using instance annotations.
2. Develop a services model, by examining activities, protocols, and safety and liveness properties of roles.
3. Develop an *acquaintance model* from the interaction model and agent model.

5. A Case Study: Agent-Based Business Process Management

This section briefly illustrates how Gaia can be applied, through a case study of the analysis and design of an agent-based system for managing a British Telecom business process (see [20] for more details). For reasons of brevity, we omit some details, and aim instead to give a general flavour of the analysis and design.

The particular application is providing customers with a quote for installing a network to deliver a particular type of telecommunications service. This activity involves the following departments: the *customer service division* (CSD), the *design division* (DD), the *legal division* (LD) and the various organisations who provide the out-sourced service of *vetting customers* (VCs). The process is initiated by a customer contacting the CSD with a set of requirements. In parallel to capturing the requirements, the CSD gets the customer vetted. If the customer fails the vetting procedure, the quote process terminates. Assuming the customer is satisfactory, their requirements are mapped against the service portfolio. If they can be met by a standard off-the-shelf item then an immediate quote can be offered. In the case of bespoke services, however, the process is more complex. DD starts to design a solution to satisfy the customer's requirements and whilst this is occurring LD checks the legality of the proposed service. If the desired service is illegal, the quote process terminates. Assuming the requested service is legal, the design will eventually be completed and costed. DD then informs CSD of the quote. CSD, in turn, informs the customer. The business process then terminates.

Moving from this process-oriented description of the system's operation to an organisational view is comparatively straightforward. In many cases there is a one to one mapping between departments and roles. CSD's behaviour falls into two distinct roles: one acting as an interface to the customer (CUSTOMERHANDLER, Figure 6), and one overseeing the process inside the organisation (QUOTEMANAGER, Figure 7). Thus, the VC's, the LD's, and the DD's behaviour are covered by the roles CUSTOMERVETTER (Figure 8), LEGALADVISOR (Figure 9), and NETWORKDESIGNER (Figure 10) respectively. The final role is that of the CUSTOMER (Figure 11) who requires the quote.

Role Schema: CUSTOMERHANDLER (CH)		
Description: Receives quote request from the customer and oversees process to ensure appropriate quote is returned.		
Protocols and Activities: <u>AwaitCall</u>, ProduceQuote, InformCustomer		
Permissions:		
reads supplied *customerDetails*	// *customer contact information*	
supplied *customerRequirements*	// *what customer wants*	
quote	// *completed quote or nil*	
Responsibilities		
Liveness:		
CUSTOMERHANDLER	=	(<u>AwaitCall</u>. GenerateQuote)$^{\omega}$
GENERATEQUOTE	=	(ProduceQuote. InformCustomer)
Safety:		
• **true**		

Figure 6. Schema for role CUSTOMERHANDLER

With the respective role definitions in place, the next stage is to define the associated interaction models for these roles. Here we focus on the interactions associated with the QUOTEMANAGER role. This role interacts with the CUSTOMER role to obtain the customer's requirements (GetCustomerRequirements protocol, Figure 12c) and with the CUSTOMERVETTER role to determine whether the customer is satisfactory (VetCustomer protocol, Figure 12a). If the customer proves unsatisfactory, these are the only two protocols that are enacted. If the customer is satisfactory then their request is costed. This costing involves enacting activity <u>CostStandardService</u> for frequently requested services or the CheckServiceLegality (Figure 12b) and CostBespokeService (Figure 12d) protocols for non-standard requests.

Having completed our analysis of the application, we now turn to the design phase. The first model to be generated is the agent model (Figure 13). This shows, for most cases, a one-to-one correspondence between roles and agent types. The exception is for the CUSTOMERHANDLER and QUOTEMANAGER roles which, because of their high degree of interdependence are grouped into a single agent type.

The second model is the services model. Again because of space limitations we concentrate on the QUOTEMANAGER role and the Customer Service Division Agent. Based on the QUOTEMANAGER role, seven distinct services can be identified (Table 3). From the GetCustomerRequirements protocol, we derive the service "obtain customer requirements". This service handles the interaction from the perspective of the quote manager. It takes the *customerDetails* as input and returns the *customerRequirements* as output (Figure 12c). There are no associated pre- or post-conditions.

The service associated with the VetCustomer protocol is "vet customer". Its inputs, derived from the protocol definition (Figure 12a), are the *customerDetails* and its outputs are *creditRating*. This service has a pre-condition that an appropriate customer vetter must be

Role Schema: QUOTEMANAGER (QM)

Description:
Responsible for enacting the quote process. Generates a quote or returns no quote (nil) if customer is inappropriate or service is illegal.

Protocols and Activities:
VetCustomer, GetcustomerRequirements, <u>CostStandardService</u>, CheckServiceLegality, CostBespokeService

Permissions:

reads	supplied *customerDetails*	// *customer contact information*
	supplied *customerRequirements*	// *detailed service requirements*
	creditRating	// *customer's credit rating*
	serviceIsLegal	// *boolean for bespoke requests*
generates	*quote*	// *completed quote or nil*

Responsibilities
Liveness:

$$\begin{aligned}\text{QUOTEMANAGER} &= \mathsf{QuoteResponse}\\ \text{QUOTERESPONSE} &= (\mathsf{VetCustomer} \parallel \mathsf{GetcustomerRequirements}) \mid\\ &\quad (\mathsf{VetCustomer} \parallel \mathsf{GetcustomerRequirements}).\\ &\quad \mathsf{CostService}\\ \text{COSTSERVICE} &= \underline{\mathsf{CostStandardService}} \mid\\ &\quad (\mathsf{CheckServiceLegality} \parallel \mathsf{CostBespokeService})\end{aligned}$$

Safety:

- $creditRating = bad \Rightarrow quote = nil$
- $serviceIsLegal = false \Rightarrow quote = nil$

Figure 7. Schema for role QUOTEMANAGER

available (derived from the TenderContract interaction on the VetCustomer protocol) and a post condition that the value of *creditRating* is non-null (because this forms part of a safety condition of the QUOTEMANAGER role).

The third service involves checking whether the customer is satisfactory (the *creditRating* safety condition of QUOTEMANAGER). If the customer is unsatisfactory then only the first branch of the QuoteRespose liveness condition (Figure 7) gets executed. If the customer is satisfactory, the CostService liveness route is executed.

The next service makes the decision of which path of the CostService liveness expression gets executed. Either the service is of a standard type (execute the service "produce standard costing") or it is a bespoke service in which case the CheckServiceLegality and CostBespokeService protocols are enacted. In the latter case, the protocols are associated with the service "produce bespoke costing". This service produces a non-nil value for *quote* as long as the *serviceIsLegal* safety condition (Figure 7) is not violated.

The final service involves informing the customer of the quote. This, in turn, completes the CUSTOMERHANDLER role.

Service	Inputs	Outputs	Pre-condition	Post-condition
obtain customer requirements	*customerDetails*	*customerRequirements*	**true**	**true**
vet customer	*customerDetails*	*creditRating*	customer vetter available	$creditRating \neq nil$
check customer satisfactory	*creditRating*	*continuationDecision*	$continuationDecision = nil$	$continuationDecision \neq nil$
check service type	*customerRequirements*	*serviceType*	$creditRating \neq bad$	$serviceType \in \{standard, bespoke\}$
produce standard service costing	*serviceType*, *customerRequirements*	*quote*	$serviceType = standard \wedge quote = nil$	$quote \neq nil$
produce bespoke service costing	*serviceType*, *customerRequirements*	*quote*, *serviceIsLegal*	$serviceType = bespoke \wedge quote = nil \wedge serviceIsLegal$	$(quote \neq nil) \vee (quote = nil \wedge \neg serviceIsLegal)$
inform customer	*customerDetails*, *quote*		**true**	customers know quote

Table 3. The services model

Role Schema: CUSTOMERVETTER (CV)		
Description: Checks credit rating of supplied customer.		
Protocols and Activities: VettingRequest, VettingResponse		
Permissions:		
reads	supplied *customerDetails*	*// customer contact information*
	customerRatingInformation	*// credit rating information*
generates	*creditRating*	*// credit rating of customer*
Responsibilities		
Liveness:		
CUSTOMERVETTER = (VettingRequest. VettingResponse)		
Safety:		
• *infoAvailable*(*customerDetails*, *customerRatingInformation*)		

Figure 8. Schema for role CUSTOMERVETTER

Role Schema: LEGALADVISOR (LA)		
Description: Determines whether given bespoke service request is legal or not.		
Protocols and Activities: LegalCheckRequest, LegalCheckResponse		
Permissions:		
reads	supplied *customerRequirements*	*// details of proposed service*
generates	*serviceIsLegal*	*// true or false*
Responsibilities		
Liveness:		
LEGALADVISOR = (LegalCheckRequest. LegalCheckResponse)		
Safety:		
• **true**		

Figure 9. Schema for role LEGALADVISOR

The final model is the acquaintance model, which shows the communication pathways that exist between agents (Figure 14).

Role Schema: NETWORKDESIGNER (ND)			
Description: Design and cost network to meet bespoke service request requirements.			
Protocols and Activities: CostingRequest, ProduceDesign, ReturnCosting			
Permissions:			
	reads	supplied *customerRequirements*	// *details of proposed service*
		serviceIsLegal	// *boolean*
	generates	*quote*	// *cost of realising service*
Responsibilities			
Liveness:			
NETWORKDESIGNER = (CostingRequest. ProduceDesign. ReturnCosting)			
Safety:			
• *serviceIsLegal = true*			

Figure 10. Schema for role NETWORKDESIGNER

Role Schema: CUSTOMER (CUST)			
Description: Organisation or individual requiring a service quote.			
Protocols and Activities: MakeCall, GiveRequirements			
Permissions:			
	generates	*customerDetails*	// *Owner of customer information*
		customerRequirements	// *Owner of customer requirements*
Responsibilities			
Liveness:			
CUSTOMER = (MakeCall. GiveRequirements)+			
Safety:			
• **true**			

Figure 11. Schema for role CUSTOMER

6. Related Work

In recent times there has been a surge of interest in agent-oriented modelling techniques and methodologies. The various approaches may be roughly grouped as follows:

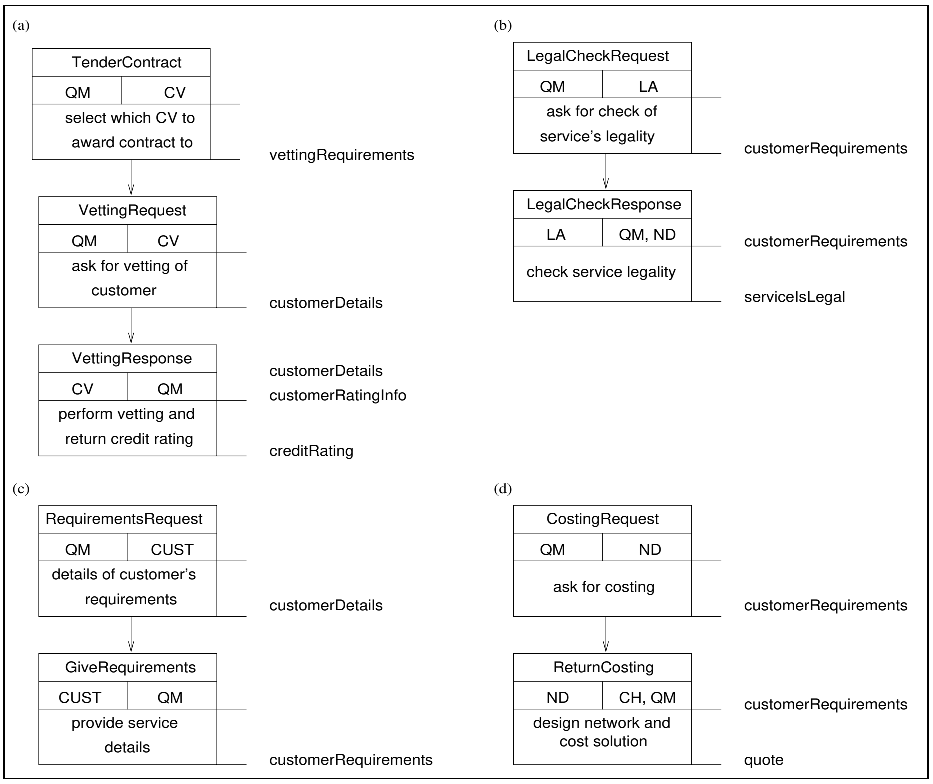

Figure 12. Definition of protocols associated with the QUOTEMANAGER role: (a) VetCustomer, (b) CheckServiceLegality, (c) GetCustomerRequirements, and (d) CostBespokeService.

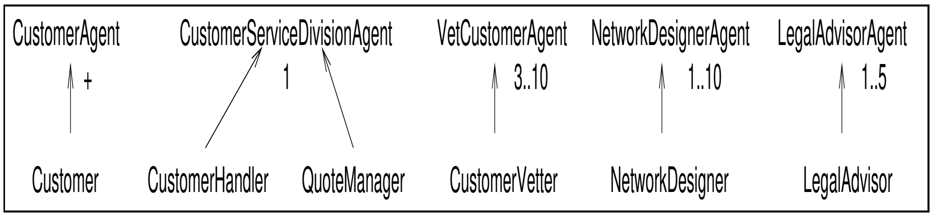

Figure 13. The agent model

- those such as [4, 24] which take existing OO modelling techniques or methodologies as their basis, seeking either to extend and adapt the models and define a methodology for their use, or to directly extend the applicability of OO methodologies and techniques, such as design patterns, to the design of agent systems,

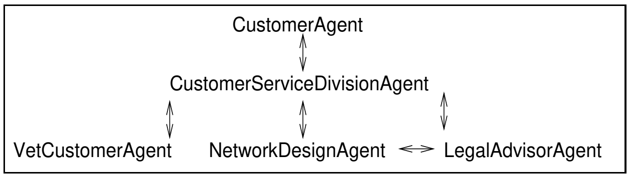

Figure 14. The acquaintance model

- those such as [3, 17] which build upon and extend methodologies and modelling techniques from knowledge engineering, providing formal, compositional modelling languages suitable for the verification of system structure and function,
- those which take existing formal methods and languages, for example Z [31], and provide definitions within such a framework that support the specification of agents or agent systems [26], and
- those which have essentially been developed *de novo* for particular kinds of agent systems. CASSIOPEIA [7], for example, supports the design of Contract Net [29] based systems and has been applied to Robot Soccer.

These design methodologies may also be divided into those that are essentially top-down approaches based on progressive decomposition of behaviour, usually building (as in Gaia) on some notion of role, and those such as CASSIOPEIA that are bottom-up approaches which begin by identifying elementary agent behaviours. A very useful survey which classifies and reviews these and other methodologies has also appeared [16].

The definition and use of various notions of role, responsibility, interaction, team and society or organization in particular methods for agent-oriented analysis and design has inherited or adapted much from more general uses of these concepts within multi-agent systems, including organization-focussed approaches such as [14, 9, 18] and sociological approaches such as [5]. However, it is beyond the scope of this article to compare the Gaia definition and use of these concepts with this heritage.

Instead, we will focus here on the relationship between Gaia and other approaches based that build upon OO techniques, in particular the kgr approach [24, 23]. But it is perhaps useful to begin by summarizing why OO modelling techniques and design methodologies themselves are not directly applicable to multi-agent system design.

6.1. Shortcomings of Object Oriented techniques

The first problem concerns the modelling of individual agents or agent classes. While there are superficial similarities between agents and objects, representing an agent as an object, i.e., as a set of attributes and methods, is not very useful because the representation is too fine-grained, operating at an inappropriate level of abstraction. An agent so represented may appear quite strange, perhaps exhibiting only one public method whose function is to receive messages from other agents. Thus an object model does not capture much useful information about an agent, and powerful OO concepts such as inheritance and aggregation become quite useless as a result of the poverty of the representation.

There are several reasons for this problem. One is that the agent paradigm is based on a significantly stronger notion of encapsulation than the object paradigm. An agent's internal state is usually quite opaque and, in some systems, the behaviours that an agent will perform upon request are not even made known until it advertises them within an active system. Related to this is the key characteristic of autonomy: agents cannot normally be created and destroyed in the liberal manner allowed within object systems and they have more freedom to determine how they may respond to messages, including, for example, by choosing to negotiate some agreement about how a task will be performed. As the underlying communication model is usually asynchronous there is no predefined notion of flow of control from one agent to another: an agent may autonomously initiate internal or external behaviour at any time, not just when it is sent a message. Finally, an agent's internal state, including its knowledge, may need to be represented in a manner that cannot easily be translated into a set of attributes; in any case to do so would constitute a premature implementation bias.

The second problem concerns the power of object models to adequately capture the relationships that hold between agents in a multi-agent system. While the secondary models in common use in OO methodologies such as use cases and interaction diagrams may usefully be adapted (with somewhat different semantics), the Object Model, which constitutes the primary specification of an OO system, captures associations between object classes that model largely static dependencies and paths of accessibility which are largely irrelevant in a multi-agent system. Only the instantiation relationship between classes and instances can be directly adopted. Important aspects of relationships between agents such as their repertoire of interactions and their degree of control or influence upon each other are not easily captured. The essential problem here is the uniformity and static nature of the OO object model. An adequate agent model needs to capture these relationships between agents, their dynamic nature, and perhaps also relationships between agents and non-agent elements of the system, including passive or abstract ones such as those modelled here as resources.

Both of these are problems concerning the suitability of OO modelling techniques for modelling a multi-agent system. Another issue is the applicability of OO methodologies to the process of analyzing and designing a multi-agent system. OO methodologies typically consist of an iterative refinement cycle of identifying classes, specifying their semantics and relationships, and elaborating their interfaces and implementation. At this level of abstraction, they appear similar to typical AO methodologies, which usually proceed by identifying roles and their responsibilities and goals, developing an organizational structure, and elaborating the knowledge and behaviours associated with a role or agent.

However, this similarity disappears at the level of detail required by the models, as the key abstractions involved are quite different. For example, the first step of object class identification typically considers tangible things, roles, organizations, events and even interactions as candidate objects, whereas these need to be clearly distinguished and treated differently in an agent-oriented approach. The uniformity and concreteness of the object model is the basis of the problem; OO methodologies provide guidance or inspiration rather than a directly useful approach to analysis and design.

6.2. Comparison with the KGR approach

The KGR approach [24, 23] was developed to fulfill the need for a principled approach to the specification of complex multi-agent systems based on the belief-desire-intention (BDI) technology of the Procedural Reasoning System (PRS) and the Distributed Multi-Agent Reasoning System (DMARS) [25, 8]. A key motivation of the work was to provided useful, familiar mechanisms for structuring and managing the complexity of such systems.

The first and most obvious difference between the approach proposed here and KGR is one of scope. Our methodology does not attempt to unify the analysis and abstract design of a multi-agent system with its concrete design and implementation with a particular agent technology, regarding the output of the analysis and design process as an abstract specification to which traditional lower-level design methodologies may be applied. KGR, by contrast, makes a strong architectural commitment to BDI architectures and proposes a design elaboration and refinement process that leads to directly executable agent specifications. Given the proliferation of available agent technologies, there are clearly advantages to a more general approach, as proposed here. However, the downside is that it cannot provide a set of models, abstractions and terminology that may be used uniformly throughout the system life cycle. Furthermore, there may be a need for iteration of the AO analysis and design process if the lower-level design process reveals issues that are best resolved at the AO level. A research problem for our approach and others like it is whether and how the adequacy and completeness of its outputs can be assessed independently of any traditional design process that follows.

A second difference is that in this work a clear distinction is made between the analysis phase, in which the roles and interaction models are fully elaborated, and the design phase, in which agent, services and acquaintance models are developed. The KGR approach does not make such a distinction, proposing instead the progressive elaboration and refinement of agent and interaction models which capture respectively roles, agents and services, and interactions and acquaintances. While both methodologies begin with the identification of roles and their properties, here we have chosen to model separately abstract agents (roles), concrete agents and the services they provide. KGR, on the other hand, employs a more uniform agent model which admits both abstract agents and concrete agent classes and instances and allows them to be organized within an inheritance hierarchy, thus allowing multiple levels of abstraction and the deferment of identification of concrete agent classes until late in the design process.

While both approaches employ responsibilities as an abstraction used to decompose the structure of a role, they differ significantly as to how these are represented and developed. Here responsibilities consist of safety and liveness properties built up from already identified interactions and activities. By contrast, KGR treats responsibilities as abstract goals, triggered by events or interactions, and adopts a strictly top-down approach to decomposing these into services and low level goals for which activity specifications may be elaborated. There are similarities however, for despite the absence of explicit goals in our approach, safety properties may be viewed as maintenance goals and liveness properties as goals of achievement. The notion of permissions, however, is absent from the KGR approach, whereas the notion of protocols may be developed to a much greater degree of detail, for example as in [22]. There protocols are employed as more generic descriptions of behaviour that may involve entities not modelled as agents, such as the coffee machine.

To summarize the key differences, the KGR approach, by making a commitment to implementation with a BDI agent architecture, is able to employ an iterative top-down approach to elaborating a set of models that describe a multi-agent system at both the macro- and micro-level, to make more extensive use of OO modelling techniques, and to produce executable specifications as its final output. The approach we have described here is a mixed top-down and bottom-up approach which employs a more fine-grained and diverse set of generic models to capture the result of the analysis and design process, and tries to avoid any premature commitment, either architectural, or as to the detailed design and implementation process which will follow. We envisage, however, that our approach can be suitably specialized for specific agent architectures or implementation techniques; this is a subject for further research.

7. Conclusions and Further Work

In this article, we have described Gaia, a methodology for the analysis and design of agent-based systems. The key concepts in Gaia are roles, which have associated with them responsibilities, permissions, activities, and protocols. Roles can interact with one another in certain institutionalised ways, which are defined in the protocols of the respective roles.

There are several issues remaining for future work.

- *Self-Interested Agents*.

  Gaia does not explicitly attempt to deal with systems in which agents may not share common goals. This class of systems represents arguably the most important application area for multi-agent systems, and it is therefore essential that a methodology should be able to deal with it.

- *Dynamic and open systems*.

  Open systems — in which system components may join and leave at run-time, and which may be composed of entities that a designer had no knowledge of at design-time — have long been recognised as a difficult class of system to engineer [15, 13].

- *Organisation structures*.

  Another aspect of agent-based analysis and design that requires more work is the notion of an organisational structure. At the moment, such structures are only *implicitly* defined within Gaia — within the role and interaction models. However, direct, explicit representations of such structures will be of value for many applications. For example, if agents are used to model large organisations, then these organisations will have an explicitly defined structure. Representing such structures may be the only way of adequately capturing and understanding the organisation's communication and control structures. More generally, the development of *organisation design patterns* might be useful for reusing successful multi-agent system structures (cf. [12]).

- *Cooperation Protocols*.

  The representation of inter-agent cooperation protocols within Gaia is currently somewhat impoverished. In future work, we will need to provide a much richer protocol specification framework.

- *International Standards*.

  Gaia was not designed with any particular standard for agent communication in mind (such as the FIPA agent communication language [11]). However, in the event of widescale industrial takeup of such standards, it may prove useful to adapt our methodology to be compatible with such standards.

- *Formal Semantics*.

  Finally, we believe that a successful methodology is one that is not only of pragmatic value, but one that also has a well-defined, unambiguous formal semantics. While the typical developer need never even be aware of the existence of such a semantics, it is nevertheless essential to have a precise understanding of what the concepts and terms in a methodology mean [33].

Acknowledgments

This article is a much extended version of [35]. We are grateful to the participants of the Agents 99 conference, who gave us much useful feedback.

Notes

1. In Greek mythology, Gaia was the mother Earth figure. More pertinently, Gaia is the name of an influential hypothesis put forward by the ecologist James Lovelock, to the effect that all the living organisms on the Earth can be understood as components of a single entity, which regulates the Earth's environment. The theme of many heterogeneous entities acting together to achieve a single goal is a central theme in multi-agent systems research [1], and was a key consideration in the the development of our methodology.
2. To be more precise, we believe such systems will require additional models over and above those that we outline in the current version of the methodology.

3. The third case, which we have not yet elaborated in the methodology, is that a single role represents the collective behaviour of a number of individuals. This view is important for modelling cooperative and team problem solving and also for bridging the gap between the micro and the macro levels in an agent-based system.
4. The most widely used formalism for specifying liveness and safety properties is temporal logic, and in previous work, the use of such formalism has been strongly advocated for use in agent systems [10]. Although it has undoubted strengths as a mathematical tool for expressing liveness and safety properties, there is some doubt about its viability as a tool for use by everyday software engineers. We have therefore chosen an alternative approach to temporal logic, based on regular expressions, as these are likely to be better understood by our target audience.
5. For the moment, we do not explicitly model the creation and deletion of roles. Thus roles are persistent throughout the system's lifetime. In the future, we plan to make this a more dynamic process

References

1. A. H. Bond and L. Gasser, editors. *Readings in Distributed Artificial Intelligence*. Morgan Kaufmann Publishers: San Mateo, CA, 1988.
2. G. Booch. *Object-Oriented Analysis and Design (second edition)*. Addison-Wesley: Reading, MA, 1994.
3. F. Brazier, B. Dunin-Keplicz, N. R. Jennings, and J. Treur. Formal specification of multi-agent systems: a real-world case. In *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS-95)*, pages 25–32, San Francisco, CA, June 1995.
4. Birgit Burmeister. Models and methodologies for agent-oriented analysis and design. In Klaus Fischer, editor, *Working Notes of the KI'96 Workshop on Agent-Oriented Programming and Distributed Systems*. 1996. DFKI Document D-96-06.
5. C. Castelfranchi. Commitments: from individual intentions to groups and organizations. In *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS-95)*, pages 41–48, San Francisco, CA, June 1995.
6. D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, and P. Jeremaes. *Object-Oriented Development: The* FUSION *Method*. Prentice Hall International: Hemel Hempstead, England, 1994.
7. Anne Collinot, Alexis Drogoul, and Philippe Benhamou. Agent oriented design of a soccer robot team. In *Proceedings of the Second International Conference on Multi-Agent Systems (ICMAS-96)*, Kyoto, Japan, 1996.
8. M. d'Inverno, D. Kinny, M. Luck, and M. Wooldridge. A formal specification of dMARS. In M. P. Singh, A. Rao, and M. J. Wooldridge, editors, *Intelligent Agents IV (LNAI Volume 1365)*, pages 155–176. Springer-Verlag: Berlin, Germany, 1997.
9. Jacques Ferber and Olivier Gutknecht. A meta-model for the analysis and design of organizations in multi-agent systems. In *Proceedings of the Third International Conference on Multi-Agent Systems (ICMAS-98)*, pages 128–135, Paris, France, 1998.
10. M. Fisher and M. Wooldridge. On the formal specification and verification of multi-agent systems. *International Journal of Cooperative Information Systems*, 6(1):37–65, 1997.
11. The Foundation for Intelligent Physical Agents. See `http://www.fipa.org/`.
12. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley: Reading, MA, 1995.
13. L. Gasser. Social conceptions of knowledge and action: DAI foundations and open systems semantics. *Artificial Intelligence*, 47:107–138, 1991.
14. L. Gasser, C. Braganza, and N. Hermann. MACE: A flexible testbed for distributed AI research. In M. Huhns, editor, *Distributed Artificial Intelligence*, pages 119–152. Pitman Publishing: London and Morgan Kaufmann: San Mateo, CA, 1987.
15. C. Hewitt. Open information systems semantics for distributed artificial intelligence. *Artificial Intelligence*, 47:79–106, 1991.
16. C. A. Iglesias, M. Garijo, and J. C. Gonzalez. A survey of agent-oriented methodologies. In J. P. Müller, M. P. Singh, and A. S. Rao, editors, *Intelligent Agents V — Proceedings of the Fifth International Workshop on Agent Theories, Architectures, and Languages (ATAL-98)*, Lecture Notes in Artificial Intelligence. Springer-Verlag, Heidelberg, 1999.
17. Carlos Iglesias, Mercedes Garijo, José C. González, and Juan R. Velasco. Analysis and design of multiagent systems using MAS-CommonKADS. In M. P. Singh, A. Rao, and M. J. Wooldridge, editors, *Intelligent Agents IV (LNAI Volume 1365)*, pages 313–326. Springer-Verlag: Berlin, Germany, 1998.

18. Toru Ishida, Les Gasser, and Makoto Yokoo. Organization self design of production systems. *IEEE Transactions on Knowledge and Data Engineering*, 4(2):123–134, April 1992.
19. N. R. Jennings, J. Corera, I. Laresgoiti, E. H. Mamdani, F. Perriolat, P. Skarek, and L. Z. Varga. Using ARCHON to develop real-world DAI applications for electricity transportation management and particle acceleration control. *IEEE Expert*, 11(6):60–88, December 1996.
20. N. R. Jennings, P. Faratin, M. J. Johnson, T. J. Norman, P. O'Brien, and M. E. Wiegand. Agent-based business process management. *International Journal of Cooperative Information Systems*, 5(2-3):105–130, 1996.
21. C. B. Jones. *Systematic Software Development using VDM (second edition)*. Prentice Hall, 1990.
22. D. Kinny. The AGENTIS agent interaction model. In J. P. Müller, M. P. Singh, and A. S. Rao, editors, *Intelligent Agents V — Proceedings of the Fifth International Workshop on Agent Theories, Architectures, and Languages (ATAL-98)*, Lecture Notes in Artificial Intelligence. Springer-Verlag, Heidelberg, 1999.
23. D. Kinny and M. Georgeff. Modelling and design of multi-agent systems. In J. P. Müller, M. Wooldridge, and N. R. Jennings, editors, *Intelligent Agents III (LNAI Volume 1193)*, pages 1–20. Springer-Verlag: Berlin, Germany, 1997.
24. D. Kinny, M. Georgeff, and A. Rao. A methodology and modelling technique for systems of BDI agents. In W. Van de Velde and J. W. Perram, editors, *Agents Breaking Away: Proceedings of the Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World, (LNAI Volume 1038)*, pages 56–71. Springer-Verlag: Berlin, Germany, 1996.
25. David Kinny. *The Distributed Multi-Agent Reasoning System Architecture and Language Specification*. Australian AI Institute, Level 6, 171 La Trobe Street, Melbourne, Australia, 1993.
26. M. Luck, N. Griffiths, and M. d'Inverno. From agent theory to agent construction: A case study. In J. P. Müller, M. Wooldridge, and N. R. Jennings, editors, *Intelligent Agents III (LNAI Volume 1193)*, pages 49–64. Springer-Verlag: Berlin, Germany, 1997.
27. A. Pnueli. Specification and development of reactive systems. In *Information Processing 86*. Elsevier Science Publishers B.V.: Amsterdam, The Netherlands, 1986.
28. Y. Shoham. Agent-oriented programming. *Artificial Intelligence*, 60(1):51–92, 1993.
29. R. G. Smith. The CONTRACT NET: A formalism for the control of distributed problem solving. In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence (IJCAI-77)*, Cambridge, MA, 1977.
30. R. G. Smith. *A Framework for Distributed Problem Solving*. UMI Research Press, 1980.
31. M. Spivey. *The Z Notation (second edition)*. Prentice Hall International: Hemel Hempstead, England, 1992.
32. M. Wooldridge. Agent-based software engineering. *IEE Proceedings on Software Engineering*, 144(1):26–37, February 1997.
33. M. Wooldridge and N. R. Jennings. Intelligent agents: Theory and practice. *The Knowledge Engineering Review*, 10(2):115–152, 1995.
34. M. Wooldridge and N. R. Jennings. Pitfalls of agent-oriented development. In *Proceedings of the Second International Conference on Autonomous Agents (Agents 98)*, pages 385–391, Minneapolis/St Paul, MN, May 1998.
35. M. Wooldridge, N. R. Jennings, and D. Kinny. A methodology for agent-oriented analysis and design. In *Proceedings of the Third International Conference on Autonomous Agents (Agents 99)*, pages 69–76, Seattle, WA, May 1999.