Web Intelligence and Agent Systems: An International Journal 8 (2010) 79–97
DOI 10.3233/WIA-2010-0181
IOS Press

Towards a systematic approach for designing autonomic systems[1]

Walamitien H. Oyenan* and Scott A. DeLoach
Department of Computing & Information Sciences, Kansas State University, 234 Nichols Hall, Manhattan, KS 66506, USA
E-mail: {oyenan, sdeloach}@ksu.edu

Abstract. An autonomic system is a system capable of managing itself and adjusting its actions in the face of environmental changes. Autonomic systems are currently developed using ad-hoc approaches, which do not promote repeatable successes. In this paper, we propose a systematic approach for designing autonomic systems. Our approach adopts a multiagent perspective based on the Organization Model for Adaptive Computational Systems, which defines the knowledge required for the system to be able to self-organize. Furthermore, a customized development process based on the Organization-based Multiagent Systems Engineering framework supports our approach. To illustrate the process, we describe the design of one autonomic system, the Autonomic Information System, and exemplify how this system fulfills desired autonomic properties. We also evaluate the performance of our autonomic system by comparing it to a non-autonomic system.

Keywords: Autonomic systems, organizations, multiagent systems, self-organization

1. Introduction

The goal of autonomic computing is to create new systems that are able to manage themselves. This requires that such systems have the ability to self-configure, self-optimize, self protect, and self-heal [10,13,16,24]. As systems become increasingly complex, they are expected to handle this complexity on their own. Therefore, it is crucial that they exhibit autonomic behavior.

While the advantages of using a multiagent approach have been recognized [6,26], many autonomic applications are developed from scratch, producing ad-hoc designs that work well for the proposed application. Unfortunately, the process is not repeatable, thus neither it nor the design can be reused in other autonomic applications. In our work, we adopt a multiagent approach for developing autonomic systems since agents are autonomous and map naturally to the autonomic computing principles [15]. In addition, our approach is based on a formal framework and is supported by a customizable multiagent development process.

The purpose of this paper is to demonstrate the effectiveness of a set of related technologies based on the Organization Model for Adaptive Computational System (OMACS) [7] for designing autonomic systems. OMACS defines the knowledge required for a system to be able to understand its own problem solving state and configuration in order to self-organize. Instead of building our system using ad-hoc methods, we defined a reusable OMACS-compliant process that incorporates all the entities required by OMACS. We designed this process using the Organization-based Multiagent System Engineering (O-MaSE) Process Framework [12], which is a framework for creating custom multiagent development processes. After designing the system according to our custom process, we implemented the system using our Organization-based Agent Architecture (OBAA). The OBAA is an architecture

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

created to support organization-based agents and to separate general autonomic reasoning from application specific tasks.

In this paper, we follow the development of one particular autonomic system, the Autonomic Information System (AIS), and illustrate how it realizes autonomic behavior. The goal of the AIS 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. 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 is available, it is fused with other related information to answer queries from the commander. A field commander uses the system interface to generate queries. To overcome the loss of sensors and continue to provide the required information, the AIS needs to adapt by replacing the failed sensors and adapting the information processing adequately without the intervention of the user. Our work has four main contributions.

1. It demonstrates the effectiveness of OMACS for building autonomic systems.
2. It defines a rigorous model-driven process, derived from the O-MaSE process framework, for designing autonomics systems.
3. It defines our generic OBAA architecture that serves as a blueprint for autonomic agents.
4. It demonstrates the validity of our approach through the design and evaluation of an exemplar autonomic system.

The remainder of this paper is organized as follows. First, we present other works related to autonomic multiagent systems in Section 2. In Sections 3 and 4 we provide an overview of the OMACS model and the O-MaSE process framework. Section 5 presents the design of our AIS organization while Section 6 introduces our OBAA architecture. In Section 7, we show the autonomic properties of the AIS via a scenario. Finally, a performance evaluation of the system is given in Section 8 while Section 9 concludes and discusses future work.

2. Related work

There have been several architectures proposed toward autonomic computing. For example, White et al. describe an architectural approach in which they suggest some required and optional behaviors, interfaces for component interaction, and some design patterns about the composition of autonomic component to insure a self-managing system [28]. Lapouchnian et al. suggest a design capable of supporting all alternative behaviors of an autonomic system by using a goal-oriented requirements engineering methodology [17]. Design templates have also been proposed for autonomic elements that monitor the system using heartbeat signals [25]. Appavoo et al. advocate a hot swapping technique to enable autonomic behavior in object-oriented systems [1]. Similarly, Schanne et al. propose to add autonomic features to object-oriented applications by incorporating proxy objects using a Java bytecode engineering toolkit [23].

Fewer works use multiagent approaches to building autonomic systems. One such approach is Unity, which is a decentralized software architecture in which autonomic elements are agents that have predefined responsibilities and reason based on some computed utility functions [26]. Like our system, Unity uses goals to initiate autonomic behaviors. However, the utility functions in Unity are tightly coupled with the agents design whereas our work offers a clear separation between application-related functionalities and autonomicity-related tasks.

Similarly, other works have their autonomic capacities tightly coupled with the agent architecture. Kumar and Cohen describe an adaptive agent architecture in which broker agents share the same knowledge of the system and are thereby aware of any agent failure [22], while Bigus et al. propose a set of agent component libraries that can be used to build autonomic systems [2]. These libraries extend the ABLE platform by adding external agents to explicitly manage and control the system, thus allowing the system to exhibit autonomic properties. Pour presents a three-tiered autonomic architecture in which different types of agents perform various tasks in several subsystems. In particular, the agents in the third tier are cognitive agent that are able to reason about the state of the system and initiate a reconfiguration [20].

All these works are similar to ours in the sense that agents have system-level knowledge that allows them to manage themselves in unpredictable environments. However, in these approaches, this system-level knowledge is tightly coupled with the application system architecture. In our approach, system-level knowledge is based on the underlying OMACS

model and is implemented using our OBAA architecture, which allows us to reuse the system-level knowledge and systematically develop autonomic applications.

While there have been several architectures proposed for autonomic system, it is also essential to develop software engineering methodologies for building those systems. Bustard et al. [5] propose integrating two systems engineering approaches, Viable Systems Modeling and Soft Systems Methodology, into a methodology for designing autonomic systems. However, their approach is not agent-based, and thus is very different from our work, which provides a rigorous methodology for designing flexible multiagent autonomic systems.

3. Overview of OMACS

OMACS [7] is a computational model that provides a metamodel and a formal framework for agent organizations. Essentially, it defines the required organizational structure that allows multiagent teams to reconfigure autonomously at runtime, thus enabling them to cope with unpredictable situations in a dynamic environment. Specifically, OMACS specifies the type of knowledge required for a multiagent system to be able to reason about its own state and configuration. Hence, multiagent teams are not limited by a predefined set of configurations and can have the appropriate information about their team, enabling them to reconfigure in order to achieve their team goals more efficiently and effectively. During the design of an OMACS-based system, the designer only provides high-level guidance about the organization, which then allows the system to self-configure based on the current goals and team capabilities. These characteristics make OMACS ideal for designing autonomic multiagent systems.

3.1. The OMACS metamodel

The OMACS metamodel is the metamodel upon which autonomic systems are designed. Figure 1 shows a simplified OMACS metamodel. Only the entities discussed in this paper are shown. OMACS defines an organization as a set of *goals* that the team is attempting to accomplish, a set of *roles* that must be played to achieve those goals, a set of *capabilities* required to play those roles, and a set of *agents* who are assigned to roles in order to achieve organization goals. In essence, each organization is an instance of the OMACS metamodel presented in Fig. 1 and is subject to all the constraints defined by OMACS. At runtime, the assignments of agents to play roles to achieve goals represent the key functionality that allows the system to be autonomic. There are more entities defined in OMACS that are not relevant for this paper. The reader is referred to [7] for the complete model.

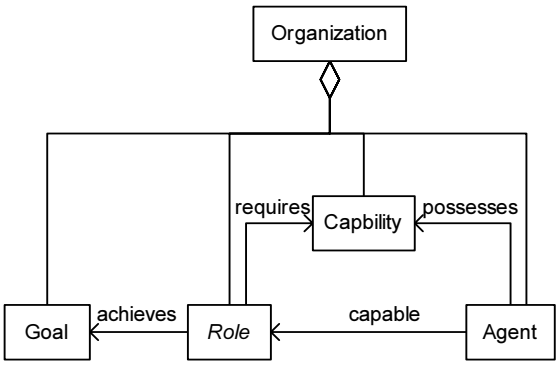

Fig. 1. Simplified OMACS metamodel.

3.2. Goals

Goals describe a desired state of the world and thus provide a high-level description of *what* the system is supposed to do [21]. Typically, each organization has a top-level goal that is decomposed into sub-goals. Eventually, this top-level goal is refined into a set of leaf goals that are pursued by agents in the organization. The set of all organizational goals is denoted as G. The *active goal set*, G_a, is the current set of goals that an organization is currently trying to achieve. G_a changes dynamically as new goals are created or existing goals are achieved.

3.3. Roles

Roles are a high-level description of the behavior required to achieve particular goals [9]. In OMACS, each organization has a set of roles that it can use to achieve its goals. The *achieves* function, which associates a score between 0 and 1 to each ⟨goal, role⟩ pair, tells how well that particular role can be used to achieve that goal (1 being the maximum score). In addition, each role *requires* a set of capabilities and agents must *possess* all the required

capabilities to be considered as a potential candidate to assume that role.

3.4. Capabilities

In OMACS, capabilities are fundamental in determining which agents can be assigned to what roles in the organization [18]. In fact, agents are *capable* of playing a role only if they posses all the required capabilities. However, the decision whether or not a capable agent is actually going to assume a role is made at runtime. Agents may possess two types of capabilities: hardware capabilities like actuator or effectors, and software capabilities like computational algorithms or resources.

3.5. Agents

OMACS agents are computational systems that have the ability to communicate with each other, accept assignments to play roles that match their capabilities, and work to achieve their assigned goals. Each agent is responsible for managing its own state and its interactions with the environment and with other agents. Once the system assigns a goal and role, the agent determines the low-level behavior necessary to fulfill the role and achieve the goal. This low-level behavior is generally provided either as part of the role definition or by a unique agent behavior specified by the designer. To capture a given agent's capabilities, OMACS defines a *possesses* function, which maps each ⟨agent, capability⟩ pair to a value between 0 and 1, describing the quality of the capability possessed by an agent (1 representing the maximum quality).

In OMACS, a tuple $\langle a,r,g \rangle$ represents the assignment of *agent a* to play *role r* in order to achieve *goal g*. The assignment set, denoted Φ, represents the set of all the current assignments in the organization.

3.6. Assignment process

The set of active goals along with the agents and their capabilities can change over time. For this reason, the process of assigning agents to play roles in order to achieve specific goals is not predefined but rather performed dynamically at runtime. This process takes into consideration the quality of each capability possessed by agents along with how well roles can achieve goals. For example, if a new goal is instantiated within the organization, a greedy algorithm could compute a new assignment by first choosing the best role for that goal then the best agent capable of playing the chosen role. However, OMACS does not prescribe any particular algorithm for computing assignments and several algorithms been investigated for this purpose [30].

4. Overview of O-MaSE

In this section, we give a brief overview of the Organization-based Multiagent System Engineering (O-MaSE) Process Framework [12]. O-MaSE is a framework that allows designers to create custom agent-oriented development processes. This custom agent-oriented process is generated following a process metamodel and then instantiated from a set of method fragments and guidelines by using a method engineering approach [4]. Method engineering is an approached that has been proposed to allow the development of software methodologies from several fragments.

Thus, O-MaSE defines a metamodel, a repository of method fragments and a set of guidelines. The O-MaSE metamodel defines general concepts used in multiagent systems along with their relationships and is based on an organizational approach. In fact, there is a 1:1 projection of the OMACS metamodel onto the O-MaSE metamodel, which allows systems developed using appropriate O-MaSE method fragments to produce valid instances of the OMACS metamodel. Organizations developed using an O-MaSE compliant process produce a set of models that specify valid instances of the O-MaSE metamodel. Method fragments are a set of activities, techniques and work products extracted from existing agent methodologies and stored in a repository. They are later combined to create a methodology instance which is used on a project. O-MaSE method fragments currently cover the requirements, analysis and design phases of a multiagent development lifecycle. Finally, O-MaSE guidelines specify a set of constraints that must be maintained when combining method fragments to create valid O-MaSE processes.

Therefore, designing a custom O-MaSE compliant process requires process engineers to select a set of methods that suit their needs from the repository and combine them into a complete process such that the constraints of each fragment are satisfied. O-MaSE provides some guidelines to help choose fragments but does not guarantee that all processes created will

necessarily be efficient. However, the O-MaSE Process Framework does allow designers to develop rigorous and repeatable processes suitable for their particular needs.

The O-MaSE Process Framework is supported by the agentTool Process Editor, which is part of the agentTool III[2] (aT^3) development environment. The agentTool Process Editor (APE) allows process designers to create custom O-MaSE processes, which can then be analyzed and designed using the (aT^3) development environment. Further details on aT^3 and APE can be found in [11]. The O-MaSE process used in this research is presented next. All the diagrams required by this process have been created using aT^3.

5. Designing the AIS organization

Our autonomic system is an organization-based multiagent system [3] built upon OMACS. Hence, to implement the system, we first need to create an O-MaSE process that captures all the concepts of OMACS. Then we can follow our custom process to design and implement an OMACS-based organization for the AIS application. In the following subsections, we define our custom process and implement each step of the process.

5.1. A process for autonomic systems

We chose relevant method fragments from the O-MaSE repository in order to derive a process for building our autonomic system. As we were interested in developing an OMACS-based system, we chose method fragments that produced the appropriate OMACS concepts. Thus, we produced a customized process that included all the necessary concepts related to the OMACS goals, roles, capabilities and agents.

The process we used (and that can be used for many autonomic systems) is shown in Fig. 2. We represent it as an activity diagram in which we show tasks as round-cornered rectangles and models as square-cornered rectangles. Arrows represent the input and output of models to and from tasks.

The process starts by translating all the system requirements into a *Goal Model* via the *Model Goals* and *Goal Refinement* tasks. Then we use the Goal Model as an input of the *Model Role* task in order to create a *Role Model* that captures all the interactions

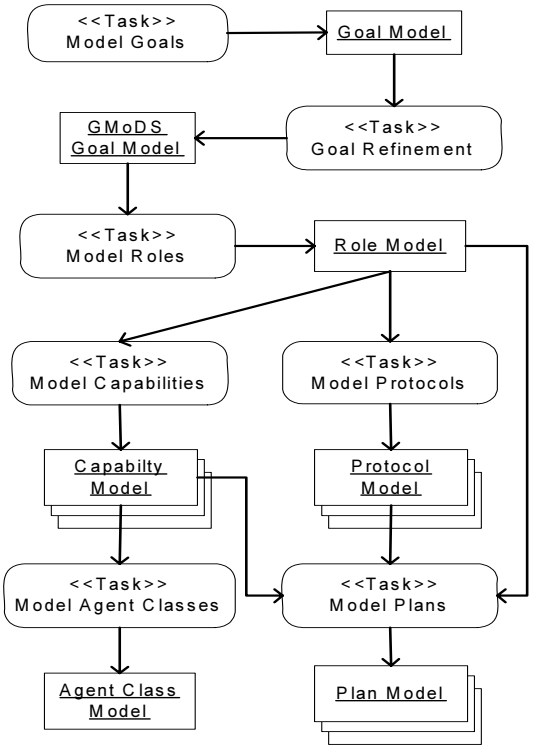

Fig. 2. Autonomic systems process: Round-cornered rectangles represent Tasks, square-cornered rectangles represent Models and arrows indicate Models Flows.

between roles and external actors. Following that, we generate a *Capability Model* using the *Model Capabilities* task that allows us to define all capabilities required in the organization. Once the Capability Model is completed, we can then design agent types that are able to play roles in the organization. Agent types are modeled via an *Agent Class Model* produced by the *Model Agent Classes* task. In parallel with the Model Capabilities task, we also specify the message passing protocols required to allow roles to interact using *Protocol Models* in the *Model Protocol* task. Then, using the Role, Capability, and Protocol Models as input, we perform the *Model Plans* task in order to generate a *Plan Model* for each role. Plan Models specify plans that the agents need to execute in order to play roles and achieve their assigned goals. All method fragments have been chosen such that the constructed organization is consistent with the OMACS metamodel.

As it is possible to create a variety of different processes that are consistent with OMACS, it is difficult to evaluate the effectiveness of all such

[2] See http://agenttool.projects.cis.ksu.edu/.

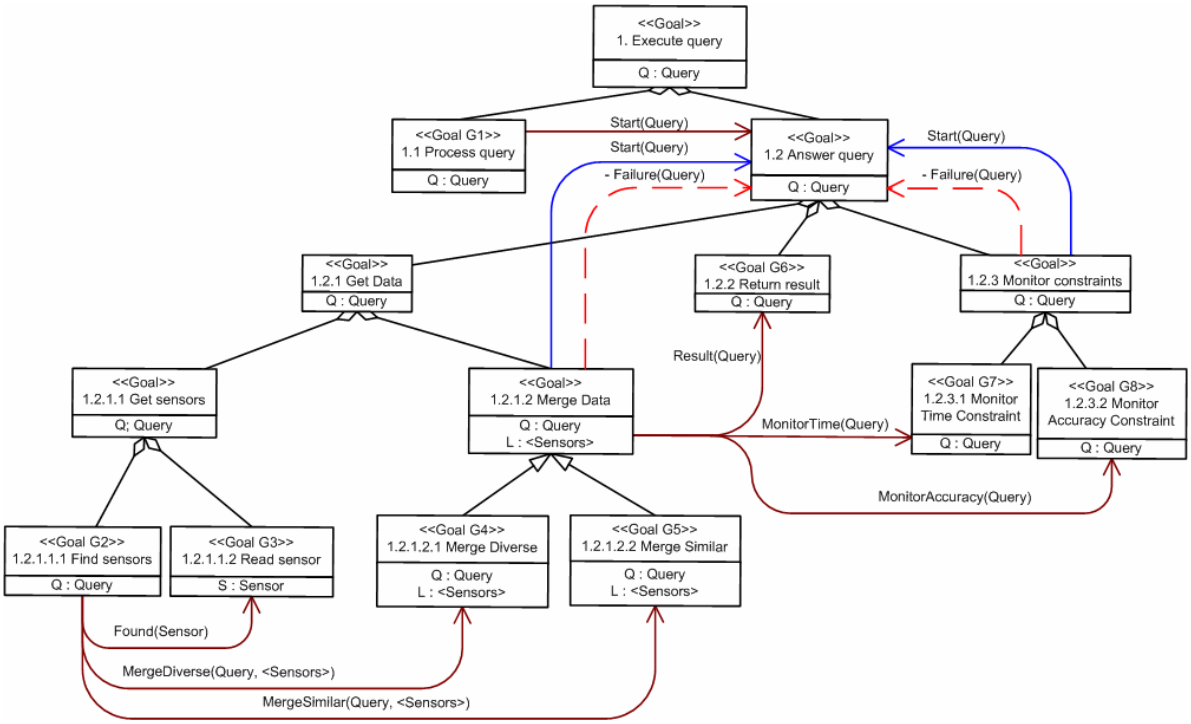

Fig. 3. GMoDS Goal Model for the AIS.

processes. However, we claim that this process, and similar processes developed using the O-MaSE Process Framework, do provide a principled, rigorous and repeatable method to systematically develop autonomic systems. A further discussion of the selection of method fragments and the verification of process consistency can be found in [11,12].

5.2. Model goals

Following our process, the first step in designing an autonomic system is to capture the system requirements in the form of a goal tree via the Model Goals and Refine Goals tasks. A Goal Model represents system-level goals and includes goal definitions and goal decomposition, which decomposes goals into a set of conjunctive or disjunctive sub-goals [27].

To capture the dynamic nature of OMACS-based systems, a time-based relationship exists between goals. This additional information is captured through a goal model based on the Goal Model for Dynamic Systems (GMoDS) [19] during the Refine Goals task. In a GMoDS goal model, we say goal g1 precedes goal g2, if g1 must be achieved before g2 can be pursued. This allows the organization to work on one part of the goal tree at a time. During the pursuit of specific goals, events may occur that cause the instantiation of new goals. These new goals may be parameterized to allow a context sensitive meaning. For instance, if an event e can occur during pursuit of goal g1 that instantiates goal g2, we say that e triggers g2 during the pursuit of g1 (as a shorthand, we often say *goal g1 triggers g2*). In addition, GMoDS proposes two types of goals: achievement goals, which the system seeks to achieve to insure normal operation and maintenance goals, which the system uses to continually monitor its own operation and check for performance improvement or failures.

The main goal of the AIS application is to answer each query presented to the system. From the requirements, we derived the GMoDS goal model presented in Fig. 3. The boxes represent the goals and their parameters. Conjunctive sub-goals are connected to their parents by a diamond shaped connector (◊) while disjunctive sub-goals are connected to their parent by a triangle shaped connector (Δ). The arrows indicate trigger events and their parameters. The dashed arrows represent *negative triggers*, which allow for the cancellation of

a goal and all its sub-goals. As queries are not predefined, most of the goals in the systems are triggered after a query has occurred. These goals and their parameters are instantiated and associated with specific queries.

From Fig. 3, we can see that the top-level AIS goal is decomposed into two conjunctive sub-goals: *Process Query* and *Answer Query*. The *Answer Query* goal is further decomposed into conjunctive goals *Get Data*, *Monitor Constraints* and *Return Result*, all of which have to be achieved in order to achieve the *Answer Query* goal. All those sub-goals, except the *Return Result* goal, are also further decomposed into sub-goals. The goal *Get Data* is decomposed into conjunctive goals *Get Sensors* and *Merge Data* and the goal *Monitor Constraints* into conjunctive goals *Monitor Time Constraint* and *Monitor Accuracy Constraint*, which are both maintenance goals. Finally, the goal *Get Sensor* is decomposed into conjunctive goals *Find Sensor* and *Read* Sensors, whereas the goal *Merge Data* is decomposed into disjunctive goals *Merge Diverse* and *Merge Similar*.

The organization only actively pursues the leaf goals as their completion implies the completion of their parent goal. The first leaf goal to be achieved by the organization is the *Process Query* goal, which gets the query from the user and triggers the *Answer Query* goal based on the event *Start* parameterized with a *Query*. Once the *Answer Query* goal is triggered, all its descendants that are not triggered by other events are also triggered. Thus, goal *Find Sensor* is triggered by the same event. This goal aims at finding all the sensors in the area of interest given by query Q. Whenever sensors are found, it triggers goals *Read Sensor*, and either *Merge Diverse* or *Merge Similar*. *Read Sensor* is a goal to read data from a sensor S passed in parameter. The *Merge Diverse* goal fuses the data received from the list of *different* sensors L for the area specified by query Q. Sensors in the list L must not all be the same type. On the opposite, goal *Merge Similar* fuses the data received from the list of *similar* sensors L for the area specified by query Q. Then, whenever constraints are specified in the query, the *Merge Data* goal triggers either goal *Monitor Time Constraint* or goal *Monitor Accuracy Constraint*. However, as *Merge Data* is not a leaf goal, the trigger is actually generated by one of its sub-goals. Goals *Monitor Time Constraint* and *Monitor Accuracy Constraint* check the validity of the data regarding the time constraint and the accuracy constraint respectively. Finally, when the data are ready, the *Merge Data* goal triggers the *Return Result* goal, which displays the results of the query Q in a user-friendly format.

In addition, in some cases of failures, the Merge Data and Monitor Constraints goals can initiate a *negative trigger* that cancels the *Answer Query* goal and all its descendants. This negative trigger is followed by a *start* event that triggers a new *Answer Query* goal and results in the organization retrying to achieve the query that previously failed.

5.3. Model roles

Once the goals of the system have been captured and translated into a dynamic goal model, we identify the required roles and their interactions through the Model Roles task.

For each leaf goal in the goal model defined in the previous task, we create a role that can achieve it. The Role Model for the AIS is presented in Fig. 4. It shows all the roles along with the protocols that exist between pairs of roles and between roles and external actors. For each role, the Role Model specifies which organizational goals can be achieved and the required capabilities. Following are the roles we have defined for the AIS organization, along with a description of their behavior.

Query Processor: Periodically gets new queries from the user via the *Interact_User* protocol. This role generates an event to notify the organization that a new query has been entered. There is no explicit protocol to send the query Q since the *Start(Query)* event triggers that creation of a new *Answer Query* goal, which has the query Q as its parameter. Each agent who gets assigned to achieve a goal with the Query parameter in effect receives a copy of Q.

Sensors Locator: Inquires the sensor database via the protocol *DB_Access* 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 results in the organization attempting to find an agent capable of reading the selected sensor. After all sensors have been selected, the role generates an event (event '*mergeSimilar*' or '*mergeDiverse*') to notify the organization that it has found all sensors capable of providing data for the query.

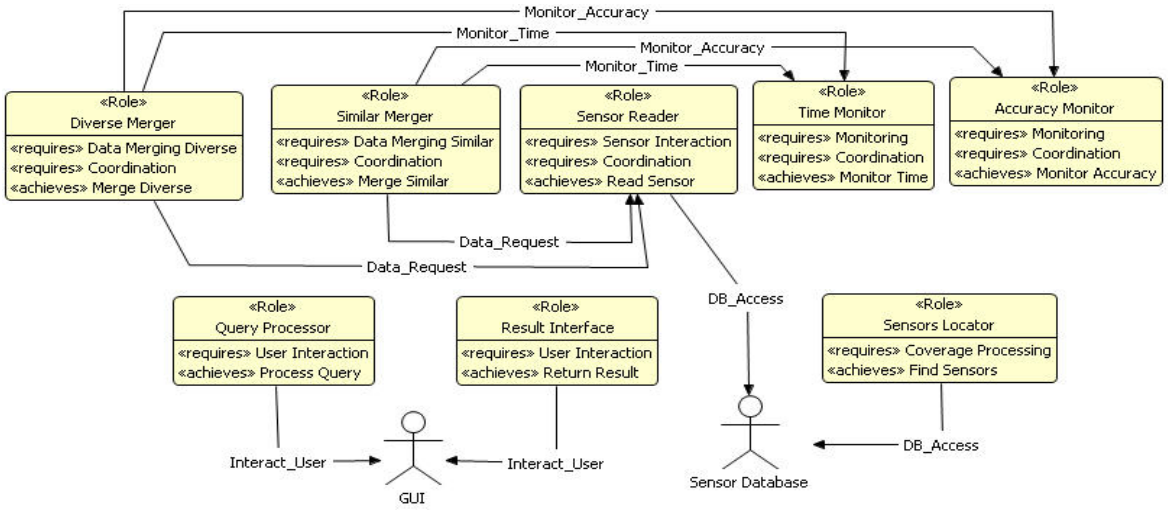

Fig. 4. AIS Role Model.

Sensor Reader: Reads 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 and updates the status of the sensor in the sensor database via the *DB_Access* protocol. The data obtained is sent to any merger agent interested in those data through the protocol *Data_Request*.

Diverse Merger: Merges 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. Data are obtained through the *Sensor Reader* role by using the protocol *Data_Request*. This role can also engage in the *Monitor_Time* and *Monitor_Accuracy* protocols in order to request validation of the data merged.

Similar Merger: Behaves like the *Diverse Merger* role. However, the difference is that this role uses a processing algorithm that allows it to efficiently merge data coming from sensors of the same type. Thus, this role cannot process data from differ types of sources.

Result Interface: Returns the results of the query to the GUI for displaying to the user via the protocol *User_Interact*. The results of the query are received from merger agents via the event *result(Query)*.

Time Monitor: Checks 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 via the *Monitor_Time* protocol and, if the constraint is violated, generates a negative trigger *failure*.

Accuracy Monitor: Behaves like the *Time Monitor* role but regarding the accuracy constraint.

5.4. Model capabilities

The next step into our process is to identify the capabilities and specify their actions on the environment through the Model Capabilities task. This task takes the role model previously defined as input.

During the Model Capabilities task, each capability is defined in a Capability Model [8]. Figure 5 shows an example of a capability model for the *User Interaction* capability and the *Coverage Processing* capability. Each capability performs some actions that are specified by their method signature. The capabilities identified for the AIS are listed below.

User Interaction: Used to interact with the user through a Graphical User Interface. As shown in Fig. 5, this capability provides actions to get a query from the user (getQuery) and to display the result of a query that has been executed (setQuery).

Coverage Processing: Used to compute the optimal set of sensors that has the maximum coverage of the area of interest and that can satisfy the time and accuracy constraints. This capability has two actions: *satisfyConstraints* and *findOptimal Coverage*. The action *satisfyConstraints* takes a query and a set of sensors and returns a list of

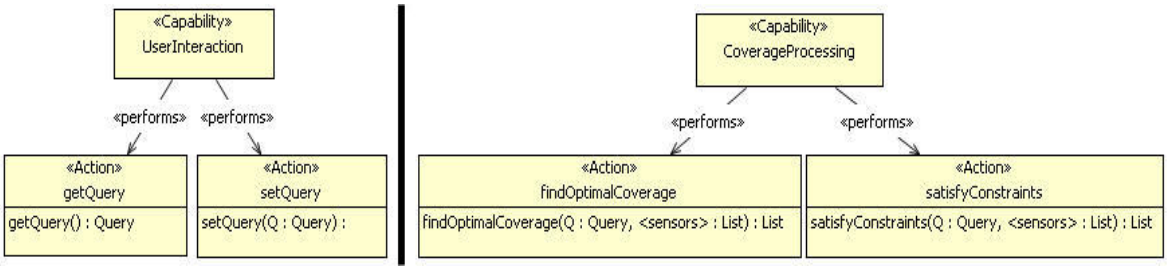

Fig. 5. AIS Capability Models for *UserInteraction* and *CoverageProcessing* capabilities.

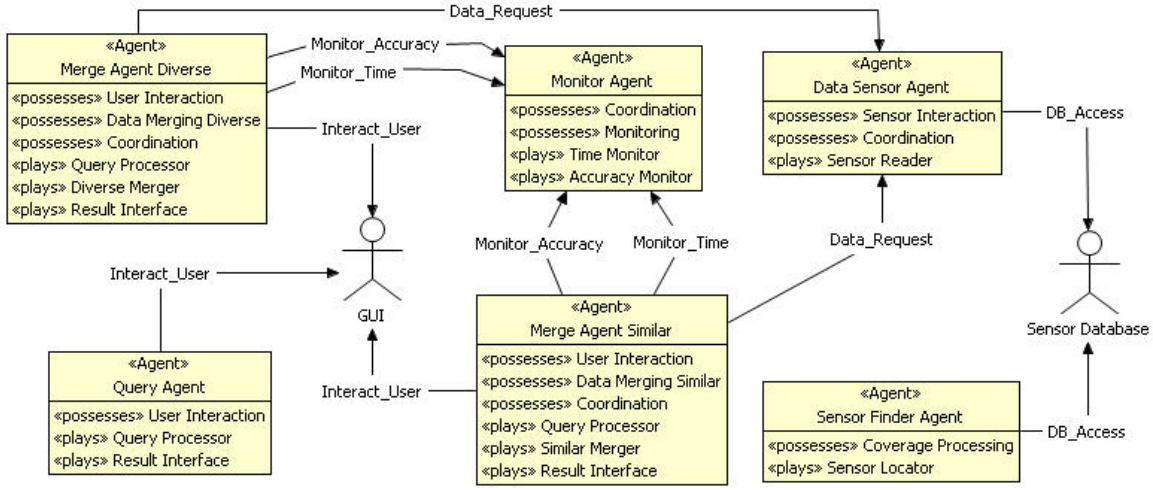

Fig. 6. AIS Agent Class Model.

sensors that satisfy the given time and accuracy constraints. The action *findOptimalCoverage* takes a query and a set of overlapping sensors and return a minimal set of sensors that has the maximum coverage of the area given in the query.

Sensor Interaction: Interacts with actual battlefield sensors. This capability provides an action to query a sensor given in parameter and retrieve its data.

Data Merging Diverse: Provides computational algorithms to merge data coming from diverse type of sensors.

Data Merging Similar: Provides fast computational algorithms to merge data coming from similar sources only.

Monitoring: Provides the ability to check the time and/or the accuracy constraints of the data. The information about accuracy and timeliness of the results are provided by the data sources (sensors).

Coordination: Provides 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 via this capability.

5.5. Model agent classes

After the goals, roles and capabilities have been identified, we need to populate our multiagent organization by creating various agents via the Model Agent Classes task. Agents represent the autonomic elements of the system.

We define a set of agent types capable of playing at least one role in the organization. Those agent types, along with the capabilities they posses, are captured in the Agent Class Model shown in Fig. 6. While assignments of agents to play roles are dynamically decided at runtime, the Agent Class Model also shows all possible roles that an agent could play. Hence, protocols between roles that were defined in the Role Model have to be mapped to the appropriate agents in the Agent Class Model.

During the actual instantiation of the organization, an agent of each type is created. For the Data Sensor Agent type, each sensor on the battlefield is associated with a unique Data Sensor Agent.

After the Agent Class Model is completed, all OMACS entities are defined. However, our process has two tasks remaining that are not related to the OMACS metamodel but are important in order to complete the low-level design of our system.

5.6. Model protocols

In the Model Protocols task, we specify all the protocols identified in the Role Model. Essentially, protocols involving roles are executed by the agents enacting those roles. The Protocol Models produced by this task are documented via AUML Interaction Diagrams [14]. Figure 7 shows two examples Protocol Models: the *Data_Request* protocol and the *Monitor_Time* protocol. The *Data_Request* protocol is used to request data from sensors on the battlefield. In this protocol, the Similar Merger role sends a *query* message to the Sensor Reader role which replies by sending the data in an *inform* message. This exchange is repeated as long as data is needed. The *Monitor_Time* protocol is used to request a validation of the results against the constraints specified in a query. In this protocol, the Similar Merger role sends a *monitor* message to the Monitor Time role that replies by sending either a *pass* or *fail* message, depending on whether the query meets the time constraints.

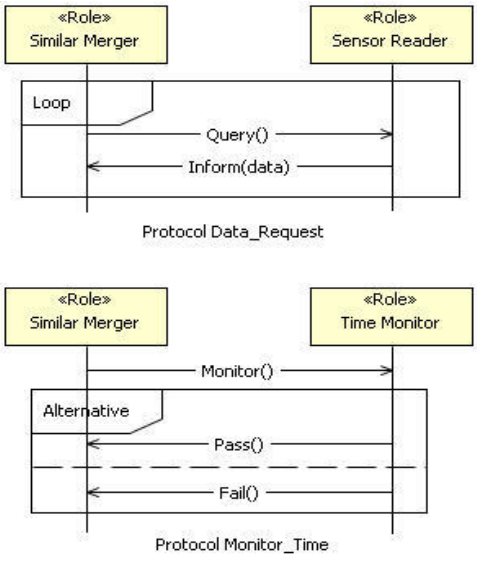

Fig. 7. Examples of AIS Protocol Models.

5.7. Model plans

For each role defined earlier in the process, we provide a plan that agents execute in order to play that role. The task Model Plans allows us to define plans based on the Role Model, the Capability Model and the Protocol Model. Essentially, a plan for a given role provides an algorithm that exhibits the behavior defined in the role. It uses the capabilities required by that role and is consistent with all the protocols defined for that role. Plans are captured in a Plan Model, which is essentially a finite state automaton.

As an example, in Fig. 8, we present a plan for the role *Sensors Locator*. When the plan starts, the agent sends a *getSensors* message to the *Sensor Database* to get a list of all sensors registered in a given area and moves to the *Wait* state. When the database returns the list of sensors requested, the plan moves into the *Find Sensors* state where it computes a new list of sensors covering the area of interest and satisfying the constraints specified in the query. If the list of sensors is empty, the plan moves to a *failure* state where the agent notifies the system that it fails to find sensors to answer the query. However, if the list is not empty, the plan moves to the *Notify Sensors* state in which a *found* event is generated for each sensor in the list. Once the events have been generated (i.e. the list is empty), a transition is made towards states *Notify Similar Merge* or *Notify Diverse Merger* depending on the type of sensors selected. In both states, an event is generated in order to activate a merger agent to merge data from the selected sensors.

6. Organization-based agent architecture

In this section, we present the Organization-based Agent Architecture (OBAA) of the AIS agents, which represent the autonomic elements of our system [16]. As Fig. 9 shows, an AIS agent typically consists of two components: the *Execution Component* (EC) and the *Control Component* (CC).

6.1. Execution component

The EC represents the *non-autonomic part* of the agent. Essentially, it corresponds to the application

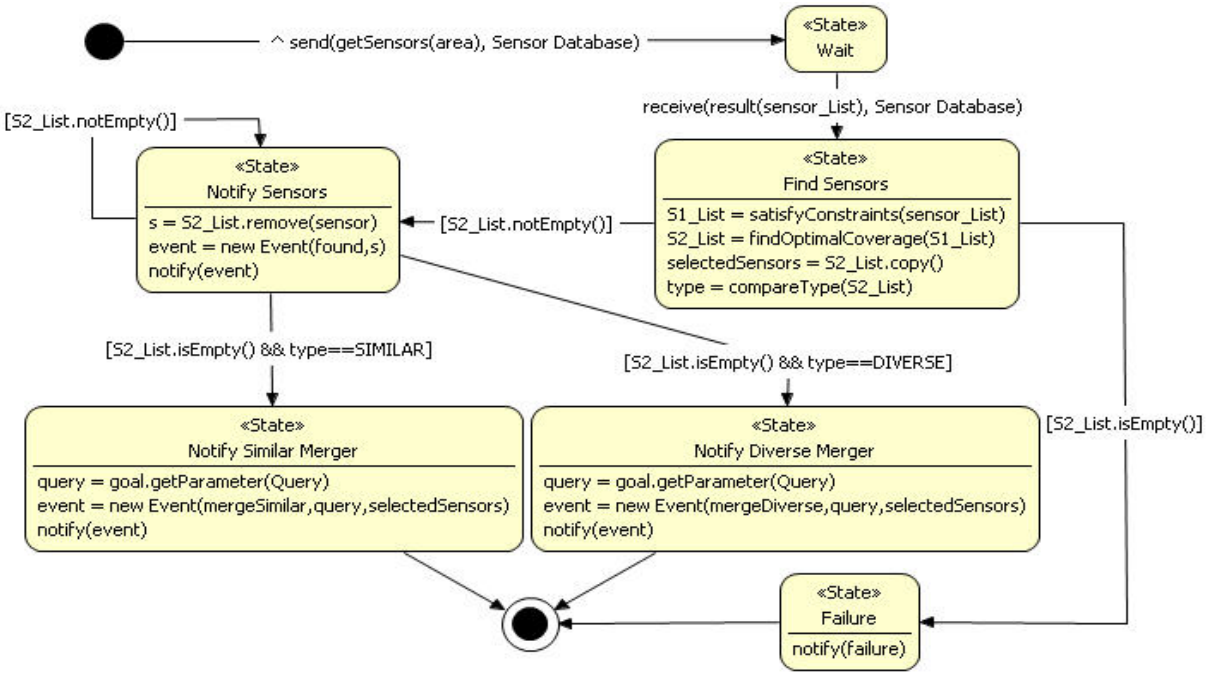

Fig. 8. AIS Plan Model for the role Sensor Locator.

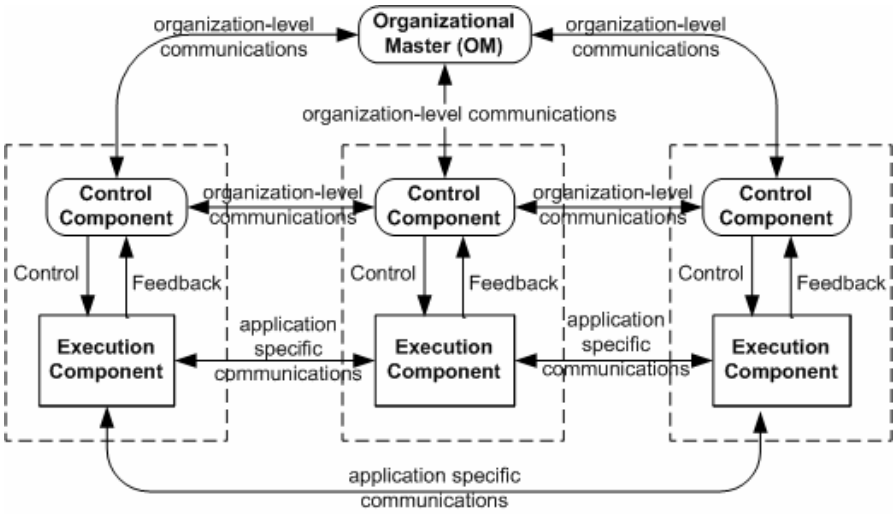

Fig. 9. Organization-based agent architecture.

specific part of the agent. It is notified by its CC about what role to play in the organization. Once it has been assigned a role, the EC plays that role according to a predefined plan provided at design time by either the role or the agent designer. During a role execution, an EC may need to coordinate with other ECs in order to exchange some data. Communication between ECs is done by message passing while the EC reports its status directly to its CC via method calls.

6.2. Control component

The CC represents the *autonomic part* of the agent. In general, the more sophisticated this component, the more autonomy the system displays. Hence, this component plays a very important role in achieving fully autonomous systems. By using the high-level specifications of the organization, the CC can reason about its own state and communicate it to others. This self-awareness of the autonomic element allows the whole system to be self-managed. Typically, the CC is an intelligent component in charge of all organization related tasks. Depending on design strategies, it can have a partial or total knowledge of the organization structure. We designed a fixed communication interface between the CC and the EC. This gives us the flexibility to plug several different CC designs into our application without having to modify other application-specific components (the ECs). Therefore, Control Components are generic and can be reused for any other autonomic applications as long as the communication interface is respected.

In general, a CC operates based on its knowledge and the information collected from other agents via their CCs. It can decide to reconfigure the organization by including or canceling goals in the organization, or by modifying the current assignments. This reconfiguration process can be distributed or centralized. A distributed reconfiguration would involve a deliberation process between all the CCs in order to reach a consensus about the next state of the organization. However, in our current implementation, we have opted for a centralized approach in which all CCs report to one particular CC that has all the knowledge to make appropriate decisions. To differentiate with other CCs, we call this particular CC the Organization Master (OM). Therefore, the OM possesses all the organizational knowledge and is in charge of all the organization-related tasks (Fig. 9). The OM reasons about the current state of the organization [29] and once it reaches a decision about a new configuration, it notifies all the CCs that are affected by this reconfiguration. This autonomous reasoning is solely based on the underlying OMACS architecture and results in a reconfiguration, which is fundamental in achieving the autonomic properties described in the following section.

7. Autonomic properties of the AIS

In this section, we present a scenario that exemplifies the autonomic behaviors of the AIS. To adapt to a variety of unpredictable situations, our AIS organization is able to detect changes in the performance of the overall organization (self-monitoring) and modify its structure accordingly (self-adjusting). Many changes occur within the environment; however, some changes occur within the organization itself (e.g., capability failure or goal completion). Hence, the AIS is not only aware of its environment but it is also aware of its own state (self-aware). These self-* properties of the AIS are facilitated by our use of OMACS, which provides all the necessary knowledge for a self-managing system.

7.1. AIS scenario

To demonstrate the AIS system, we use a simulated battlefield with sensors and enemy targets. In our battlefield simulator, there are five different types of vehicles that the system is trying to locate and identify: truck, halftrack, tank, artillery, and launcher.

For the specific scenario described in this paper, we have defined two types of sensors: ground sensors and airborne automatic target recognition (ATR) sensors. Sensors do not all provide the same accuracy in identifying and locating enemy targets and do not refresh their data at the same rate. The ground sensors have a fixed location and provide information about location and type of enemy vehicles with an accuracy of 75%. They are also capable of providing requested data within 5 minutes. The ATR sensors are obviously mobile and also very accurate, providing location and enemy vehicles type information with an accuracy of 95%. Unfortunately, ATR sensors are not very fast; they typically can only provide their information in 15 minutes. Therefore, the simulator can provide erroneous or outdated data that might not be of any interest for the commander. For this reason, the commander can specify some constraints for the query.

The screenshot in Fig. 10 shows the simulated battlefield along with the sensors represented by circles and enemy targets represented by small squares. There are four ground sensors (S1, S2, S3, S4) and one ATR sensor (S5). There are also five enemy vehicles. We assume that the system is only

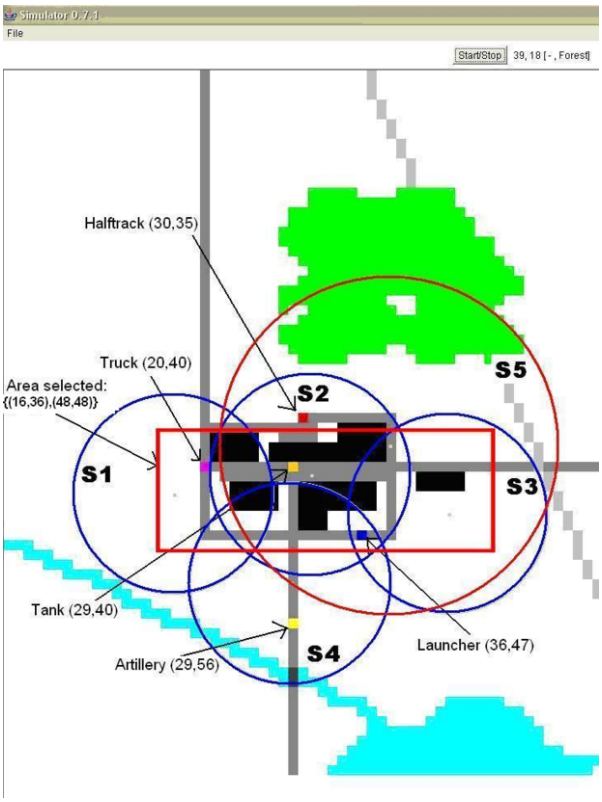

Fig. 10. Battlefield map: Circles S1 to S5 represent sensors and enemy vehicles are denoted by small squares.

trying to answer one persistent query and omit the query parameter for goals and triggers. The persistent query is: "*Show the location and type of all enemy vehicles in the selected area*" (the area selected is defined by a rectangle in Fig. 10).

In what follows, all goals, roles, capabilities and agent types are referred to by their id as presented in Table 1.

7.2. Initialization

At the initialization of the system, all the agents interested in participating in the organization register with the OM. For this scenario, we have created one agent for each agent type except for the agent type DSA for which one DSA agent has been created for each sensor in the battlefield. Agents are named after their types and DSA agents are numbered to match their sensor number. The organization only actively pursues the leaf goals as their completion implies the completion of their parent goal. At initialization, all the leaf goals that have no predecessors and do not require any triggers are inserted in G_a (the active goal set) and thereby pursued by the organization. Based on the goal model (Fig. 3), only goal G1 is active initially. Once G1 is active, the OM chooses the best role to achieve G1. R1 is chosen to achieve G1 because the pair ⟨G1,R1⟩ has the highest *achieves score*. In fact, R1 is the only role capable of achieving G1 as there is no other ⟨goal, role⟩ pair with a non-null *achieves score*. Then the organization chooses the QA agent to play R1. This choice is motivated by the fact that QA possess all the required capabilities to play R1. Hence, at initialization, the AIS organization assigns QA to play role R1 to achieve goal G1. Figure 11 shows the successive states of the organization after the occurrence of events. States contain the active goal

Table 1
Mapping between goals, roles, capabilities, agents and their id

	Name	Id
Goals	Process Query	G1
	Find Sensors	G2
	Read Sensor	G3
	Merge Diverse	G4
	Merge Similar	G5
	Return Result	G6
	Monitor Time Constraint	G7
	Monitor Accuracy Constraint	G8
Roles	Query Processor	R1
	Sensors Locator	R2
	Sensor Reader	R3
	Data Merger Diverse	R4
	Data Merger Similar	R5
	Result Interface	R6
	Time Monitor	R7
	Accuracy Monitor	R8
Capabilities	User Interaction	C1
	Coverage Processing	C2
	Sensor Interaction	C3
	Data Merging Diverse	C4
	Data Merging Similar	C5
	Monitoring	C6
	Coordination	C7
Agents	Query Agent	QA
	Sensor Finder Agent	SFA
	Data Sensor Agent	DSA
	Merger Agent Diverse	MAD
	Merger Agent Similar	MAS
	Monitor Agent	MON

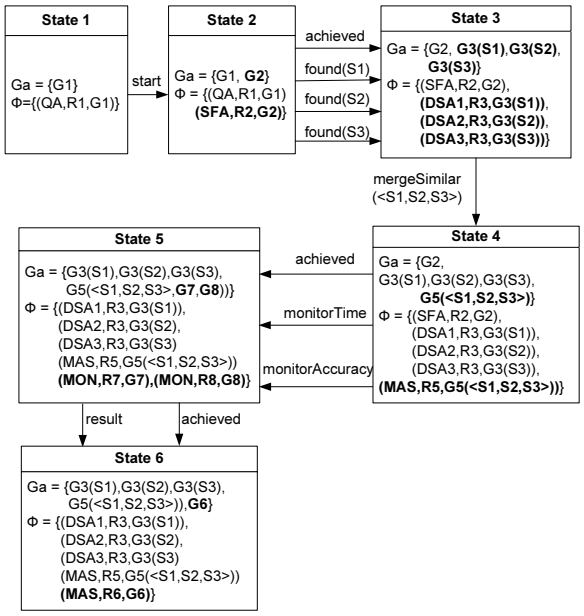

Fig. 11. Organization states during a normal execution. Changes from the previous state are in bold.

set (G_a) and the set of current assignments (Φ). Arrows between states represent events that occurred during the transition. The initial assignment we have just described corresponds to State 1 in Fig. 11.

Once the QA retrieves the query from the GUI, it triggers an event *start(Query)*. This trigger results in the activation of G2. Upon this activation, the system must reconfigure itself to achieve this new active goal. By taking the best role and agent to achieve goal G2, the SFA is assigned to play role R2 to achieve goal G2 (State 2 in Fig. 11).

When the query has been retrieved, the QA agent terminates by sending an *achieved* message to the OM, which causes the goal and its related assignment to be removed from G_a and Φ. Next, the SFA agent chooses sensors S1, S2, S3 for the query as those sensors maximize the area of interest coverage. The following events are then generated: *found(S1)*, *found(S2)*, *found(S3)*. Each *found* event triggers a parameterized goal G3 having the parameter of that event. In our case, goals G3(S1), G3(S2), and G3(S3) become active, which again requires a reconfiguration resulting in (DSA1,R3,G3(S1)), (DSA2,R3,G3(S2)), and (DSA3,R3,G3(S3)) being inserted into Φ (State 3 in Fig. 11).

As all the sensors found by the SFA agent are of the same type, an event *mergeSimilar(⟨S1,S2,S3⟩)* is triggered, which results in the activation of the parameterized goal G5(⟨S1,S2,S3⟩). After computing the best assignment, the assignment of the Merger Agent Similar (MAS) to play role R5 to achieve goal G5 is chosen (State 4 in Fig. 11).

Once all the events have been triggered, the SFA agent notifies the OM that it has successfully completed its role by sending an *achieved* message. At this point, the MAS agent starts getting data from the DSA agents and merges them to extract the necessary information. In order to have the results of the query checked against the constraints that may have been specified by the commander, the MAS agent triggers a *monitorTime* and *monitorAccuracy* event. These events result in the activation of G7 and G8.

Following a system reconfiguration, the assignment ⟨MON,R7,G7⟩ and ⟨MON,R8,G8⟩ are inserted into Φ. Note actually that the MON agent is playing R7 and R8 because it has all the required capabilities for both roles (State 5 in Fig. 11). As no constraints have been specified for the query, the constraints are trivially validated and the MON agent sends a message to the MAS agent, notifying it that it can proceed and then terminates.

When the results are ready, the MAS agent triggers a *result* event, which results in the activation of goal G6. The MAS agent, which has the capability to interact with the GUI, is selected to play role R6 to achieve goal G6 and sends the results to the GUI (State 6 in Fig. 11). When a query update is required, the MAS agent coordinates with the same DSA agents to get new data.

For this scenario, the results reported a coverage representing 100% of the area of interest and the system effectively detected all three targets in the selected area: Tank at 29,40, Truck at 20,40 and Launcher at 36,47.

Therefore, this scenario shows an important attribute of our autonomic system: self-configuration. The system is able to reconfigure itself when new goals appear in the organization. Every newly activated goal in the organization requires the AIS to take action in order to achieve this new goal. Our autonomic system is also capable of self-optimizing in the case of a goal completion. The achievement of a goal can free an agent to take on a new role and goal assignment. When this occurs, the organization may make new assignments in order to optimize the performance of the system.

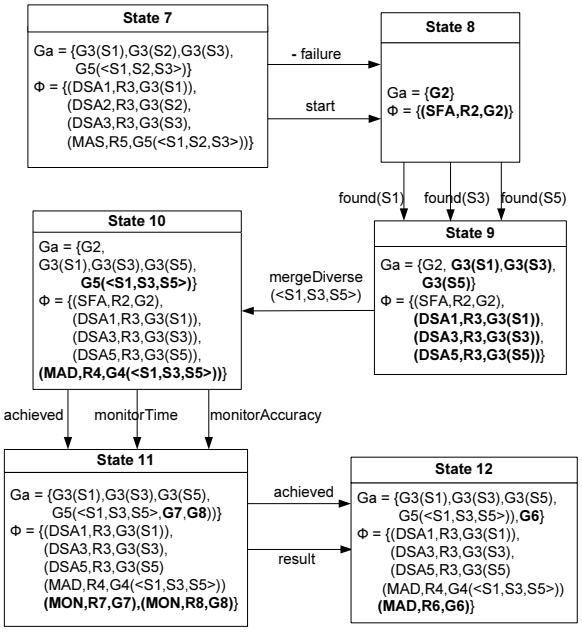

Fig. 12. Organization states before and after a sensor failure.

7.3. Sensors failure

The AIS simulator allows us to fail specific sensors. If we make S2 fail, the corresponding DSA agent (DSA2), which is the only agent capable of playing R3 to achieve G3(S2), can no longer achieve its goal. The organization states related to that sensor failure are presented in Fig. 12. The first state, State 7, corresponds to a state from the normal execution in which data are being refreshed. Whenever the DSA2 fails, the MAS agent, which was coordinating with the DSA2 agent to gather the data, interrupts its task and generates a negative trigger *failure* and a trigger *start(query)*. The negative trigger causes all the goals related to that query to be removed, resulting in the cancellation of all their current assignments. Thus, goals G3(S1), G3(S2), G3(S3), G5(⟨S1,S2,S3⟩) are all removed. The *start(query)* event causes the activation of a new instance of goal G2 that is achieved by the SFA agent playing role R2 (State 8 in Fig. 12).

Taking into account the loss of capability of the DSA2 agent, the SFA agent selects sensors S1, S3, S5 as the new optimal set of sensors for the query. The SFA agent then triggers the following events: *found(S1)*, *found(S3)*, *found(S5)* which result in the activation of goal G3(S1), G3(S3), and G3(S5) (State 9 in Fig. 12). As this set of sensors contains sensors of different types (S1, S3 are ground sensors whereas S5 is an ATR sensor), the SFA agent triggers an event *mergeDiverse(⟨S1,S3,S5⟩)*, which results in the activation of goal G4(⟨S1,S3,S5⟩). To achieve this new goal, the system chooses role R4, which is played by the Merger Agent Diverse (MAD). Then, the SFA sends an *achieved* message to the OM and terminates (State 10 in Fig. 12). The system then continues its execution as described in the previous section, except that the merger in charge of the query is now the MAD agent (States 11 and 12 in Fig. 12). As the coverage provided by the new set of sensors is also 100%, the AIS detects all the enemies in the area of interest: Tank at 29,40, Truck at 20,40 and Launcher at 36,47.

Therefore, by effectively detecting the sensor that failed and replacing it, the AIS has demonstrated its self-healing capability. The failure triggered a reconfiguration of the system to allow sensor reading tasks to be redistributed among all the DSA agents still operational and capable of covering the area of interest. Through this reconfiguration, the DSA5 agent has replaced the DSA2 agent that failed. In addition, during this scenario, an illustration of the AIS self-optimizing behavior has also occurred when the AIS organization has decided to replace the MAS agent, which was merging the data prior to the failure, by the MAD agent in order to insure a better performance. Hence, even though a loss of a sensor used to provide information for the query occurred, the AIS was able to reconfigure itself and maintain the flow of information without the intervention of the user. In a rigid system, the loss of a sensor would mean an irreversible loss of performance.

7.4. Maintenance goal failure

When specifying a query, the commander can stipulate some time or accuracy constraints that the query needs to satisfy. By default, a query does not have any constraints. For the remainder of our scenario, we assume that the commander has specified that the system should provide query results within 8 minutes.

We continue our example with the system currently running the query using the DSA1, DSA3, DSA5 and MAD agents as described in the previous section. Figure 13 shows the successive states of the organization for the events described in this section. State 13 in Fig. 13 corresponds to a state in which data are being refreshed after the sensor failure. After

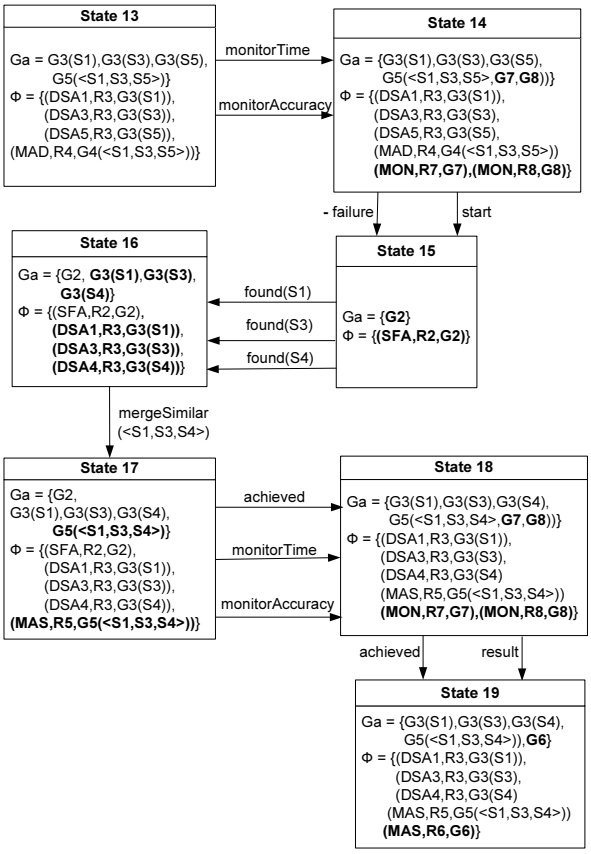

Fig. 13. Organization states before and after a maintenance goal failure.

the data from the battlefield are refreshed, the MAD agent triggers *monitorTime* and *monitorAccuracy* events in order to check the query against the time and accuracy constraints that have been updated. These events result in the activation of goals G7 and G8. During reconfiguration, the OM assigns the MON agent to play both R7 and R8 to achieve G7 and G8 (State 14 in Fig. 13).

Once the data is sent to the MON agent for checking the constraints, it generates a negative trigger *failure* because the query, as executed, does not meet the 8 minutes constraint. In fact, sensor S5 used in that query, which is an ATR sensor, can only provide data within 15 minutes. Therefore, the maintenance goal G7 fails. This negative trigger causes all the goals related to that query to be removed, resulting in the cancellation of all related assignments. Thus, goals G3(S1), G3(S3), G3(S5), G4(⟨S1,S3,S5⟩), G7, and G8 are all removed. The *failure* event is immediately followed by a *start* event generated by the MON agent. This event triggers the activation of goal G2 that is achieved by the SFA agent, playing role R2 (State 15 in Fig. 13).

Taking into account the time constraint for the query, the SFA agent selects 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 minutes. The SFA agent then triggers the following events: *found(S1)*, *found(S3)*, *found(S4)* which result in the activation of goal G3(S1), G3(S3), and G3(S4) (State 16 in Fig. 13). As the set of sensor contains only ground sensors, the SFA agent triggers an event *mergeSimilar(⟨S1,S3,S4⟩)*, which results in the activation of goal G5(⟨S1,S3,S4⟩). To achieve this new goal, the system chooses role R5 which is played by the MAS agent (State 17 in Fig. 13). The execution then continues as described in the normal execution (States 18 and 19 in Fig. 13).

Nevertheless, this reconfiguration has resulted in a lost of coverage and the Tank located at (29,40) can no longer be detected as the set of sensors selected in order to satisfy the time constraint covers only 87% of the area. The AIS detects the following enemies: Truck at 20,40 and Launcher at 36,47.

Hence, through this scenario, we can see how the AIS exhibits its self-protection and self-optimizing properties. The MON agent, in charge of monitoring the system for performance degradation, protects the system against undesirable behavior. When a violation of the query constraints was detected, the system decided to reconfigure itself by replacing the DSA5 agent by the DSA4 agent in order to insure the effectiveness of the query concerning the time constraint. This reconfiguration has also replaced the MAD by the MAS, which yields a better performance in merging data coming from different sensors.

8. Experimental results

To evaluate the performance of our autonomic information system, we performed 100 persistent queries executed 20 times each for different sensor failure rates, which we define as the ratio of sensor failures to the total number of sensors originally available on the battlefield. We are interested in evaluating the cost associated with the implementation of OMACS-based systems. For that, we compare our autonomic system with a non-autonomic system. This non-autonomic system is also designed with a simple O-MaSE compliant

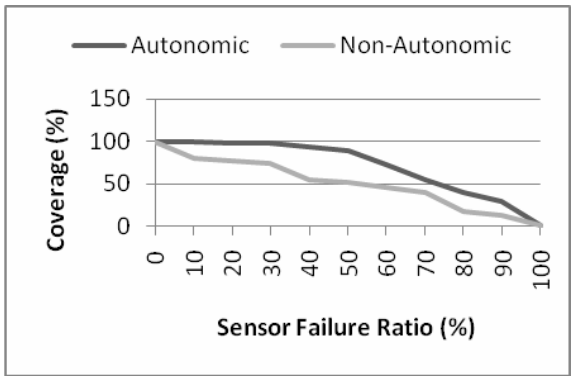

Fig. 14. Comparison of area coverage.

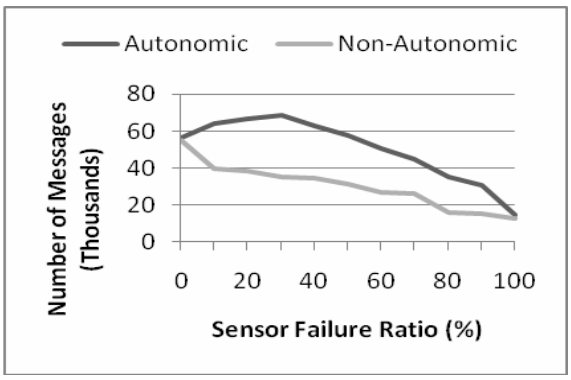

Fig. 15. Comparison of number of messages sent.

process in which assignments are predefined at design time by directly assigning agents to achieve goals. Hence, this process is not OMACS compliant.

Our first sets of experiments attempted to evaluate the system robustness. We created a battlefield containing 40 sensors at fixed locations and 100 merger agents. The sensors were placed such that each point in the area of interest was covered by at least two sensors. For each run, we submitted 100 persistent queries and the experiment ran until each query had been executed 20 times. Experiments were run on both our autonomic system and a non-autonomic system that we simulated. Basically, once the queries were submitted, both systems chose the optimal set of Data Sensor Agent (DSA) to provide maximum coverage. However, when a DSA failed, the non-autonomic system ignored the failure and continued to provide results whereas the autonomic system reconfigured. The robustness of the system was measured in terms of the average coverage obtained at the end of the experiment for various sensor failure rates.

As expected, the results, as shown in Fig. 14, clearly indicate that for any failure rate, the autonomic system was significantly more robust. Observe that after 30% failure rate, the autonomic system was still able to provide a coverage of 100% whereas this coverage had dropped to 75% for the non-autonomic system. However, when too many sensors were lost (around 90% failure rate), the coverage provided by both system was very close as the autonomic system could not find any other sensors to replace the failed sensors. On average, the autonomic system achieved 25% more coverage than its counterpart.

In our second set of experiments, we were interested in characterizing the cost of reconfiguration in terms of the number of messages sent by the agents. As in the first experiment, we used 40 data sensor agents and 100 merger agents trying to answer 100 persistent queries. The results in Fig. 15 show that for the non-autonomic system, the number of messages decreased as the failure rate increased. This is due to the fact that as more sensors fail, the system interacts with less data sensor agents resulting in fewer messages. For the autonomic system, we expected a monotonically increasing function due to the messages involved in an increasing number of reconfigurations. However, the number of messages only increased until we reached a failure rate of 30% and then decreased. These results suggest that before the 30% failure rate, the number of reconfiguration messages was larger than the number of messages generated by sensors prior to their failure. As more sensors were lost (30% failure rate and larger), the increase in messages due to reconfiguration was less than the loss of messages caused by the reduction of sensor interactions, globally resulting in less messages sent in the system. Overall, the autonomic system generated approximately 50% more messages.

Our final set of experiments attempted to confirm the influence of the number of data sensor agents on system performance. We ran the previous experiments with 3 different numbers of data sensor agents: 20, 30 and 80. For each set, we measured the number of messages sent at different failure rates. The results are shown in Fig. 16. We note that there is a direct correlation between the number of data sensor agents and the number of message sent. However, as the number of data sensor agents doubles, the number of messages only increases by a constant factor. This is an important result that ensures an effective scalability of our system.

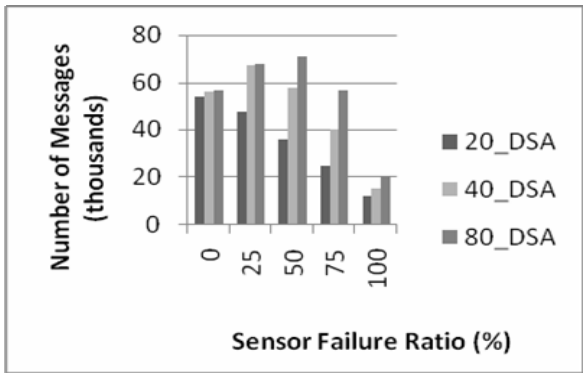

Fig. 16. Impact of the number of data sensor agents used.

9. Conclusions and future work

We have described a comprehensive multiagent approach for building an Autonomic Systems. Our approach takes advantage of OMACS, which fits well in the autonomic computing perspective and allows autonomic systems to achieve many self-* properties. Furthermore, we have designed a customized multiagent development process from the O-MaSE process framework. Our process fits with OMACS organizational concepts and provides means to create a rigorous process to design autonomic systems. In addition, we have described our OBAA agent architecture that separates general autonomic reasoning from application specific tasks. This allows applications to reuse various autonomic reasoning strategies without changing their implementation.

Moreover, through a specific scenario, we have illustrated the self-configuring, self-optimizing, self-healing and self-protecting properties of our exemplar system, the autonomic information system. As a result of those properties, the system is able reason about system goals, recover from failures, recognize and prevent undesirable behaviors and optimize performance without disrupting the flow of information and requiring the intervention of the user.

Finally, we have shown experimentally that our autonomic information system was more robust than a non-autonomic version, achieving on average 25% more coverage after some sensors had failed. Then we have measured that this increase in robustness comes with an increase number of messages generated due to reconfiguration. Our experiments have demonstrated that the autonomic system generated 50% more messages than the non-autonomic one. Lastly, we have shown that, despite the cost of reconfiguration, our autonomic system exhibits a good scalability.

In the future work, we want to evaluate our proposed design process by comparing it with other autonomic methodologies. In addition, we are investigating ways to allow a distributed reconfiguration, as oppose to the centralized approach used in this paper. This would allow all the agents to participate in the reconfiguration process, resulting in a more robust autonomic system.

We are also working on extending the aT^3 tool to support design metrics, which allows the designer to explore alternative designs and verify its design for robustness, flexibility and efficiency thorough the entire development lifecycle.

Finally, as systems become larger and more complex, we are also looking into ways to simplify the design of such systems by allowing reuse of simpler and smaller organizations through some composition mechanisms. This would allow the designer to focus on smaller organizations and would facilitate the verification process.

References

[1] J. Appavoo, et al., *Enabling autonomic behavior in systems software with hot swapping.* IBM SYSTEMS JOURNAL, 2003. **42**(1): p. 60–76.

[2] J.P. Bigus, et al., *ABLE: A toolkit for building multiagent autonomic systems.* IBM Journal of Research and Development, 2002. **41**(3): p. 350.

[3] P.M. Blau and W.R. Scott, *Formal organizations*. 1966, Routledge & Kegan Paul.

[4] S. Brinkkemper, *Method engineering: engineering of information systems development methods and tools.* Information and Software Technology, 1996. **38**(4): p. 275–280.

[5] D.W. Bustard, et al., *Autonomic system design based on the integrated use of SSM and VSM.* Artificial Intelligence Review, 2006. **25**(4): p. 313–327.

[6] T. De Wolf and T. Holvoet. *Towards autonomic computing: agent-based modelling, dynamical systems analysis, and decentralised control*. In *Industrial Informatics, 2003. INDIN 2003. Proceedings. IEEE International Conference on*. 2003.

[7] S. DeLoach, W. Oyenan, and E. Matson, *A capabilities-based model for adaptive organizations.* Autonomous Agents and Multi-Agent Systems, 2008. **16**(1): p. 13–56.

[8] S. DeLoach and J. Valenzuela, *An Agent-Environment Interaction Model.* In *Agent-Oriented Software Engineering VII*. 2007. p. 1–18.

[9] J. Ferber, et al. *Organization models and behavioural requirements specification for multi-agent systems*. In *MultiAgent Systems, 2000. Proceedings. Fourth International Conference on*. 2000.

[10] A.G. Ganek and T.A. Corbi, *The dawning of the autonomic computing era.* IBM SYSTEMS JOURNAL, 2003. **42**(1): p. 6.

[11] J.C. Garcia-Ojeda, S.A. DeLoach, and Robby. *agentTool Process Editor: Supporting the Design of Tailored Agent-based Processes*. In *Proceedings of the 24th Annual ACM Symposium on Applied Computing* 2009. Honolulu, Hawaii.

[12] J.C. Garcia-Ojeda, et al. *O-MaSE: A Customizable Approach to Developing Multiagent Development Processes*. In *The 8th International Workshop on Agent Oriented Software Engineering* 2007.

[13] P. Horn, *Autonomic Computing: IBM's Perspective on the State of Information Technology.* IBM TJ Watson Labs, NY, 15th October, 2001.

[14] M.P. Huget and J. Odell. *Representing agent interaction protocols with agent UML*. In *Autonomous Agents and Multiagent Systems, 2004. AAMAS 2004. Proceedings of the Third International Joint Conference on*. 2004.

[15] N.R. Jennings, *On agent-based software engineering.* Artificial Intelligence, 2000. **117**(2): p. 277–296.

[16] J.O. Kephart and D.M. Chess, *The vision of autonomic computing.* Computer, 2003. **36**(1): p. 41–50.

[17] A. Lapouchnian, et al., *Towards requirements-driven autonomic systems design.* In *Proceedings of the 2005 workshop on Design and evolution of autonomic application software*. 2005, ACM: St. Louis, Missouri.

[18] E. Matson and S. DeLoach. *Capability in Organization Based Multi-agent Systems*. In *Proceedings of the Intelligent and Computer Systems (IS'03) Conference*. 2003.

[19] M. Miller, *A Goal Model for Dynamic Systems*. 2007, Kansas State University: Manhattan.

[20] G. Pour. *Expanding the Possibilities for Enterprise Computing: Multi-Agent Autonomic Architectures*. In *Enterprise Distributed Object Computing Conference Workshops, 2006. EDOCW'06. 10th IEEE International*. 2006.

[21] S.J. Russel and P. Norvig, *Artificial intelligence*. 2003, Prentice-Hall.

[22] K. Sanjeev and P.R. Cohen, *Towards a fault-tolerant multi-agent system architecture.* In *Proceedings of the fourth international conference on Autonomous agents*. 2000, ACM: Barcelona, Spain.

[23] M. Schanne, T. Gelhausen, and W.F. Tichy. *Adding autonomic functionality to object-oriented applications*. In *Database and Expert Systems Applications, 2003. Proceedings. 14th International Workshop on*. 2003.

[24] R. Sterritt. *Towards autonomic computing: effective event management*. In *Software Engineering Workshop, 2002. Proceedings. 27th Annual NASA Goddard/IEEE*. 2002.

[25] R. Sterritt and D. Bustard. *Towards an autonomic computing environment*. In *Database and Expert Systems Applications, 2003. Proceedings. 14th International Workshop on*. 2003.

[26] G. Tesauro, et al. *A multi-agent systems approach to autonomic computing*. In *Autonomous Agents and Multiagent Systems, 2004. AAMAS 2004. Proceedings of the Third International Joint Conference on*. 2004.

[27] A. van Lamsweerde, R. Darimont, and E. Letier, *Managing conflicts in goal-driven requirements engineering.* Software Engineering, IEEE Transactions on, 1998. **24**(11): p. 908–926.

[28] S.R. White, et al. *An architectural approach to autonomic computing*. In *Autonomic Computing, 2004. Proceedings. International Conference on*. 2004.

[29] C. Zhong, *An Investigation of Reorganization Algorithms*. 2006, Kansas State University.

[30] C. Zhong and S.A. DeLoach, *An Investigation of Reorganization Algorithms.* In *International Conference on Artificial Intelligence (IC-AI'2006)*. 2006, CSREA Press: Las Vegas, Nevada.