O-MaSE: An Customizable Approach to Designing and Building Complex, Adaptive Multiagent Systems

Abstract. The complexity and scope of software systems continues to grow. One approach to dealing with this growing complexity is the use of intelligent, multiagent systems. However, due in part to its relative infancy when compared to other software paradigms, the use of multiagent systems has yet to be used extensively in industry. One reason is the lack of industrial strength methods and tools to support multiagent development. This paper presents the Organization-based Multiagent Software Engineering (O-MaSE) methodology framework, which integrates a set of concrete technologies aimed at facilitating industrial acceptance. Specifically, O-MaSE is a customizable agent-oriented methodology based on consistent, well-defined concepts supported by plug-ins to an industrial strength development environment, agentTool III.

Keywords: agent-oriented methodology, method engineering, integrated development environments, software analysis, software design.

1. Introduction

Today’s software industry is tasked with building ever more complex software applications. While software development methods and techniques have made great strides over the last thirty years, the demand being placed on software is increasing even more rapidly. Businesses today are demanding applications that operate autonomously, adapt in response to dynamic environments, and interact with other distributed applications in order to provide wide-ranging solutions (Jennings, Sycara, and Wooldrige, 1998; Luck, et. al., 2005). This insatiable demand has left the software industry constantly looking for new computing metaphors and approaches to allow it to cope.

Multiagent system (MAS) technology is a promising approach capable of meeting these new demands (Luck, et. al., 2005). Its central notion – the intelligent agent – encapsulates the appropriate characteristics (i.e., autonomy, proactivity, reactivity, and interactivity) necessary to meet the requirements of these new applications. Unfortunately, there is a disconnect between the advanced technology being created by the multiagent community and its application in industrial software. The obstacles to industrial adoption have been the focus of several discussions. Jennings, Sycara, and Wooldrige (1998) mention two major obstacles to
widespread adoption of agent technologies in the industry: (1) the lack of complete methodologies and processes to help designers to specify, analyze, and design agent-based applications, and (2) the lack of industrial-strength agent-based toolkits. Luck, et. al. (2005) also suggest that the lack of mature methodologies and programming tools are the culprit. In a special session at AAMAS 2008, leading MAS researchers and engineers were asked to discuss the obstacles currently impeding industrial adoption of MAS technology. While there were a variety of opinions, Georgeff (2009) and DeLoach (2009a) suggested that standard definitions of agent concepts and agent-oriented methodologies are one of the keys to advancing MAS into the mainstream while Winikoff (2009) and Calisti and Rimassa (2009) both argued for producing concrete tools to support MAS techniques and methodologies.

Odell, Parunak, and Bauer (2001) advise that acceptance of any new technology requires techniques to reduce the inherent risk of that technology. They go on to assert that acceptance of new software development methods requires standard representations for artefacts supporting analysis, specification, and design. Thus, they propose two approaches for gaining industry acceptance of MAS technology. First, they suggest presenting the new methods as incremental extensions to known and trusted methods. Second, they recommend providing engineering tools to support the new methods that are similar to existing industrial practice. Instead of presenting new methods as extensions of existing methods, Bernon et. al. (2004) suggest integrating existing agent-oriented methods into one highly defined methodology.

An alternative approach to defining industrial strength methodologies that has gained support in the agent-oriented software engineering community is Situational Method Engineering, which promotes flexibility in MAS methods and processes (Low, et. al., 2009; Molesini, et. al., 2009; Cossentino, et. al., 2007). Henderson-Sellers (2005) was one of the first to argue that Situational Method Engineering was the key to creating industrial strength methodologies as it allows the creation of standard approaches that are widely supported while continuing to allow innovation and research. Situational Method Engineering allows method engineers to construct methods (a.k.a. methodologies) from a set of existing method fragments (Brinkkemper, 1996).

As Method Engineering is a young field, several terms are used ambiguously in the literature. Chief among these are method, methodology, process model and process. In this paper, the terms method and methodology are used synonymously with process model while the
O-MaSE: An Customizable Approach to Designing and Building Complex, Adaptive Multiagent Systems

term *process* is used to denote an instance of a process model or method that is enacted to develop a software system. Some exceptions to this convention exist in the naming of tool components as they have retained their historical names (e.g., the ‘agentTool Process Editor’).

This paper presents an overview of the Organization-based Multiagent Software Engineering (O-MaSE) methodology framework, which integrates a set of concrete technologies aimed at facilitating industrial acceptance through Situational Method Engineering. Specifically, O-MaSE is a customizable agent-oriented methodology based on consistent, well-defined concepts supported by plug-ins to an industrial strength development environment.

The goal of the O-MaSE methodology framework is to allow method engineers to build custom agent-oriented methods using a set of method fragments, all of which are based on a common meta-model. To achieve this, O-MaSE is defined in terms of a meta-model, a set of method fragments, and a set of Method Construction Guidelines. The O-MaSE **meta-model** defines a set of analysis, design, and implementation concepts and a set of constraints between them. The **method fragments** define a set of work products, a set of activities that produce work products, and the performers of those activities. Finally, **Method Construction Guidelines** define how the method fragments may be combined to create O-MaSE compliant methods. In general, an O-MaSE compliant method is an instance of the O-MaSE methodology in which appropriate method fragments are assembled into a method such that the Method Construction Guidelines are satisfied. Critical to the O-MaSE methodology framework is the agentTool III (aT³) integrated development environment that supports the creation of custom O-MaSE compliant methods as well as providing the editors, verification tools, and code generators for creating complex, adaptive systems using MAS technology.

O-MaSE has its roots in the original Multiagent Systems Engineering (MaSE) methodology (DeLoach, Wood and Sparkman, 2001). While MaSE provided a good starting point for developing multiagent systems, it had several problems. First, MaSE produced multiagent systems with a fixed organization. Agents developed in MaSE played a limited number of roles and had a limited ability to change those roles, regardless of their individual capabilities. In addition, MaSE did not include the notion of sub-teams and had no mechanism for modeling interactions with the environment. Finally, MaSE was utterly inflexible. MaSE prescribed a strict set of models that built upon each other; there were no guidelines to help a developer deviate from the established method. The aT³ toolset is
the successor to the original agentTool that was developed in 2000 – 2001 to support MaSE (DeLoach and Wood, 2001). The aT³ toolset is a plug-in to the Eclipse platform and extends the Eclipse Process Framework (EPF) to handle method customization.

While many pressing issues have been tackled in O-MaSE, at least for the moment, many tasks critical for a complete software methodology such as management, product deployment, and testing and evaluation have been intentionally ignored. Management and deployment issues are generally applicable over a wide variety of software projects and thus existing approaches can and should be applied. Testing and evaluation is not yet included in O-MaSE, as current work has focused strictly on the analysis, design, and implementation of multiagent systems; while many traditional techniques can be applied to multiagent systems, the need for unique approaches and tools is recognized. Existing research can be used to extend O-MaSE in this area (Poutakidis, et. al., 2009; Nguyen, Perini, and Tonella, 2008; Lam, and Barber, 2005; Coelho, et. al., 2006).

Following a discussion of background material in Section 2, O-MaSE is introduced in Section 3 in terms of its meta-model, method fragments, and guidelines. The aT³ toolkit is introduced in Section 4 while Section 5 illustrates the use of O-MaSE on two examples. Section 6 presents a comparison of O-MaSE with related methodologies, while Section 7 provides a final discussion and describes future work.

2. Background

Method Engineering is an approach where method engineers construct methods (a.k.a. methodologies) from a set of method fragments as opposed to modifying or tailoring monolithic, “one-size-fits-all” methods to suit their needs. Method fragments are generally created by extracting useful tasks and techniques from existing methods and redefining them in terms of a common meta-model. The fragments are then stored in a repository for later use. During method creation, method engineers select suitable method fragments from the repository and assemble them into complete methods meeting project specific requirements (Brinkkemper, 1996).

While intuitively straightforward, the application of Method Engineering for developing agent-oriented applications is non-trivial. Specifically, there is currently no consensus on the main elements distinguishing multiagent systems. While concepts such as agents, roles, and goals appear in many MAS techniques and methodologies, the definitions of
those concepts are inconsistent and often unrelated. Thus, Beydoun, et. al. (2005) (along with others) have suggested that prior to developing a set of method fragments, a well defined meta-model of common agent-oriented concepts should be developed and agreed upon similar to the object-oriented community.

Three similar metamodels exist to help apply Method Engineering to the production of custom methods: SPEM 2.0, OPEN, and SEMDM (ISO/IEC 24744). The Software and Systems Process Engineering Meta-model (SPEM) is “a process engineering meta-model as well as conceptual framework, which can provide the necessary concepts for modeling, documenting, presenting, managing, interchanging, and enacting developments processes” (Object Management Group, 2008). SPEM distinguishes between reusable method content and the way it is applied in actual methodologies. SPEM method content captures and defines the key Tasks, Roles, and Work Products\(^1\) that are used in a software development methodology. As shown in Figure 1, Tasks define the work that is performed by Roles to use an input set of Work Products to create and output set of Work Products.

Development methodologies are assembled into a set of Activities, populated with Tasks and their associated Roles and Work Products. Thus, Activities are aggregates of either basic content or other Activities. SPEM defines three special types of Activities: Phases, Iterations and Processes. Phases are special Activities that take a period of time and end with a major milestone or set of Work Products. Iterations are Activities that groups other Activities that are often repeated. Finally, Processes are special Activities that specify the structure of a software development project.

\(^1\) Technically, SPEM 2.0 defines Task Definitions, Role Definitions, and Work Product Definitions as Method Content with Task Uses, Role Uses, and Work Product Uses being instances of those definitions in actual methods. This paper refers to both forms as Tasks, Roles or Work Products.
In a similar vein, the OPEN Process Framework (OPF) uses a meta-model based framework that allows designers to select method fragments from a repository in order to construct custom methods (Firesmith and Henderson-Sellers, 2002). The OPF is defined in three layers: M2, M1, and M0. The M2 layer includes the OPF meta-model, which defines the types of method fragments that can be created. The OPF meta-model defines methodologies as consisting of Stages, Work Units (Activities, Tasks, and Techniques), Producers, Work Products, and Languages. The M1 layer includes a repository of method fragments and a methodology specific meta-model defining the concepts used within those fragments. The method engineer uses predefined method fragments from M1 to creating custom methods that are enacted at the M0 level on a specific project.

ISO/IEC 24744 defines the Software Engineering Metamodel for Development Methodologies (SEMDM), a competing metamodel for defining methodologies. SEMDM is unique in its ability to formalize the notion of dual-layer modeling using powertypes (Gonzalez-Perez, Henderson-Sellers 2006). Dual layer modeling refers to the situation where instances of methodology concepts (e.g., requirements specification, architectural design) are used as classes by developers to create instances of those classes (e.g., specific specifications and designs) during the enactment of the methodology. SEMDM defines methodologies as consisting of templates of stages, work units, work products, model units, and producers along with a set of resources, which define the languages, notations, constraints, outcomes and guidelines used. SEMDM also defines an Action that captures whether particular task of a work unit creates, modifies, or uses specific work products.

The core concepts of SPEM, OPF and SEMDM are parallel. SPEM Roles are essentially OPF and SEMDM Producers; SPEM Activities are similar, but not identical, to OPF and SEMDM Work Units; Work Products are analogous between the three. The difference between Activities and Work Units is that OPF and SEMDM Work Units describe what is to be done, but not when while SPEM mixes the two. O-MaSE was originally defined using the OPF. However, due to the popularity of SPEM in the agent-oriented software engineering community and the use of the SPEM-based Eclipse Process Framework to implement the aT³ Process Editor (see Section 4), O-MaSE has been redefined here in terms of SPEM 2.0.

In a related effort, the Foundation for Physical Agents Technical Committee (FIPA-TC) Methodology group attempted to define reusable method fragments from existing agent-oriented methodologies (Seidita,
O-MaSE: An Customizable Approach to Designing and Building Complex, Adaptive Multiagent Systems

Cossentino, and Gaglio, 2006). As part of this effort, the group is currently defining a Design Process Documentation Template1, which uses SPEM 2.0 as its base.

3. The O-MaSE Methodology Framework

The O-MaSE methodology framework is based on two metamodels: SPEM 2.0 and the O-MaSE metamodel. The SPEM metamodel defines methodology-related concepts while the O-MaSE metamodel defines the product related concepts. As shown in Figure 2, the definition of O-MaSE consists of three main components: the O-MaSE meta-model, method fragments, and guidelines. In general, a method engineer creates new O-MaSE compliant methods in aT3 by selecting O-MaSE Fragments and combining them into a method that is consistent with the Method Construction Guidelines. O-MaSE fragments are instances of SPEM elements such as Tasks, Work Products, and Roles, and are defined in terms of concepts from the O-MaSE metamodel. For example, the O-MaSE Role Model is an instance the SPEM Work Product and is defined in terms of Roles, Goals and Capabilities, each of which are defined in the O-MaSE metamodel. In this section, the three O-MaSE components are defined. First, the O-MaSE meta-model is defined. Next, a discussion of O-MaSE Phases is given followed by an explanation of the method

1 http://www.fipa.org/subgroups/DPDF-WG.html
fragments. Finally, the guidelines governing the construction of O-MaSE compliant methods are examined.

3.1. Meta-model

The O-MaSE meta-model defines the main concepts and relationships used to define multiagent systems. The O-MaSE meta-model is based on an organizational approach (DeLoach and Valenzuela, 2007; DeLoach, Oyenan and Matson, 2008) and includes notions that allow for hierarchical, holonic, and team-based decomposition of organizations (Horling and Lesser, 2004). The O-MaSE meta-model was derived from the Organization Model for Adaptive Computational Systems (OMACS). OMACS captures the knowledge required of a system’s organizational structure and capabilities to allow it to organize and reorganize at runtime (DeLoach, Oyenan, and Matson, 2008). The key decision in OMACS-based systems is determining which agent to assign to which role in order to achieve which goal.

Using models such as OMACS at runtime has recently become an important research area as it and allows efficient and effective runtime adaptation (Blair, Bencomo, and France, 2009). While O-MaSE does not focus solely on OMACS based systems, O-MaSE does provide direct support for such systems. As shown in Figure 3, an Organization is composed of five entities: Goals, Roles, Agents, Domain Model, and Policies. (Shaded entities correspond directly to OMACS entities and
In the traditional artificial intelligence sense, a *Goal* represents a desirable state (Russel and Norvig, 2003) or the objective of a computational procedure (van Lamsweerde, Darimont, and Letier, 1998). In agent-oriented circle, van Riemsdijk, Dastani, and Winikoff (2008, p.714) define a goal as “a mental attitude representing preferred progressions of a particular multi-agent system,” which captures the concepts individually distinct goals that require action to reach a particular state. As such, O-MaSE uses goals to define the objectives of the organization. A *Role* defines a position within an organization whose behaviour is expected to achieve a particular goal or set of goals. (Due to the naming conflict between O-MaSE Roles and methodology-related roles, the term method-role is used to refer to methodology-related roles throughout the remainder of this paper.) Agents are assigned to play those roles and perform the behaviour expected of those roles. *Agents* are autonomous entities that can perceive and act upon their environment (Russel and Norvig, 2002). To carry out perception and action, an agent possesses a set of capabilities. *Capabilities* can be used to capture soft abilities (i.e., algorithms) or hard abilities (i.e., physical sensors or effectors). An agent that possesses all the capabilities required to play a role, may be assigned that role in the organization. Capabilities can be defined as (1) a set of sub-capabilities, (2) a set of actions that may interact with the environment, or (3) a plan that uses actions in specific ways.

*Organizational Agents* (OAs) are organizations that act as agents in a higher-level organization and thus capture the notion of organizational hierarchy. As agents, OAs may possess capabilities, coordinate with other agents, and be assigned to play roles. OAs are similar to the notion of non-atomic holons in the ASPECS methodology (Cossentino, et. al., 2009). Therefore, OAs represent an extension to the traditional Agent-Group-Role (AGR) model (Ferber and Gutknecht, 1998) (Ferber, Gutknetcht, and Michel, 2003) and the organizational meta-model proposed by Odell et al. in (Odell, Nodine, and Levy, 2005).

The *Domain Model* is used to capture the key elements of the environment in which agents will operate. These elements are captured as *Domain Object Types* from the environment, which includes agents, and the *relationships* between those object types. It can also be used to capture general *Environment Properties* that describe how the objects behave and interact (DeLoach and Valenzuela, 2007). A designer may use entities defined in the O-MaSE model (goals, roles, agents, etc.) along with multiplicities of 0..* are omitted for clarity.) Each of these entities is discussed below.
entities defined in the Domain Model to specify organizational *Policies* to constrain how an organization may behave in a particular situation. Policies are often used to specify liveness and safety properties of the system being designed.

*Protocols* define interactions between roles or between the organization and external *Actors*. Protocols are generally defined as patterns of communication between such entities (Odell, Parunak, and Bauer, 2001; Odell, Parunak, and Bauer, 2000). A protocol can be of two types, External or Internal. *External Protocols* specify interactions between the organization and external actors (i.e., humans or other software applications), while *Internal Protocols* specify interactions between agents playing specific roles in the organization. Either messages or actions can be used to define protocols. Messages are typically used for communications; however, actions may be used to modify the environment as a means of communication (Holland and Melhuish, 1999).

3.2. Phases

SPEM uses Phases to organize the various Activities of a development method. While O-MaSE explicitly defines Activities and Tasks (see overview in Table 1), it does not define specific Phases. Because there are numerous ways to organize Activities, O-MaSE makes no commitments to a predefined set of Phases. Instead, O-MaSE allows method engineers to organize Activities in different ways based on project need. For instance, O-MaSE has been used to support modern iterative, incremental approaches as proposed by Royce (Royce 1998) and as implemented in the popular Rational Unified Process (RUP) (Kroll and Kruchten, 2003; Kruchten, 2000). However, O-MaSE has also been used in several projects using much simpler approaches such as the classical Waterfall model (Royce 1970).

Figure 4 shows an example of using an iterative, incremental approach with O-MaSE. Here, the goal of the *Inception Phase* is to establish what is and is not part of the product. The inception phase is broken into two iterations, the first focusing solely on Problem Analysis, while the second continues to refine the Problem Analysis while doing some preliminary Solution Analysis. The *Elaboration Phase*, whose goal is to demonstrate an architecture that can support key requirements, is also broken into two iterations. In Iteration 3, the Solution Analysis is further refined while initial Architecture Design work begins. In Iteration 4, Solution Analysis is finalized, more Architecture Design is carried out, and preliminary Low Level Design is done to support an executable prototype. The goal of the
Construction Phase is to produce an acceptable version of the system within cost and schedule. It starts with Iteration 5 where the Architecture Design is finalized and the Low Level Design and Code Generation of the initial features is performed. Iteration 6 continues with the Low Level Design and Code Generation for the next set of features.

Figure 5 shows an example of using O-MaSE with a Waterfall approach. In this case, there are three main Phases: Requirements Analysis, Design, and Implementation. In this case, the main Activities are allocated as expected, with Problem and Solution Analysis done in the Requirements Analysis Phase, Architecture and Low Level Design done during the

<table>
<thead>
<tr>
<th>Activities</th>
<th>Tasks</th>
<th>Work Products Created/Modified</th>
<th>Responsible Method-Roles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requirements Gathering</td>
<td>Requirements Specification</td>
<td>Requirements Spec</td>
<td>Requirements Engineer</td>
</tr>
<tr>
<td></td>
<td>Model Goals</td>
<td>Goal Model</td>
<td>Goal Modeler</td>
</tr>
<tr>
<td></td>
<td>Refine Goals</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Model Domain</td>
<td>Domain Model</td>
<td>Domain Modeler</td>
</tr>
<tr>
<td></td>
<td>Model Organization Interfaces</td>
<td>Organization Model</td>
<td>Organization Modeler</td>
</tr>
<tr>
<td></td>
<td>Model Roles</td>
<td>Role Model</td>
<td>Role Modeler</td>
</tr>
<tr>
<td></td>
<td>Define Roles</td>
<td>Role Description Document</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Define Role Goals</td>
<td>Role Goal Model</td>
<td></td>
</tr>
<tr>
<td>Architecture Design</td>
<td>Model Agent Classes</td>
<td>Agent Class Model</td>
<td>Agent Class Modeler</td>
</tr>
<tr>
<td></td>
<td>Model Protocols</td>
<td>Protocol Model</td>
<td>Protocol Modeler</td>
</tr>
<tr>
<td></td>
<td>Model Policies</td>
<td>Policy Model</td>
<td>Policy Modeler</td>
</tr>
<tr>
<td>Low Level Design</td>
<td>Model Plans</td>
<td>Agent Plan Model</td>
<td>Plan Modeler</td>
</tr>
<tr>
<td></td>
<td>Model Capabilities</td>
<td>Capabilities Model</td>
<td>Capabilities Modeler</td>
</tr>
<tr>
<td></td>
<td>Model Actions</td>
<td>Action Model</td>
<td>Action Modeler</td>
</tr>
<tr>
<td>Code Generation</td>
<td>Generate Code</td>
<td>Source code</td>
<td>Programmer</td>
</tr>
</tbody>
</table>
Design Phase and Code Generation done during the Implementation Phase.

Therefore, the definition of a complete O-MaSE compliant method requires the method engineer to distribute Activities and Tasks to Phases, as defined by the overall approach (iterative, incremental, waterfall, etc.). As this will be unique for each system being developed, there are no hard and fast rules on what activities should be placed in which phase. However, there are dependencies between the various fragments that must be maintained. These dependencies are captured as Method Construction Guidelines as described in Section 3.6. As the construction of O-MaSE compliant methods can be somewhat confusing, method construction is supported by the aT³ Process Editor as described in Section 4.2; this support includes automated validation of methods using the Method Construction Guidelines.

<table>
<thead>
<tr>
<th>Problem Analysis</th>
<th>Problem Analysis</th>
<th>Solution Analysis</th>
<th>Solution Analysis</th>
<th>Solution Analysis</th>
<th>Architecture Design</th>
<th>Low Level Design</th>
<th>Code Generation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iteration 1</td>
<td>Iteration 2</td>
<td>Iteration 3</td>
<td>Iteration 4</td>
<td>Iteration 5</td>
<td>Iteration 6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inception</td>
<td>Elaboration</td>
<td>Construction</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 4. Using Iterative, Incremental Phases in O-MaSE**

**Figure 5. Using Waterfall Phases with O-MaSE**
3.3. Activities

Table 1 shows six Activities currently covered by O-MaSE: Requirements Gathering, Problem Analysis, Solution Analysis, Architecture Design, Low Level Design, and Code Generation. Requirements Gathering is the process of identifying software requirements from a variety of sources. Typically, requirements are classified as either functional requirements, which define the functions required by the software, or non-functional requirements, which specify traits of the software such as performance quality and usability that are not directly related to software function.

The goal of the Problem Analysis is to capture the purpose of the product and document the environment in which it will be deployed. O-MaSE captures this information in a Goal Model, which captures the purpose of the product, and a Domain Model that captures the environment in which the product exits. The objective of Solution Analysis is to translate the purpose and environment of the project into a description of the required system behaviour and interactions with external entities such as users and existing systems. This behaviour is captured as roles and interactions in the Organization Model and Role Model.

Once the goals, environment, behaviour, and interactions of the system are known, Architecture Design is used to create a high-level description of the main system components and their interactions. The architecture is captured in Agent Class Models, Protocols, and Policies. This high-level description is then used to drive Low Level Design, where the detailed specification of the internal agent behaviour is defined. Low-level agent behaviour is captured in Plan, Capability and Action models. This low-level specification of agent behaviour is then used to implement the individual agents during Code Generation. While not currently defined in O-MaSE, system creation ends with testing, evaluation, and deployment of the system. Fortunately, the nature of the O-MaSE framework allows it to be extended based on current research and state of practice methods and techniques and thus incorporation of these activities is straightforward.

3.4. Tasks

Next, the typical Tasks, Work Products, and Method-roles used in O-MaSE are defined. While Table 1 shows Tasks as being associated with specific Activities, this is not always the case. As with the allocation of Activities to Phases, O-MaSE does not require specific Tasks to be performed in specific Activities. For instance, even though the Model Protocols task is generally part of the Architecture Design Activity, there is nothing to preclude a method engineer from including it in a Solution
Analysis Activity to define the protocols between roles defined in a Role Model. The only hard and fast requirements are contained in the Method Construction Guidelines in Section 3.6.

Each task is defined below with a general description of the Task objective along with a description of the steps used to produce the associated Work Products.

Throughout this paper, O-MaSE concepts, tasks, and models are illustrated using a Temperature Monitoring System (TMS) example as derived from (Bakshi, et. al., 2005). The TMS is a distributed, sensor system, where each node has a processor and a temperature system. During operation, each node monitors the temperature gradient between itself and its neighbours (those within 1-hop). If this temperature gradient exceeds a given threshold, a local alarm occurs; if the node can corroborate this reading with a larger set of neighbours (those within 10 meters) it triggers a global alarm. Each node is responsible to “push” its temperature reading to its neighbours at a set rate. However, when a node needs to corroborate a temperature gradient, the node is required to “pull” that data from all nodes within 10 meters.

3.4.1. Requirements Specification

There are several techniques for gathering software requirements. In general, there are several steps in requirements specification including elicitation, analysis, specification, negotiation, and validation. In many cases, traditional techniques (Pressman, 2010) for gathering requirements (e.g., data flow diagrams, use cases, and event-response tables) will be sufficient, while in other cases newer approaches focused toward multiagent systems are applicable (Castro, Kolp and Mylopoulos, 2002; Fuentes-Fernández, Gómez-Sanz and Pavón, 2009). O-MaSE assumes that either traditional or multiagent focused requirements gathering techniques are sufficient and thus does not stipulate a specific technique; the method engineer is free to use any existing technique deemed appropriate.

3.4.2. Model Goals

The objective of the Model Goals task is to transform the initial system requirements into a set of structured goals for the system. Goal models are widespread in many agent-oriented methodologies (DeLoach, Wood and Sparkman, 2001; Giorgini, Mylopoulos and Sebastiani, 2005; Padgham and Winikoff, 2002). The deliverable of the Model Goals task is an initial Goal Model.
The typical approach to modeling goals is AND/OR decomposition (van Lamsweerde and Letier, 2000). The objective of this approach is to refine the overall goal of the system into a set of subgoals. If all the subgoals must be achieved in order to achieve the parent goal, the parent is AND-refined, while if the subgoals represent alternative ways to achieve the parent goal, the parent goal is OR-refined.

An O-MaSE goal model for the TMS system is shown in Figure 6. The overall goal, Monitor Temperature, is AND-refined into three subgoals: MonitorTemp, CorroborateTemp, and NotifyUser. Essentially, the goal model creates a high-level specification of what the system should do. Each goal in the model is annotated by the keyword «Goal». A line between two goals with an «and» keyword at the parent end is used to represent AND-refinement while a line with an «or» keyword at the parent is used to represent OR-refinement.

3.4.3. Refine Goals

The Refine Goals task captures the dynamic aspects of the Goal Model and further defines each goal using a technique called Attribute-Precede-Trigger Analysis. The result is a refined version of the Goal Model called a GMoDS (Goal Model for Dynamic Systems) goal model (DeLoach and Miller, 2010).

The Refine Goals task is used to (1) capture any sequential constraints among goals, (2) determine which goals should be created in response to events that occur at runtime, and (3) document parameters required to define a unique goal state. If goal A must be completed before goal B can be pursued, then it is said that goal A precedes goal B. As the TMS system operates in parallel, there are no precedence relations in the goal model. New goals are often generated in response to specific events that occur within the environment or system and multiple instances of such goals may be active at any time. In the TMS system, new instances of the
CorroborateTemp and NotifyUser goals are created whenever a local alarm or global alarm is raised. When multiple instances of a goal may exist, parameters are used to uniquely define and identify each goal. With a NotifyUser goal, which is created each time a global alarm is raised, a user would need to know the temperature reading as well as the location of the node that raised the alarm.

A GMoDS version of the Goal Model for the TMS system is shown in Figure 7. Triggers are represented by arrows decorated with an event name and a set of event parameters. When instantiated, the Initialize goal is assigned to an agent to determine how many MonitorTemp goals should be created to monitor the entire area. These MonitorTemp goals are assigned to agents who use their sensing capabilities to monitor the temperature. When the sensed temperature exceeds the preset threshold $t$, the agent raises the localTempAlarm($temp, loc$) event that triggers the instantiation of the CorroborateTemp goal. This goal is assigned to an appropriate agent who attempts to corroborate the reading. If it does, the agent raises a globalAlarm($temp, loc$) event, which causes the instantiation of a NotifyUser goal. The NotifyUser goal is then assigned to an agent capable of interacting with the user.

3.4.4. Model Domain

The aim of the Model Domain task is to capture the object types, relationships, and behaviours that define the domain in which agents will sense and act. O-MaSE uses a simple Domain Model to capture the object types that agents interact and reason about. The Domain Model captures
the environment as a set of Object Types and Agents that are situated in the environment. Object types are defined by a name and a set of attributes. In O-MaSE, domain object types are similar to object classes rather than instances.

The domain model is developed using traditional domain modeling or domain analysis techniques common to many object oriented development methodologies (Prieto-Diaz and Arango, 1991). Object types from the Domain Model are commonly used to specify goal and event parameters in the Goal Model, to define message parameters in the Protocol Model, to specify constraints in the Policy Model, and to specify the result of agent actions in the Action Model.

The Domain Model of the TMS system is shown in Figure 8. As this is a simple system, the model is somewhat small. However, in order to be able to understand the Goal Model of Figure 7, one must understand the semantics of each attribute and parameter. Thus, the Domain Model defines the object types Temperature, Threshold, and Rate as base floating point types while Area is defined as a circle with a radius. Each Location is denoted by an xLoc and yLoc attribute.

3.4.5. Model Organization Interfaces

The objective of the Model Organization Interfaces task is to identify the organization’s interfaces with external entities, whether they are other agents, organizations, or actors external to the system.

To capture the organization’s interfaces, various classes of external entities are scrutinized to determine if the organization needs to interact with them. If the organization is a sub-organization (an OA) of a higher-
level organization, the interactions between the roles/agents in the higher-level organization and the OA define the initial set of interactions with this sub-organization. However, if this is a stand-alone, or top-level organization, the developer should consider interactions required with users as well as existing systems or databases to find the appropriate interfaces. Once identified, protocols are identified between the organization and the external entities. There should be a protocol for each type of interaction and thus there can be more than one protocol with a given external entity. The interfaces are defined in an Organization Model, which depicts a single organization interacting with a set of external actors. All external entities are modeled as external actors. The details of the protocols are defined later via the Model Protocols task.

Figure 9 shows the Organization Model for the TMS system. The TMS system is a single-level organization that interfaces directly with a single User. The User issues controls the system via the commands protocol, while the system provides feedback to the User via the alarms interaction protocol.

3.4.6. Model Roles

The Model Roles task identifies all the roles in the organization as well as their interactions with each other and with external actors. The result of the Model Roles task is a Role Model. The goal of role modeling is to assign each leaf goal from the organization Goal Model to a specific role. As a first cut, a single role is often created for each leaf goal. However, it is sometimes beneficial to enable a single role to achieve multiple types of goals. However, it is also true that organizations that are more flexible can be designed by having multiple roles capable of achieving the same type of goal. The designer must also identify interactions between roles as well as with external actors. Interactions with external actors can be derived directly from the Organization Model if provided.

The TMS Role Model in Figure 10 defines four roles, one for each leaf goal in Figure 7: Initiator, TempMonitor, TempCorroborator, and UserInterface. Each role requires various capabilities, which include hardware sensors, such as ReadTemperature, as well as software
algorithm such as GradientComputation. Although not stipulated in the model itself, the TempMonitor and TempCorroborator roles are designed to run on remote sensor platforms while the UserInterface and InitiatorRoles can execute on any capable computer. Notice that the User actor defined in the Organization Model is also in the Role Model. Each role is further defined using either the Define Roles or the Define Role Goals tasks described next.

3.4.7. Define Roles

The purpose of the Define Roles task is to define the behaviour and capabilities required for an agent to play a role. In addition, constraints may also be specified. In the Define Roles task, the designer specifies the capabilities required by a role, the goals the role is able to achieve, constraints associated with the role, and the plan(s) that implement the role. (If the required capabilities and goals that can be achieved by the roles have already been defined in the Role Model, these may be omitted.) These plans are developed using Model Plan task as described in Section 3.4.12. The Role Description Document for the TMS system is shown in Table 2. In this case, there is a single plan associated with each role. If a role may be used to achieve multiple goals, then the role may possess

<table>
<thead>
<tr>
<th>Role</th>
<th>Achieves</th>
<th>Requires</th>
<th>Plan</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initiator</td>
<td>Initialize</td>
<td>AreaDivision</td>
<td>DivideArea</td>
<td>None</td>
</tr>
<tr>
<td>UserInterface</td>
<td>NotifyUser</td>
<td>UserInterface</td>
<td>ControlSys</td>
<td>None</td>
</tr>
<tr>
<td>TempMonitor</td>
<td>MonitorTemp</td>
<td>ReadTemperature</td>
<td>GradientComputation</td>
<td>None</td>
</tr>
<tr>
<td>TempCorroborator</td>
<td>CorroborateTemp</td>
<td>TempSensor</td>
<td>Corroborate</td>
<td>None</td>
</tr>
</tbody>
</table>

![Figure 10. Role Model](image)
3.4.8. Define Role Goals

In the Define Role Goals task, role behaviour is defined in terms of a role Goal Model. The starting point for a role Goal Model is the leaf goal from the organization that is to be achieved by the role. Thus, the top goal of a role Goal Model is a leaf goal from the organization Goal Model.

The role Goal Models have the same semantics as the organization goal models created with the Model Goals and Refine Goals tasks described in Sections 3.4.2 and 3.4.3. In fact, the approach taken to define the Goal Model is the same as well. The key difference between an organization goal model and a role Goal Model is in the level of functionality that can be used to achieve the leaf goals. At the role level, each leaf goal is associated with a capability that can achieve that goal; at the organization-level, each leaf goal is associated with a role capable of achieving it.

The TempCorroborator role Goal Model is shown in Figure 11. Precedence is denoted by a «precedes» arrow; in this case, the Corroborate goal cannot be pursued until the PullTemps goal has been achieved. When started, the role must pull temperature readings from all agents with a certain distance. Once that is accomplished, it must corroborate its reading against those it has pulled. Finally, if the high temperature reading is corroborated, then the RaiseSystemEvent goal will cause the agent to raise a NotifyUser event at the system level.

Figure 11. TempCorroborator Role Goal Model
3.4.9. Model Agent Classes
The Model Agent Classes task identifies the types of agents that may participate in the organization. Agent classes may be defined to play specific roles, or they may be defined in terms of capabilities, which implicitly define the types of roles that may be played. An Agent Class is a template for a type of agent in the system. Each agent class identifies the capabilities that it possesses or the roles it can play (or both). In an open system where specific agents are not known a priori, an Agent Class Model may not be used as agents register themselves and their capabilities directly with the system; the roles these agents may play is based entirely on the capabilities required for the various roles.

Figure 12 shows an agent class model for the TMS system. As the system consists of homogeneous sensor nodes and a user interface device, there are only two agent types in the system: TempSensor and Notifier. A functioning TempSensor agent is implicitly capable of playing both the TempMonitor and TempCorroborator roles, while a Notifier agent is capable of playing the Initiator and UserInterface roles. Notice that the protocols specified in the Role Model are inherited by the appropriate agent types in the Agent Class Model and that the User actor is also included.

3.4.10. Model Protocols
The purpose of the Model Protocols task is to define the details of the interactions between agents or roles. Since protocols can be specified in Organization Models, Role Models and Agent Class Models, the method engineer may decide which set of protocols to define. If the Role Model protocols are defined via Protocol Models, agent classes playing those roles should inherit those protocols. When using $\text{aT}^3$ to design systems, $\text{aT}^3$ provides automated checks to ensure the consistency of these protocols between the various models. The Protocol Model produced defines the types of messages sent between the two entities and is essentially the same as the AUML (Bauer, Müller, and Odell, 2000) and
UML (Rumbaugh, Jacobson, and Booch, 2004) interaction models. In each of these models, messages are specified on arrows between lifelines and allowing looping and alternative control flows.

Figure 13 shows the Protocol Model for the monitorArea protocol in the TMS system. In this case, the Initiator sends a monitor(area) request to the TempMonitor. The “Alternative” frame provides an option for the TempMonitor role to return either a refuse() or accept() message.

3.4.11. Model Capabilities

The Model Capabilities task is used to define the internal structure of the capabilities possessed by agents in the organization. The result of the Model Capabilities task is a Capability Model. Each capability may be modeled as an Action or a Plan. An action is an atomic functionality possessed by an Agent and defined using the Model Actions task as described in Section 3.4.13. A plan is an algorithmic definition (defined via a state machine) of a capability that uses actions and implements protocols. Each plan is defined using the Model Plans task as presented in Section 3.4.12.

A portion of the Capability Model for the TMS system is shown in Figure 14. Notice that the AreaDivision capability is represented as a Plan, specifically the DivideAreaPlan, while ReadTemperature is represented as a complex capability with a plan, getTemperature, and a sub-capability, TempSensor. In this case, the getTemperature plan uses the TempSensor by calling its readSensor action.
3.4.12. Model Plans

The purpose of the Model Plans task is either to capture how an agent can achieve a specific type of goal using a set of actions (which includes sending and receiving messages) or to define a soft capability. The result of the Model Plans task is a Plan Model.

A Plan Model is specified in terms of a simple Finite State Machine where states contain action sequences and transitions contain inter-agent communications. Two special actions, send and receive, are used to denote sending and receiving of messages on transitions. User defined actions are carried out sequentially within states. Each action must be defined as part of a capability possessed by the agent performing the plan. Once in a state, the task remains in that state until processing is complete and a transition out of the state is enabled. Variables used in actions and messages are globally visible within the Plan.

The getTemperature Plan (which is part of the ReadTemperature capability) is shown in Figure 15. It is initialized by receiving a monitor message from the Initiator. It uses a timer to access its temperature sensor (via the readSensor() action) at the appropriate rate. It then computes the gradient. If the gradient exceeds the threshold it calls raiseAlarm, otherwise, it returns to the Wait state. While in the Wait state, the plan can respond to requests from Corroborator roles to get the current temperature, once it is defined.

3.4.13. Model Actions

The Model Actions task defines the low-level actions used by agents to perform plans and achieve goals. Actions belong to capabilities possessed by agents. Actions are typically defined as a function with a signature and a set of pre and post-conditions. In some cases, actions may be modeled by providing detailed algorithmic information. If using automatic code
generation techniques, this information is generally captured as a function or operation in the language being generated. In either case, the Action Model is usually just a textual document.

readSensor()  
Pre: true  
Post: readSensor > minTemp ∧ readSensor < maxTemp

In the readSensor example, since the action reads a sensor inputs, there is no precondition and the only guarantee about the output is that it will fall within the advertised sensor range.

3.4.14. Model Policies

The Model Policies task defines a set of formally specified rules that describe how an organization may or may not behave in particular situations. During the organization design, the Policy Modeler captures the desired and/or required properties of the system and writes them in natural language. Once all the policies have been identified, they can be formally specified if needed. For example, the following policy specifies that each TempSensor agent should be assigned to play both the TempMonitor and TempCorroborator roles.

∀a:TempSensor, ∃g1:MonitorTemp, g2:CorroborateTemp |  
    assigned(a, TempMonitor, g1) ∧ assigned(a, TempCorroborator, g2)

Policies have been used in multiagent system engineering for some time and several languages, frameworks, enforcement and checking
mechanisms have been developed (Bradshaw, et. al, 2003; Shoham, Y. and Tennenholtz 1995; Harmon, DeLoach, and Robby 2007, Harmon, et. al., 2008). In general, policies are used to restrict agent behaviour and may be enforced at design time or at runtime. How policies are enforced is a critical decision that affects the way the Policy Model is used during development. If there is no runtime mechanism designed or provided by the runtime environment, designs and implementations must be evaluated to ensure they conform to the policies.

3.4.15. Generate Code

The purpose of the Generate Code task is to take all the design models created during the development and convert them into code that correctly implements the models. Obviously, there are numerous approaches to code generation based on the runtime platform and implementation language chosen.

The aT³ toolkit includes an automatic code generation framework. Currently, the only platform supported is JADE (Bellifemine, Caire, and Greenwood, 2007) coupled with our Cooperative Robotics Organization-based Simulator (DeLoach, 2009b). To support OMACS-based systems, the Organization-based Agent (OBA) architecture (Figure 16) was created. The Control Component uses XML specifications of the organization.
Goal, Role, and Agent models to perform reasoning about goals, the organization state, and the assignment of agents to roles. The O-MaSE models produced during Low Level Design are used to define the role behaviour in the *Execution Component*. The OBA architecture supports significant reuse as much of the OMACS reasoning is standard and thus much of the Control Component code is reusable. A complete description of the architecture can be found in (DeLoach, 2009b)

3.5. Method-roles

Twelve Method-roles have been identified as part of the O-MaSE methodology: Requirements Engineer, Goal Modeler, Domain Modeler, Organization Modeler, Role Modeler, Agent Class Modeler, Protocol Modeler, Policy Modeler, Plan Modeler, Capabilities Modeler, Action Modeler, and Programmer. Each O-MaSE Method-role is responsible for carrying out the Tasks by applying the appropriate techniques to produce the Work Products shown in Table 1. Obviously, this requires the ability to apply the various techniques and to understand the Work Products that are both inputs to and outputs from those Tasks.

3.6. Method Construction Guidelines

Table 3 shows the Method Construction Guidelines (called Process Construction Guidelines in previously published papers) for the Tasks defined in Table 1. These Method Construction Guidelines are defined in terms of a pre-condition and post-condition. The pre-condition specifies the set of Work Products that must be available prior to the Task being undertaken while the post-conditions specify the Work Products produced by the task. For example, for the Model Goals task, either a Requirements Spec must be available or a Goal Model/GMoDS and a Role Model must be available. The Requirements Spec is used when the Model Goals task is used to model system-level goals while the Goal Model/GMoDS and Role Model are used when the task is used to model role-level goals. Disjunctive pre-conditions generally specify alternate ways the Task can be used. However, it does not limit what information can be used in the definition of a model. For instance, the Model Domain task only requires a Requirements Spec as input; however, that does not mean that other Work Products such as Goal Models cannot be used in the Task. This additional information is generally documented in the individual task definitions.
The agentTool III (aT³) development environment is built on the Eclipse platform (Garcia-Ojeda, DeLoach, Robby, 2009b; DeLoach et. al., 2009). The core elements of aT³ are the model creation tools that support the analysis, design, and implementation of multiagent systems following the O-MaSE methodology. aT³ also provides verification and metrics computation components, as well as the ability to compose, verify, and maintain custom O-MaSE compliant methods. The aT³ project webpage (http://agentTool.cis.ksu.edu/) contains the latest version of aT³ for download and includes tutorials, documentation, and examples.

aT³ is based on agentTool 1 and 2, which supported the original MaSE methodology. The original versions of agentTool were written as standalone Java tools that supported graphical model creation, protocol

<table>
<thead>
<tr>
<th>Task</th>
<th>Pre-condition</th>
<th>Post-condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requirements Specification</td>
<td>True</td>
<td>Requirements Spec</td>
</tr>
<tr>
<td>Model Goals</td>
<td>Requirements Spec ∨ ((Goal Model ∨ GMoDS) ∧ Role Model)</td>
<td>Goal Model</td>
</tr>
<tr>
<td>Refine Goals</td>
<td>Goal Model</td>
<td>GMoDS</td>
</tr>
<tr>
<td>Model Domain</td>
<td>Requirements Spec</td>
<td>Domain Model</td>
</tr>
<tr>
<td>Model Organization Interfaces</td>
<td>Requirements Spec ∧ GMoDS</td>
<td>Organization Model</td>
</tr>
<tr>
<td>Model Roles</td>
<td>GMoDS ∧ Organization Model</td>
<td>Role Model</td>
</tr>
<tr>
<td>Define Roles</td>
<td>Role Model</td>
<td>Role Description</td>
</tr>
<tr>
<td>Model Agent Classes</td>
<td>GMoDS ∨ Role Model ∨ Organization Model</td>
<td>Agent Class Model</td>
</tr>
<tr>
<td>Model Protocols</td>
<td>Role Model ∨ Agent Class Model</td>
<td>Protocol Model</td>
</tr>
<tr>
<td>Model Policies</td>
<td>GMoDS ∨ Organization Model ∨ Role Description ∨ Agent Class Model</td>
<td>Policy Model</td>
</tr>
<tr>
<td>Model Plans</td>
<td>(GMoDS ∧ Role Model) ∨ (GMoDS ∧ Agent Class Model)</td>
<td>Plan Model</td>
</tr>
<tr>
<td>Model Capabilities</td>
<td>Role Model ∧ Agent Class Model ∨ Domain Model</td>
<td>Capability Model</td>
</tr>
<tr>
<td>Model Actions</td>
<td>Capability Model ∧ Domain Model</td>
<td>Action Model</td>
</tr>
</tbody>
</table>
verification, semi-automatic analysis to design transformations, and code generation.

aT³ is a completely new development and was developed as a set of Eclipse plug-ins. Eclipse¹ is an open-source integrated development environment that supports easy extension through its plug-in based architecture. Eclipse was chosen as the base for aT³ due to this extensibility, support for graphic based editors, and for the ability to create methods, designs, and code within the same environment. In addition, the Eclipse Process Framework² (EPF) provides basic tools that support building custom methods.

In aT³, there is a separate plug-in for each O-MaSE model and each plug-in accesses a single Core plug-in that implements the O-MaSE meta-model. This multi-plug-in architecture supports the goal of allowing O-MaSE to be highly tailorable and extensible. None of the models are required and new models may be incorporated into the tool by adding a new plug-in to create/edit the model and new consistency rules to verify consistency with other models.

The aT³ development environment actually includes four components that are integrated into a single tool. These components are the Graphical Editor, the Process Editor, the Verification Framework, and a Code Generation Facility. Each component is discussed below.

4.1. Graphical Editor

The aT³ Graphical Editor supports the graphical editing of each of the O-MaSE models described in Section 3.4. These models, when combined, define the design of a multiagent system using concepts from the O-MaSE meta-model. A designer creates models in aT³ by dragging model elements from a palette and placing them onto the drawing panel. Built-in validation ensures that only valid connections are made between the appropriate model elements. To edit the internal details of model elements, aT³ also provides pop-up panels for items such as agent attributes and event parameters.

A screenshot of aT³ is shown in Figure 17. On the left side of the screen, the Eclipse Package Explorer allows the user to organize and store O-MaSE models in projects. Generally, subdirectories within projects refer to sub-organizations in the system design, thus the Package Explorer file

¹ http://www.eclipse.org/
² http://www.eclipse.org/epf/
structure mimics the hierarchical structure of the system. The model shown is an Agent Class Diagram. The icons shown in the Palette on the right side of the screen show the valid model elements and relations that may be added to the model. To add a model element to the model, users simply click on the model element in the Palette and then click where they want to place the model element in the model. Once the model element has been placed in the model, it may be edited or moved to another location. The protocol model elements are slightly different in that they are added between two actors or agents. To add a protocol, the user first clicks on the protocol icon in the Palette and then on the two actors/agents that participate in the protocol. After placing the protocol, the name may be edited. To add relationships between model elements, the user also clicks on the desired relationship in the Palette and then click on two model elements already in the model. Relationships have fixed names that may not be edited.

4.2. Process Editor

The at³ Process Editor (APE) is based on the EPF and allows method engineers to compose O-MaSE compliant methods (Garcia-Ojeda, DeLoach, and Robby, 2009a). APE provides five basic structures: a Method fragment Library, the Process Editor, a set of Task Constraints, a Process Consistency checker, and a Process Management tool as shown in

![Figure 17. at³ Graphical Editor](image-url)
Figure 18. The *Library* is a repository of O-MaSE compliant method fragments, which can be extended by APE users. The *Process Editor* allows users to create and maintain O-MaSE compliant methods. The *Task Constraints* view helps Method engineers specify Method Construction Guidelines to constrain how tasks can be assembled, while the *Process Consistency* mechanism verifies the consistency of custom methods against those constraints. Finally, the *Process Management tool* provides a way to measure progress using Earned Value Analysis. For more details, see (Garcia-Ojeda, DeLoach, and Robby, 2009a).

4.3. Verification Framework

The aT³ Verification Framework gives designers a way to maintain consistency between their O-MaSE models using a predefined set of rules. Since methods are customized, this rule set can also be customized by turning on and off certain rules. Each time a model is saved, the Verification Framework checks that document against all related documents in the current project using the currently enabled rules. Verification problems are shown to the user through the Eclipse Problems panel similar to compiler errors and warnings as shown in Figure 17.

4.4. Code Generation Facility

Automatic code generation is also available in aT³. Currently, the only platform targeted has been JADE (Bellifemine, Caire, and Greenwood, 2007). However, a framework has been created consisting of the Organization, Operation, Social, and Environment levels. At the Organization level, agents and roles are chosen for achieving specific goals. At the Operation level, agents achieve goals by performing actions based on their available capabilities. At the Social level, agent’s interactions are captured via messaging, while at the Environment level, the knowledge of object types and relationships are generated. Due to the detail of the O-MaSE models, the aT³ JADE generator is capable of generating 100% of the code necessary to create functional JADE systems. The generated code relies on pre-written Java code for each action specified in the Action Model.

5. Examples

In order to demonstrate our approach to assembling customized methods using O-MaSE, two examples deriving custom O-MaSE methods are presented. Readers can find applications of O-MaSE in other fields such as Information Systems (DeLoach, Oyenan, and Matson, 2008), Robotics
(Garcia-Ojeda, et. al., 2008), and Cooperative Software Agents (Garcia-Ojeda, DeLoach, and Robby, 2009a). The first example is an Adaptive Sensor Network that, while highly adaptive, is computationally expensive. The second example is a much more straightforward sensor-based Building Monitoring system whose operation relies on relatively simple sensors with little computational overhead.

Figure 18. agentTool Process Editor
5.1. Adaptive Sensor Networks

The first example is the development of an Adaptive Sensor Network (ASN) system. The ASN is designed to be able to detect and track vehicles moving over a large area. Multiple sensor types will be deployed including motion detectors, magnetometers, and heat detectors. In addition, special radiation sensors will be deployed to determine if any vehicles are radioactive. Sensors will be deployed in overlapping patterns based on probability of vehicles actually being in that area. To maximize battery life, sensors will be turned off as much as possible and only awakened when needed. Generally, a few motion detector sensors will be on to detect possible vehicles. When a vehicle is detected, additional sensors will be turned on to verify its location and track the vehicle as it moves.

Therefore, an ASN system must be able respond to specific events that occur in the environment as well as be able to reason about individual sensor capabilities and re-organize to achieve the desired system functionality. This highly adaptive behaviour is exactly what OMACS systems are designed to achieve and thus the system must be designed to include all OMACS required entities as shown in Figure 3.

Figure 19 shows an overview of the method for developing an Adaptive Sensor Network. As shown, the method uses an iterative approach with three phases (Inception, Elaboration, and Construction) and four iterations. Since the ASN systems requires an OMACS-based approach, it is necessary to ensure that all OMACS entities are produced as Work Products. In general, an OMACS-based method should produce a Goal Model, Role Model, Agent Class Model, and Policy Model. The additional Domain, Organization, Protocol, Capability, Plan, and Action models were added to support development of the base models or to support code generation. The right side of Figure 19 shows the Process Consistency checker for the APE. As shown, there are no inconsistencies and thus the method is O-MaSE compliant. As this method is designed to produce a highly adaptive, OMACS-based system, 14 out of the 15 O-MaSE tasks, or 93%, are used to define the method. As demonstrated here, methods used to create systems that are more complex tend to be larger in terms of number of O-MaSE tasks required.

5.2. Building Monitoring System

The second example is also taken from sensor network domain to illustrate the flexibility of O-MaSE within a single domain. In this case, a Building Monitoring System (BMS) will be developed. In this case, the BMS will have a predefined set of sensor types and each sensor will be deployed to a
O-MaSE: An Customizable Approach to Designing and Building Complex, Adaptive Multiagent Systems

fixed location. Each sensor will be hardwired to the building’s electrical supply, so power consumption is not an issue. Each sensor will sense at regular intervals and send its data to its neighbours for verification. All verified data will be sent via a predefined path to a central computer. Each sensor will be modeled as an agent that achieves a specific set of goals; the system will not need to reason about its capabilities or reorganize. While another contractor will design the internal operation of the individual agents, each agent will have to conform to system specific policies to ensure compatibility between agents.

Figure 19. Adaptive Sensor Network Method
Given the well-defined nature of the system, a straightforward waterfall development approach is chosen. Appropriate models are selected based on the implementation needs. Because there is no need for adaptivity in terms of reassigning agent responsibility, an organizational approach is not required. Therefore, a straightforward agent-centred approach is taken where agents are design to achieve specific goals. Figure 20 shows the method developed for the project. The method consists of three different Phases: Requirements Engineering, Analysis, Design, and Implementation. In this case, only 8 out of 15, or 53% of tasks defined by O-MaSE are used in this method. When compared to the previous example, this method is much simpler and thus more appropriate to this specific system development. Again, the right side of Figure 20 shows that the method is O-MaSE compliant.

6. Related Work

This section provides a comparison of O-MaSE against several other well-known agent-oriented software engineering methodologies in three different categories: Process Features, Model Features, and Supportive Features. These categories are taken from the evaluation of the ASPECS methodology when compared to PASSI, INGENIAS, ANEMONA, Gaia, ROADMAP, Tropos, Prometheus, and ADELFE (Cossentino, et. al., 2009). ASPECS is a modern agent-oriented methodology focused on complex, organization-based system following a holonomic perspective; it
is similar in many aspects to O-MaSE as is discussed below. Instead of reproducing a complete evaluation here (most of which would essentially duplicate the ASPECS evaluation), a discussion of the unique aspects of O-MaSE is provided as related to these categories and the other methodologies. For the complete evaluation of the other methodologies, the reader is referred to Section 6 of Cossentino et. al. (2009), which includes evaluations of the aforementioned methodologies with respect to the same three categories. The three basic categories were derived from the four categories used by Tran and Low (2005). Each category is defined below along with a discussion of the unique aspects of O-MaSE as compared to the other methodologies.

6.1. Process Features

The Process Features category attempts to judge generality and completeness of a methodology. Questions used to evaluate methodologies in this category include (1) standard lifecycle(s) supported, (2) standard development activities included, and (3) whether or not the methodology is domain dependent or independent.

Each of the methodologies studied claim to allow iterative and incremental lifecycles based on modern approaches. However, the methodologies generally fail to specify the exact relationships between the various activities that would allow activities to be placed appropriately in varying iterations. Support is available for methodologies whose processes have been formally defined such as INGENIAS, PASSI, and O-MaSE.

Most of modern methodologies considered cover the entire development lifecycle from requirements through design and implementation and/or deployment. Only the early methodologies such as Gaia and Tropos cover only analysis and design activities. A unique aspect of O-MaSE is the ease with which it can be extended. Due to its inherent design as a set of fragments, new tasks and models may be easily added without causing changes to existing methods. For example, related work on design metrics for OMACS-based systems resulted in the addition of several additional Tasks and Work Products to current version of O-MaSE as released in aT³ such as System Flexibility (Robby, DeLoach, and Kolesnikov, 2006).

6.2. Model Features

The Model Features category attempts to judge the focus of the process models and their completeness in terms of handling non-agent concepts. Criteria used to evaluate methodologies in this category include (1) agent focused versus organization focused, (2) support for levels of system
decomposition, (3) support for modeling interactions with the environment, (4) support for modeling of domain knowledge, and (5) formal foundation and semantics.

In terms of focus, several of the earlier methodologies such as PASSI, Prometheus, and ADELFE are agent focused, while the newer methodologies tend to be organization focused. O-MaSE appears to be unique in its support for both points of view as demonstrated in the example methods of Sections 0 and 5.2. As Cossentino et al. (2009) point out, within the organization based approaches, some focus on concepts of agents, roles, and groups while other highlight norms, which moves toward the concepts of Electronic Institutions (Noriega and Sierra, 2002). Again, O-MaSE shows its flexibility by supporting either approach.

In terms of modeling complex systems via levels of system decomposition, only the holonomic methodologies, ANEMONA and ASPECS provide such support. Here, O-MaSE provides a hierarchical decomposition approach using agent organizations (AOs), which are closely related to holarchies.

O-MaSE also provides support for modeling of the environment and interactions with the environment. One of the main purposes of the O-MaSE Domain model is for capturing object types and their relationships in the environment while Actions allow developers to specify the effects of agent operations on those environment objects. Because Protocols may be specified in terms of Actions (as well as messages), complex interactions with the environment may also be modeled.

While O-MaSE does not use formal notations except in the case of policies, the formalization of its meta-model does allow the use of formal model checking techniques to provide predictive metrics. For instance, Robby, DeLoach, and Kolesnikov (2006) describe a system flexibility metric that measures how many different ways an OMACS-based system can achieve its overall goal. Automated techniques for computing such metrics have been incorporated into aT3.

6.3. **Supportive Features**

Finally, the Supportive Features category looks at the methodology’s support for standards, tools, and complex system concepts. Criteria used to evaluate methodologies in this category include (1) tool and library support, (2) support for open versus closed agent systems, and (3) support for dynamic, self-organizing, and reconfiguring systems.
A distinguishing aspect of O-MaSE is its tool support. As discussed above, aT³ provides an integrated environment (with Eclipse) that supports Method Engineering, model development and verification, design level predictive metrics, and automatic and manual code generation. Because it is based on the Eclipse plug-in approach, aT³ is extremely extensible; new method fragments, new models, and new code generation, deployment, and testing tools can be added by adding new plug-ins. The INGENIAS IDK also supports similar aspects although the process editor is not fully integrated with IDK.

Cossentino et al. (2009) point out that APSECS is “the only process that supports both open and dynamic systems and merges an agent-oriented approach with a knowledge-engineering approach.” Clearly, O-MaSE also supports these areas. While ASPECS focuses more heavily on the use of domain knowledge by specific references to its ontology, O-MaSE, being based on OMACS, has a more precisely defined mechanism to support dynamic, reconfigurable open systems. In addition, the second main use of the O-MaSE Domain model is to capture ontologies in support of open systems.

7. Conclusions and Future Work

The O-MaSE methodology framework integrates a suite of technologies aimed at removing impediments to the industrial acceptance of agent technology. O-MaSE provides a customizable agent-oriented methodology based on consistent, well-defined concepts supported by plug-ins to an industrial strength development environment. The O-MaSE methodology framework allows developers to create custom agent-oriented methods using a set of well-defined method fragments that support a variety of system types and complexities. This is achieved in O-MaSE via a meta-model, a set of method fragments, and a set of Method Construction Guidelines. Each aspect of the O-MaSE methodology framework is supported by the aT³ integrated development environment, which supports method creation and maintenance, model creation and verification, and code generation and maintenance.

The main advantages of this approach are

1. O-MaSE supports agent-centered, organization centered, closed or open agent systems, based on the method fragments used in an appropriate custom method.

2. Each O-MaSE method fragment is defined over a common meta-model that also directly supports complex adaptive systems based
on the OMACS organization model and its associated architectures and algorithms.

3. The O-MaSE Method Construction Guidelines define how method fragments may be combined to assemble O-MaSE compliant methods.

4. O-MaSE is fully supported by aT3, which supports the creation and implementation of O-MaSE compliant methods as well as supporting the creation and verification of systems using those methods.

Because O-MaSE and aT3 provide a comprehensive environment for developing multiagent and organization-based systems, it also provides an excellent platform for additional research and development. We plan to continue investigating formal compositional approaches for building complex, adaptive systems using semi-automatic design-time metrics as well as automatic runtime composition using O-MaSE models (Oyenan, DeLoach, and Singh, 2009). This work will be integrated by extending O-MaSE with new method fragments as well as adding new functionality in aT3.

A very important area that should be investigated further is the integration of O-MaSE concepts with other MAS metamodels, which have been the subject of much research (Azaiez, Huget, Oquendo 2006; Bernon, Cossentini, Pavon 2005; Beydoun, Low, Henderson-Sellers, Mouratidis, Gomez-Sanz, Pavon, Gonzalez-Perez 2009). When development of O-MaSE started in 2005 (DeLoach 2006), there were no metamodels that captured the key elements required to support OMACS-based systems, namely a direct relation between agents, roles, goals, and capabilities. Thus, the O-MaSE metamodel was developed in parallel with recently published metamodels. However, new work in synthesizing common MAS metamodels, specifically the FAML metamodel (Beydoun, Low, Henderson-Sellers, Mouratidis, Gomez-Sanz, Pavon, Gonzalez-Perez 2009), provides great promise of producing a general metamodel capable of supporting standardization of concepts across the agent community. While FAML does not currently support all the required concepts and relations to support an OMACS-based system (there is currently no notion of capabilities at design or runtime that are possessed by agents and required to play specific roles), the extensibility of FAML has been shown. Future work on O-MaSE should include a detailed study of extending more general metamodel such as FAML to replace the O-MaSE metamodel. Such use of common metamodels would significantly enhance
the overall goal of many MAS researchers of combining reusable method fragments from multiple MAS methodologies.

Another area of future work would be to recast O-MaSE in terms of ISO/IEC 24744, the Software Engineering Metamodel for Development Methodologies (SEMDM). Using SEMDM as the basis of O-MaSE would allow a more precise description of the relationship between the modeling of methodology elements and their instances during methodology enactment. It would also allow a more precise description of the relationships between the process and products currently captures in the O-MaSE Method Construction Guidelines via the SEMDM Action element. Finally, SEMDM would allow for a more precise definition of how the O-MaSE (or extended FAML) metamodel and modeling notations relate to specific work products.

We are also interested in approaches for dealing with human agents as part of the system. We are currently studying how humans and agents can exist together within multiagent teams. We are looking at extending the OMACS model and thus, by extension the O-MaSE (or an extended version of FAML) metamodel. Clearly, such an extension should ensure backward compatibility and will likely require the integration of new method fragments into O-MaSE.

We are also investigating how to model and reason about agent interactions at runtime. Again, this will likely require introducing new concepts into the O-MaSE (or FAML) meta-model, such as first-class interactions and interaction goals, as well as providing new models for capturing such information.

8. Acknowledgements

This work was supported by grants from the US National Science Foundation (0347545) and the US Air Force Office of Scientific Research (FA9550-06-1-0058 and FA9550-09-1-0108).

9. References


