INESC
Software Engineering Group

A Methodology for Agent-Oriented Analysis and Design

Paulo Dias

Paulo.Dias@inesc.pt
Index

- Introduction
- Conceptual Framework
- Analysis
- Design
- Future work
- References
Introduction

Agent – a powerful abstraction

Progress in software engineering over the past two decades has primarily been made through the development of increasingly powerful and natural abstractions with which to model and develop complex systems. **Procedural abstraction, abstract data types, and, most recently, objects**, are all examples of such abstractions. In this way, **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 realize their potential as a software engineering paradigm, then it is necessary to develop software engineering techniques that are specifically tailored for them. There is a fundamental mismatch between the concepts used by object-oriented developers and the agent-oriented view. 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 organizational structures.
The methodology is appropriate for the development of systems (large-scale real-world applications) following these main characteristics:

- Agents are coarse-grained computational systems, each making use of significant computational resources.
- It is assumed that the goal is to obtain a system that maximizes some global quality measure, but which may be sub-optimal from the point of view of the system components.
- Agents are heterogeneous, in that different agents may be implemented using different programming languages and techniques.
- The overall system contains a comparatively small number of agents (less than 100).
The methodology borrows some terminology and notation from object-oriented analysis and design. However, it is not simply a naive attempt to apply such methods to agent-oriented development. Rather, the methodology provides an agent-specific set of concepts through which a software engineer can understand and model a complex system. In particular the methodology encourages a developer to think of building agent-based systems as a process of organizational design.
The main concepts can be divided in two categories: abstract and concrete.

**Abstract entities** are those used during analysis to conceptualize the system, but which do not necessarily have any direct realization within the system. **Concrete entities**, in contrast, are used within the design process, and will typically have direct counterparts in the run-time system.
The most abstract entity in the concept hierarchy is the system. This term is used to mean society or organization.

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 organizational view of the world.
Conceptual Framework

The role concept

A role is defined by three attributes:

- Responsibilities
- Permissions
- Protocols
Responsibilities determine functionality and, as such, are perhaps the key attribute associated with a role. Responsibilities are divided in two types: **liveness properties** and **safety properties**.

Liveness properties intuitively state that “something good will happen”. They describe those states of affairs that an agent must bring about, given certain environmental conditions.

In contrast, safety conditions are invariants. Intuitively, a safety property states that “nothing bad happens”.


Permissions are the “rights” associated with a role. The permissions of a role identify the resources that are available to that role in order to realize its responsibilities. In the kinds of system typically modeled, permissions tend to be information resources. For example, a role might have associated with it the ability to read a particular item of information, or to modify another piece of information. A role can also have the ability to generate information.
Conceptual Framework

Protocols

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.
Conceptual Framework

The concept hierarchy

Figure 1: Analysis Concepts
Conceptual Framework

Main models

Figure 2: Relationships between the methodology’s models
The objective of the analysis stage is to **develop an understanding of the system and its structure** (without reference to any implementation detail).

In this case, this understanding is captured in the system’s **organization**.

To define an organization, it therefore suffices to define the roles in the organization, how these roles relate to one another, and how a role can interact with other roles.

The organization model is comprised of two further models: the **roles model** and the **interaction model**.
Analysis

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.

The roles are characterized by two types of attribute:

- The permissions/rights associated with the role.
- The responsibilities of the role.
In order to illustrate the various concepts associated with roles, we will use a simple running example of a “coffee filler” role – the purpose of this role is to ensure that a coffee pot is kept full of coffee for a group of workers. Examples of liveness responsibilities for a 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.
Analysis

Liveness expression

In this model, an agent’s liveness properties are specified via a liveness expression which define the “life-cycle” of the role.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x.y$</td>
<td>$x$ followed by $y$</td>
</tr>
<tr>
<td>$x^*$</td>
<td>$x$ occurs 0 or more times</td>
</tr>
<tr>
<td>$x.c_0$</td>
<td>$x$ occurs infinitely often</td>
</tr>
<tr>
<td>$x \parallel y$</td>
<td>$x$ and $y$ interleaved</td>
</tr>
<tr>
<td>$x \mid y$</td>
<td>$x$ or $y$ occurs</td>
</tr>
<tr>
<td>$x \leftarrow y$</td>
<td>$x$ occurs 1 or more times</td>
</tr>
<tr>
<td>$[x]$</td>
<td>$x$ is optional</td>
</tr>
</tbody>
</table>

Table 1: Operators for liveness expressions
The general form of a liveness expression is:

\[ RoleName = \text{expression} \]

The atomic components of a liveness expression are protocols.

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

\[ \text{CoffeeFiller} = \text{(Fill.InformWorkers.CheckStock.AwaitEmpty)} \]
Analysis

Safety conditions

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. These invariants are called safety conditions, because they usually relate to the absence of some undesirable condition arising.
Safety requirements in this methodology 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. This can be done by means of the following safety expression:

- $coffeeStock \geq 0$
Analysis

Role schemata

It is now possible to precisely define the roles model. A roles model is comprised by a set of role schemata, one for each role in the system. A role schema draws together the various attributes discussed above into a single place.

<table>
<thead>
<tr>
<th>ROLE SCHEMA</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>short English description of the role</td>
</tr>
<tr>
<td>Protocols</td>
<td>protocols in which the role plays a part</td>
</tr>
<tr>
<td>Permissions</td>
<td>“rights” associated with the role</td>
</tr>
<tr>
<td>Responsibilities</td>
<td>liveness responsibilities</td>
</tr>
<tr>
<td>Liveness</td>
<td></td>
</tr>
<tr>
<td>Safety</td>
<td>safety responsibilities</td>
</tr>
</tbody>
</table>

Figure 3: Template for Role Schemata
Analysis

The CoffeeFiller’s role schema

| Role Schema: | CoffeeFiller |
| Description: | This role involves ensuring that coffee is kept filled, and informing the workers when fresh coffee has been brewed. |
| Protocols: | Fill, InformWorkers, CheckStock, AwaitEmpty |
| Permissions: | reads supplied coffeeMaker // name of coffee maker |
| | coffeeStatus // full or empty |
| | changes coffeeStock // stock level of coffee |
| Responsibilities | |
| Liveness: | CoffeeFiller = (Fill, InformWorkers, CheckStock, AwaitEmpty)* |
| Safety: | coffeeStock ≥ 0 |

Figure 4: Schema for role CoffeeFiller
There are inevitably dependencies and relationships between the various roles in a multi-agent organization. In this case, 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 institutionalized pattern of interaction.
In more detail, protocol definitions consist of the following set of attributes:

- **purpose**: brief description of the nature of the interaction;
- **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 description of any processing the protocol initiator performs during the course of the interaction.
Analysis

The *Fill* protocol definition

Figure 5: The *Fill* Protocol Definition
The analysis stage of the methodology can now be summarized:

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

- For each role, identify and document the associated **protocols**.
  
  **Output**: An interaction model, which captures the recurring patterns of inter-role interaction.

- 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, and the protocols in which they take part.

- Iterate the previous stages.
The aim of the design stage is to transform the analysis models into a sufficiently low level of abstraction that traditional design techniques (including object-oriented techniques) may be applied.

The design process involves generating three models:

- the agent model
- the services model
- the acquaintance model
The purpose of the agent model is to document the various agent types that will be used in the system under development, and the agent instances that will realize 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 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 or efficiency.
The agent model is defined using a simple agent type tree, in which root nodes correspond to roles, 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 $t_3$.

The agent instances that will appear in a system are documented by annotating agent types in the agent model. Note that inheritance plays no part in the agent models. The view is that agents are coarse grained computational systems, and an agent system will typically contain only a small number of roles and types, with often a one-to-one mapping between them.
The aim of the services model is to identify the services associated with each agent role, and to specify the main properties of these services.

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- 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.
The services that an agent will perform are derived from the list of protocols and responsibilities associated with a role, and in particular, from the liveness definition of a role.

For example, returning to the coffee example, there are four protocols associated with this role: Fill, InformWorkers, CheckStock and AwaitEmpty. There will be at least one service associated with each protocol. In the case of CheckStock, the service 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 condition is one of the safety conditions of the CoffeeFiller.
Design
The acquaintance model

Acquaintance models simply define the communication links that exist between agent types.
In particular, the purpose of an acquaintance model is to identify any potential communication bottlenecks, which may cause problems at run-time.
It is generally regarded as good practice to ensure that systems are loosely coupled, and the acquaintance model can help in doing this.
Design

A directed graph

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 can be derived in a straightforward way from the roles, protocols and agent models.
The design stage of the methodology can now be summarized:

- 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.

- Develop a services model, by examining protocols and safety and liveness properties of roles.

- Develop an acquaintance model from the interaction model and agent model.
Future Work

There are many issues remaining for future work:
Perhaps most importantly, this methodology does not attempt to deal with truly open systems, in which agents may not share common goals.

Another aspect of agent-based analysis and design that requires more work is the notion of an organizational structure. More generally, the development of organization design patterns might be useful for reusing successful multi-agent system structures.

Finally, a successful methodology is one that is not only of pragmatic value, but one that also has a well-defined, unambiguous formal semantics.
References