Kansas State University
234 Nichols Hall
Manhattan, KS 66506-2302

Phone: (785) 532-6350
Fax: (785) 532-7353
E-mail: sdeloach@cis.ksu.edu

Technical Report

An Organizational Model and Dynamic Goal Model for Autonomous, Adaptive Systems

by

Scott A. DeLoach & Walamitien H. Oyenan

MACR-TR-2006-01

March 13, 2006

This technical report is the final performance report for AFOSR grant number F49620-02-01-0427. The grant period was 1 September 2002 – 31 December 2005.

TABLE OF CONTENTS

Chapter 1 - Introduction

Systems are becoming more complex, in part due to increased customer requirements and the expectation that applications should be seamlessly integrated with other existing, often distributed applications and systems. In addition, there is an increasing demand for these complex systems to exhibit some type of intelligence as well. No longer is it "good enough" to be able to access systems across the internet, but customers require that their systems know how to access data and systems, even in the face of unexpected events or failures.

The goal of our research is to provide a framework for constructing complex, distributed systems that can autonomously adapt to their environment. Multiagent systems have become popular over the last few years for providing the basic notions that are applicable to this problem. A multiagent system uses teams of self-directed agents working together to achieve a common goal. Such multiagent teams are widely proposed as replacements for sophisticated, complex, and expensive stand-alone systems for similar applications. Multiagent teams tend to be more robust and, in many cases, more efficient (due to their ability to perform parallel actions) than single monolithic applications. In addition, the individual agents tend to be simpler to build, as they are built from a single agent's perspective.

However, unpredictable application environments make multiagent teams susceptible to individual failures that can significantly reduce the ability of the team to accomplish its goal. The problem is that multiagent teams are typically designed to work within a limited set of configurations. Even when the team possesses the resources and computational ability to accomplish its goal, it may be constrained by its own structure and knowledge of team member's capabilities, which may change over time. In most multiagent design methodologies [13, 32, 46], the system designer *analyzes* the possible organizational structure – which determines which roles are required to accomplish which goals and sub-goals – and then *designs* one organization that will suffice for most anticipated scenarios. Unfortunately, in dynamic applications where the environment as well as the agents may undergo changes, a designer can rarely account for, or even consider, all possible situations. Attempts to overcome these problems include large-scale redundancy using homogenous capabilities and centralized/distributed planning. However, homogenous approaches negate many of the benefits of using a multiagent approach are not applicable in complex applications where specific capabilities are often needed by only one or two agents. Centralized and distributed planning approaches tend to be brittle and computationally expensive due to their required level of detail (individual actions in most cases).

To overcome these problems, we have developed a framework that allows *the team to design its own organization at runtime*. In essence, we propose to provide the team with organizational knowledge and let the team design its own organization based on the current goals and team capabilities. While the designer can provide guidance, supplying the team with key organizational information will allow the team to redesign, or reorganize, itself to match its scenario. This paper defines a theoretical framework that shows how the appropriate knowledge of a team's organizational structure and capabilities can allow multiagent teams to reorganize at runtime thus enabling them to achieve their top-level team goals more efficiently and effectively in the face of a changing environment and team capabilities.

1.1 Motivating Examples

1.1.1 Cooperative Robotic Search and Rescue

Consider the case where a team of heterogeneous robots is performing a cooperative search and rescue operation. Some robots will have better capabilities for searching due to their enhanced sensor package while some robots may be better suited for rescuing due to their specific effectors such as grippers and robotic arms, each with differing payload amounts. However, robots with enhanced rescue abilities can also perform searching, since they must have some type of minimal object detection system in order to perform rescues. Thus at the onset of the search and rescue operation, since the team has not yet found

any victims, all the robots are available for searching. Once a victim is found, one of the robots must switch to a rescuer role and attempt to rescue the victim. However, choosing the correct robot to perform the rescue is dependent on many properties of both the victim and their current situation, which may include size of the victim, access to victim, etc.

In addition, the capabilities of the individual robots may change over time. This must be accounted for when organizing the team. For instance, what happens if there are three robots performing the rescue role and one of those robots happens to break down? Should the team get another robot to take its place? Or, should the team continue with its two rescue robots? These are decisions that can only be made within the context of what is best for the team and its current state in terms of the problem being solved. What is needed is a mechanism that the robot team can use to determine the best robot for the job in terms of overall team performance. This mechanism must take into account the current state of the team, which includes the goals being pursued, the available team members, and the team member's capabilities.

1.1.2 Battlefield Information Systems

The goal of a battlefield information system is to provide the commander with both tactical and strategic intelligence. To accomplish this, various types of sensors are used to detect events and objects of interest. The sensor data is then be combined, or fused, with other sensor data to provide a commander with a more complete picture of the battlefield. Due to the nature of war, there is a high probability that some of these sensors will become disabled. However, when sensors are lost, their information is still required in order to provide the battlefield commander with a complete picture. Thus, the battlefield information system must detect sensor failures and adapt its processing in a timely manner. An example of such a system is one in which air, satellite and ground-based sensors must be monitored to evaluate enemy force deployment and strategy. To operate effectively in this scenario, the battlefield information system must adapt to changes in both the queries from the commander as well as sensor availability.

As an example, assume we have a system with three types of agents as shown in Figure 1: data sensor agents, synthesis agents, and query agents. *Data Sensor Agents* (DS) provide the interface between the hardware sensors and the *Synthesis Agents* (SA), which fuse data from various sensor types to formulate answers to requests for information of the *Query Agents* (QA). The Query Agent translates, manages and communicates the query to the Synthesis Agents and returns results to the commander.

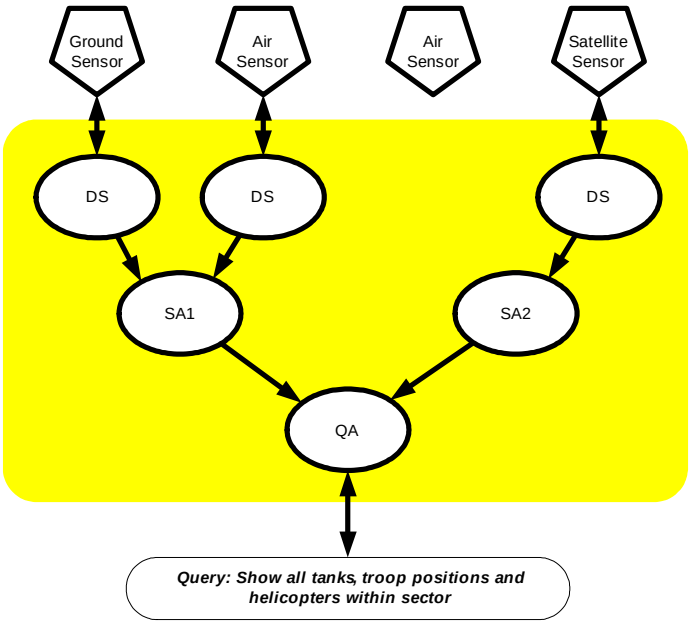


Figure 1. Original System Organization

When the system receives a query, there is no assurance the sensors required to execute the query will be available. If a sensor is damaged, the system may be required to re-evaluate its ability to satisfy the query requirements. If the requirements are not met, the system must reorganize to produce a new structure that can meet the query goal requirements. Figure 1 shows the initial layout of the system as set up to answer the query, "Show all tank, troop and helicopter movement within sector". Answering this query requires the minimal capabilities of three sensors and 3 DS agents to interpret the raw data. SA1 possesses the capability to accept data from ground and air sensors and synthesize it for return to the QA. SA2 accepts and passes data from the satellite via the DS agent.

A problem arises when the Air Sensor that the system is using to answer the query becomes unavailable. The system reacts to this event by reorganizing itself as shown in Figure 2. Notice that instead of simply replacing the lost Air Sensor with another one and integrating its data via SA1 as might be expected, the system chose to integrate the Air Sensor via SA2. The answer to why the system chose this particular organization lies in the capabilities of the various sensors and the agents that are combining the data. Even though the new Air Sensor provides data similar to the one that was lost, the system realized that due to its lower quality, combining it with the Satellite data first and then combining it with the Ground Sensor data would yield a better result (either in terms of timeliness or quality) than simply replacing the failed Air Sensor with the new Air Sensor. Analysis of this type requires detailed knowledge about the capabilities of the sensors as well as the agents used to combine the data. The goal of our research is to provide a framework that provides systems with this knowledge and analysis capability.

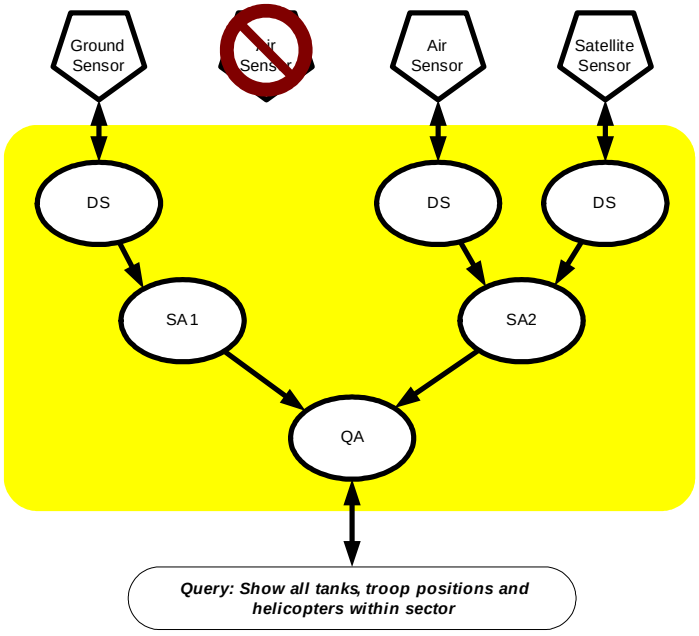


Figure 2. System after Sensor Failure

Chapter 2 - Organization Model for Adaptive Computational Systems

They key result of this project was the development of a model for artificial organizations, called the Organization Model for Adaptive Computational Systems (OMACS). This model allows MAS developers to define a structure to support specific applications. OMACS does not assume a specific goal model. (Note: In the following definitions, $\mathbf{P}(S)$ is used to denote the powerset of S.)

2.1 General Organization Definition

OMACS defines an organization as a tuple

$$O = \langle G, R, A, C, \Phi, P, \Sigma, oaf, achieves, capable, requires, possesses, potential \rangle$$

where

- G goals of the organization
- R set of roles
- A set of agents
- C set of capabilities
- Φ relation over $G \times R \times A$ that defines the current set of agent/role/goal assignments
- P set of constraints on Φ
- Σ domain model used to specify objects in the environment, their inter-relationships, and the operations that can be performed upon them
- *oaf* function $\mathbf{P}(G \times R \times A) \rightarrow [0.. \infty]$ that defines the quality of a proposed set of assignments
- *achieves* function $G \times R \rightarrow [0..1]$ that defines how well a role achieves a goal
- *capable* function $A \times R \rightarrow [0..1]$ that defines how well an agent can play a role
- *requires* function $R \rightarrow \mathbf{P}(C)$ that defines the set of capabilities required to play a role
- *possesses* function $A \times C \rightarrow [0..1]$ that defines the quality of an agent's capability
- *potential* function $A \times R \times G \rightarrow [0..1]$ that defines how well an agent can play a role to achieve a goal

Each of the above components is described below in detail.

2.2 Goals

Artificial organizations are designed with a specific purpose, which defines the overall function of the organization. *Goals* are defined as a desirable situation [39] or the objective of a computational process [65]. Within OMACS, each organization has a set of goals, `G`, that it seeks to achieve. OMACS makes no assumptions about these goals except that they can be assigned to individual agents. (For simplicity, this report refers to a role *achieving* a goal, although there are many types of goals such as goals of achievement, maintenance, etc.) A detailed description of the Goal Model for Dynamic Systems (GMoDS) used in this research is given in Chapter 3.

2.3 Roles

Within OMACS, each organization contains a set of roles (`R`) that it can use to achieve its goals. A *role* defines a position within an organization whose behavior is expected to achieve a particular goal or set of goals. Roles are analogous to roles played by actors in a play or by members of a typical corporate structure. A typical corporation has roles such as "president", "vice-president", and "mail clerk". Each role has specific responsibilities, rights and relationships defined in order to help the corporation perform

various functions towards achieving its overall goal. Specific people (agents) are assigned to fill those roles and carry out the role's responsibilities using the rights and relationships defined for that role.

OMACS roles consist of a name and a role capability function, *rcf*. Each role, $r \in R$, is a tuple ⟨name, *rcf*⟩ where

- name a string
- *rcf* function $A \rightarrow [0..1]$ that defines how well a given agent can play the role

While an agent that is assigned a role may chose to play that role in any way it wishes, OMACS does have certain expectations for agents assigned to roles. First, the agent is expected to play that role in order to achieve a specific goal. Thus, OMACS assumes that each role implies some minimal expected behavior. For instance, it would be assumed that someone playing the "mail clerk" role in a company would pick up mail from the mailroom and eventually deliver that mail to its addressee. This minimal behavior defines the functionality associated with the role. Although an understanding of this behavior is critical to the design and operation of the actual system, it is not critical to the definition of the organization of the system and is not specified further in OMACS.

A role's *rcf* determines the ability of an agent to play that role; it is user defined and computed in terms of the agent's capabilities. Having the required capabilities is not necessarily sufficient to determine whether an agent can actually play the role or decide which agent can best play the role. Some capabilities may be more important to the role than others. To capture this on a role-by-role basis, the designer must define a role specific *rcf*, which computes a value in the range of 0 to 1. The role capability function allows the role designer to specify how specific capabilities affect the ability of an agent to play that role. OMACS uses the notation *r.rcf(a)* to denote the application of the role capability function for role *r* on agent *a*.

2.4 Agents

OMACS also includes a set of heterogeneous agents (A) in each organization. As described by Russell and Norvig, an agent is an entity that perceives and can perform actions upon its environment [39], which includes humans as well as artificial (hardware or software) entities. For our purposes, we define agents as computational systems that inhabit some complex dynamic environment, sense and act autonomously in this environment, and by doing so realize a set of goals. Thus, we assume that agents exhibit the attributes of autonomy, reactivity, pro-activity, and social ability. *Autonomy* is the ability of agents to control their actions and internal state. *Reactivity* is an agent's ability to perceive its environment and respond to changes in it, whereas *pro-activeness* ensures agents do not simply react to their environment, but that they are able to take the initiative in achieving their goals. Finally, *social ability* allows agents to interact with other agents, and possibly humans, either directly via communication or indirectly through the environment.

Within the organization, agents must have the ability to communicate with each other, accept assignments to play roles that match their capabilities, and work to achieve their assigned goals.

2.5 Capabilities

The set of capabilities, C, in an organization is the sum of all the capabilities required by roles or possessed by agents in the organization.

$$\forall\ c{:}C\ \ (\exists\ a{:}A\ \ (a,c) \in \text{possesses}\ \vee\ \exists\ r{:}R\ \ (r,c) \in \text{requires}) \qquad (1)$$

Capabilities are the key to determining exactly which agents can be assigned to which roles within the organization. *Capabilities* are atomic entities that are used define to the abilities of various agents in relation to other agents as well as roles. Capabilities can be used to capture *soft* abilities such as the access to/control over specific resources, the ability to communicate with other agents, the ability to migrate to a new platform, or the ability to use computational algorithms that allow the agent to carry out

specific functional computations. Capabilities also capture the notion of *hard* capabilities that are often associated with hardware agents such as robots. These hard capabilities are generally described as sensors, which allow the agent to perceive a real world environment, and effectors, which allow the agent to act upon a real world environment.

2.6 Assignment Set

The assignment set Φ is a subset of all the potential assignments of agent to play roles to achieve organizational goals. The selection of the actual subset members is left up to the organization.

$$\Phi \subseteq \{\langle a,r,g\rangle \mid a \in A \wedge r \in R \wedge g \in G \wedge \text{potential}(a,r,g) > 0\} \quad (2)$$

However, just because an agent has the potential to play a role in order to achieve a goal, it does not indicate that the actual assignment of agent `a` to role `r` to achieve goal `g`, has been made. It simply indicates that if the assignment of agent `a` to role `r` is made, it has the potential to achieve goal `g`.

To capture the notion of the actual assignments that have been made in the organization, we define an *assignment set* Φ which consists of agent-role-goal tuples, `<a,r,g>`. If `<a,r,g>` $\in \Phi$, then agent `a` has been assigned by the organization to play role `r` in order to achieve goal `g`. The only inherent constraint on Φ is that it must contain only assignments whose potential value is greater than zero.

2.7 Policies

In general, policies are a set of formally specified rules that describe how an organization may or may not behave in particular situations. In OMACS, we distinguish between three specific types of policies: *assignment* policies (P_{Φ}) *behavioral* policies (P_{beh}), and *reorganization* policies (P_{reorg}).

2.7.1 Assignment Policies

In general, OMACS allows the assignment of any agent `a` to any role `r` in order to achieve any goal `g`, as long as `potential(a,r,g)` > 0. However, in a specific application, there may be additional constraints that the assignment set, Φ must satisfy. These constraints are captured in the form of *assignment policies.* Thus, assignment policies, P_{Φ}, constrain the assignment set Φ.

In many cases, generic policies such as "an agent may only play one role at a time" or "agents may only work on a single goal at a time" are useful and are shown below.

$$\forall\ a1,a2{:}A\ r1,r2{:}R\ g1,g2{:}G\ \ \langle a1,r1,g1\rangle \in \Phi \wedge \langle a2,r2,g2\rangle \in \Phi\ \ \wedge a1 = a2 \Rightarrow r1 = r2$$

$$\forall\ a1,a2{:}A\ r1,r2{:}R\ g1,g2{:}G\ \ \langle a1,r1,g1\rangle \in \Phi \wedge \langle a2,r2,g2\rangle \in \Phi\ \ \wedge a1 = a2 \Rightarrow g1 = g2$$

However, policies are often application specific, such as requiring particular agents to play specific roles or that the correct number of agents are playing specific roles. In an information system application, it might be necessary to ensure that no more than two agents are assigned to play roles that interface to a specific database in order to reduce resource contention. If the specific role that has access to the database is named DBAccess, then we could specify such a policy as (where # is the cardinality operator)

$$\#(\{a \mid \langle a,\text{DBAccess},g\rangle \in \Phi\ \}) \leq 2$$

The language used to define policies will be implementation specific and will consist of names for the entities and relationships from our organizational framework (e.g., potential, related, etc.) as well as application specific terms such as role, goal, and capability names, which are defined in the organization's ontology/domain model, Σ.

To determine if an individual assignment, ϕ, or an assignment set, Φ, are legal according to the current organizational policies, OMACS defines two `legal` operations. Here **P**(P) refers to the set of all policies of the organization.

$$\text{legal: } \phi, \mathbf{P}(P) \rightarrow \text{Boolean}$$
$$\text{legal: } \Phi, \mathbf{P}(P) \rightarrow \text{Boolean}$$
$$\text{legal: } \Phi, \langle \Phi', E' \rangle, \mathbf{P}(P) \rightarrow \text{Boolean}$$

The first operation takes a single assignment and determines its legality according to a set of policies while the second operation determines the legality of a set of assignments according to a set of policies. It will often be the case that a single assignment is legal by itself, however, when it is included into a set of assignments, it may become illegal due to policies such as the first two presented in this section.

Finally, we define the legality of a sequence of legal assignments, $\Phi' = [\Phi_1, \Phi_2, \ldots \Phi_n]$, in terms of the legality of the individual assignments.

$$\text{legal: } \Phi, \langle \Phi', E' \rangle, P(P) \rightarrow \text{Boolean}$$

where

$$\text{legal}(\Phi', P) = \text{legal}(\Phi_1, P) \wedge \text{legal}(\Phi_2, P) \ldots \text{legal}(\Phi_n, P) \quad (3)$$

2.7.2 Behavioral Policies

Behavioral policies P_{beh} define how agent in the organization should behave in relation to one another. For instance, in a conference review system, we would want to describe responsibilities of agent playing specific roles and their relationships to other roles. Although behavioral policies have been identified as part of OMACS, there was not time to define a formal language and semantics. However, the following presents a notional overview of how behavioral policies might be used in OMACS.

To refer to an agent playing a particular role, organizational predicates (`achieves`, `capable`, `possesses`, and `potential`) or assignment set membership can be used. Thus, to test whether a particular agent is playing a particular set of roles (e.g., the agent making final decisions cannot be an author of any papers for the conference), we can test for inclusion in Φ as follows.

$$\forall\ a{:}A,\ g1,g2{:}G\ \ \neg(\langle a,\text{Author},g1\rangle \in \Phi\ \wedge \langle a,\text{DecisionMaker},g2\rangle \in \Phi)$$

Although it is possible to state some requirements using only concepts from OMACS, other cases require the use of relationships between roles based on system/environment data. For instance, in the conference management system the relationships between roles based on the papers submitted, reviewed, or collected are vital. Thus, we must be able to talk about the data in the system as well, which is defined by the domain model, Σ.

2.7.3 Reorganization Policies

Application specific approaches to reorganization allow the designer to define heuristics to guide the system in its reorganization. For instance, instead of using a generic algorithm, the designer could specify the order in which agents should fill roles. OMACS models these heuristics as a special set of organizational polices called *reorganization* policies, P_{reorg}. Reorganization policies the designer to specify default reorganization strategies that are used prior to expensive computational approaches (see Section 2.16.2). Reorganization can first be attempted using these reorganization policies. If reorganization fails, these policies may be ignored and reorganization attempted using general purpose

(and more expensive) approaches. Application specific rules increase the reasoning efficiency in anticipated scenarios while providing robustness for unknown or uncommon cases [69][1].

Assignment policies (P_Φ) are only used to constrain the interactions between roles, protocols, and robots; they do not prescribe actions to be taken by the organization. To create reorganization policies, the organizational policies logic is used directly for carrying out reorganization actions. An example of a possible application specific reorganization rule is shown below.

$$<a1,r1,g1> \in \Phi \wedge \neg capable(a1,r1) \wedge capable(a2,r1) \Rightarrow <a2,r1,g1> \in \Phi' \wedge <a1,r1,g1> \notin \Phi'$$

Here, Φ' refers to the assignment set after the reorganization occurs. In this case, the rule specifies that if agent `a1` is playing `r1` to achieve goal `g1` and `a1` becomes incapable of playing role `r1`, then if `a2` is capable of playing role `r1`, it should be assigned to goal `g1` and `a1` should be de-assigned.

2.8 Domain Model

The domain model, Σ, is used to define object types in the environment and the relations between those types. The domain model is based on traditional object oriented class diagrams. They include object classes that each have a set of attribute types. Relations between object classes include general purpose associations as well as generalization-specialization and aggregation. Relations may also include multiplicities to constrain the number of object classes participating in any given relation.

The domain model Σ is a tuple ⟨O, *Rel*⟩ where

- O set of object types, which consists of public attributes
- *Rel* relation over O × O that defines various relationships between object types

An *object* O is a tuple of ⟨Attrs, C⟩ where

- Attrs set of datum
- C set of constraints over Attributes

A *datum*, d, is defined as a tuple ⟨name, type, value⟩ where

- name a string
- type a data type
- value a valid instance of the attribute data type

There are three types of relations in *Rel*,

- Rel_{Agg} a type of *Rel* denoting general aggregation relations between object types
- Rel_{Gen} a subset of *Rel* containing generalization-specialization relations between object types
- Rel_{Ass} a subset of *Rel* containing general associations between object types

The associations in a domain model can be used to define functions for talking about relations between environment object types. For instance, Figure 3 shows an ontology/domain model for a conference review system.

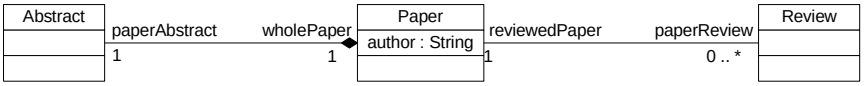


Figure 3. Conference Management Ontology/Domain Model

[1] An excellent example of this is given in [69] where human intuition led operators to propose reorganization when the automated algorithm deemed it unnecessary. Later analysis showed the operator's proposed reorganization was globally optimal.

The ontology/domain model defines an environment with a set of papers, each with an associated abstract and a set of reviews. Using the relations defined in the model, we can talk about the reviews a paper has received *paperReview(p)* or a paper's abstract *paperAbstract(p)*, etc. Multiplicities may constraint the number of allowable environment objects or the number of objects that may be related. Figure 3 defines a model where each abstract must have exactly one paper and each paper must have exactly one abstract. It also specifies that a review must be related to a single paper, while a paper may have any number of reviews on it (including none). Thus several organizational constraints can be defined in the domain model itself.

2.9 Organizational Assignment Function

Ideally, an organization will select the best set of assignments to maximize its ability to achieve its goals. As with the `rcf`, the selection of assignments may be application specific. Thus, each organization has its own application specific organization assignment function, `oaf`, which computes the *goodness* of the organization based on Φ.

$$\text{oaf: } \Phi \rightarrow 0 \,..\, \infty$$

With the `oaf`, the organization designer can specify how to make assignments based on a variety of organization specific constraints such as the importance of the specific goals or whether the assignment of multiple agents to a given role and goal will improve goal satisfaction. In the absence of an organization-specific organizational assignment function, we often define the `oaf` as the sum the scores in some assignment set Φ.

$$oaf = \sum_{<a,r,g> \in \Phi} potential\ (a, r, g)$$

2.10 Achieves

The *achieves* function defines how well a role achieves a goal. It can be predefined by the organization designer or learned before or during system operation. Each role is responsible for achieving specific system goals and may actually be able to achieve multiple goals. However, since some roles are better for achieving certain goals than other roles, OMACS must have an approach to determine which roles are preferred for which goals. Therefore, OMACS defines an `achieves` function that describes how well a particular role achieves a specific goal. The `achieves` function is a total function from the cross product of roles and goals to a real value in the range of 0 to 1.

$$\text{achieves: R, G} \rightarrow 0 \,..\, 1$$

Thus a role that cannot achieve a particular goal has an `achieves` value of 0, while any role that can achieve a goal would have an `achieves` value greater than zero. The achieves function is used along with the capable function (defined in Section 2.13) to define the potential of a specific assignment (see Section 2.14).

2.11 Requires

In order to perform a particular role, agents must possess a sufficient set of *capabilities* that allow the agent to carry out the role and achieve its assigned goals. For instance, to play the "president" role, a person would be expected to have knowledge of the corporation's domain, experience in lower-level jobs in similar types of companies, and experience in managing people and resources; an artificial organization is no different. Roles *require* a certain set of capabilities while agents possess a set of capabilities (see Section 2.12).

To capture the notion of a role requiring a specific capability, we have defined the `requires` predicate. If a role, `r`, requires a specific capability, `c`, then `requires(r,c)` is true.

$$\text{requires: } R, C \rightarrow \text{Boolean}$$

All roles require some level of capability, even if it is purely computational or communicative. Therefore, OMACS dictates that all roles *require* at least one capability.

$$\forall\ r{:}R \quad \#\{c \mid \text{requires}(r,c) > 0\} \geq 1 \tag{4}$$

2.12 Possesses

To be able to play a specific role, an agent must *possess* the capabilities required for that particular role. To capture a given agent's capabilities, we define a `possesses` function, which returns a value in the range of 0 to 1. The *possesses* function defines the quality of each capability that an agent has; 0 represents no capability while a 1 represents a high quality capability.

$$\text{possesses: } A, C \rightarrow 0 \mathinner{..} 1$$

As agent capabilities improve or degrade over time, the output of the possesses function is dynamic. Agents may learn and thus (hopefully) improve an agent's capability. However, an agent's capability may also degrade through either hardware failure or loss of access to/competition over a particular resource.

2.13 Capable

Using the capabilities required by a particular role and capabilities possessed by a given agent, we can compute the ability of an agent to play a given role, which we capture in the *capable* function. The capable function returns a value from 0 to 1 based on how well a given agent may play a specific role.

$$\text{capable: } A, R \rightarrow 0 \mathinner{..} 1$$

As described above, since the capability of an agent, `a`, to play a specific role, `r`, is application and role specific, OMACS provides a role capability function, `rcf` to compute the capable function for each agent-role pair. Thus, the *capability score* of an agent playing a particular role is defined via the designer defined role capability functions (rcf) for each organizational role.

$$\forall\ a{:}A\ r{:}R \quad \text{capable}(a,r) = r.\text{rcf}(a) \tag{5}$$

While the *rcf* is user defined, it must conform to one OMACS constraint. To be *capable* of playing a given role in the current organization, an agent must *possess* all the capabilities that are *required* of that role.

$$\forall\ a{:}A, r{:}R \quad \text{capable}(a,r) > 0 \Leftrightarrow \{c \mid \text{requires}(r,c)\} \subseteq \{c \mid \text{possesses}(a,c)\} \tag{6}$$

2.14 Potential

One of the goals of an organization is to provide a mechanism to distribute goals in such a way that agents work together toward accomplishing the top-level organization goal. As described above, these goals are achieved by assigning agents to specific roles in the organization. However, because the agents in an organization may be heterogeneous, some agents may play a particular role better than others. The *potential* function captures the ability of an agent to play a role in order to achieve a specific goal; it maps each agent-role-goal tuple to a real value ranging from 0 to 1, where 0 indicates that the agent-role-goal tuple cannot be used to achieve the goal. A non-zero values indicates how well an agent can play a role in order to achieve a goal.

$$\text{potential: } A, R, G \rightarrow 0 \mathinner{..} 1$$

The potential of an agent to play a specific role in order to achieve a specific goal is defined by combining the capable and achieves functions.

$$\forall\ a{:}A\ r{:}R\ g{:}G \quad \text{potential}(a,r,g) = \text{achieves}(r,g) * \text{capable}(a,r) \tag{7}$$

2.15 Organizational Agents

Organizational agents (OA) are organizations that function as agents in a higher-level organization. OAs allow OMACS to represent a hierarchy of organizations, providing OMACS with both flexibility and scalability. As agents, OAs may possess capabilities, coordinate with other agents, and be assigned to play roles. They represent an extension to the traditional Agent-Group-Role (AGR) model developed by Ferber [15, 18] and is similar to the organizational meta-model proposed by Odell [35].

OMACS defines two relationships between the higher-level organization and the OA's internal organization. First, there must be a connection between the role being played in the higher-level organization and the OA's internal `oaf` function. Second, a specific relationship must exist between the OA's internal capabilities and those of the higher-level organization.

Because the role the OA is playing will affect the internal organization of the OA, there must be a way to relate the organizational assignment function of the OA to its role. However, the `oaf` is defined as having no parameters and only has access to the local organizational components (see Section 2.9). Therefore, an OA must extend the definition of an organization by adding a new `oaf` function that allows it to take a parameter that includes a set of roles from the higher-level organization. Thus, an OA is an organization with one extension, a polymorphic `oaf` function that takes as input an assignment set along with a set of roles it has been assigned to play in the higher-level organization. Again, the polymorphic `oaf` function is application specific and must be written to take into account the specific roles the OA can take on in the higher-level organization.

$$\text{oaf} : \Phi, R \rightarrow 0 \,..\, \infty$$

The relationship between the capabilities in an OA and those of the higher-level organization is actually straightforward. Essentially, if a capability belongs to an agent that is part of the OA's internal organization, then those capabilities also exist in the higher-level organization by inclusion. Thus, if an OA, *a*, exists as an agent in an organization, *o*, then the capabilities possessed by *a* in *o* must be equivalent to the entire set of capabilities possessed by the individual agents in *a*'s internal organization. (Dot notation is used to differentiate between the capabilities of the organizations represented by *a* and *o* respectively.)

$$a.C \subseteq o.C \tag{8}$$

$$\forall\ ag{:}a.A,\ c{:}a.C \quad a.\text{possesses}(ag, c) > 0 \Rightarrow o.\text{possesses}(a, c) > 0 \tag{9}$$

Notice that we stop short of defining the actual possesses score for these capabilities in the higher-level organization. This is because there may be multiple agents in the OA's internal organization with the same capability. Thus, the actual possesses score will be application specific.

2.16 Organization and Reorganization

Each organization has an implicitly defined *organization transition function* that describes how the organization may transition from one organizational state to another over the lifetime of the organization. Since the team members (agents) as well as their individual capabilities may change over time, this function cannot be predefined, but must be computed based on the current state, the active goal set, G_{Active}, and the current policies. In our present research with purely autonomous teams, we have only considered reorganization that involves the *state* of the organization. However, we have defined two distinct types or reorganization: *state reorganization*, which only allows the modification of the organization state, and *structure reorganization*, which allows modification of the organization structure (and may require state reorganization to keep the organization consistent). We define the *state* of the organization as the set of agents, `A`, the `possesses`, `capable`, and `potential` functions, and the assignment set, Φ. However, not all these components may actually be under the control of the organization. For our purposes, we assume that agents may enter or leave organizations or relationships,

but that these actions are triggers that cause reorganizations and are not the result of reorganizations. Likewise, `possesses` (and thus `capable` and `potential` as well) is an automatic calculation that determines the possible assignments of agents to roles and goals in the organization. The calculation of `possesses` is totally under control of the agent (i.e. the agent may lie) and the organization can only use this information in deciding how to make assignments. This leaves one element that can be modified via state reorganization: Φ.

2.16.1 Reorganization Triggers

Various events may occur in the lifecycle of a multiagent team that may require it to reorganize. In general, *reorganization* is initiated when an event occurs that changes the current goals of the organization or forces a change in Φ. We discuss these two situations in detail below.

2.16.1.1 Goal Set Changes

Any change in `G` may cause reorganization. There are three basic types of events that can cause a change in `G`: (1) insertion of a new goal, (2) goal achievement, and (3) goal failure. Each of these is discussed below.

The first situation deals with new goals being added to `G`. However, we cannot say with certainty that reorganization will occur based on a new goal in `G`. It is possible that the organization will choose to forego reorganization for a number of reasons, the most likely being that it has simply chosen not to pursue any new goals added to `G` at the present time.

The second case deals with goal achievement. When a goal `g` is achieved, `G` is changed to reflect that event by (1) removing `g` from `G` and (2) possibly adding new goals, which are enabled by the achievement of `g`, into `G`. Obviously, the agent assigned to achieve goal `g` is now free to pursue other goals.

The third instance involves goal failure, which really has two forms: agent-goal failure and goal failure. When a specific agent cannot achieve goal `g` but `g` might still be achievable by some other agent, *agent-goal failure* occurs. When agent-goal failure occurs, reorganization must occur to allow the organization to (1) choose another agent to achieve `g`, (2) not pursue `g` at the current time, or (3) choose another goal to pursue instead of `g`. In any of these situations, `g` is not removed from `G` since it has not been achieved. In the case where the organization or the environment has changed such that a goal `g` can never be achieved, then *goal failure* occurs. In this case, `g` is removed from `G` and the organization must attempt to assess whether it can still achieve the overall system goals. Reorganization may occur to see if the agent assigned to achieve `g` can be used elsewhere. In all cases, the selection of the appropriate strategy is left to the organization.

When a maintenance goal `g` fails, it signifies the violation of a constraint that must be maintained. Typically, `g` requires a special agent/role to monitor for conditions that violate the constraint. When this constraint is violated, the maintenance goal fails. In this case, the organization must re-establish the violated constraint.

Ideally, there would be a way to annotate which goals should be pursued when maintenance goal is failed in order to re-establish the required condition; however, this requires additional research and extension of the goal model as presented in Chapter 3.

2.16.1.2 Assignment Set Changes

The second type of change that triggers reorganizations are changes within a team that force a change in Φ. Currently, we have limited our investigation of these changes to the set of agents, `A`, and their individual capabilities. When an agent that is part of Φ is removed from the organization, a reorganization must occur, even if only to remove the agent and its assignment(s) in Φ. Likewise, when

an agent that is part of Φ loses a capability that negates its ability to play a role that it is assigned, reorganization must occur as well.

In general, when changes occur in an agent's capability, a reorganization may or may not occur, based on the agent's `capable` relation. We have identified four specific types of changes in an agent's capabilities that may indicate a need for reorganization: (1) when an agent gains the ability to play a new role, (2) when an agent loses the ability to play a role, (3) when an agent increases its ability to play a specific role, or (4) when an agent decreases its ability to play a specific role. While case 2 requires reorganization if the agent is currently assigned to play the role for which it no longer has the capability to play, whether or not to reorganize is left up to the organization when the other three cases (along with 2 when the agent is not currently assigned that role) occur.

2.16.2 Reorganization

Reorganization is the process of changing the assignments of agents to roles to goals as specified in Φ. The organization's oaf function is used to determine the best new Φ; however, total team reorganization may not be necessary or efficient. (In the absence of any information or policies, a total reorganization would take on the order of $2^{A \times G \times R}$.)

One approach is to take a *local view*, in which the organization looks at the OMACS state and reorganizes in a locally optimal fashion (i.e. hill climbing). However, when dealing with dynamic environments, it is often desirable to reorganize so that the team will operate efficiently and effectively in its present situation as well as being adaptable to its changing environment. Thus, we would like to take a long-range or *global* view. Unfortunately, it has been shown that in the general case globally optimal reorganizations are NEXP-complete and, thus impractical for most applications with any time constraints. Therefore, OMACS provides a mechanism for augmenting the locally optimal algorithm with application specific rules to make reasoning more efficient and to enable globally better solutions.

2.16.2.1 Local Reorganization Techniques

For our general-purpose reorganization, we developed several reorganization algorithms, which provide a default reorganization capability. When a reorganization trigger occurs, the general-purpose reorganization algorithms are used to find appropriate organizations to achieve the organizations goals, if possible.

To compute the best reorganization, a centralized algorithm that optimizes the organization's oaf might seem appropriate; however, this approach is short sighted. First, it does not deal with the cost associated with reorganizing and, second, it does not consider the reason reorganizing was initially undertaken. Exploiting reorganizing costs requires a distributed solution since the cost for robots to change roles is not globally known. For instance, if an agent is required to perform a complex computation, any effort toward that computation would be lost if the agent was reassigned to another role/goal. Considering the reason for reorganization may enable a less extensive (and less costly) reorganization. If the reason for reorganizing is to fill a single role, then a total reorganization may be a waste of time and resources.

2.16.2.2 Application Specific Approaches

See the discussion of reorganization policies in Section 2.7.3 for a discussion of application specific approaches to reorganization.

Chapter 3 - Goal Model for Dynamic Systems

OMACS assumes that the organizational goals are simply a set of goals that need to be achieved; however, to capture the dynamic nature of an OMACS system, a goal model that allows designers to define adaptive behavior based on dynamic problem was required. A second major result of this research was the definition of the Goal Model for Dynamics Systems (GMoDS), which defines the operational semantics of a dynamically changing model of system goals.

In GMoDS, there are two main representations of system goals: the goal specification model, G_{Spec}, and a goal instance model, $G_{Instance}$. The goal specification model is a static representation of system goals specified by the system designer and includes the definitions of when goals may be created and pursued. The goal instance model is a dynamic model used at runtime to define the actual goals generated during system operation.

GMoDS is a tuple $\langle G_{Spec}, G_{Instance}, \textit{instanceOf} \rangle$ where

- G_{Spec} a goal specification model
- $G_{Instance}$ a goal instance model
- *instanceOf* function that maps goal instances from $G_{Instance}$ to goal classes in G_{Spec}

The goal specification tree, G_{Spec}, is defined as the tuple: $\langle G_S, E, Params, \textit{children}, \textit{precedes}, \textit{triggers}, \neg\textit{triggers} \rangle$

- G_S set(Goal)
- E set(Event)
- Params set(Parameters)
- *children* $\subseteq G_S \times G_S$
- *precedes* $\subseteq G_S \times G_S$
- *triggers* $\subseteq G_S \times E \times P(Params) \times G_S$
- $\neg$*triggers* $\subseteq G_S \times E \times P(Params) \times G_S$

A single *goal*, g, is defined as a tuple $\langle$state, attr$\rangle$ where

- state expression defining a desired state of the world parameterized by attr
- attr set(Parameter)

There are also a set of predicates defined over a goal:

- conjunctive : $G_S \rightarrow$ Boolean
- disjunctive : $G_S \rightarrow$ Boolean
- achievement : $G_S \rightarrow$ Boolean
- maintenance : $G_S \rightarrow$ Boolean
- achieved : $G_S \rightarrow$ Boolean
- failed : $G_S \rightarrow$ Boolean
- preceded : $G_S \rightarrow$ Boolean
- obviated : $G_S \rightarrow$ Boolean

An *event*, e, simply has a $\langle$name$\rangle$ attribute where

- name a string

A *parameter*, p, is defined as a tuple $\langle$name, type, value$\rangle$ where

- name a string
- type a data type
- value a valid instance of the attribute data type

To make the discussions easier, the following functions are defined over the relations defined in GMoDS:

Function Signature	Function Definition
parent : $G_S \rightarrow P(G_S)$	parent(g) = $\{g_1{:}G_S \mid (g_1,g) \in \text{children}\}$
children : $G_S \rightarrow P(G_S)$	children(g) = $\{g_1{:}G_S \mid (g,g_1) \in \text{children}\}$
precedes : $G_S \rightarrow P(G_S)$	precedes(g) = $\{g_1{:}G_S \mid (g,g_1) \in \text{precedes}\}$
triggers : $G_S \times \text{Event} \rightarrow P(G_S)$	triggers(g,e) = $\{g_1{:}G_S \mid \forall\ \text{p:Params}\ (g,e,p,g_1) \in \text{triggers}\}$
triggers : $G_S \times \text{Event} \times P(\text{Params}) \rightarrow P(G_S)$	triggers(g,e,p) = $\{g_1{:}G_S \mid (g,e,p,g_1) \in \text{triggers}\}$
$\neg$triggers : $G_S \times \text{Event} \rightarrow P(G_S)$	$\neg$triggers(g,e) = $\{g_1{:}G_S \mid \forall\ \text{p:Params}\ (g,e,p,g_1) \in \neg\text{triggers}\}$
$\neg$triggers : $G_S \times \text{Event} \times P(\text{Params}) \rightarrow P(G_S)$	$\neg$triggers(g,e,p) = $\{g_1{:}G_S \mid (g,e,p,g_1) \in \neg\text{triggers}\}$
triggeredBy : $G_S \times \text{Event} \times P(\text{Params}) \rightarrow P(G_S)$	triggeredBy(g,e,p) = $\{g_1{:}G_S \mid (g_1,e,p,g) \in \text{triggers}\}$
$\neg$triggeredBy : $G_S \times \text{Event} \times P(\text{Params}) \rightarrow P(G_S)$	$\neg$triggeredBy(g,e,p) = $\{g_1{:}G_S \mid (g_1,e,p,g) \in \neg\text{triggers}\}$

3.1 Goals

Every artificial organization is designed with a specific purpose, which defines the overall function of the organization. *Goals* can be thought of as a desirable situation [39] or the objective of a computational process [65]. In GMoDS, there are two types of goals: goal classes and goal instances (which are analogous to object classes and object instances). Goal classes define templates from which goal instances are created. A goal class consists of a *state* expression that defines the state that the goal represents and a set of goal *attributes* that are used to parameterize the state expression. When a goal is instantiated, all its attributes must be given explicit values, which in turn concretely define the goal expression.

Because many goals are general in nature and apply to many situations, we allow goals to be parameterized. These parameters typically define attributes of the goal that are required when determining if a goal has been achieved. For example, the goal "search area" requires that the "area" be defined. Instead of defining a general goal `g`, we include parameters such as `g(area)`. During the operation of the organization, parameterized goals must be instantiated with concrete values for each parameter. Thus we might instantiate this particular goal as `g(room1)`, which would be interpreted as "search room1". Only instantiated goals may be assigned to agents to be achieved.

Each goal in an organization can be classified as either an achievement goal or a maintenance goal. *Achievement goals* require the organization to take action to produce a state that meets the goal requirements whereas *maintenance goals* require the organization to ensure that a certain state is maintained as long as the maintenance goal's siblings are active. Thus, if the organization wants to ensure that a specific state exists at all times, then its root goal will have a maintenance subgoal. If a goal has a maintenance sub-goal, then its achievement can only occur when all achievement sub-goals have been achieved and its maintenance sub-goals are maintained. To determine whether a goal, g, is an achievement goal or a maintenance goal, we define two predicates, `achievement` and `maintenance`. The achievement and maintenance attributes are exclusive; each goal must be defined as either an achievement goal or a maintenance goal.

$$\forall\ g{:}G_S \mid g.\text{achievement} \otimes g.\text{maintenance} \qquad (1)$$

The predicates achieved, failed, preceded, and obviated refer to the state of a particular goal instance. The predicate values are defined upon goal instantiation. A goal is *achieved* when the goal state becomes true and *failed* when it is determined that the goal state cannot be achieved. A goal is *preceded* when all goals that precede it have been achieved and it can be pursued. Finally, a goal is *obviated* when achieving it no longer is useful in terms of reaching the overall goal g_0. Thus, there are a few obvious constraints on these values (G_I is the set of goal instances):

$$\forall\ g{:}G_I \mid achieved(g) \Rightarrow preceded(g) \quad (2)$$
$$\forall\ g{:}G_I \mid \neg\ preceded(g) \Rightarrow \neg\ achieved(g) \quad (3)$$
$$\forall\ g{:}G_I \mid obviated(g) \Rightarrow \neg\ achieved(g) \quad (4)$$
$$\forall\ g{:}G_I \mid achieved(g) \Rightarrow \neg\ obviated(g) \quad (5)$$

3.1.1 Goal Specification Tree

3.1.1.1 Children

The *children* relation defines the structure of the goal specification tree. The root of the tree is g_o, which has no parents; every other goal class in the specification tree has exactly one parent and descends from g_o.

$$\forall\ g{:}G_S \mid g \notin children^+(g) \quad (6)$$
$$\exists\ g_o{:}G_S, \forall g{:}G_S \backslash \{g_o\} \mid g \in children^+(g_o) \wedge parent(g_o)=\{\} \wedge \#parent(g)=1 \quad (7)$$

Each non-leaf goal class is refined conjunctively or disjunctively; leaf goals are neither. Thus, each goal class is statically defined as conjunctive or disjunctive while leaf goals are neither conjunctive nor disjunctive. This is defined formally as follows ($\otimes$ is the exclusive-or operator).

$$\forall g{:}G_S \mid children(g) \neq \{\} \Rightarrow g.conjunctive \otimes g.disjunctive \quad (8)$$
$$\forall g{:}G_S \mid children(g) = \{\} \Rightarrow \neg g.conjunctive \wedge \neg g.disjunctive \quad (9)$$

3.1.1.2 Precedes

Conceptually, the *precedes* relation determines which goals must be achieved before a given goal may even be attempted. As defined above, *precedes(g)* returns a set of goals that may be achieved after g has been achieved. Precedence is not valid between ancestors or descendants in the goal tree. Likewise, we cannot have a circular precedence relation.

$$\forall\ g_1,g_2{:}G_S \mid g_2 \in precedes(g_1) \Rightarrow g_2 \notin children^+(g_1) \wedge g_1 \notin children^+(g_2) \quad (10)$$
$$\forall\ g{:}G_S \mid g \notin precedes^+(g) \quad (11)$$

3.1.1.3 Triggers

The *triggers* relation is similar to *precedes* in that it restricts specific goals from being pursued until a specific event occurs. For instance, assume that goal *A* triggers goal *B* based on event *E*. Instead of requiring the system to achieve goal A before pursuing goal B, the trigger relation requires the system to instantiate a new instance of goal B when event E occurs *during* the achievement of goal A. This relationship is quite natural and useful. For example, in a search and rescue system when an agent is searching an area, a rescue goal should be instantiated for each victim found. Thus, the event "victim found", which occurs during achievement of the "search area" goal, triggers the creation of a new instance of the "rescue victim" goal. Since when an event triggers a parent goal, the entire subtree under the parent goal is triggered, children of the parent goal cannot be triggered independently of the parent unless triggered by a sibling of the same parent.

$$\forall\ g_1,g_2,g_3{:}G_S, \forall\ e_1,e_2{:}E \mid g_2 \in triggers(g_1,e_1) \wedge g_3 \in children^+(g_2) \Rightarrow triggeredBy(g_3,e_2) \subseteq children(g_2) \quad (12)$$

As with precedence, it is also not allowable to have a loop of goal classes that trigger each other, nor is triggering allowed between ancestors or descendants.

$$\forall\ g{:}G_S \mid g \notin triggers^+(g) \quad (13)$$
$$\forall\ g_1,g_2{:}G_S \mid g_2 \in triggers(g_1) \Rightarrow g_2 \notin children^+(g_1) \wedge g_1 \notin children^+(g_2) \quad (14)$$

Generally, goals that are triggered by an event are parameterized based on information related to that event. For example, the triggered goal "rescue victim", is parameterized on the location of the victim to

be rescued. When a victim is discovered, an instance of the "rescue victim" goal is created and the parameter is given an explicit value. Thus, it is required that the parameters of an event match one-to-one with the attributes of the goal that it triggers.

3.1.1.4 Negative Triggers

Negative triggers are the opposite of goal triggers. Instead of adding new goals to the goal instance tree, negative triggers remove existing goal instances from the goal instance tree. Like triggers, they are activated on a specific event. They also have a set of parameters, whose values must exactly match those of an existing goal instance for the negative trigger to remove a goal. Within the goal specification tree there are few restrictions on the use of negative triggers as there can be loops and negative triggers can be used to remove ancestor as well as predecessor goals.

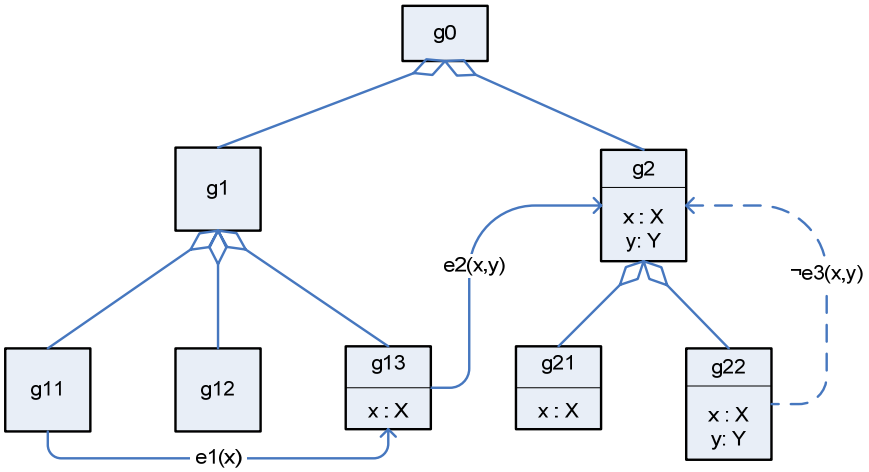


Figure 4. Goal Specification Tree

Figure 4 is an example of a goal specification tree with both triggers and negative triggers. In this example, during the pursuit of goal g_{11} the occurrence of event e_1 will trigger the creation of a new instance of g_{13} parameterized with the value x from event e_1. Likewise, during the pursuit of each instance of g_{13}, each occurrence of the e_2 (with parameters x and y) will cause a new instance of the subtree headed by goal g_2 (including goals g_2, g_{21}, and g_{22}) to be created.

The negative trigger $\neg e_3$ indicates that during the pursuit of g_{22}, if event e_3 occurs, then an instance of the g_2 subtree (including subgoals g_{21}, and g_{22}) whose parameters match e_3's x and y values will be removed from the current set of goal instances. If there is no g_2 goal with matching values of x and y, then no goals are removed.

3.1.1.5 Annotated Goal Specification Tree

Each goal in G_S must be triggered for an instance to be placed in G_I. There are however not all goal classes in the goal specification tree have an explicit trigger. Some goals, such as g_0, are instantiated once when the system is initialized. Other goals are instantiated when their parent goals are triggered. Since g_0 cannot have an explicit trigger, a distinguished trigger, the *initial trigger*, is automatically added to the goal specification tree. Thus, Figure 5 is the original goal specification tree while in Figure 6, the initial trigger, $e_0()$, has been added to g_0. Each goal that does not have a trigger is implicitly triggered when its parent is triggered as is explained below.

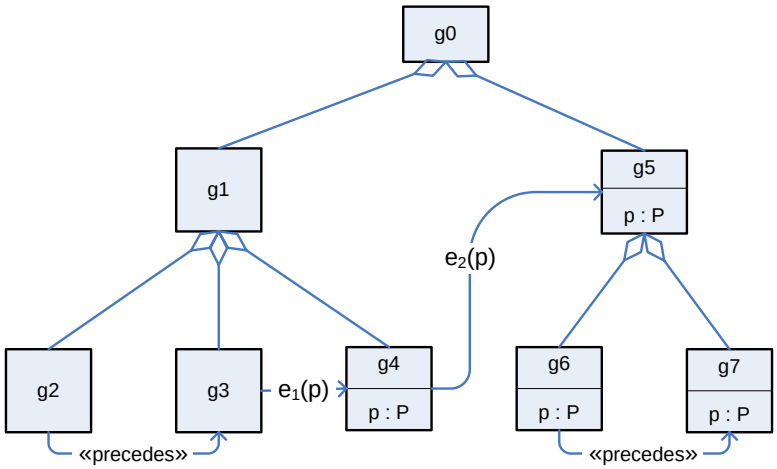


Figure 5. Example Goal Specification Tree

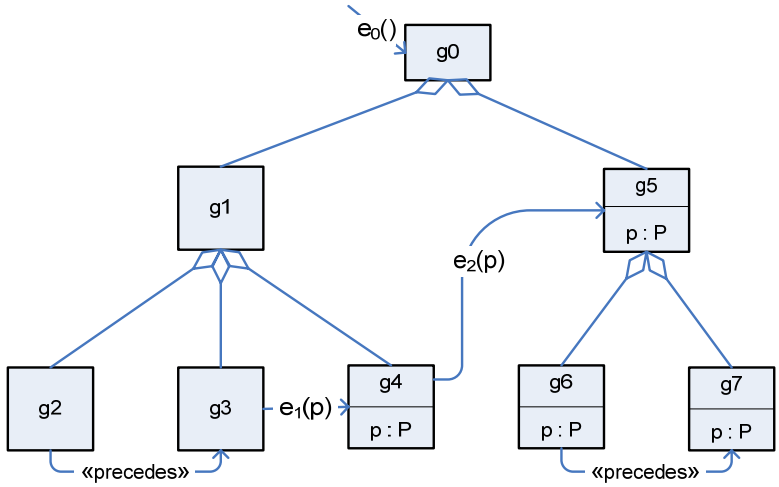


Figure 6. Annotated Goal Specification Tree

3.1.2 Goal Instance Tree

As described above, the goal specification tree allows the designer to specify a set of goal classes that can be used as a template for creating goal instances at runtime. The system can dynamically create and destroy goals in the instance tree. The instance tree ($G_{Instance}$) retains the static tree structure but still allows for dynamism of goal triggering.

$G_{Instance}$, is defined as a tuple: $\langle G_I, E, Params, \textit{children}, \textit{precedes}, \textit{triggers}, \neg\textit{triggers}\rangle$

- G_I set(Goal)
- E set(Event)
- Params set(Parameters)
- *children* $\subseteq G_I \times G_I$
- *precedes* $\subseteq G_I \times G_I$
- *triggers* $\subseteq G_I \times E \times P(Params) \times G_I$
- $\neg$*triggers* $\subseteq G_I \times E \times P(Params) \times G_I$

A single *goal*, g, is defined as a tuple ⟨state, attr⟩ where

- state expression defining a desired state of the world parameterized by attr
- attr set(Parameter)

There are also a set of predicates defined over a goal:

- conjunctive : $G_I \rightarrow$ Boolean
- disjunctive : $G_I \rightarrow$ Boolean
- achievement : $G_I \rightarrow$ Boolean
- maintenance : $G_I \rightarrow$ Boolean
- achieved : $G_I \rightarrow$ Boolean
- failed : $G_I \rightarrow$ Boolean
- preceded : $G_I \rightarrow$ Boolean
- obviated : $G_I \rightarrow$ Boolean

An *event*, e, simply has a ⟨name⟩ attribute where

- name a string

A *parameter*, p, is defined as a tuple ⟨name, type, value⟩ where

- name a string
- type a data type
- value a valid instance of the attribute data type

Functions similar to those defined for G_S are defined over the relations for goal instances in G_I:

Function Signature	Function Definition
parent : $G_I \rightarrow P(G_I)$	parent(g) = $\{g_1{:}G_I \mid (g_1,g) \in \text{children}\}$
children : $G_I \rightarrow P(G_I)$	children(g) = $\{g_1{:}G_I \mid (g,g_1) \in \text{children}\}$
precedes : $G_I \rightarrow P(G_I)$	precedes(g) = $\{g_1{:}G_I \mid (g,g_1) \in \text{precedes}\}$
triggers : $G_I \times \text{Event} \rightarrow P(G_I)$	triggers(g,e) = $\{g_1{:}G_I \mid \forall\ p{:}\text{Params}\ (g,e,p,g_1) \in \text{triggers}\}$
triggers : $G_I \times \text{Event} \times P(\text{Params}) \rightarrow P(G_I)$	triggers(g,e,p) = $\{g_1{:}G_I \mid (g,e,p,g_1) \in \text{triggers}\}$
$\neg$triggers : $G_I \times \text{Event} \rightarrow P(G_I)$	$\neg$triggers(g,e) = $\{g_1{:}G_I \mid \forall\ p{:}\text{Params}\ (g,e,p,g_1) \in \neg\text{triggers}\}$
$\neg$triggers : $G_I \times \text{Event} \times P(\text{Params}) \rightarrow P(G_I)$	$\neg$triggers(g,e,p) = $\{g_1{:}G_I \mid (g,e,p,g_1) \in \neg\text{triggers}\}$
triggeredBy : $G_I \times \text{Event} \times P(\text{Params}) \rightarrow P(G_I)$	triggeredBy(g,e,p) = $\{g_1{:}G_I \mid (g_1,e,p,g) \in \text{triggers}\}$
$\neg$triggeredBy : $G_I \times \text{Event} \times P(\text{Params}) \rightarrow P(G_I)$	$\neg$triggeredBy(g,e,p) = $\{g_1{:}G_I \mid (g_1,e,p,g) \in \neg\text{triggers}\}$

When a goal is triggered, either a single leaf goal or an entire subtree is created depending on whether the triggered goal is a leaf goal or not. The children of a triggered goal are implicitly triggered by the creation of the parent goal. The implicit triggering of a child goal creates a single new instance upon the triggering of a parent goal. In the example in Figure 6, g_6 and g_7 are also triggered when g_5 is triggered, as they implicitly inherit the $e_2(p)$ trigger.

Goal instances are created from the goal classes defined in the goal specification tree. Because it is illogical to try to achieve the same goal instance twice, each goal instance must be unique in terms of goal class and actual parameter values. For instance, if a goal class exists for rescuing victims at location (x,y), it does not make sense to have two concurrent goal instances for rescuing a victim at the exact same location. Thus, an attempt to trigger a duplicate goal instance will result in no change to the current goal instance tree.

Upon system startup, the initial trigger, $e_0()$, is implicitly triggered causing the creation of all goals triggered by $e_0()$. Because g_1, g_2, and g_3 inherited the initial trigger, those goals, along with goal g_0 are created in the initial goal instance tree as shown in Figure 7.

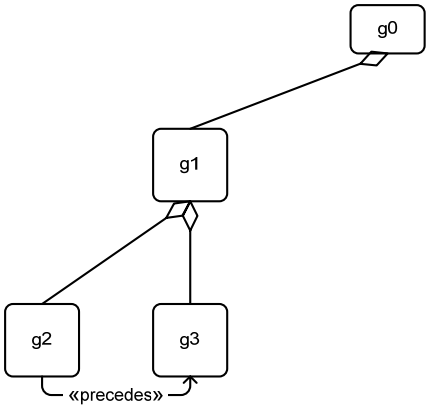


Figure 7. Goal Instance Tree after the Initial Trigger

Once the initial instance tree is created, there are actually two goals, g2 and g3, that are available to be achieved. However, because g_2 precedes g_3, only g_2 may actually be pursued. Goal instances that can actually be pursued are part of the *active goal set*, G_{Active}, which is discussed in detail in Section 3.2. Since g_2 is the only goal in G_{Active} and it cannot trigger any new goal instances, it must be either be achieved or failed. Once goal g_2 is achieved (denoted by a grey box in Figure 8), it is marked as achieved in the goal instance tree as shown in Figure 8, and goal g_3 is moved into G_{Active}.

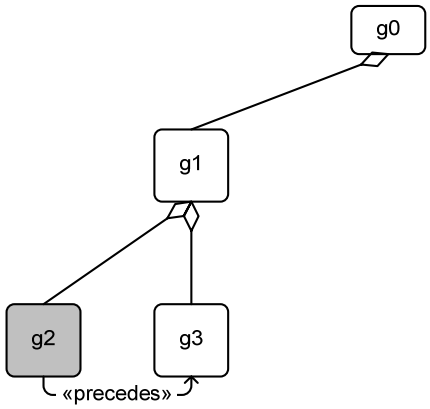


Figure 8. Goal Instance Tree after the achievement of g2

During the pursuit of g_3, if an event of type e_1 occurs, then a new instance of goal g_4 is added to the goal instance tree under g_1. Assuming event $e_1(a)$ occurs, the goal instance tree in Figure 8 is transformed into the goal instance tree in Figure 9. In this case, a new instance of g_4 is added to the tree with the parameter value of *a*, which is the value from the event $e_1(a)$. At this point, if another an event of type e_1 occurs during the pursuit of g_3, a second instance of g_4 will be added to the goal instance tree under g_1. Note, however, that this new instance of event e_1 must have a value other than *a*. A second $e_1(a)$ event will attempt to trigger a second instance of goal g_4 with parameter *a*, which is illegal given our requirement that goal instances be unique based on goal type and parameter values.

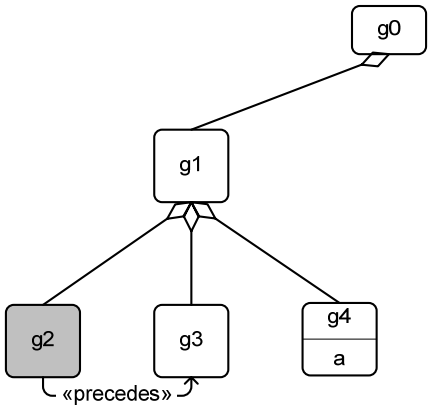


Figure 9. Goal Instance Tree after event e1(a)

Once goal g_4 becomes active, there are two types of events that can trigger the creation of new goal instances: $e_1$and e_2. For event e_2, notice from Figure 6 that it actually triggers the creation of three new goals: g_5, g_6 and g_7. Thus, if event e_2(b) occurs during the pursuit of goal g_4, an entire subtree under g_0 is created as shown in Figure 10.

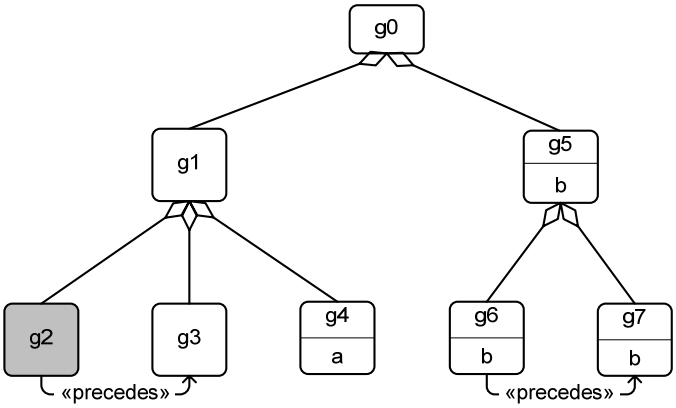


Figure 10. Goal Instance Tree after event e2(b)

As any number of any specific event may occur during pursuit of a goal, it is likely that similar events will occur (albeit with different parameters). Thus, if event e_2(c) occurs during the pursuit of goal g_4, an second unique subtree under g_0 is created as shown in Figure 11.

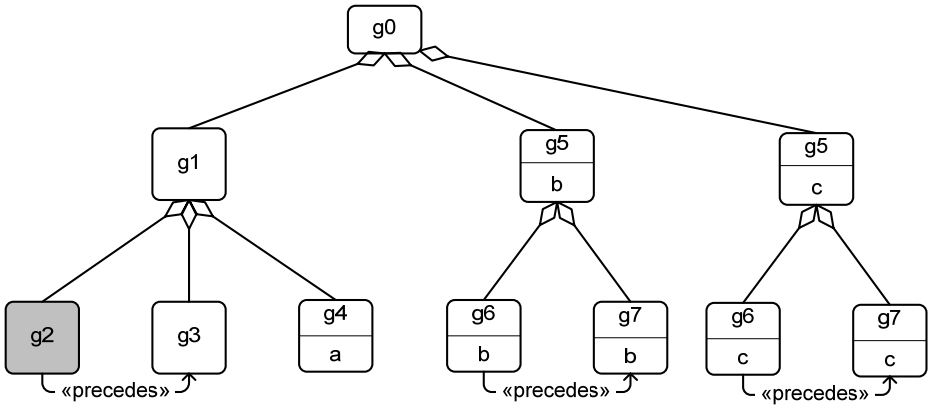


Figure 11. Goal Instance Tree after Event e2(c)

3.1.2.1 Goal Instance Achievement

As discussed above, non-leaf goals are statically defined as either conjunctive or disjunctive and therefore, the condition for their achievement is also statically defined. However, because subgoals may not be triggered at all or be triggered multiple times, the actual achievement condition cannot be defined directly in terms of subgoals from the goal specification tree. For instance, in the example in Figure 6, one cannot say that

$$\text{achieved}(g_1) = \text{achieved}(g_2) \wedge \text{achieved}(g_3) \wedge \text{achieved}(g_4)$$

since there may be zero or many instances of g_4. Thus, the achievement conditions must be specified in more general terms.

GMoDS defines the achievement of non-leaf goals as follows.

$$\forall\ g{:}G_I \mid \text{children}(g) \neq \{\} \wedge \text{conjunctive}(g) \Rightarrow$$
$$[\text{achieved}(g) \Leftrightarrow \text{completed}(g) \wedge \forall\ g_1{:}\text{children}(g) \mid \text{achieved}(g_1)] \quad (15)$$

$$\forall\ g{:}G_I \mid \text{children}(g) \neq \{\} \wedge \text{disjunctive}(g) \Rightarrow [\text{achieved}(g) \Leftrightarrow \exists\ g1{:}\text{children}(g) \mid \text{achieved}(g_1)] \quad (16)$$

The completed(g) predicate defines when no more children of a goal g can be instantiated. For instance, in the example in Figure 12, determining when g_1 has been completed requires determining whether the $e_3(p)$ trigger can occur. Of course, event $e_3(p)$ may occur as long as there is an instance goal of g_{32} active or if a new goal of type g_{32} can be instantiated. Since goal g_{31} can trigger instances of goal g_{32} on event $e_2(p)$, goals of type g_{32} may be instantiated as long as there is still an active instance of goal g_{31}. Furthermore, since goal g_2 may instantiate instances of g_3 and thus g_{31} and g_{32}, no instances of g_2 may be active either.

$$\forall\ g, g_1{:}G_I \mid \text{completed}(g) \Leftrightarrow [g_1 \in \text{triggeredBy}^+(g) \Rightarrow \text{achieved}(g_1) \vee \text{failed}(g_1) \vee \text{obviated}(g_1)] \quad (17)$$

Where the goal instance version of triggeredBy is defined as follows.

$$\forall\ g{:}G_I, e{:}E \mid \text{triggeredBy}(g) \equiv$$
$$\{g_1{:}G_I \mid g \in \text{triggers}(g_1,e)\} \cup \{g_2{:}G_I \mid \exists\ g_1{:}\text{parent}^+(g)\ \ g1 \in \text{triggers}(g_2,e)\} \quad (18)$$

Notice that in Figure 11, the achievement condition of goal g_0 does not change as it already requires the achievement of all its children. As more goals are triggered, g_0 simply gains more children. Thus for Figure 11 achievement of g_0 requires the achievement of g_1, $g_5(b)$, and $g_5(c)$.

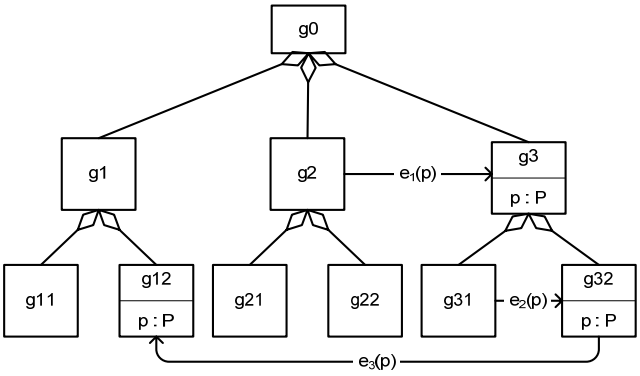


Figure 12. Example 2 Goal Specification Tree

3.1.2.2 Goal Instance Precedence

While the example above had some simple goal precedence in it, goal precedence is not always so straightforward. Due to the ability to instantiate new goal instances that might be defined as preceding

another existing goal instance, care must be taken to ensure that all possible cases are considered. For instance, in Figure 13 goal g_3 precedes g_4 (the left hand side of Figure 13 is the goal specification tree and the right hand side of Figure 13 is a goal instance tree). Given the goal instance tree in Figure 13, the questions is whether g_4 can become active when g_3 is achieved?

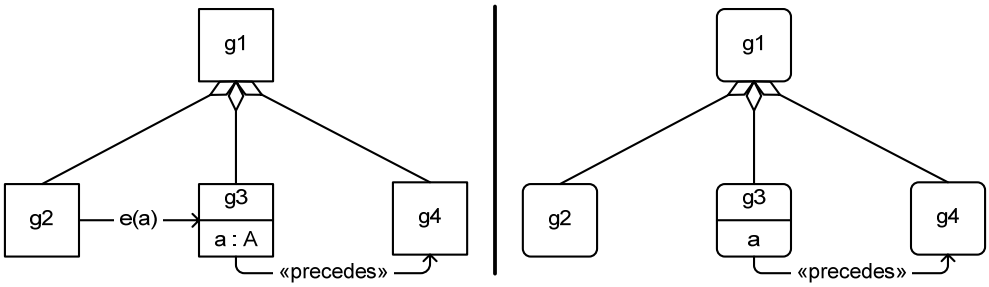


Figure 13. Goal Precedence Specification Problem

Initially one is tempted to state that g_4 should become active once g_3 is achieved. However, this is problematic since event e(b) might still occur during the pursuit of g_2 thus generating another instance of g_3 as shown in Figure 14. In this case, it is clear that g_4 cannot become active until all instances of g_3 are achieved and there is no longer a chance that a new g_3 can be instantiated.

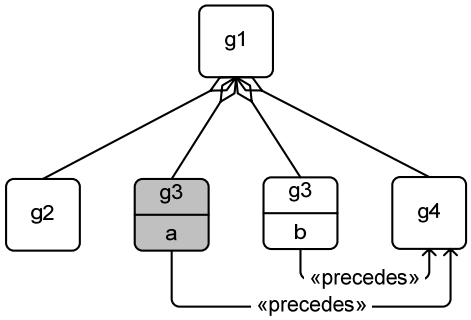


Figure 14. Precedence with Two Preceding Goal Instances

In order for a preceded(g) to become true, all of the instances (existing and future) of the goals that precede that goal must be achieved. Intuitively, this means that in order for a goal instance g to be *preceded* all goals that could possibly trigger a goal instance that precedes g must be achieved.

$$\forall\, g{:}G_I \mid \text{preceded}(g) \Leftrightarrow$$
$$\forall\, g_1{:}\text{precedes}(g), e{:}E, g_2{:}\text{triggeredBy}^{+}(g_1,e) \mid \text{achieved}(g_1) \wedge \text{achieved}(g_2) \qquad (19)$$

3.2 Goal Lifecycle

While a goal tree is useful for stating what a multiagent system should do, the tree itself is not sufficient to determine what goals should be pursued by the system or when specific goals have been achieved. To help keep track of the status of the various system goals, a group of goal sets is used as shown in Figure 15.

Leaf goal instances in G_I are partitioned into five sets: G_T, G_{Active}, $G_{Obviated}$, G_{Failed}, and $G_{Achieved}$. G_T is the set of all goals that have been triggered, but are not yet preceded. The second set of interest is the set goals that have been *triggered* by some event, G_T. The active goal set G_{Active} denotes those goals that have been triggered and have all their *precedence* requirements satisfied (i.e., all goals that precede them have been achieved). Intuitively, G_{Active} represents all the goals that the system can work on (or pursue) at any given time. Once in the G_{Active}, a goal can be achieved, failed, or obviated. Both goal achievement and goal failure are determined by the agent assigned to achieve the goal. Thus for GMoDS purposes, if

the agent assigned to achieve a goal instance states that it is achieved or failed, it is placed into the *Achieved Goal Set*, $G_{Achieved}$ or the *Failed Goal Set*, G_{Failed}.

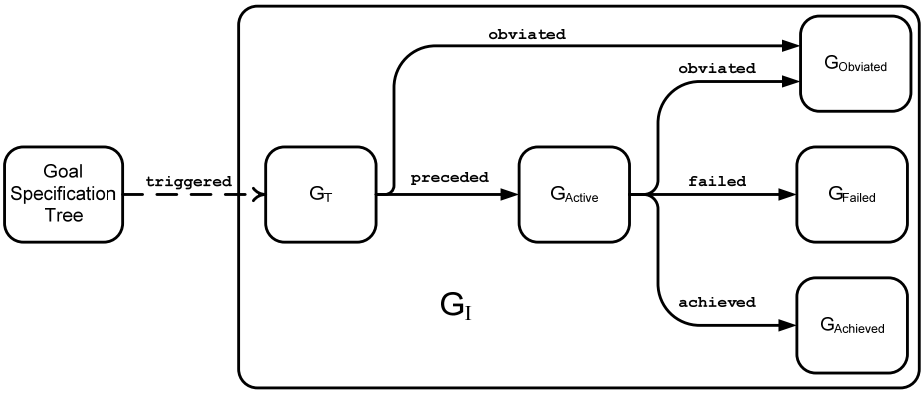


Figure 15. Goal Lifecycle

3.2.1 Triggered Goal Set

The triggered goal set, G_T, contains all goals that have been triggered (implicitly or explicitly) by some event during the system operation. The *triggers* relation defined above controls when goals are inserted into G_T. When a triggering event occurs, it causes the creation of a new leaf goal instance that is moved into G_T. For example, in a search and rescue system, when an agent is searching a particular area a "rescue victim" goal should be created and placed into G_T for each victim found. Thus, the event "victim found", which occurs during achievement of the "search area" goal, triggers the creation of a new instance of the "rescue victim" goal.

$$\forall\, g{:}G_I \mid g \in G_T \Leftrightarrow \text{preceded}(g) \wedge \neg\text{achieved}(g) \wedge \neg\text{obviated}(g) \wedge \neg\text{failed}(g) \qquad (20)$$

At system startup, the implied *init* event places instances of all non-triggered leaf goals into G_T, thus effectively stating which goals may be available to be pursed immediately (assuming they are not preceded by some other goal).

3.2.2 Active Goal Set

The *active goal set*, G_{Active} is the set of goal instances that are available to be pursued by an organization. Once a goal is inserted into G_T, it is moved into G_{Active} as soon as all of the goals that precede it are achieved. Formally, we define the transition from G_T to G_{Active} as

$$\forall\, g{:}G_I \mid g \in G_{Active} \Leftrightarrow \neg\text{preceded}(g) \wedge \neg\text{achieved}(g) \wedge \neg\text{obviated}(g) \wedge \neg\text{failed}(g) \qquad (21)$$

Later, the notion of a sequence of valid active goals sets will be crucial in defining important properties of an organization. A sequence of active goal sets what goals are available to be pursued by the system over time. We denote a sequence of active goal sets G_{Active}' as

$$G_{Active}' = [G_{Active1}, G_{Active2}, \ldots G_{Activen}]$$

3.2.3 Obviated Goal Set

Goals are placed in the obviated set when their achievement is no longer necessary in order to achieve the top level goal, g_0. Unlike achieved or failed goals, goal instances may be obviated without input from an agent. Goal obviation refers to the case where the goal instance is no longer needed to achieve the overall system goal g_0, which occurs when a parent goal has already been achieved due to conjunctive goals. Thus, goal instance obviation is defined as follows.

$$\forall\, g{:}G_I \mid \text{obviated}(g) \Leftrightarrow \exists\, g_1{:}\text{parent}^{+}(g) \mid \neg\text{achieved}(g) \wedge \neg\text{failed}(g) \wedge \text{achieved}(g_1) \qquad (22)$$

Thus, when a goal becomes obviated, it is moved into the obviated set.

$$\forall\ g{:}G_I \mid g \in \mathtt{G}_{\mathtt{Obviated}} \Leftrightarrow \neg\text{achieved}(g) \wedge \text{obviated}(g) \wedge \neg\text{failed}(g) \tag{23}$$

3.2.4 Failed Goal Set

A goal is moved into the failed goal set when the agent assigned to pursue it determines that it is failed and can never be achieved. As described in Section 2.16.1.1, *goal failure* is different from an individual agent failing to achieve the goal (agent-goal failure). In the case of *agent-goal failure*, the goal may still be achieved and the goal stays in $\mathtt{G}_{\mathtt{Active}}$ and reorganization occurs. To be categorized as a *goal failure*, the goal must have been in the active goal set and had not been achieved or obviated.

$$\forall\ g{:}G_I \mid g \in \mathtt{G}_{\mathtt{Failed}} \Leftrightarrow \neg\text{preceded}(g) \wedge \neg\text{achieved}(g) \wedge \neg\text{obviated}(g) \wedge \text{failed}(g) \tag{24}$$

3.2.5 Achieved Goal Set

When an agent is assigned to a goal, it will attempt to achieve that goal until (1) the goal is achieved, (2) goal failure occurs, (3) agent-goal failure occurs, (4) the goal is reassigned to another agent, or (5) the goal becomes obviated. When an agent *achieves* a goal, it is moved into $\mathtt{G}_{\mathtt{Achieved}}$. Non-leaf goals may also be moved into $\mathtt{G}_{\mathtt{Achieved}}$ based on the achievement of their children. Thus, when a goal is achieved by an agent, its parent goal achievement condition is checked to see if it is also achieved. This check is recursive and continues up the tree until the parent goal has not been achieved. When g_o is moved into $\mathtt{G}_{\mathtt{Achieved}}$, the system has achieved its goal.

$$\forall\ g{:}G_I \mid g \in \mathtt{G}_{\mathtt{Achieved}} \Leftrightarrow \neg\text{preceded}(g) \wedge \text{achieved}(g) \wedge \neg\text{obviated}(g) \wedge \neg\text{failed}(g) \tag{25}$$

Chapter 4 Viability with OMACS and GMoDS

The constraints in Chapter 2 define the validity of the organization structure and its instances. However, we are also interested in whether or not an assignment of agents to roles satisfying all the organizational policies exists that *can* allow the team to reach its top-level team goals, which we refer to as *organizational viability*. Although an organization may be structurally valid, there is no guarantee that an instance of that organization exists that *can* achieve its goals. In actuality, we can never guarantee that the team *will* ever reach its top-level goal due to the dynamic nature of the environment in which the organization operates. To achieve the organizational goals, the team must have the right mix of agents to play the right roles to achieve those goals. Essentially, a viable organization is a valid organization that has been populated with the right types and numbers of agents so that it might potentially achieve its goals.

> ***Viability*** – an organization, O, is viable if there exists a series of assignments of agents to roles to goals – which is consistent with its goal model, `G`, and its policies, `P` – that can achieve its top-level goal.

As implied by the definition, we must know the structure and semantics of the organization's goal model to define viability formally. Thus, in the remainder of this chapter, we discuss the viability of systems designed using OMACS and GMoDS.

For an organization to be viable, according to the definition above, it must have the roles and agents to achieve its top-level goal, under ideal conditions (no agent failures, etc.). Therefore, we define a viable organization as an organization that is able to show that the top-level organization goal is achievable by some set of assignments of goals, roles, and agents. When a given goal (or sub-goal) is achievable by a set of assignments, we term that goal *satisfiable*. Therefore, viability refers the satisfiability of an organization's top-level, or overall goal g_o.

Notice that viability does not require all goals be satisfiable since we allow the notion of disjunctive goals in which only a single subgoal must be satisfiable. It merely requires that it is possible to find a suitable set of assignments so that the top-level goal of the organization may be achieved.

However, viability does require that the set of assignments used to determine satisfiability is consistent, or legal with respect to the organizations policies, `P`. Thus, to show viability, we must show that an organization's top-level goal is satisfiable using only legal sets of assignments.

While viability does not require goals to be achieved in a specific order, it does require that the order of goal achievement is consistent with the precedence rules in the goal hierarchy. This requirement implies that we actually need to consider a sequence of assignment sets that are consistent with the current goals in the active goal set, which is determined by the `triggers` and `precedes` relations.

To formalize the notion of viability, we need to introduce the notion of a sequence of assignments of agents to roles to goals, Φ'. First, we note that we have already recursively defined the notion of a sequence of active goal sets, $\mathtt{G}_{\mathtt{Active}}' = [\mathtt{G}_{\mathtt{Active0}}, \mathtt{G}_{\mathtt{Active1}}, \ldots \mathtt{G}_{\mathtt{Activen}}]$. Thus, Φ' will be defined based on the sequence of active goal sets.

$$\Phi' = [\Phi_0, \Phi_1, \ldots \Phi_n]$$

Where Φ_i is the set of assignments corresponding to the active goal set $\mathtt{G}_{\mathtt{Activei}}$.

$$\Phi_i \subseteq \{<a,r,g> \mid g \in \mathtt{G}_{\mathtt{Activei}} \wedge r \in R \wedge a \in A \wedge potential(a,r,g) > 0\}$$

Finally, we can define viability as a predicate that determines satisfiability of a given organization, where the *viable* predicate is defined as

$$viable(O) = \exists\, \Phi' : \Phi \mid satisfiable(g_o, \Phi') \wedge legal(\Phi', P) \wedge ordered(\Phi', G) \qquad (10)$$

Each element of viability, the satisfiability of a goal, the legality of an assignment set with organizational policy, and the orderliness of the goal tree are discussed individually below.

4.1.1 Satisfiable

Essentially, if a goal is a leaf goal, then it is satisfiable if we can find a role that can achieve that goal and an agent that can play that role. This assignment must be part of some assignment set in the assignment sequence. If a goal is not a leaf goal, then its satisfiability is based on the satisfiability of its sub-goals. If the goal is conjunctive, then all its leaf goals must be satisfiable; if the goal is disjunctive, then at least one of its leaf goals must be satisfiable. Formally, we define *satisfiable* as

$$\begin{aligned}&\text{satisfiable}(g, \Phi') = \\ &\quad \text{children}(g) = \{\} \Rightarrow (\exists\, \Phi_i{:}\Phi' \mid \langle a,r,g\rangle \in \Phi_i \wedge \text{achieves}(r,g) \wedge \text{capable}(a,r) \geq 1) \wedge \\ &\quad \text{children}(g) \mathrel{!}= \{\} \wedge g.\text{conjunctive} \Rightarrow (\forall\, g'{:}\text{children}(g)\ \ \text{satisfiable}(g', \Phi')) \wedge \\ &\quad \text{children}(g) \mathrel{!}= \{\} \wedge g.\text{disjunctive} \Rightarrow (\exists\, g'{:}\text{children}(g)\ \ \text{satisfiable}(g', \Phi'))\end{aligned}$$

4.1.2 Legal

We define the *legality* of a sequence of legal assignments, $\Phi' = [\Phi_1, \Phi_2, \ldots \Phi_n]$, in terms of the legality of the individual assignments as described in Section 2.7.1. Essentially, a sequence of assignments if legal if each of assignment set in the sequence is legal.

$$\text{legal}(\Phi',P) = \text{legal}(\Phi_1,P) \wedge \text{legal}(\Phi_2,P) \ldots \text{legal}(\Phi_n,P)$$

4.1.3 Ordered

The requirement for the correct ordering of goals in the active goal set sequence is handled by the ordered predicate. This predicate must ensure that goal precedence is appropriate and that maintenance goals hold at the appropriate places.

$$\text{ordered}(\Phi', G) = \text{ordered}^{PT}(\Phi', G) \wedge \text{ordered}^{M}(\Phi', G)$$

To be correctly ordered in terms of the `precedes` relations, an instance of a goal may only appear in the active goal set sequence if there is an instance of all preceding goals in previous active goal sets. To be correctly ordered in terms of the `triggers` relations, an instance of a goal may only appear in the active goal set sequence if there is at least one instance of a triggering goal that appears in the immediately preceding active goal set sequence. In the case of a triggered goal, the triggering goal must appear in the goal set immediately preceding it.

$$\begin{aligned}\text{ordered}^{PT}(\Phi', G) = \forall\, g,g'{:}G,\, e{:}E\ (&g \in G_{\text{Active}j} \wedge g \in \text{precedes}(g') \Rightarrow g' \in \texttt{G}_{\texttt{Active}i} \wedge i < j) \\ &\wedge\ (g \in G_{\text{Active}j} \wedge g \in \text{triggers}(g',e) \Rightarrow g' \in \texttt{G}_{\texttt{Active}i} \wedge j\text{-}i = 1)\end{aligned}$$

If a goal has a maintenance sub-goal, then its achievement can only occur when all achievement sub-goals have been achieved and its maintenance sub-goals are maintained. Thus, maintenance goals must be active whenever their sibling goals are active.

$$\begin{aligned}&\text{ordered}^{M}(\Phi', G) = \forall\, g,g1,g2{:}G \\ &\quad g1 \in \text{children}(g) \wedge g2 \in \text{children}(g) \wedge \text{maintenance}(g1) \wedge g2 \in \texttt{G}_{\texttt{Active}i} \Rightarrow g1 \in \texttt{G}_{\texttt{Active}i}\end{aligned}$$

Chapter 5 Organization-Based Multiagent Systems Engineering

5.1 Original MaSE

MaSE was originally designed to develop general-purpose multiagent systems and has been used to design systems ranging from computer virus immune systems to cooperative robotics systems [14], [56]. Each phase is presented below.

Analysis Phase. The goal of the MaSE analysis phase is to define a set of roles that can be used to achieve the system level goals. This process is captured in three steps: capturing goals, applying use cases, and refining roles.

- Capturing Goals. The first step is to capture the system goals by extracting them from the requirements, which is done by Identifying Goals and Structuring Goals. The purpose of the Identifying Goals is to derive the overall system goal and its subgoals. This is done by extracting scenarios from the requirements and then identifying scenarios goals. After the goals have been identified, the second step, Structuring Goals, categorizes and structures the goals into a goal tree, which results in a Goal Hierarchy Diagram that represents goals and goal/subgoal relationships.
- Applying Use Cases. In this step, goals are translated into use cases, which capture the previously identified scenarios with a detailed description and set of sequence diagrams. These use cases represent desired system behaviors and event sequences.
- Refining Roles. Refining Roles organizes roles into a Role Model, which describes the roles in the system and the communications between them. Each role is decomposed into a set of tasks, which are designed to achieve the goals for which the role is responsible. These tasks are documented using finite state automata-base Concurrent Task Diagrams. Concurrent tasks consist of a set of states and transitions that represent internal agent reasoning and communications.

Design Phase. The purpose of the design phase is to take roles and tasks and to convert them into a form more amenable to implementation, namely agents and conversations. The MaSE design phase consists of four steps: designing agent classes, developing conversation, assembling agents and deploying the agents.

- Construction of Agent Classes. The first step in the design phase identifies agent classes and their conversations and then documents them in Agent Class Diagrams. The Agent Class Diagram that results from this step is similar to object-oriented class diagrams with two differences: (1) agent classes are defined by the roles instead of attributes and methods and (2) relations between agent classes are conversations.
- Constructing Conversations. Once the agent classes and the conversations are identified, the detailed conversation design is undertaken. Conversations model communications between two agent classes using a pair of finite state automata similar in form and function to concurrent tasks. Each task usually generates multiple conversations, as they require communication with more than one agent class.
- Assembling Agent Classes. Assembling Agent Classes involves defining the agents' internal architecture. MaSE does not assume any particular agent architecture and allows a wide variety of existing and new architectures to be used. The architecture is defined using components similar to those defined in UML.
- Deployment Design. The final design step is to choose the actual configuration of the system, which consists of the number and types of agents in the system and the platforms on which they should be

deployed. These decisions are documented in a Deployment Diagram, which is similar to a UML Deployment Diagram.

5.1.1 MaSE Weaknesses

While MaSE provides many advantages for building multiagent systems, it is not perfect. It is based on a strong top-down software engineering mindset, which makes it difficult to use in some application areas.

1. MaSE fails to provide a mechanism for modeling multiagent system interactions with the environment. While we examined this topic in [13], it has never been fully integrated into MaSE.
2. MaSE also tends to produce multiagent systems with a fixed organization. Agents developed in MaSE tend to play a limited number of roles and have a limited ability to change those roles, regardless of their individual capabilities. As discussed above, a multiagent team should be able to design its own organization at runtime. While MaSE already incorporates many of the required organizational concepts such as goals, roles and the relations between these entities, it cannot currently be used to define a true multiagent organization.
3. MaSE also does not allow the integration of sub-teams into a multiagent system. MaSE multiagent systems are assumed to have only a single layer to which all agents belong. Adding the notion of sub-teams would allow the decomposition of multiagent systems and provide for greater levels of abstraction.
4. The MaSE notion of conversations can also be somewhat bothersome, as it tends to decompose the protocols defined in the analysis phase into small, often extremely simple pieces. When the original protocol involves more than two agents, it often results in conversations with only a single message. This makes comprehending how the individual conversations fit together more difficult.

5.2 O-MaSE

To avoid designing static multiagent systems, we have extended MaSE to allow designers to design a multiagent organization, which provides a structure within which the multiagent system may adapt. This extended version of MaSE is called Organization-based MaSE (O-MaSE). A preliminary proposal for the O-MaSE methodology is described below. In general, many of the diagrams used in O-MaSE are variants of the UML class diagrams and use keywords to denote the difference between goals, roles, capabilities, agent classes, etc.

Throughout this section, an Information Flow Monitoring System (IFMS) is used as an example of an organization-based multiagent system. The overall goal of the IFMS is to keep track of the information producers and consumers along with the actual flow of information through a dynamically reconfigurable enterprise information system. The information producers and consumers use a publish/subscribe mechanism that allows consumers to find and subscribe to appropriate information producers. Therefore, the IFMS must keep track of the various information paths between the producers and consumers as well as monitor the actual data flowing along those paths. The IFMS provides data in the form of current paths and information flow statistics to enterprise system operators who monitor the system for problems.

5.2.1 Requirements

Requirements are translated into system level goals, which are documented in the form of an AND/OR goal tree. Figure 16 shows the goal tree for the IFMS described above. Given goal precedence relations, it is possible to design goal structures that cannot be achieved, thus we would like to provide the assurance that the top-level goal can be achieved. We have replaced the non-specific MaSE goal tree with a tree with specific AND/OR decompositions to match the organization metamodel.

The syntax has also changed in the O-MaSE goal model. We use standard UML class notation with the keyword «Goal». Each goal may be parameterized, with parameters annotated as attributes of the goal

class. When goals are instantiated, they are given specific values for these attributes. The aggregation notation is used to denote AND refined goals whereas the generalization notation is used to denote OR refined goals. This notation is somewhat intuitive as AND refined goals require a composition of its subgoals to be achieved. Subgoals of an OR refined parent goal can be thought of as alternative ways to achieve the parent goal, or that they can be substituted in place of the parent goal.

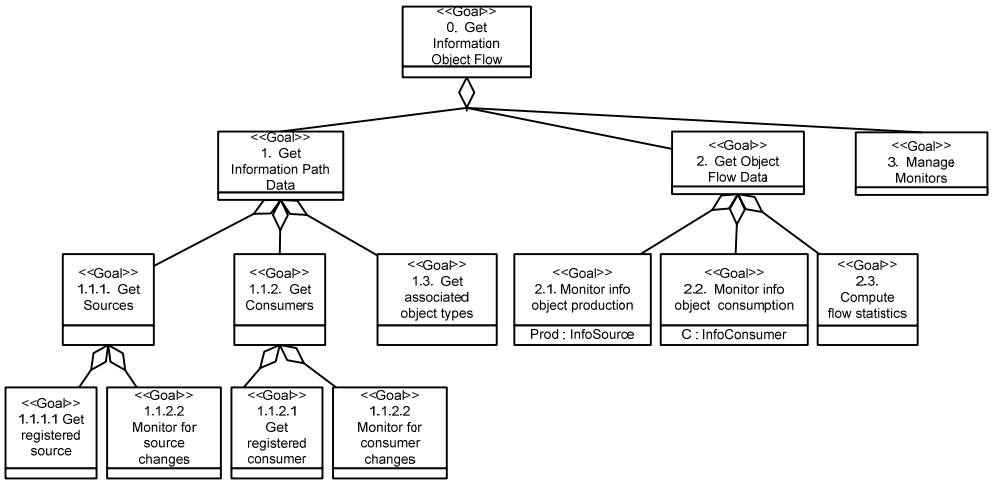


Figure 16. Goal Hierarchy Diagram

5.2.2 Analysis

Analysis begins by creating OMACS, which defines the organization's interactions with external actors. Generally, there is one organization at the top level (denoted by the «Organization» keyword) and that organization becomes responsible for the top goal in the goal tree. Each organization can achieve goals and provide services, which are further refined via activity diagrams (similar to UML activity diagrams, not included in this report). The designer can also use sequence diagrams for describing use cases at the system level, similar to the original version of MaSE. Each organization may also include sub-organizations to allow for abstraction during the design process.

While we allow the use of *services* in O-MaSE to help define the activities that agents carry out while performing roles, they do not map directly to the organization metamodel as presented earlier. For the purposes of this report, we only mention them for completeness, but do not elaborate on them, as their use in defining organizations is not required.

An example of an Organization Model is shown in Figure 17, where there are three actors making up the system's environment: the *ClientAPI*, the *ServerAPI*, and the *Admin*. The arrows connecting the organization to the actors denote protocols that define the agent class's interactions with the environment (these protocols are defined in detail in the high-level design stage). The relations between the organization and the goal and service classes (classes denoted by «Goal» and «Service» keywords) are fixed relation types. An organization provides services while achieving goals. These relations may be shown via explicit relations between organizations and goals; however, the relations may also be embedded in a class as shown in Figure 18 (where «achieves» relations are shown embedded within roles).

Next, the organization model is refined into a role model (Figure 18) that defines the roles in the organization, the services they provide, and the capabilities required to play them. Each role is designed to achieve specific goals from the Goal Model and provide specific activities refined from top-level services in the Organization Model. Again, the arrows between actors and roles and between two roles indicate protocols that are fully defined later in the design stage. The Role Model may also include

capabilities (denoted by the «Capability» keyword), which are attached to the roles that require them by the «requires» keyword.

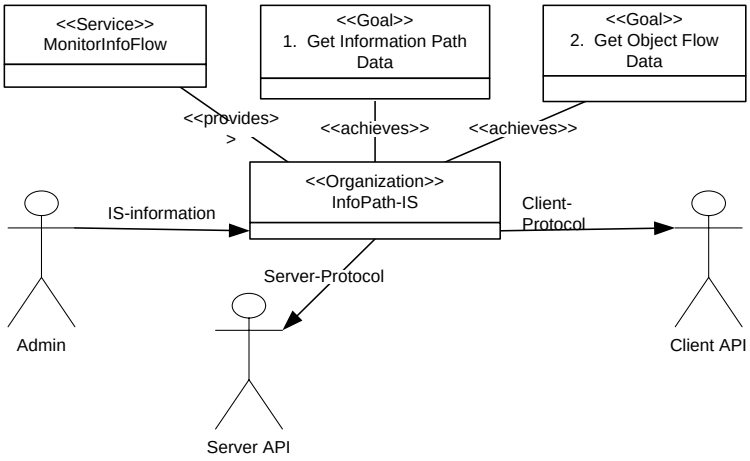


Figure 17. Organization Model

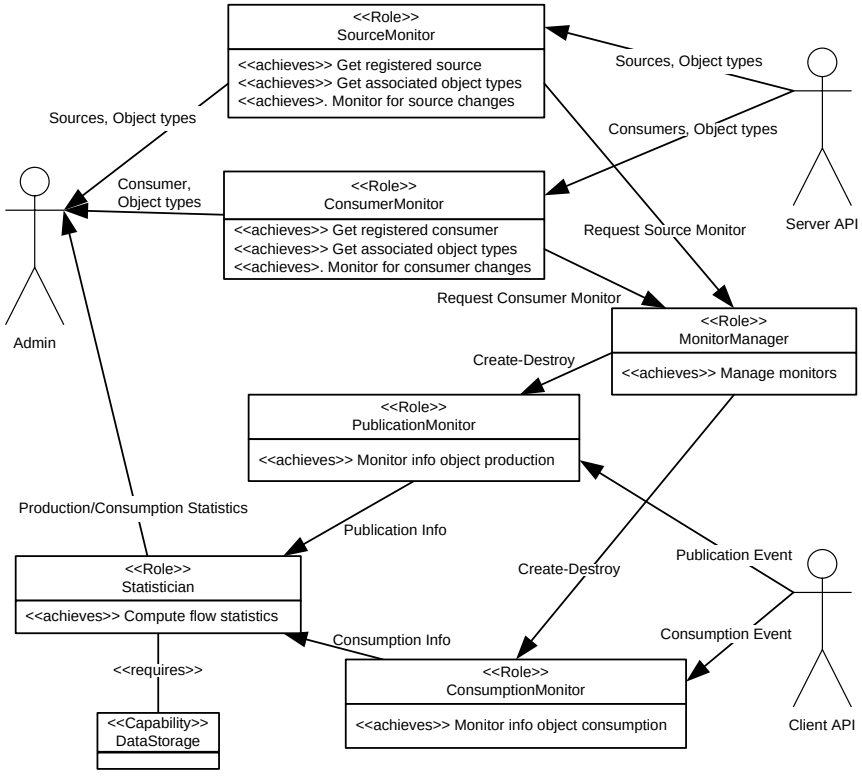


Figure 18. Role Model

At this point, O-MaSE differs from MaSE in that O-MaSE does not require the analyst to create concurrent task diagrams to describe the behavior of each role. This task is more appropriately carried out at the low-level design stage. The use of activities, which are refined via activity diagrams, allow the analyst to specify high-level behavior without resorting to low-level details required by concurrent task diagrams.

Throughout the analysis phase, the analyst should also capture and document the ontology that will be used within the system as part of the Domain Model. We have explored the integration of domain models into MaSE in [12]. The Domain Model allows the analyst to model domain entities, their attributes, and their relationships. Figure 19 shows a simple example of a domain model using standard UML notation to show the relationships between two types of Clients: Source and Consumer. Sources produce *InfoObjects* while Consumers consume *InfoObjects*.

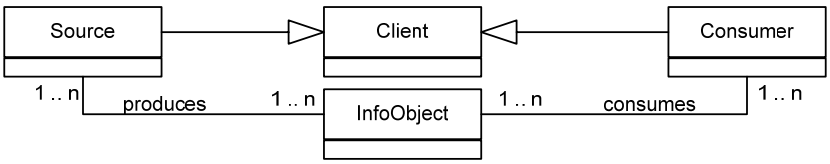


Figure 19. Domain Model

5.2.3 High-level design

The first step in the high-level design is for the designer to use the Role Model and service activity diagrams to define the Agent Class Model as shown in Figure 20. In the Agent Class Model, agents classes and lower-level organizations are defined by the roles played (which determines the goals they can achieve), capabilities possessed (which determines the roles they can play), or services provided. Figure 18 shows the use of both explicit and embedded relations. The «plays» and «provides» keywords in the agent classes (denoted by the «Agent» keyword) define which roles instances of the agent class can play as well as the services it can provide. The «possesses» relation between agent classes and capabilities (denoted by the «Capability» keyword) indicates the capabilities possessed by instances of that class of agent.

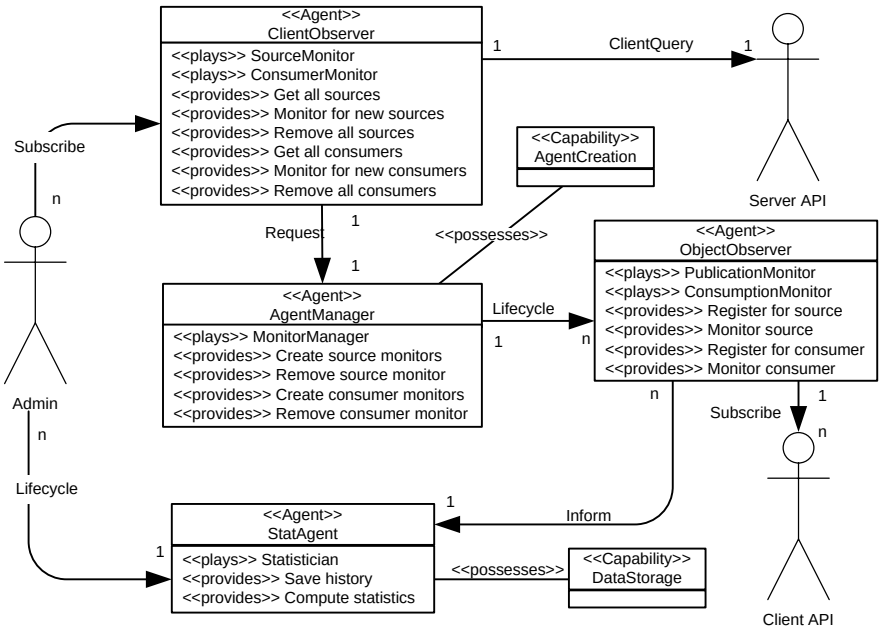


Figure 20. Agent Class Model

One the Agent Class Model is complete, Protocol Models (Figure 21) are used to define the message-passing protocols between agent classes. These Protocol Models follow the currently proposed AUML protocol diagrams [59], which allow the ability to show alternative and repetitive message structures.

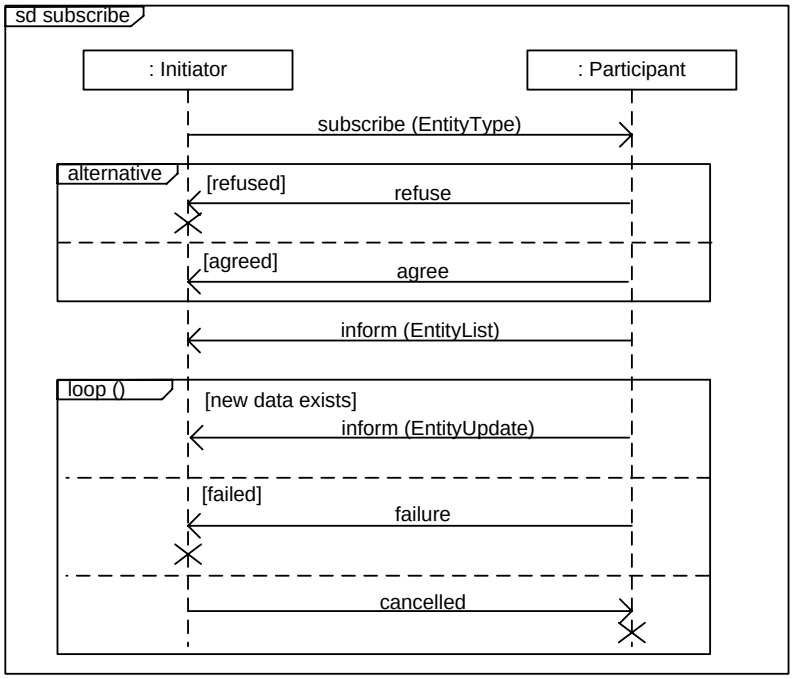


Figure 21. Protocol Model

Figure 21 captures a subscription protocol where the initiator wants to subscribe to information published periodically by the participant. After the initial subscribe message, the participant may either refuse or agree. If the participant refuses, the protocol terminates, which is denoted by the **X** symbol. Assuming the participant agrees, the participant sends an inform message with the current subscription information. The protocol then enters a loop where, typically, the participant sends an inform message with new information. However, the participant may send a failure message or the initiator a cancelled message, both of which end the protocol.

5.2.4 Low-level design

In low-level design, we define agent behavior using an Agent Plan Model, which is based on finite state automata (Figure 22). The Agent Plan Model is similar to the original MaSE concurrent task diagrams, as it captures internal behavior and message passing between multiple agents. They feature an explicit send and receive actions to denote sending and receiving messages. The remainder of the syntax and semantics is defined in [14]. The difference is that each Agent Plan Model is designed to achieve a specific goal by playing a given role and, thus, an agent may have multiple plan models. When an agent is assigned to play a particular role to achieve a goal, it uses the appropriate plan. If an agent is assigned multiple role/goals to play/achieve, it is up to each agent to decide whether plans are executed sequentially or concurrently.

5.3 Conclusions

In this section, we have discussed the current version of MaSE and some of its shortcomings. With the extension of MaSE to O-MaSE, we have dealt with each of these problems. Specifically, we have provided a mechanism for defining the multiagent systems interactions with the environment by adding external actors and defining the interactions protocols between the system and the actors.

Second, we have extended MaSE to capture the organizational concepts identified in our organization metamodel. New concepts include AND/OR refinement of goals, integration of capabilities and the ability to model sub-teams, or sub-organizations. This feature allows designers greater levels of abstractions and directly complements the notion of organizational agents in our organization metamodel.

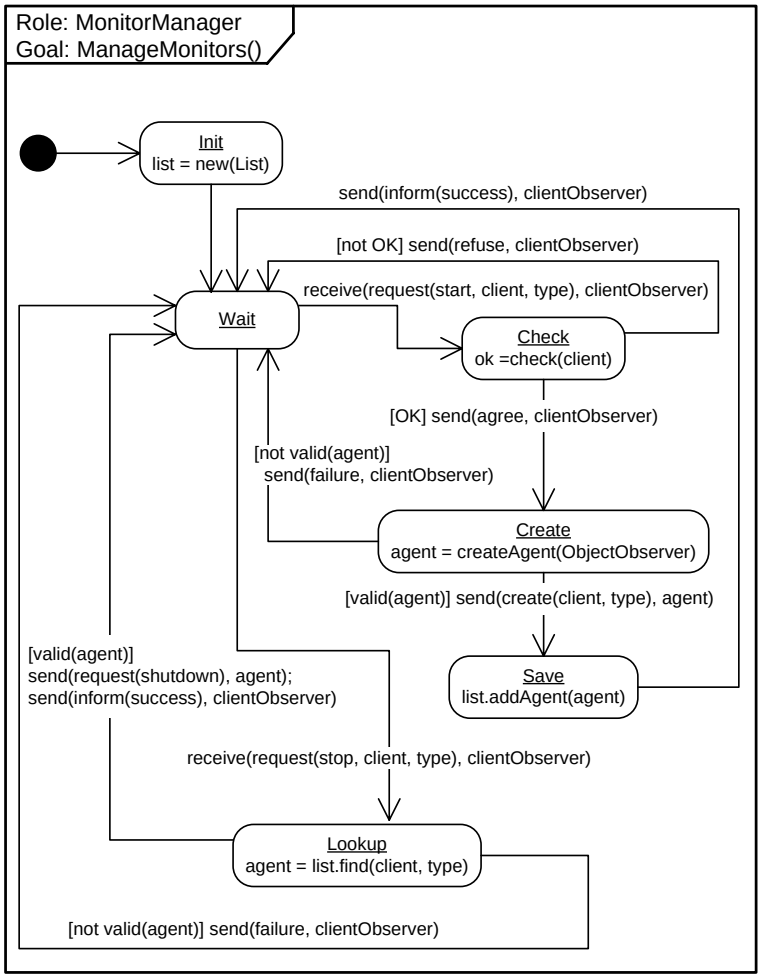


Figure 22. Agent Plan Model for AgentManager

Finally, we took the notion of concurrent tasks out of the analysis phase and integrated concurrent tasks with conversations into Agent State Models in the low-level design phase. We are currently using O-MaSE and our organization metamodel in several projects including an adaptive Battlefield Information System [56], cooperative robotic teams [60] and a system to monitor and control a large-scale information system.

We are continuing to evolve O-MaSE to provide a flexible methodology that can be used to develop both traditional and organization-based systems. A long-term goal is to provide a tailorable methodology that is fully supported by automated tools. We are currently building a new version of agentTool (aT^3) within the Eclipse IDE to support O-MaSE[2]. Plans include code generation for various platforms as well as integration with the Bogor model-checking tool [64] to provide model validation and performance prediction metrics.

[2] The current status of O-MaSE and the aT^3 project can be found at the Multiagent and Cooperative Robotics Laboratory web site (http://macr.cis.ksu.edu/).

Chapter 6 Battlefield Information System Demonstration

To demonstrate the effectiveness of OMACS and GMoDS, we implemented a simulation Battlefield Information System (BIS). The goal of the BIS is to provide an information system that can adjust its processing algorithms and/or information sources to provide required information at various levels of efficiency and effectiveness [1]. In this system, various types of sensors at different locations are used to detect enemy vehicles. These sensors are subject to failure and erroneous outputs and typically have a delay in getting the information categorized. When sensor data of interest are available, they are fused in order to answer queries from the commander. Queries are sent to the system using a user interface and they are of two types: transient and persistent. Transient queries are executed only once whereas persistent queries are carried out repeatedly until canceled. To be able to overcome the loss of sensors and continue to provide the required information, the Battlefield Information System needs to adapt by replacing the failed sensors and adapting the information processing adequately. We use OMACS and GMoDS, which provide the knowledge required for information agents to reorganize and adapt to the changes in the environment.

6.1 Organization Design

We define the goals, roles, capabilities and agents in order to create an organization meeting the requirements stated above.

6.1.1 Goal Model

The main goal of the application is to answer each query presented to the system. From the requirements, we derived a GMoDS goal specification tree as shown in Figure 23. Non-leaf goals are decomposed using AND-refinement and OR-refinement. All the goals are conjunctive except goals 1.2.1.2.1 and 1.2.1.2.2 that are disjunctive. Following are the leaf goals that the organization must attempt to satisfy.

Goal 1.1 – Process Query: Get the query from the user.

Goal 1.2.1.1.1 – Find Sensors<Q: Query>: Find all the sensors in the area of interest for the Query Q given in parameter.

Goal 1.2.1.1.2 – Read Sensor<S: Sensor>: Read the data from the sensor S given in parameter.

Goal 1.2.1.2.1 – Merge Diverse<Q: Query, L:<Sensors>>: Fuse the data received from the list of sensors L for the area specified by query Q. Sensors in L must be of different types.

Goal 1.2.1.2.2 – Merge Similar<Q: Query, L:<Sensors>>: Fuse the data received from the list of similar sensors L for the area specified by query Q. Sensors in L must all be the same type.

Goal 1.2.2.1 – Filter Information<Q: Query>: Filter the merged data based on the information required by the query Q.

Goal 1.2.2.3 – Correlate Data<Q: Query>: Compare data with historical data in order to extract persistence information if the query Q is a persistent query.

Goal 1.2.2.2 – Add Information<Q: Query>: Look up additional information if required by the query Q.

Goal 1.2.3 – Return Result<Q: Query>: Display the result of the query Q in a user-friendly format.

Goal 1.2.4.1– Monitor Time Constraints<Q: Query>: Check the validity of the data regarding the time constraint specified by query Q.

Goal 1.2.4.2 – Monitor Accuracy Constraints<Q: Query>: Check the validity of the data regarding the accuracy constraint specified by query Q.

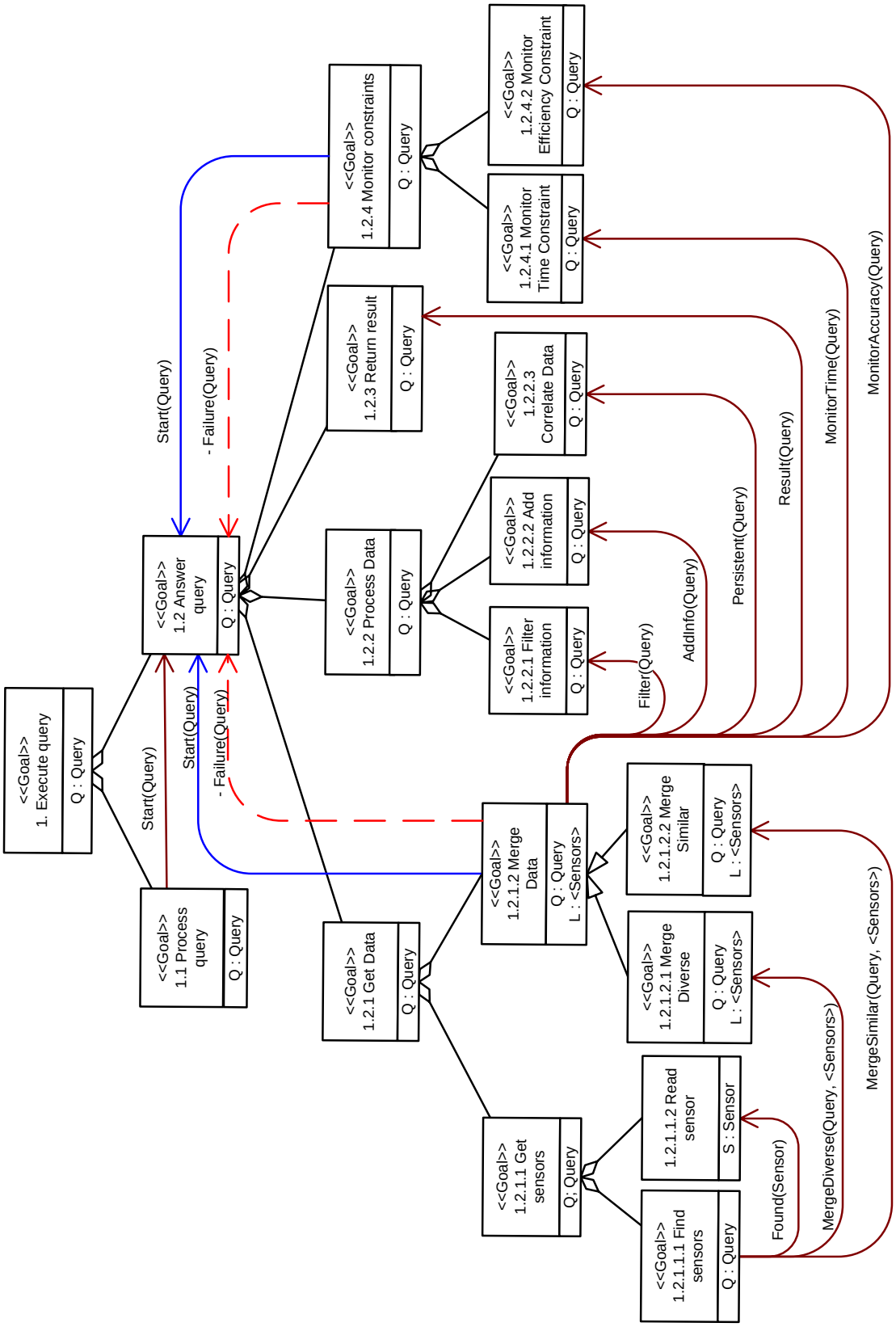


Figure 23. BIS Goal Model

6.1.2 Role Model

The roles for the BIS have been derived from the goal tree. In order to have a valid organization, each leaf goal should have at least one role that can achieve it. Therefore, for each goal in this organization, we have created a role that can achieve it. A role achieving a parameterized goal will be able to achieve all different parameterized instances of this goal. Following are the roles we have defined for the BIS organization, along with the goals they can achieve. We also describe the behavior of each of those roles.

R1 – Query Processor (G1.1): Periodically queries the GUI to get any new queries entered by the user. This role triggers an event to notify the system that a new query has been entered. This role achieves a goal that does not depend on any queries. It never terminates except for reorganization purposes.

R2 – Sensors Locator (G1.2.1.1.1): Query the sensor database in order to find all sensors available in the area specified by the parameter of the goal it achieves. Then it executes an algorithm to find the best coverage based on the set of available sensors. For each sensor selected, an event is triggered (event '*Found(S: Sensor)*'). This event will result in the organization attempting to find an agent capable of reading the selected sensor. After all sensors have been selected, the role triggers an event to notify the organization that it has found all sensors capable of providing data for the query and terminates. The event – which will be *mergeSimilar* or *mergeDiverse* depending on the sensors selected – will result in the instantiation of a new data merger goal to merge the results coming from the sensors selected.

R3 – Sensor Reader (G1.2.1.1.2): Read the data from the sensor given in parameter of the Read Sensor goal. This role interacts with the battlefield simulator in order to get the appropriate data. The data will be sent to any agents (typically a data merger agent) requesting those data.

R4 – Data Merger Diverse (G1.2.1.2.1): Merge the data collected from various sensors covering the area of interest. This role uses a processing algorithm that allows it to merge data coming from sensors of different type. Once the data are fused, it can decide to trigger any of the sub goals of the Process Data goal. Depending on the information required by the query, the agent playing this role will collaborate with some other agents to process the data fused. This coordination will allow the agent to formulate adequate answers to the query. Once all the processing is done, it triggers an event with the result and terminates.

R5 – Data Merger Similar (G1.2.2): Behave like the previous role. However, this role uses a processing algorithm that allows it to merge data coming from sensors of the same type efficiently. Thus, this role will not be able to process data from different sources.

R6 – Object Filter (G1.2.2.1): Filter the data based on the type of information required by the query given as parameter for the Filter Information goal. When the filtering is done, the agent playing this role will return the processed query to the Data Merger agent in charge of that query and terminate.

R7 – Intelligence Provider (G1.2.2.2): Look up additional information about the enemies in the area selected by the query given in parameter of the goal it is achieving (Add Information goal). When the processing is done, the agent playing this role will return the processed query to the data merger agent in charge of that query and terminate.

R8 – Persistence Validator (G1.2.2.3): Compare data with historical data in order to extract persistence information. This would be the case if, for example, we are monitoring the entrance of new vehicles in a given area. This role only applies to persistent queries. When the correlation is done, the agent playing this role will return the results to the Data Merger agent in charge of that query and terminate.

R9 – Result Interface (G1.2.3): Return the results of the query to the GUI that will display them.

R10 – Time Monitor (G1.2.4.1): Check the validity of the data regarding the time constraint if specified by the user. It communicates the results to the Data Merger agent in charge of the query and triggers a *failure* event if the constraint is violated.

R11 – Accuracy Monitor (G1.5.2): Behave like the previous role but regarding the accuracy constraint.

6.1.3 Capabilities

In order to have a valid organization, each role needs to require at least one capability. Some capabilities are used to interact with the environment while some others just allow the agent to carry out specific functional computations within the system. The capabilities identified for the BIS and the roles that require them are listed below.

C1 – User Interaction (R1, R9): Used to interact with the GUI. This capability provides actions to get a query from the user and to display the result of a query that has been executed.

C2 – Coverage Processing (R2): Used to compute the optimal set of sensors that has the maximum coverage of the area of interest and that can satisfy the efficiency and accuracy constraints. This capability has two actions: satisfyConstraints and findOptimalCoverage. The action satisfyConstraints takes a set of sensors and return those that can satisfy some given constraints. The action findOptimalCoverage takes a set of overlapping sensors and return a minimal set of sensors which has the maximum coverage of a given area.

C3 – Sensor Interaction <sensor> (R3): Used to interact with the actual sensors on the battlefield. This capability provides an action to query a sensor given in parameter and read its data.

C4 – Data Merging Diverse (R4): Provides computational algorithms to merge data coming from diverse type of sensors.

C5 – Data Merging Similar (R5): Provides fast computational algorithms to merge data coming from similar sources only.

C6 – Data Filtering (R6): Used to filter out some information that is not needed to answer the given query.

C7 – Intelligence Processing (R7): Used to search for additional information not provided by the sensors but needed to answer the given query. This capability will access a database and look up for the extra information needed.

C8 – Correlation Processing (R8): Provide the ability to correlate data from two successive results of a given query. Correlation exhibits differences in the results of a query obtained at different times.

C9 – DB access <name> (R2, R7, R8): Provide permission to access a database. The name of the database is given as a parameter. This capability also provides actions to read or write in the databases the BIS can interact with (e.g. sensors database, intelligence database).

C10 – Monitoring (R10, R11): Provide the ability to check the time and/or the accuracy of a query. The information about accuracy and times of the results are provided by the data sources.

C11 – Coordination (R3, R4, R5, R6, R7, R8, R10): Provide the ability to communicate with other agents. This capability provides actions to send/receive messages to/from specific agents in the organization. Agents can only communicate between them using this capability.

6.1.4 Agents

In order to have a viable organization, we define a specific types of agent capable of playing at least one role in the organization. To be able to play a specific role, an agent must possess the capabilities required for that particular role. The agent types with their capabilities are listed below.

- *QA* – Query Agent (C1)
- *SFA* – Sensor Finder Agent (C9, C2)
- *DSA* – Data Sensor Agent (C3, C11)
- *MAD* – Merger Agent Diverse (C1, C4, C5, C11)
- *MAS* – Merger Agent Similar(C1, C5, C11)

- *DFA* – Data Filter Agent (C6, C11)
- *IA* – Intelligence Agent (C9, C7, C11)
- *DCA* – Data Correlation Agent (C8, C9, C11)
- *MON* – Monitor Agent (C10, C11)

During the actual instantiation of the organization, an agent of each type will be created. For the Data Sensor Agent type, each sensor on the battlefield will be associated with a unique agent of type DSA. We have designed our system with all capabilities equally important. For this reason, the role capability function for an agent playing that role is 1 if that agent possesses all the capabilities required by the role, or 0 if it doesn't.

6.1.5 Organization State Model

Given the defined goals, roles, and capabilities, we can show in Figure 24 OMACS populated with an appropriate set of agents. The boxes at the top of the diagram represent the leaf goals, the circles represent the roles, the pentagons represent capabilities, and the ellipses are agents identified by their types. The arrows between the entities represent the achieves, requires, and possesses functions/relations. Each achieves and possesses arrow has a value of 1.

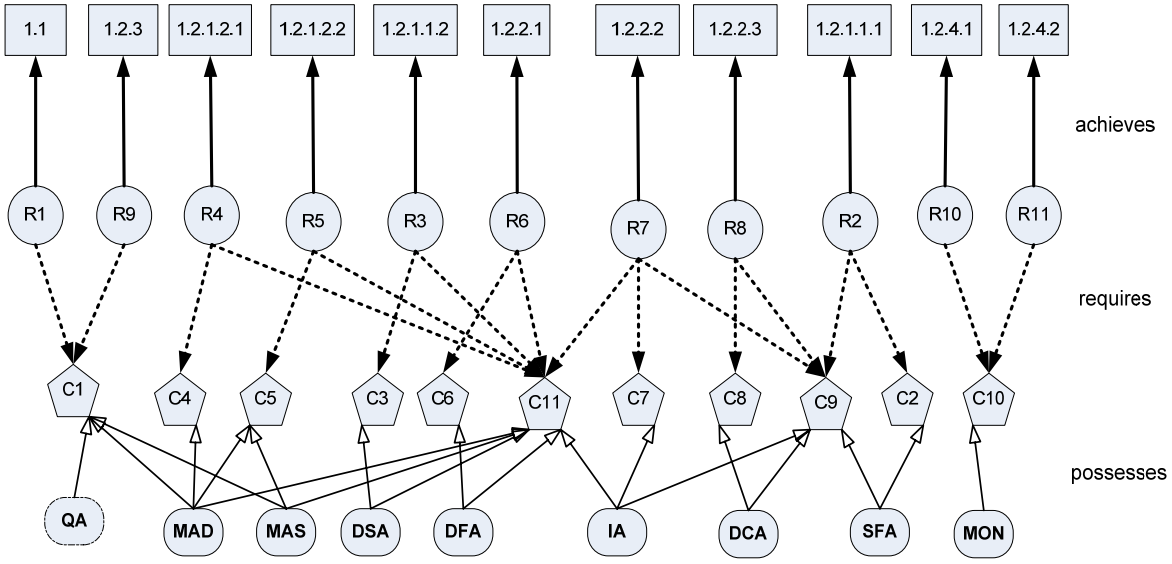


Figure 24. BIS Organization Model

6.1.6 Implementation Architecture

The BIS system has been designed using a centralized approach (Figure 25). This approach allows reusability of various organization reasoning algorithms by decoupling the organization reasoning part of the application, which can be generic, from the actual BIS system composed of application-specific agents. The system has the following entities:

- the Organization Master (OM),
- the Agent Reasoning (AR)
- the Agent Body (AB)

The Organization Master (OM) is a specialized agent that is in charge of all organization-related tasks. Currently, this agent is not part of the organization and cannot be assigned any role to play. The OM agent is the only agent that possesses the organization knowledge and that is able to execute reorganizations algorithms. This agent finds goals to be in the Active Goal set, creates appropriate

assignments (<A, R, G>) and sends them to the agents via their Agent Reasoning component. The OM also gets the agents status and reorganizes appropriately if needed.

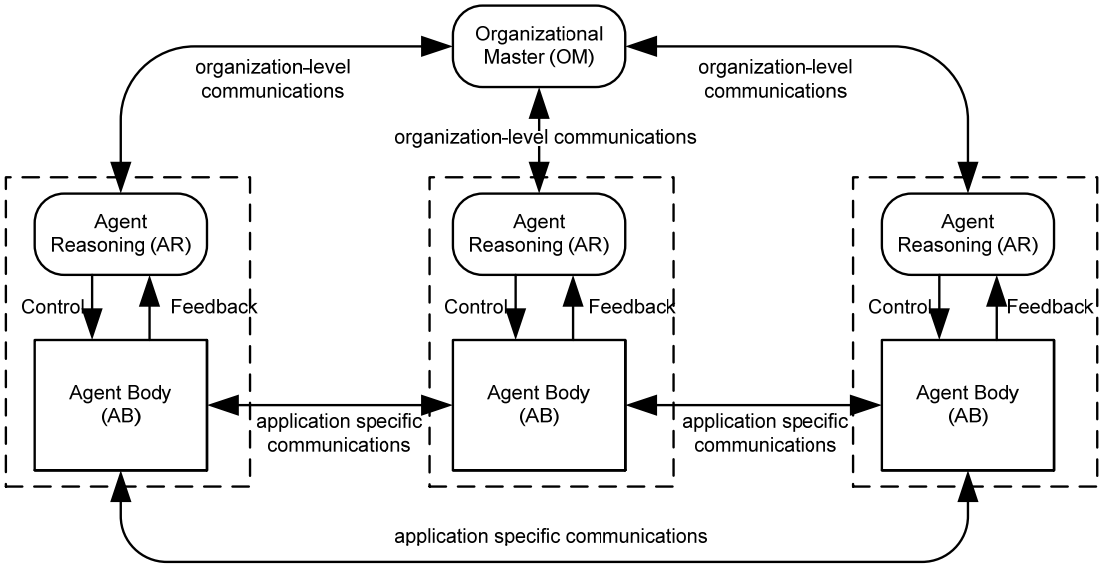


Figure 25. Centralized Architecture

Each agent is composed of two components: an Agent Reasoning (AR) component and an Agent Body (AB) component. In our centralized approach, the Agent Reasoning component of the agent serves as an interface between the OM and the Agent Body. It represents the part of the agent in charge of all organization related tasks. The AR receives assignments from OM and forwards them to the Agent Body. It also reports status/failures of its attached Agent Body to the OM.

The Agent Body is the actual agent as defined in OMACS. It accepts assignments from the AR, plays its assigned roles and reports its status to its reasoning component.

Communication between the OM and the agents is done by message passing via the AR. As the AR is actually part of the agent, the entities AR and AB can communicate directly via methods calls, thereby reducing the communication overheads produced by message passing. This architecture allows us to plug various organization reasoning algorithms into the systems while leaving the agents intact.

A distributed organization reasoning would involve a partial or total distribution of the organization knowledge among all or some of the actors of this system. Therefore, a consensus would have to be reached about how the organization should adapt. In both approaches (centralized or distributed), we can easily attach various organization reasoning modules to the actual application.

6.2 Reorganization Triggers

To be able to adapt to a variety of unpredictable situations, our BIS organization is able to detect changes in the performance of the overall organization and modify its structure accordingly. Those changes are happening while the system is running. Most of them are environment changes, but some changes can happen internally to the organization (a goal completion for example). Such changes will be reorganization triggers when they cause the organization to fail in achieving some goals or to be less effective or efficient. These trigger events will initiate the reorganization process. As a result, the reorganized instance will perform better than the original configuration. Currently, the organization may decide to reorganize for the followings reasons:

- Sensor Failure
- Goal Completion
- Maintenance Goal Failure

We now describe how the system reacts to those reorganization triggers.

6.2.1 Sensors Failure

Each Data Sensor Agent (DSA) is linked to one physical sensor from the battlefield. The failure of a sensor (S1) is taken as a goal failure. The corresponding DSA, which is the only agent capable of playing R3 to achieve G1.2.1.1.2(S1), cannot satisfy its goal anymore. When a sensor involved in a query fails, the Data Merger agent in charge of that query becomes aware of this goal failure and notifies the OM of its lost of performance due to the failure of one agent. Then, the system redistributes the sensor reading tasks among all the sensor reader agents still working and capable of covering the area of interest. In some cases, this reorganization process will require the reassignment of the Data Merger agents to ensure optimal performance. When the reorganization is achieved, the query is executed again and the results are sent to the user. In a rigid system, the loss of a sensor would mean an irreversible loss of performance in the system.

6.2.2 Goal Completion

The achievement of a goal can free up an agent and trigger the insertion of some new goals in the active goal set. In those cases, the organization needs to take proper decisions in order to optimize the performance of the system. The insertion of a new goal into the organization will require the team to take action to satisfy this new goal. If the organization is able to find a role that achieves this new goal, and an agent that can play this role, the selected agent will be assigned to the goal-role pair. In some cases, in order to find a valid assignment, the organization will have to reassign some agents already playing some roles. Similarly, an agent freed up after completion of its goal can be assigned to play new roles in the organization.

6.2.3 Maintenance goal failure

The user can specify time or accuracy constraints that the query needs to satisfy. To monitor the validity of those constraints, we have defined two maintenance goals: Monitor Time Constraint and Monitor Accuracy Constraint. The agents assigned to achieve those goals will monitor for conditions violating the constraints for a particular query. If a constraint violation is detected, the agent playing the monitoring role will notify the organization that will try to reorganize to meet the constraints. If the constraints cannot be satisfied, the user will be notified and the query will be executed without any constraints.

6.3 Example Scenario

To show an example of how the BIS operates, we define a scenario that will exemplify some of the adaptive behaviors explained above. We will describe the state of the organization each time a reorganization trigger is detected. We will exhibit the goals in the Active Goal Set (G_{Active}) and the Achieved Goal Set ($G_{Achieved}$), and all the assignments currently active in the system (Φ). We will assume that the system is only trying to answer one persistent query and omit the query parameter for goals and triggers. The BIS organization will answer the following persistent query: "*Show the location and type of all enemy vehicles in the selected area*"(the area selected is defined by a rectangle as shown in Figure 26). Once the query is entered, it is stored by an external agent (not part of the BIS organization) in charge of the GUI. The screenshot in Figure 26 represents the battlefield along with the sensors and enemy targets. We have defined four ground sensors (S1, S2, S3, S4), 1 ATR sensor (S5) and 5 enemy vehicles in our scenario.

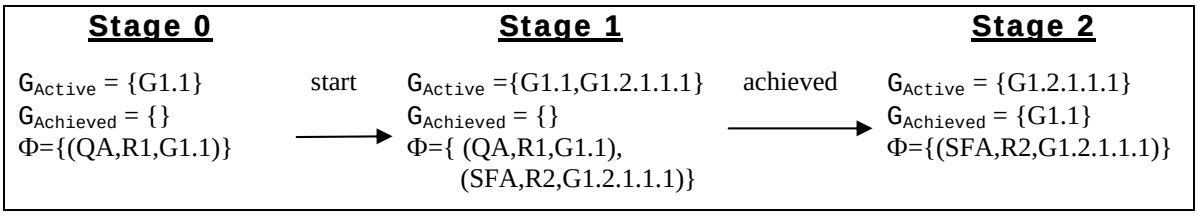
6.3.1 Normal Execution

At the initialization of the system, all the agents interested in participating in the organization will register with the Organization Master. All the goals that have no predecessors and do not require any explicit triggers are inserted in G_{Active}. Once those goals are in G_{Active}, roles and agents will be assigned to satisfy these goals. Given the goals, roles, capabilities and agents defined in section 3**,** G1.1 is the only goal that can be inserted in G_{Active}. When G1.1 is inserted into G_{Active}, the system will assign the Query Agent (QA) to play role R1 to achieve goal G1.1. Thus, we obtain the following state for the organization:

Stage 0

G_{Active} = {G1.1}
$G_{Achieved}$ = {}
Φ = {(QA, R1, G1.1)}

Once the QA retrieves the query from the GUI, it triggers the event *start(Query)*. This trigger results in the insertion of G1.2.1.1.1 in G_{Active}. Upon this insertion, the system must reorganize to satisfy this newly inserted goal. The Sensor Finder Agent (SFA) will be assigned to play role R2 to achieve goal G1.2.1.1.1. As we assumed that the system has only one query, the QA can terminate. Once the agent terminates, it sends an *achieved* message to the Organization Master. This message will result in the removal of the goal G1.1 from G_{Active} and its insertion into $G_{Achieved}$. The corresponding assignment is also removed from the list of current assignments. The corresponding organizational state transition is shown below.



In our scenario, the SFA chooses sensors S1, S2, S3 as optimal sensors for the current query. It then triggers the following events: *found(S1), found(S2), found(S3), mergeSimilar(<S1, S2, S3>)*. Each *found* event will trigger a parameterized goal G1.2.1.1.2 having the parameter of the trigger. In our case, goal G1.2.1.1.2(S1), G1.2.1.1.2(S2), and G1.2.1.1.2(S3) will be triggered. As the sensors given in parameter for the event *mergeSimilar* are all similar sensors (ground sensors), this event will result in the insertion of the parameterized goal G1.2.1.2.2(<S1, S2, S3>). To satisfy the new goal G1.2.1.2.2, the system will choose role R5 which will be played by the Merger Agent Similar (MAS). When all the events have been triggered, the SFA sends an *achieved* message to the Organization Master. This message will result in the removal of the goal G1.2.1.1.1 from G_{Active} and its insertion in $G_{Achieved}$. The corresponding assignment is also removed from the list of current assignments. We have the following state transitions.

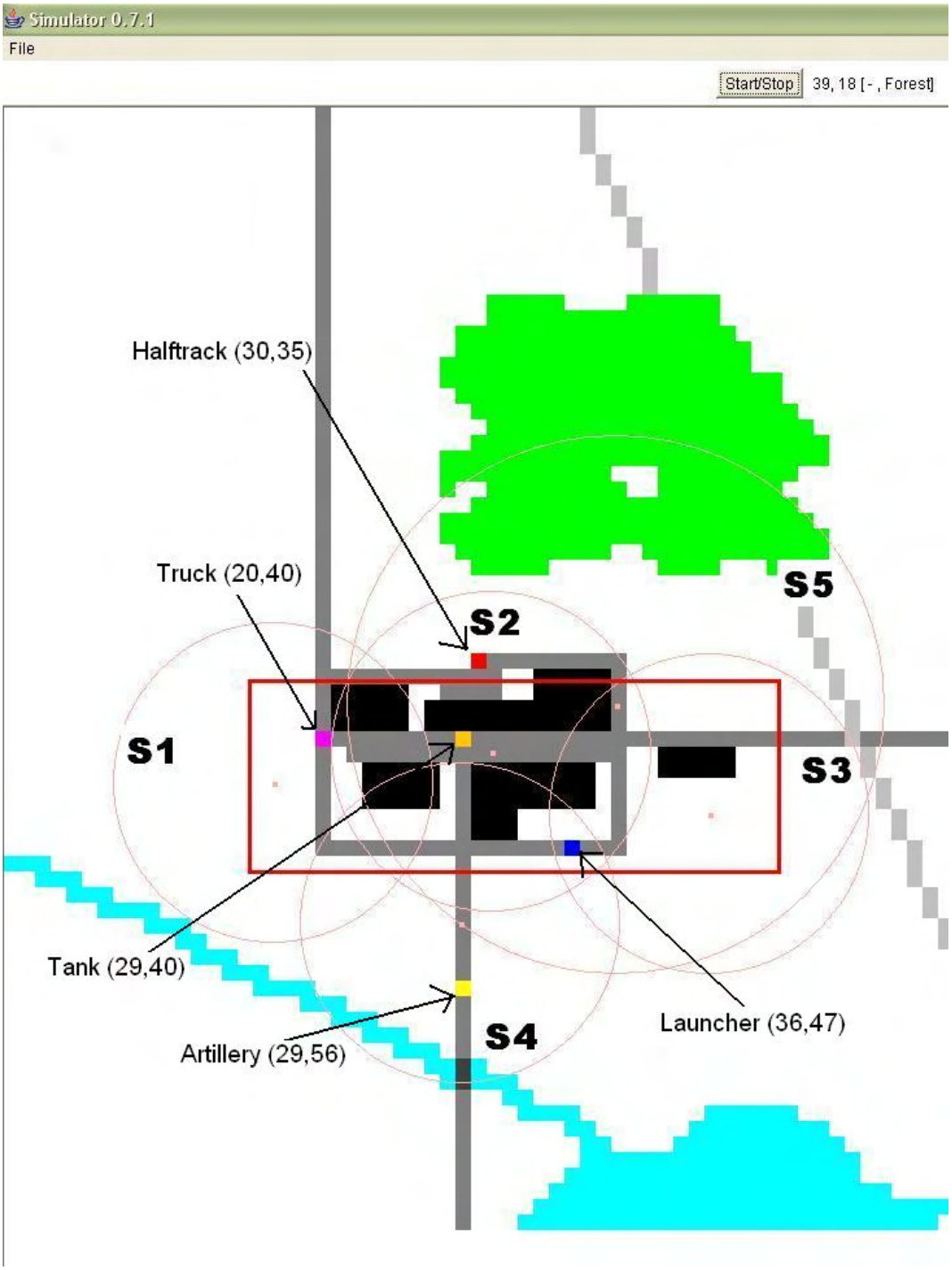
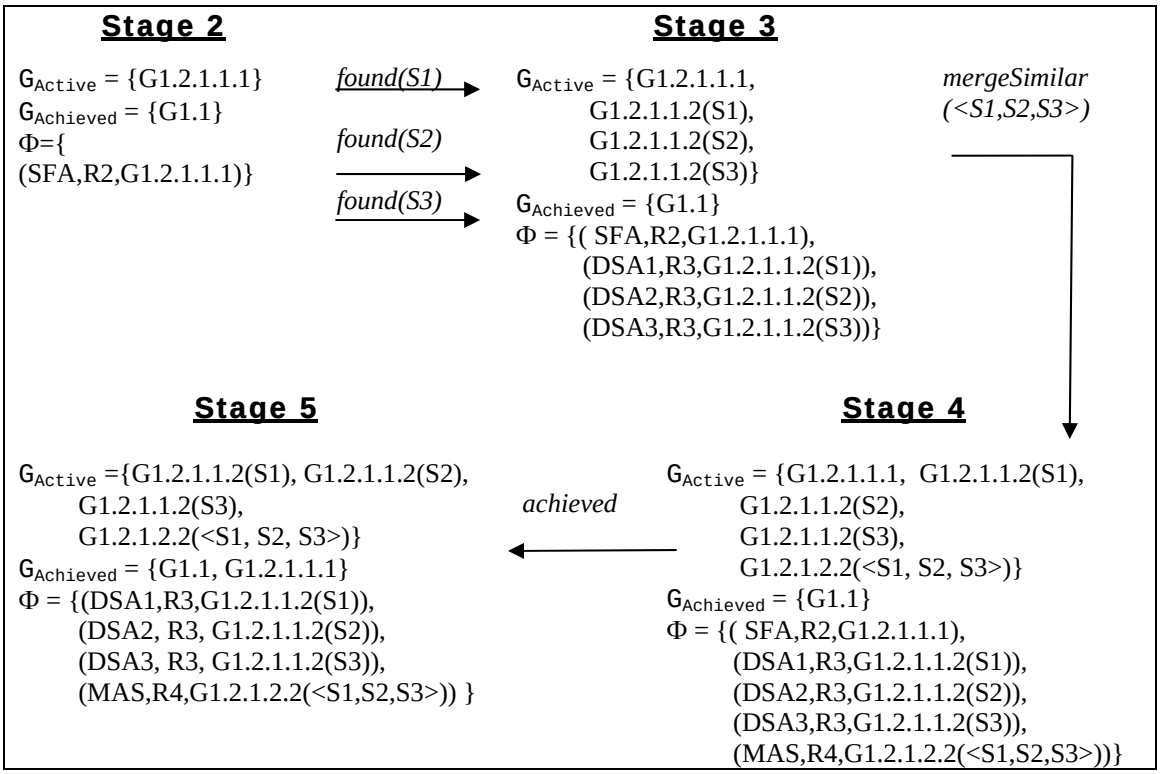
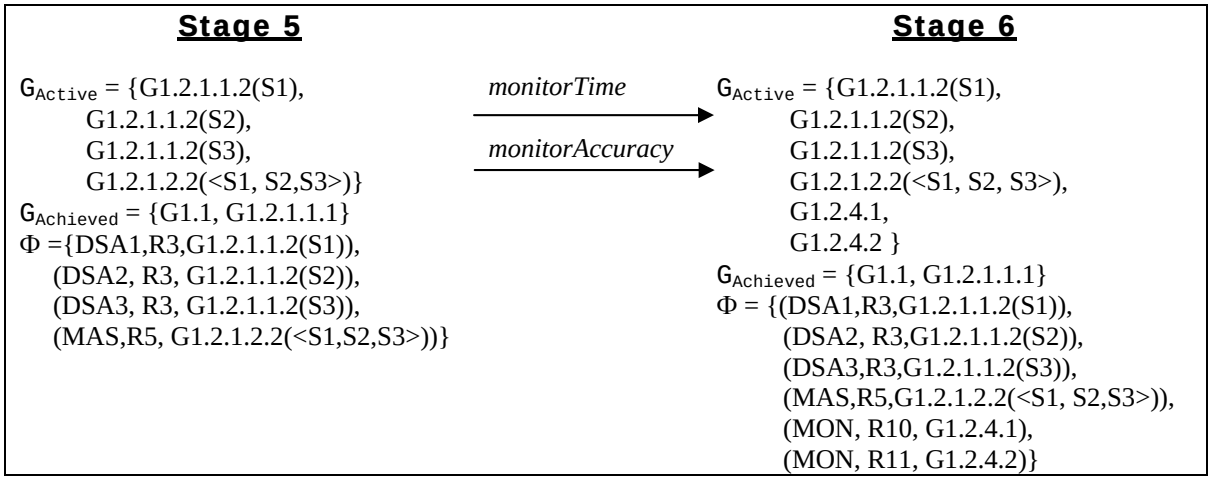


Figure 26. Battlefield Map

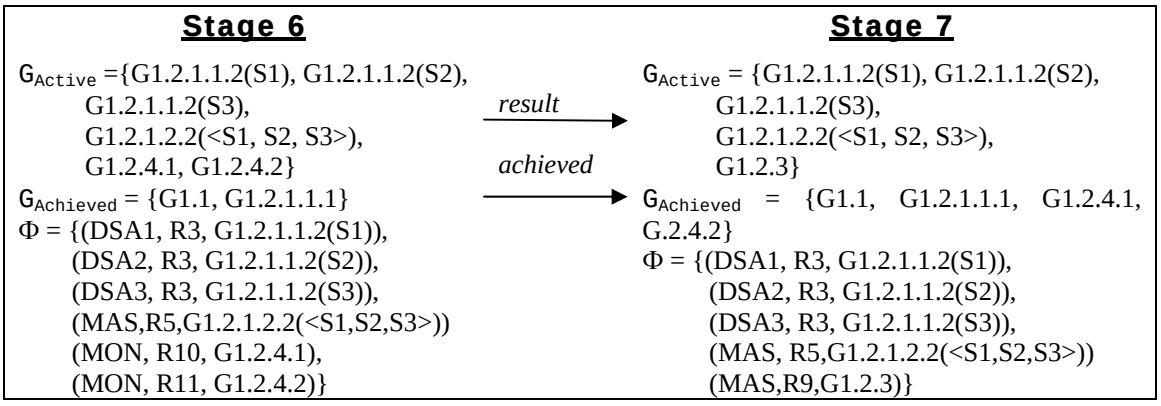


The fusing of data is done after coordination between the MAS and the DSAs in charge of that query. Note that no filtering, additional data or correlation is necessary for this query. In order to have the results of the query checked against the constraints, the MAS will trigger a *monitorTime* and *monitorAccuracy* event. Those events will result in the insertion of G1.2.4.1 and G1.2.4.2 in G_{Active}. Role R10 and R11 will be selected to achieve respectively G1.2.4.1 and G1.2.4.2. These roles will both be achieved by the Monitor Agent (MON) which has all the required capabilities to play both roles. Thus we obtain the following transitions.



If none of the constraints is violated, the MON will send a message to the MAS notifying it that it can proceed and will send an achieved message to the Organization Master. This message will result in the removal of the goal G1.2.1.1.1 from G_{Active}, its insertion in $G_{Achieved}$ and the removal of the assignment related to G1.2.1.1.1 from Φ. The MAS will then trigger a *result* event, which will result in the insertion

of G1.2.3 in G_{Active}. As the MAS has the capability to interact with the GUI, it will play role R9 to achieve goal G1.2.3 and send the result of the query to the GUI.



Once the results of the query have been sent, the MAS will terminate its assignment to play role R9 to achieve G1.2.3 by sending an *achieve* message to the Organization Master. This message will result in the removal of the goal G1.2.3 from G_{Active} and its insertion in $G_{Achieved}$. The corresponding assignment is also removed from the list of current assignments. The MAS will not terminate its assignment for R4 as the query is persistent and will need to be executed again.

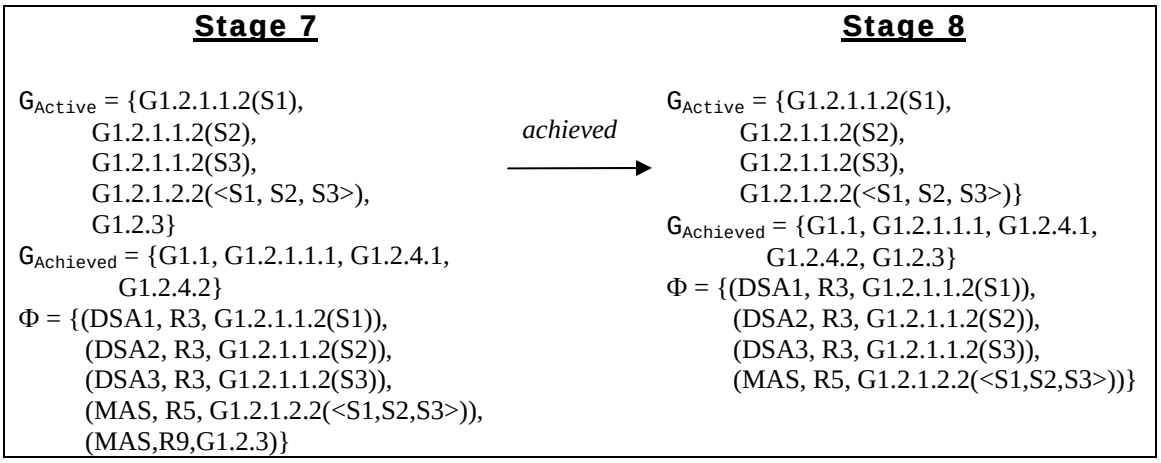


Figure 27 shows the results obtain from the GUI. The answer for the query covers 100% of the area of interest. The BIS detected the following three enemies.

- Tank at 29,40
- Truck at 20,40
- Launcher at 36,47

The system effectively detected all the targets in the selected area.

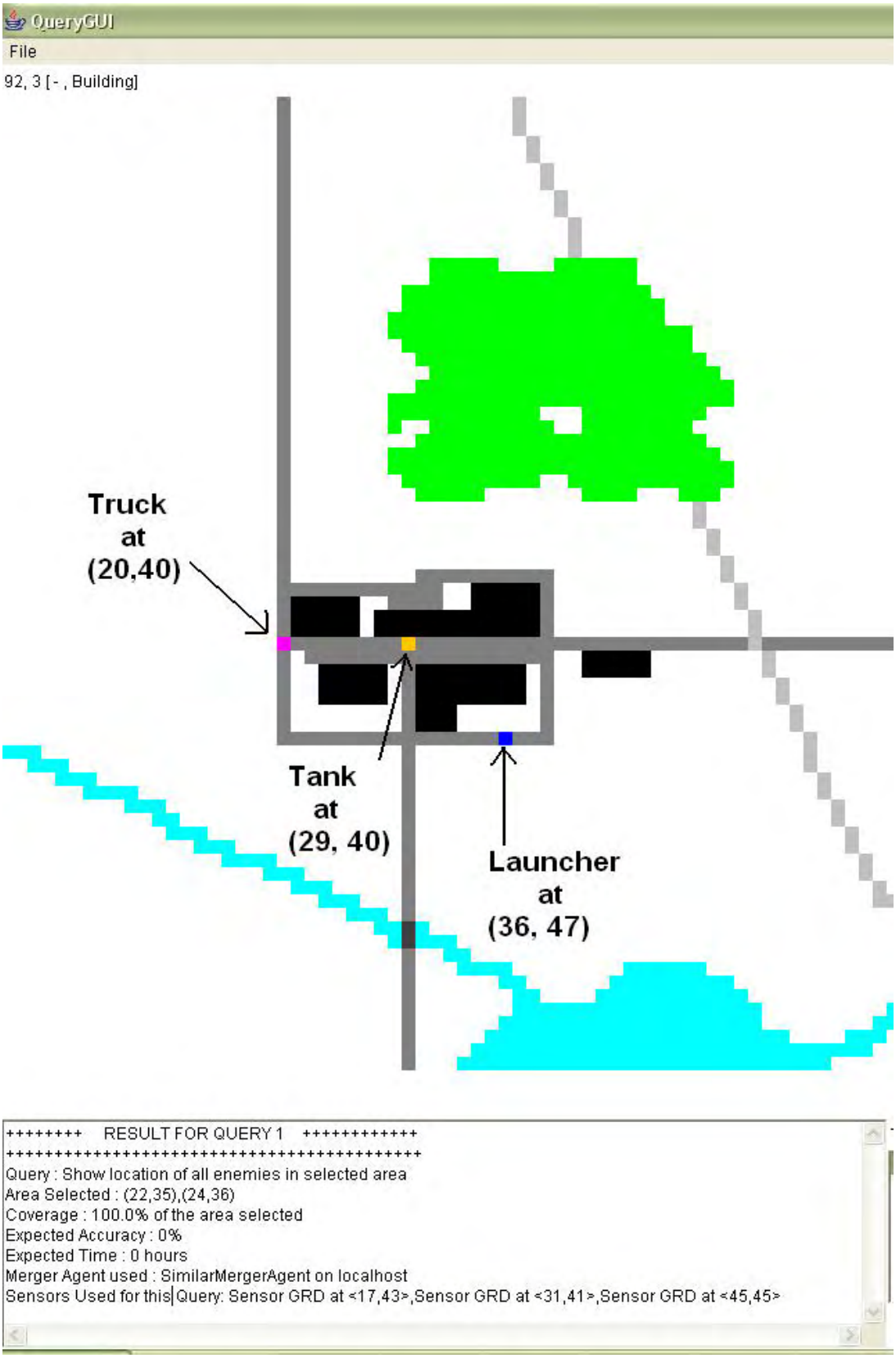
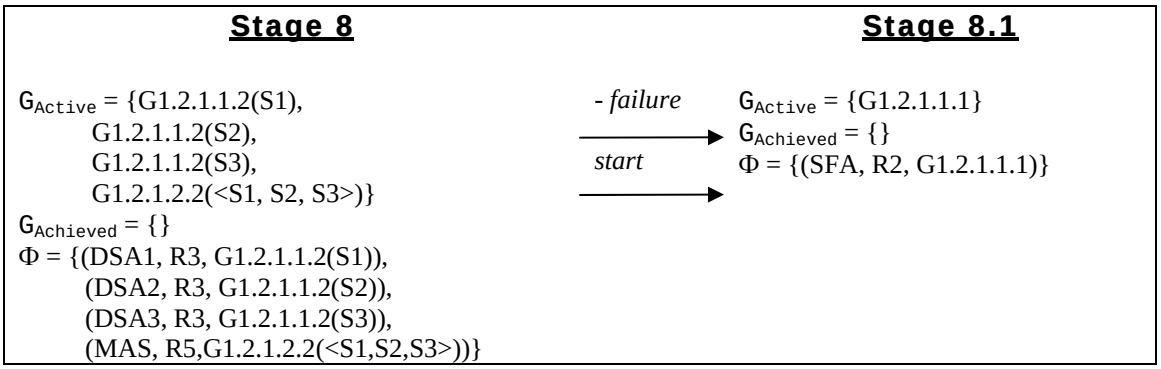


Figure 27. Results from the GUI for a Normal Execution

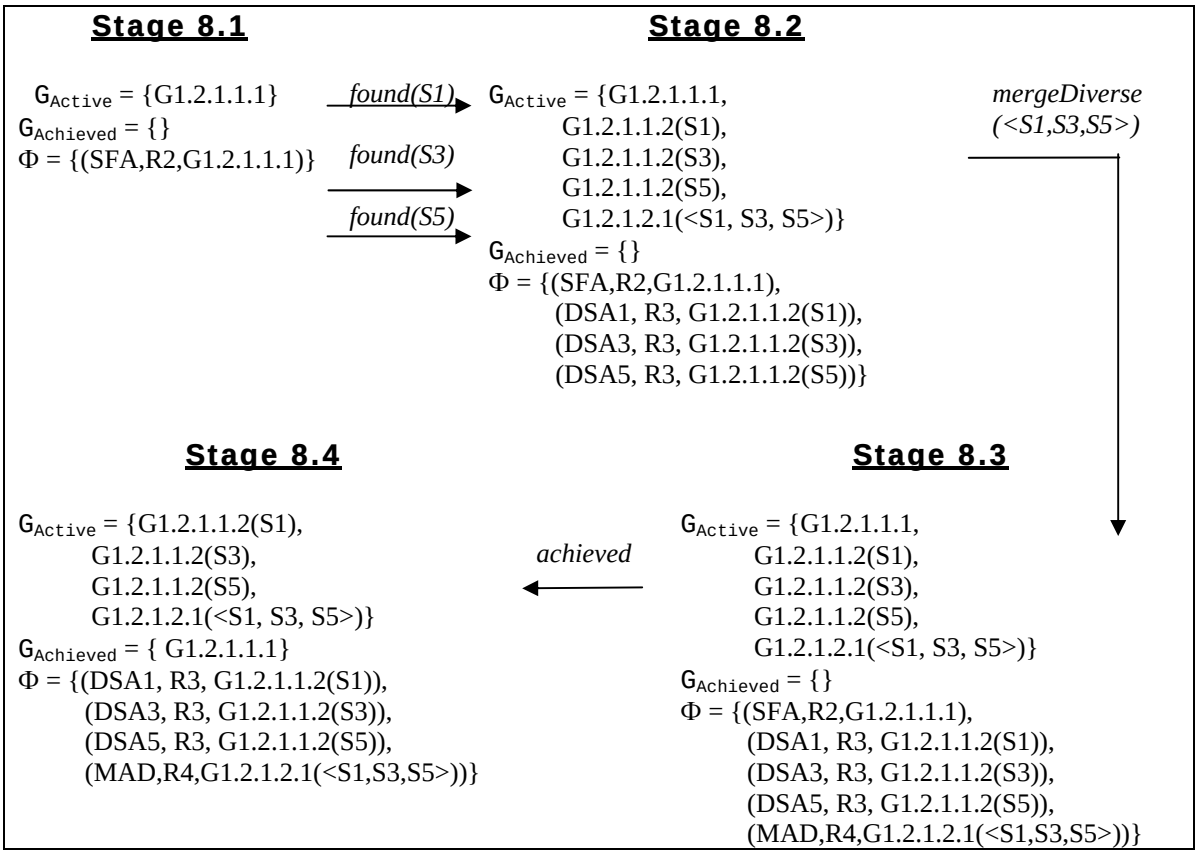
6.3.2 Sensor Failure

The BIS simulator allows us to fail specific sensors. If we make S2 fail, the attached DSA2 will be unable to achieve goal G.2.1.1.2(S2). As DSA2 can no longer gather the data, the MSA, which was coordinating with DSA2, will have to interrupt its task and will generate a negative trigger *failure*. This negative trigger will cause all the subgoals of G1.2 to be removed from G_{Active}, resulting in the cancellation of all their current assignments. Thus, goals G1.2.1.1.2(S1), G1.2.1.1.2(S2), G1.2.1.1.2(S3), G1.2.1.2.2(<S1, S2, S3>) will all be removed from G_{Active}. The negative trigger *failure* will immediately be followed by a *start* event generated by the DSA. The *start* event is parameterized with the initial query and causes the insertion of goal G1.2.1.1.1 in G_{Active}. Achieving this goal will help to reselect appropriate coordinating agents for the query. The organization is treating the query that it failed to answer due to a loss of sensor like a new query. The BIS will then choose appropriate agents to overcome this loss in order to provide the best results. The state transitions after a sensor failure are shown below. For simplicity, we assume that $G_{Achieved}$ is empty when the failure occurs.



Taking into account the loss of capability of the DSA for S2, the SFA will indicate sensors S1, S3, S5 as the new optimal set of sensor for the query. It will then trigger the following events: *found(S1), found(S3), found(S5), mergeDiverse(<S1, S3, S5>)*. Each *found* event will trigger a parameterized goal G1.2.1.1.2 having the parameter of the trigger. In our case, goal G1.2.1.1.2(S1), G1.2.1.1.2(S3), and G1.2.1.1.2(S5) will be triggered.

As the sensors given in parameter for the event *mergeDiverse* are different sensors (S1, S3 are ground sensors whereas S5 is an ATR sensor), this event will result in the insertion of the parameterized goal G1.2.1.2.1(<S1, S3, S5>).To satisfy the new goal G1.2.1.2.1, the system will choose role R4 which will be played by the Merger Agent Diverse(MAD). When all the events have been triggered, the SFA sends an *achieved* message to the Organization Master. This message will result in the removal of the goal G1.2.1.1.1 from G_{Active} and its insertion in $G_{Achieved}$. The corresponding assignment is also removed from the list of current assignments. The corresponding states of the organization are described below.



The execution then continues as described in the normal execution where stage 8.4 would be equivalent to Stage 5 as described in Section 6.3.1. The BIS detects the following enemies.

- Tank at 29,40
- Truck at 20,40
- Launcher at 36,47

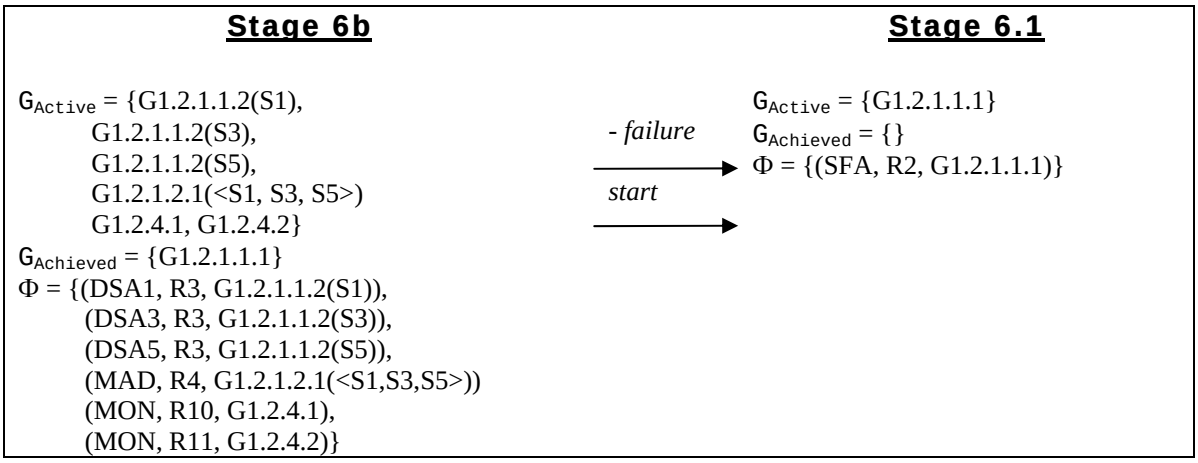
Therefore, in this reorganization, DS2 has been replaced by DS5 and the BIS organization decided to use the MAD for the merging instead of the MAS in order to insure a better performance. Even though a loss of a sensor used to provide information for the query has occurred, the system was able to reorganize and maintain the flow of information without the intervention of the user.

6.3.3 Maintenance Goal Failure

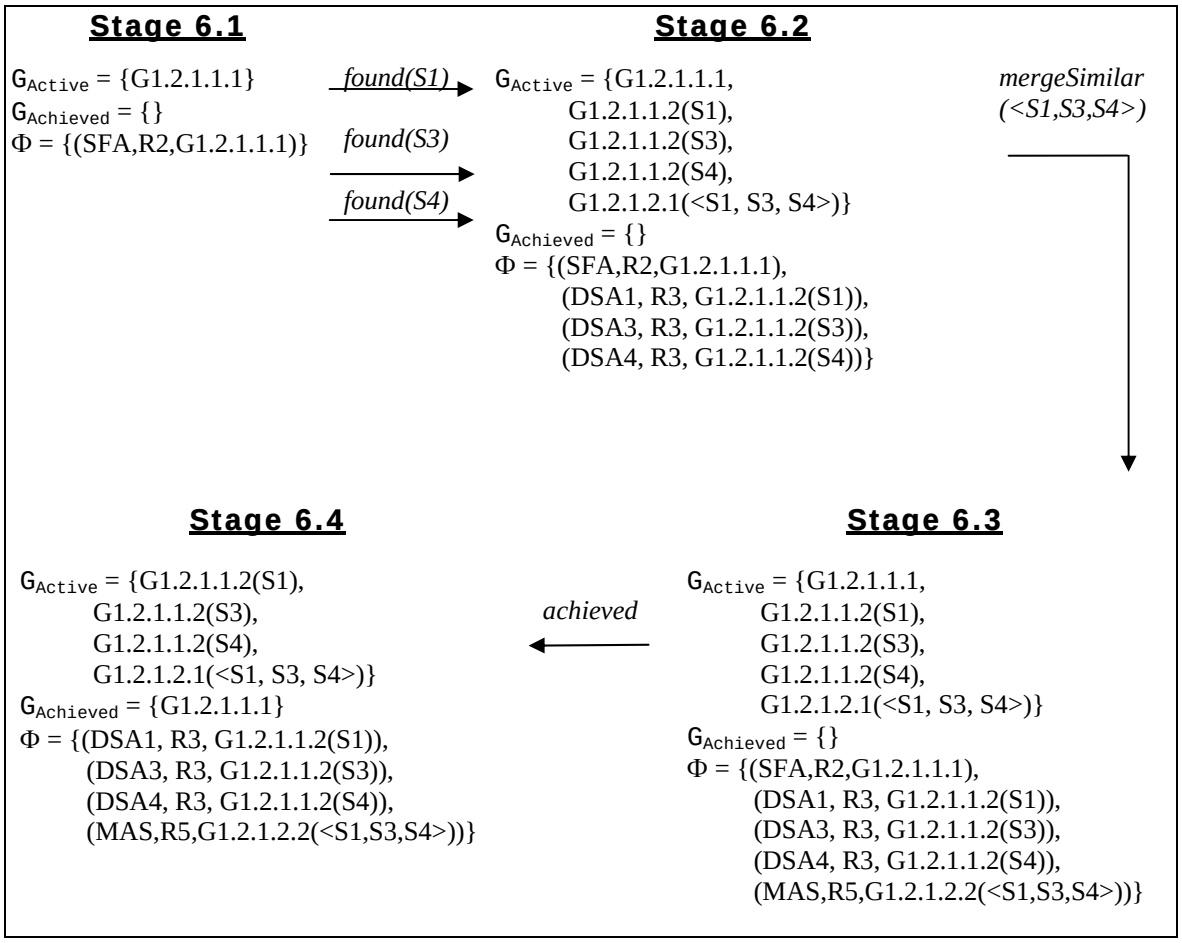
Sensors in the battlefield simulator do not all provide the same accuracy for data and do not refresh their data at the same rate. In the BIS simulator, the ground sensors provide an accuracy between 65% and 75% and will provide data within 5 hours. ATR sensors are very accurate (80%-99%) but they are not very fast at providing data (10-15 hours). Therefore, the simulator can provide erroneous or outdated data that might not be of any interest for the commander. In this case, the commander can specify some constraint for the query. By default, the query does not have any constraints. In our scenario, we will require the system to provide data for the query within 8 hours. The system is currently running the query using DSA1, DSA3, DSA5 and MAD and the time and accuracy monitor roles R10 and R11 have been assigned to MON. This phase correspond to Stage 6 in Section 6.3.1 with some differences in the sensors used to answer the queries. We will denote this state by ‘stage 6b’.

Once the data are sent to the Monitor Agent for checking the time constraint as described in the normal execution, the MON will generate a negative trigger *failure* because the query, as executed, do not meet

the 8 hours constraint. In fact, S5, which is an ATR sensor, can only provide data within 10 to 15 hours. Therefore, the maintenance goal G1.2.4.1 will fail. The negative trigger will cause all the subgoals of G1.2 to be removed from G_{Active}, resulting in the cancellation of all their current assignments. Thus, goals G1.2.1.1.2(S1), G1.2.1.1.2(S3), G1.2.1.1.2(S5), G1.2.1.2.1(<S1, S3, S5>), G1.2.4.1, and G1.2.4.2 will all be removed from G_{Active}. The negative trigger *failure* will immediately be followed by a *start* event generated by the monitor agent (MON). This event is parameterized with the initial query and causes the insertion of goal G1.2.1.1.1 in G_{Active}. The system will then treat the query that failed as a new query and try to choose the appropriate organization in order to provide result to the query meeting the time constraint defined above. The state transition after the constraint violation is described below. For simplicity, we will assume that $G_{Achieved}$ is empty when the violation occurs.



Taking into account the time constraint for the query, the SFA will select sensors S1, S3, S4 as the new optimal set of sensor for the query because they are all ground sensors capable of covering the area of interest and providing data within 5 hours. It will then trigger the following events: *found(S1)*, *found(S3)*, *found(S4)*, *mergeSimilar(<S1, S3, S4>)*. Each *found* event will trigger a parameterized goal G1.2.1.1.2 having the parameter of the trigger. In our case, goal G1.2.1.1.2(S1), G1.2.1.1.2(S3), and G1.2.1.1.2(S4) will be triggered. As the sensors given in parameter for the event *mergeSimilar* are all ground sensors, this event will result in the insertion of the parameterized goal G1.2.1.2.2(<S1, S3, S4>). To satisfy this new goal, the system will choose role R5 which will be played by the Merger Agent Similar(MAS). When all the events have been triggered, the SFA sends an *achieved* message to the Organization Master. This message will result in the removal of the goal G1.2.1.1.1 from G_{Active} and its insertion in $G_{Achieved}$. The corresponding assignment is also removed from the list of current assignments. The corresponding state of the organization is as follow.



The execution then continues as described in the normal execution where stage 6.4 would be equivalent to stage 5 (section 5.1). In this case, the BIS detects only the following enemies.

- Truck at 20,40
- Launcher at 36,47

Therefore, in this reorganization, DS5 has been replaced by DS4 to insure the effectiveness of the query with regards to the time constraint. This reorganization has also replaced the MAD by the MAS which yields a better performance in merging data coming from the new set of sensors. In this scenario, we can see how the BIS system has been able to reorganize in order to satisfy a maintenance goal in the system. However, this reorganization has resulted in a lost of coverage and the Tank located at 29,40 cannot be detected anymore. The set of sensor chosen to satisfy the time constraint covers only 87% of the area.

6.3.4 Execution Summary

Figure 28 through Figure 30 summarize the information sources changes the BIS made to overcome the sensor failure and to satisfy the time constraint imposed by the commander. We can see that the BIS was able to switch its information sources from the set <S1, S2, S3> in Figure 28 to <S1, S3, S5> in Figure 29 and finally to <S1, S3, S4> in Figure 30. The system was also able to change its fusing algorithms by assigning agents to play one of the two merging roles available in the organization (R4 and R5).

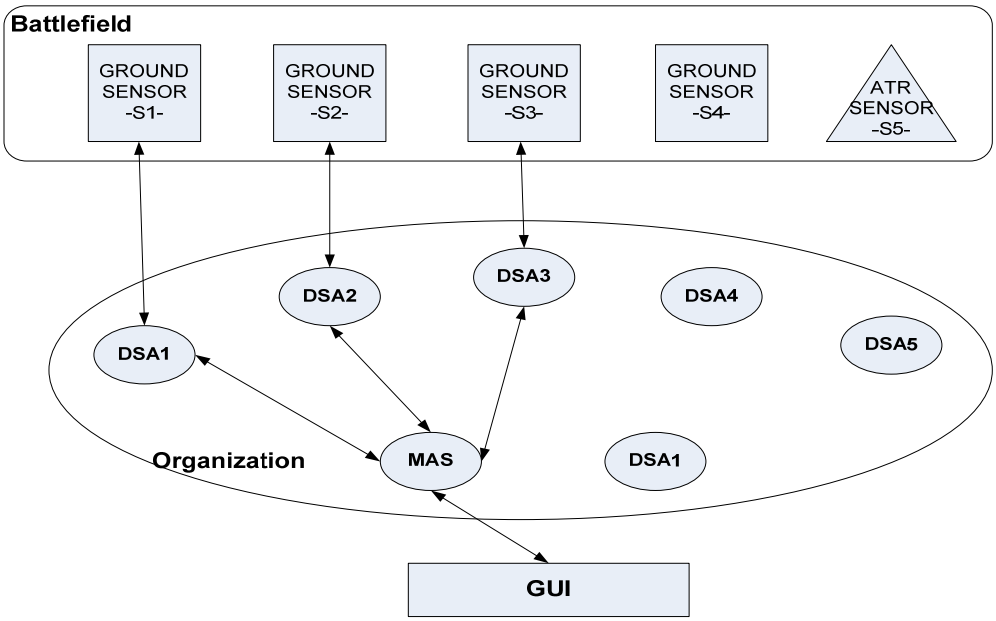


Figure 28. Normal Execution

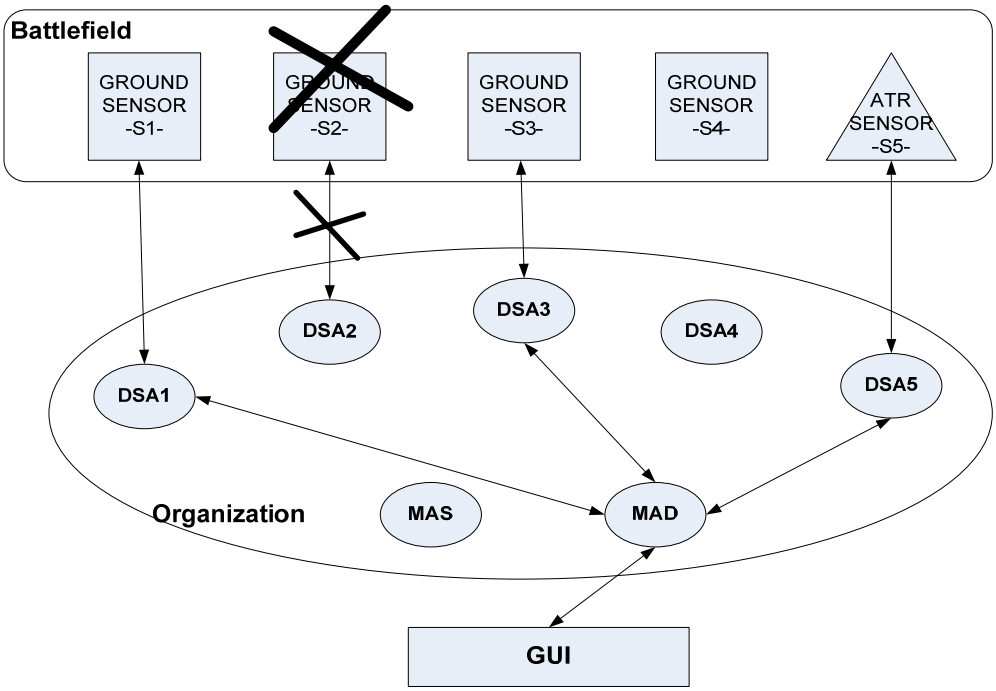


Figure 29. Execution with Sensor Failure

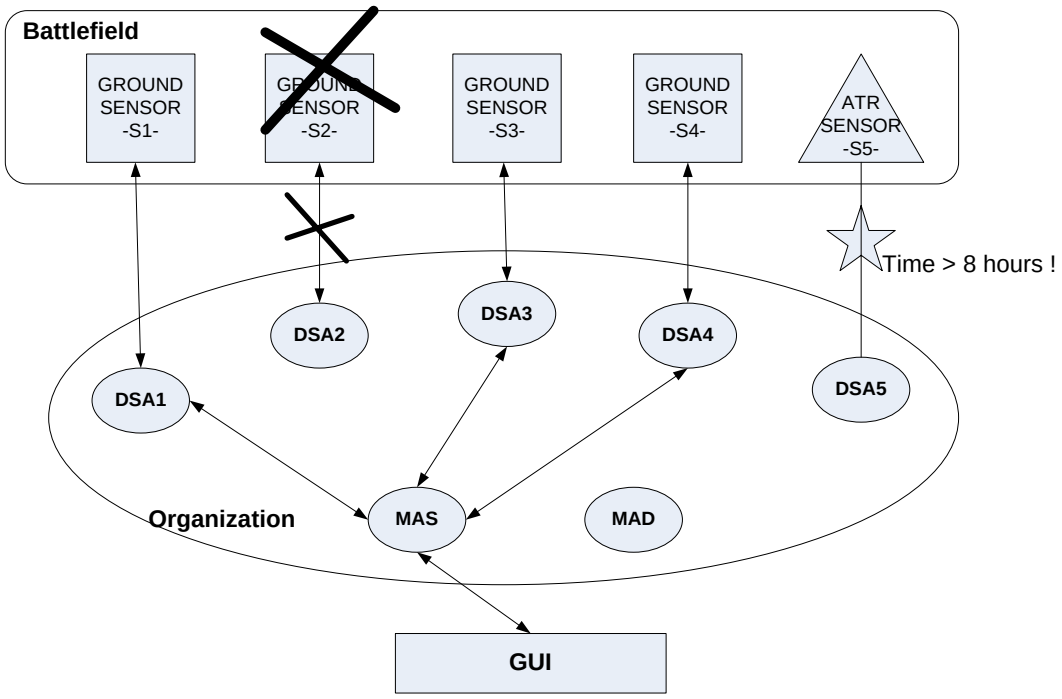


Figure 30. Execution with Maintenance Goal Failure

6.4 Conclusion

We designed a Battlefield information System capable of adapting to changes in the environment and modify its information processing algorithms consequently. The organizational framework on which the BIS is based, allows us to develop a scalable and flexible system, where agents can reorganize to overcome unpredictable situations and ensure the best performance of the system.

Chapter 7 - Conclusions and Future Work

7.1 Conclusions

There were four significant results from AFOSR grant F49620-02-01-0427.

1. The creation of a model useful for developing distributed, adaptive systems based on organization theory, the Organization Model for Adaptive Computational Systems.
2. The creation of a model – the Goal Model for Dynamic Systems – for specifying goals for dynamic systems that allows for ordering of goals and the creation of new goal instances based on events that occur during system operation.
3. The initial investigation of a software engineering methodology for designing distributed, adaptive systems using a dynamic goal model.

The result of this work has been documented in this report as well as 11 other publications.

1. Scott A. DeLoach, et. al. A Capabilities Based Theory of Artificial Organizations. In development for submission to journal.
2. Scott A. DeLoach. Multiagent Systems Engineering of Organization-based Multiagent Systems. 4th International Workshop on Software Engineering for Large-Scale Multi-Agent Systems (SELMAS'05). May 15-16, 2005, St. Louis, MO.
3. Eric Matson & Scott A. DeLoach. Formal Transition in Agent Organizations, IEEE International Conference on Knowledge Intensive Multiagent Systems (KIMAS '05), Waltham, MA, April 18-21, 2005.
4. Eric Matson & Scott A. DeLoach. Autonomous Organization-Based Adaptive Information Systems, IEEE International Conference on Knowledge Intensive Multiagent Systems (KIMAS '05), Waltham, MA, April 18-21, 2005.
5. David Gustafson, Venkata Prashant Rapaka, Scott DeLoach. A Comparison of Algorithms for Teams of Robots. Proceedings of the 2004 International Conference on Systems, Man and Cybernetics. October 10-13 2004 The Hague, The Netherlands.
6. Scott A. DeLoach, Eric Matson. An Organizational Model for Designing Adaptive Multiagent Systems. The AAAI-04 Workshop on Agent Organizations: Theory and Practice (AOTP 2004). Technical Report WS-04-02. AAAI Press. pp. 66-73. July 25-29, 2004, San Jose, California.
7. Eric Matson & Scott A. DeLoach. Integrating Robotic Sensor and Effector Capabilities with Multi-Agent Organizations. Proceedings of The 2004 International Conference on Artificial Intelligence (IC-AI'04). Las Vegas, Nevada, USA. June 21 - 24, 2004.
8. Eric Matson & Scott A. DeLoach. Enabling Intra-Robotic Capabilities Adaptation Using an Organization-Based Multiagent System. Proceedings of the 2004 IEEE International Conference on Robotics and Automation (ICRA 2004). April 26 – May 1, 2004. New Orleans, LA.
9. Eric Matson & Scott A. DeLoach. An Organization-Based Adaptive Information System for Battlefield Situational Analysis. Proceedings of the International Conference on Integration of Knowledge Intensive Multi-Agent Systems: KIMAS'03: Modeling, Exploration, and Engineering. 30 Sep – 3 Oct 2003. Boston, MA
10. Eric Matson and Scott DeLoach. Using Dynamic Capability Evaluation to Organize a Team of Cooperative, Autonomous Robots. Proceedings of The 2003 International Conference on Artificial Intelligence (IC-AI'03) June 23-26, 2003, Las Vegas, Nevada, USA.

11. Eric Matson, Scott A. DeLoach. Organizational Model for Cooperative and Sustaining Robotic Ecologies. Proceedings of Robosphere 2002, a workshop on Self Sustaining Robotic Ecologies, pp. 5-9. NASA Ames Research Center November 14-15, 2002.

7.2 Future Work

While this research created two basic models and an initial methodology for developing dynamic, organization-based systems, it also instigated additional investigation into other areas of interest.

1. Complete definition of the initial methodology proposed under this research. This research is being funded by AFOSR grant BG0229, Organization-based Model-driven Development of High-assurance Multiagent Systems. In this research, the Organization-based Multiagent Systems Engineering (O-MaSE) methodology will be completely defined and integrated with advanced verification and validation techniques to (1) predict system performance and (2) ensure the system operates within preset bounds. The project will integrate the verification and validation capabilities via the agentTool III analysis and design tool being developed to support O-MaSE.
2. Adapt the OMACS and GMoDS models to cooperative robotics area and sensor networks area. While there are many similarities between agent-based software systems and cooperative robotics and sensor networks, that fact that robots and sensors are hardware systems adds some complexity and rigidity that is not there in purely software systems. Specifically, the interaction with the environment is much more critical in hardware-based systems. The adaptation to cooperative robotics is being funded by the National Science Foundation under grant 0347545. The adaptation towards sensor networks is being funded by a Kansas State University Target Excellence grant and NSF Computing Research Infrastructure grant (to be awarded).
3. As the OMACS and GMoDS models allow the development of complex software and hardware agent systems, the ability to control these systems becomes increasingly important. The organizational paradigm allows a unique approach towards human-based control of distributed agent/robot teams. Current Air Force UAVs such as Predator and Global Hawk require multiple operators to carry out a single sortie due to the concurrent demands of image management, navigation, systems monitoring and decision making processes. This reality makes the task of controlling teams of robots seem daunting. Thus, employing *organizational control* should permit the human operator to focus on a single entity representing a team of agents/UAVs, rather than individual agents. The objective of this work would be to evaluate the ability of a human operator to interact with a team employing organizational control.

REFERENCES

1. Eric Matson, Scott DeLoach. *Organization-based Adaptive Information Systems for Battlefield Situational Analysis*, Proceedings of the IEEE International Conference on Integration of Knowledge Intensive Multi-Agent Systems (IEEE KIMAS '03), Boston, MA, October 1-3, 2003.
2. Scott A. DeLoach, Eric Matson. *An Organizational Model for Designing Adaptive Multiagent Systems*. The AAAI-04 Workshop on Agent Organizations: Theory and Practice (AOTP 2004), San Jose, California, July 25-29, 2004.
3. Blau, P.M. & Scott, W.R., Formal Organizations, Chandler, San Francisco, CA, 1962, 194-221.
4. Bradshaw, J. M., Uszok, A., Jeffers, R., Suri, N., Hayes, P., Burstein, M. H., Acquisti, A., Benyo, B., Breedy, M. R., Carvalho, M., Diller, D., Johnson, M., Kulkarni, S., Lott, J., Sierhuis, M., & Van Hoof, R. (2003). Representation and reasoning for DAML-based policy and domain services in KAoS and Nomads. Proceedings of the Autonomous Agents and Multi-Agent Systems Conference (AAMAS 2003). Melbourne, Australia, New York, NY: ACM Press, 2003.
5. Cabri, G., Leonardi, L., and Zambonelli, F. Implementing Agent Auctions using MARS. Technical Report MOSAICO/MO/98/001.
6. Carley, K. M. *Organizational Adaptation*. Annals of Operations Research, 1998. 75: 25-47.
7. Carley, K.M., and Gasser, L. *Computational Organization Theory*. In G. Weiss, ed. Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence. MIT Press, Cambridge, MA, 1999.
8. Carley, K. M. *Computational and Mathematical Organization Theory: Perspective and Directions.* Computational and Mathematical Organization Theory, 1995. 1(1): 39 – 56.
9. Chaimowicz, L., Sugar, T., Kumar, V. and Campos, M. F. M., *An Architecture for Tightly Coupled Multi-Robot Cooperation*, Proceedings of the 2001 IEEE International Conference on Robotics and Automation, pp. 2292-2297. Seoul – Korea, May, 2001.
10. Cohen, P. R. and Levesque, H. J. 1991. *Teamwork*. Nous 25(4), Special Issue on Cognitive Science and Artificial Intelligence, 1991 pp. 487-512.
11. Cohen, P.R., and Levesque, H.J. *Intention is Choice with Commitment*. Artificial Intelligence, 42(3), 1990.
12. DeLoach, S. A. *Modeling Organizational Rules in the Multiagent Systems Engineering Methodology*. Proceedings of the 15th Canadian Conference on Artificial Intelligence. Calgary, Alberta, Canada. May 27-29, 2002.
13. DeLoach, S. A. *Analysis and Design of Multiagent Systems Using Hybrid Coordination Media*. Proceedings of Software Engineering in Multiagent Systems (SEMAS 2002). Orlando, Florida. July 18, 2002.
14. DeLoach, S. A., Wood, M. F. and Sparkman, C. H., *Multiagent Systems Engineering*, The International Journal of Software Engineering and Knowledge Engineering, Volume 11 no. 3, pp. 231-258, June 2001.
15. Ferber, J., and Gutknecht, O.. A meta-model for the analysis and design of organizations in multi-agent systems. In Proceedings of Third International Conference on MultiAgent Systems (ICMAS'98), pages 128-135, IEEE Computer Society, 1998.
16. Ferber, J. Multi-Agent Systems: An Introduction to Distributed Artificial Intelligence. Addison-Wesley, 1999.

17. Ferber, J., Gutknecht, O., Jonker, C.M., Müller, J.P., and Treur, J., Organization Models and Behavioral Requirements Specification for Multi-Agent Systems. In: Y. Demazeau, F. Garijo (eds.), Multi-Agent System Organizations. Proc. of the 10th European Workshop on Modeling Autonomous Agents in a Multi-Agent World, MAAMAW'01. Lecture Notes in AI, Springer Verlag. To appear, 2002.

18. Ferber, J., Gutknecht, and Michel, F. From Agents to Organizations: An Organizational View of Multi-agent Systems. In Paolo Giorgini, Jörg P. Müller, James Odell (Eds.): Agent-Oriented Software Engineering IV, 4th International Workshop, AOSE 2003. pp 214-230. LNCS 2935. Springer-Verlag, 2003.

19. Ficher, M. J., Lynch, N. A., and Paterson, M. S. ``Impossibility of Distributed Consensus with One Faulty Process," *J. ACM 32*, 2 (April 1985), 374--382.

20. Grosz, B. and Sidner, C. *Plans for Discourse*. In P.R. Cohen, J. Morgan, and M. Pollack, eds., Intentions in Communication. pages 417—444. MIT Press, 1990.

21. Grosz, B. J., and Kraus, S. *Collaborative plans for complex group action*. Artificial Intelligence 86(2): 269—357, 1996.

22. Ishida, T., Gasser, L., and Yokoo, M. *Organization self design of production systems*. IEEE Transactions on Knowledge and Data Engineering, 4(2): 123--134, April 1992.

23. Jennings, N. R. Commitments and Conventions: The Foundation of Coordination in Multiagent Systems. In The Knowledge Engineering Review, 8(3): 223-250, 1993.

24. Jennings, N.R. Controlling Cooperative Problem Solving in Industrial Multi-Agent Systems Using Joint Intentions. Artificial Intelligence, 75 (2) 195-240, 1995.

25. Jennings, N.R. *Towards A Cooperation Knowledge Level For Collaborative Problem Solving*. In Bernd Neumann, editor, Proceedings of the 10th European Conference on Artificial Intelligence, pages 224-228, Vienna, Austria, August 1992. John Wiley & Sons Ltd.

26. Hyuckchul Jung, Jeffrey M. Bradshaw, Shri Kulkarni, Maggie Breedy, Larry Bunch, Paul Feltovich, Renia Jeffers, Matt Johnson, James Lott, Niranjan Suri, William Taysom, Gianluca Tonti, & Andrzej Uszok, An Ontology-Based Representation for Policy-Governed Adjustable Autonomy, Proceedings of the AAAI Spring Symposium on Knowledge Representation and Ontology for Autonomous Systems, 2004.

27. Kinny, D., Ljungberg, M., Rao, A. S., Sonenberg, E., Tidhar, G. and Werner, E. *Planned Team Activity*. In Artificial Social Systems - Selected Papers from the Fourth European Workshop on Modeling Autonomous Agents in a Multi-Agent World (MAAMAW-92), C. Castelfranchi and E. Werner, Eds. pp. 226--256. Volume 830 of Lecture Notes in Artificial Intelligence, Springer-Verlag, Heidelberg, 1992.

28. Lawrence, P.R., and Lorsch, J.W., *Organization and Environment: Managing Differentiation and Integration*, Division of Research, Graduate School of Business Administration, Harvard University, 1967.

29. Levesque, H. J., Cohen, P. R., and Nunes, J. *On acting together*. In Proceedings of the National Conference on Artificial Intelligence. Menlo Park, Calif.: AAAI press, 1990.

30. Eric Matson & Scott A. DeLoach. *An Organization-Based Adaptive Information System for Battlefield Situational Analysis*. Proceedings of the International Conference on Integration of Knowledge Intensive Multi-Agent Systems: KIMAS'03: Modeling, Exploration, and Engineering. 30 Sep – 3 Oct 2003. Boston, MA.

31. Eric Matson & Scott A. DeLoach. *Integrating Robotic Sensor and Effector Capabilities with Multi-Agent Organizations*. Proceedings of The 2004 International Conference on Artificial Intelligence (IC-AI'04). Las Vegas, Nevada, USA. June 21 - 24, 2004.

32. *MESSAGE: Methodology for Engineering Systems of Software Agents*. Deliverable 1. Initial Methodology. July 2000. EURESCOM Project P907-GI.

33. Moses, Y. and Tennenholtz, M. *Artificial Social Systems Part I: Basic Principles*. Technical Report CS90-12, Weizmann Institute, 1990.

34. Moses, Y., and Tennenholtz, M. *Artificial Social Systems*, Computers and Artificial Intelligence, 14(3): 533-562, 1995.

35. Odell, J., Nodine, M., Levy, R. A Metamodel for Agents, Roles, and Groups. To be published in Agent-Oriented Software Engineering V, 5th International Workshop, AOSE 2004. 2004.

36. Onn S. and Tennenholtz, M. *Determination of social laws for multi-agent mobilization*. Artificial Intelligence, 95:155--167, 1997.

37. Parker, L. ALLIANCE: An Architecture for Fault-Tolerant Multi-Robot Cooperation. IEEE Transactions on Robotics and Automation, 14 (2), 220—240, 1998.

38. Pollack, M.E. *Plans as complex mental attitudes*. In Phillip R. Cohen, Jerry Morgan, and Martha E. Pollack, editors, Intentions in Communication, pages 77-- 103. MIT Press, Cambridge, MA, 1990.

39. Russell, S. and Norvig, P. Artificial Intelligence a Modern Approach. 2nd Ed. Pearson Education, Inc. 2003.

40. Shoham, Y. and Tennenholtz, M. *On Social Laws for Artificial Agent Societies: Off-line Design*. Artificial Intelligence, 73, 1995. 47

41. Sonenberg, E., Tidhar, G., Werner, E., Kinny, D., Ljungberg, M., and Rao, A. *Planned Team Activity*. Technical Report 26, Australian Artificial Intelligence Institute, Australia, 1992.

42. Tambe, M. *Towards flexible teamwork*. Journal of Artificial Intelligence Research (JAIR), 7:83--124, 1997.

43. Turner, R. M., and Turner, E. H. A Two-Level, Protocol-Based Approach to Controlling Autonomous Oceanographic Sampling Networks. IEEE Journal of Oceanic Engineering, vol. 26 (4), pp. 654-666, October 2001.

44. van Lamsweerde, A., Darimont, R., Letier, E. Managing conflicts in goal-driven requirements engineering. *IEEE Transactions on Software Engineering*. 24(11), pp 908-926, 1998.

45. Wagner, G. Agent-Oriented Analysis and Design of Organizational Information Systems. Proceedings of the 4th IEEE International Baltic Workshop on Databases and Information Systems, Vilnius, Lithuania, May 2000.

46. Wooldridge, M., Jennings, N.R., and Kinny, D. *The Gaia Methodology for Agent-Oriented Analysis and Design*. Journal of Autonomous Agents and Multi-Agent Systems. Volume 3(3), 2000.

47. Zambonelli, F., Jennings, N.R., and Wooldridge, M. Organisational abstractions for the analysis and design of multi-agent systems. In P. Ciancarini and M. Wooldridge, editors, in Agent-Oriented Software Engineering – Proceedings of the First International Workshop on Agent-Oriented Software Engineering, 10th June 2000, Limerick, Ireland. P. Ciancarini, M. Wooldridge, (Eds.) Lecture Notes in Computer Science. Vol. 1957, Springer Verlag, Berlin, pages 207-222, January 2001.

48. Zambonelli, F., Jennings, N.R., and Wooldridge, M.J. Organisational Rules as an Abstraction for the Analysis and Design of Multi-Agent Systems. International Journal of Software Engineering and Knowledge Engineering. Volume 11, Number 3, June 2001. Pages 303-328.

49. Zambonelli, F., Jennings, N.R., Omicini, A., and Wooldridge M.J. Agent-Oriented Software Engineering for Internet Applications. Coordination of Internet Agents: Models, Technologies, and Applications, Chapter 13. Springer-Verlag, March 2001.

50. Bernon, C., Camps, V., Gleizes M.P., Picard G. Engineering Adaptive Multi-Agent Systems: the ADELFE Methodology. In B. Henderson-Sellers and P. Giorgini (Eds.), Agent-Oriented Methodologies. Idea Group Pub, June 2005, pp.172-202.

51. Blau, P.M. & Scott, W.R., Formal Organizations, Chandler, San Francisco, CA, 1962, 194-221.

52. Cranefield, S. & Pruvis, M. UML as an Ontology Modelling Language. Proc of the Workshop on Intelligent Information Integration, 1999.

53. DeLoach, S. A. Modeling Organizational Rules in the Multiagent Systems Engineering Methodology. Proc of the 15th Canadian Conference on Artificial Intelligence. 2002.

54. DeLoach, S. A. Analysis and Design of Multiagent Systems Using Hybrid Coordination Media. Proceedings of Software Engineering in Multiagent Systems (SEMAS 2002). 2002.

55. DeLoach, S. A., Wood, M. F. and Sparkman, C. H., "Multiagent Systems Engineering". The International Journal of Software Engineering and Knowledge Engineering, 11(3), pp. 231-258, June 2001.

56. DeLoach, S.A., & Matson, E. An Organizational Model for Designing Adaptive Multiagent Systems. The AAAI-04 Workshop on Agent Organizations: Theory and Practice (AOTP 2004). 2004.

57. Dignum, V. A Model for Organizational Interaction: Based on Agents, Founded in Logic. PhD thesis, Utrecht University, 2004.

58. Ferber, J., and Gutknecht, O. A meta-model for the analysis and design of organizations in multi-agent systems. In Proceedings of Third International Conference on MultiAgent Systems (ICMAS'98), pages 128-135, IEEE Computer Society, 1998.

59. Huget, M.P., Bauer, B., Odell, J., Levy, R., Turci, P., Cervenka, R., and Zhu, H. http://www.auml.org/. FIPA Modeling: Interaction Diagrams, Working Draft. 2002.

60. Matson, E., DeLoach, S. Capability in Organization Based Multi-agent Systems, Proceedings of the Intelligent and Computer Systems (IS '03) Conference, 2003.

61. MESSAGE: Methodology for Engineering Systems of Software Agents. Deliverable 1. Initial Methodology. July 2000. EURESCOM Project P907-GI.

62. Odell, J., Nodine, M., Levy, R. A Metamodel for Agents, Roles, and Groups. Agent-Oriented Software Engineering V, 5th International Workshop, AOSE 2004. 2004.

63. Picard, G. and Gleizes, M.-P. The ADELFE Methodology – Designing Adaptive Cooperative Multi-Agent Systems. In Bergenti, F. and Gleizes, M-P. and Zambonelli, F., editor, Methodologies and Software Engineering for Agent Systems. Kluwer Publishing, 2004.

64. Robby, Dwyer, M.B., & Hatcliff, J. Bogor: An Extensible and Highly-Modular Model Checking Framework, Proceedings of the Fourth Joint Meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2003).

65. van Lamsweerde, A., Darimont, R., Letier, E. Managing conflicts in goal-driven requirements engineering. IEEE Transactions on Software Engineering. 24(11), pp 908-926, 1998.

66. Wagner, G. Agent-Oriented Analysis and Design of Organizational Information Systems. Proceedings of the 4th IEEE International Baltic Workshop on Databases and Information Systems, May 2000.

67. Zambonelli, F., Jennings, N.R., and Wooldridge, M.J. Developing Multiagent Systems: The Gaia Methodology. In AMC Transactions on Software Engineering Methodology 12(3), 317-370, 2003.

68. Zambonelli, F., Jennings, N.R., and Wooldridge, M.J. Organisational Rules as an Abstraction for the Analysis and Design of Multi-Agent Systems. IJSEKE. 11(3) pp. 303-328, June 2001.

69. Tambe, M., and Zhang, W. *Towards flexible teamwork in persistent teams: extended report*. Journal of Autonomous Agents and Multi-agent Systems, Vol. 3, 2000, 159-183.