Applying a UML-based Agent Modeling Language to the Autonomic Computing Domain

Ivan Trencansky Radovan Cervenka Dominic Greenwood
Whitestein Technologies, Paneska 28, SK-811 03 Bratislava, Slovakia
{itr, rce, dgr}@whitestein.com

Abstract

As agent technology practitioners, some time ago we determined to develop an extension to UML 2.0 that addressed our specific needs, such as modeling autonomicity, proactivity and role-based behavior. We called this extension the Agent Modeling Language (AML) and have recently published the metamodel and specification for public use. In a recent project, we realized that AML could also be applied to the domain of autonomic computing and so decided to publish some of our findings in this paper. AML can be directly used by designers of autonomous and autonomic computing systems to visually model their architectures and behaviors. Herein we provide an overview of the scope, approach taken, the specific language structure and optional extensibility. The core modeling constructs of AML are explained using a series of didactic examples describing the IBM Unity architecture, an well-grounded exemplar of an autonomic system. We thus focus on the features of AML that differentiate it from UML 2.0 with a specific focus on those aspects that support the autonomic principles of self-healing and survivability.

Categories and Subject Descriptors D.2.1 [*Software Engineering*]: Requirements/Specification—languages, tools; I.2.11 [*Artificial Intelligence*]: Distributed Artificial Intelligence—languages and structures, intelligent agents, multiagent systems

General Terms Languages, Design, Reliability

Keywords Autonomic computing, autonomous system, agent, multi-agent system, modeling language, agent-oriented software engineering, AML

1. Introduction

By definition an autonomic system consists of a set of computational elements that are capable of managing themselves and thereby the system within which they are embedded, given some objectives or policies provided by administrators. Drawn from the biological metaphor of the mammalian autonomic nervous system [13], autonomic computing systems are comprised of multiple interacting and self-governing components, autonomic elements, that can often themselves contain embedded populations of self-governing components.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
OOPSLA'06 October 22–26, 2006, Portland, Oregon, USA.
Copyright © 2006 ACM 1-59593-491-X/06/0010. . . $5.00.

While the implementation of autonomic systems can be achieved by many means, the control and management aspects of individual autonomic elements and complete autonomic systems can, and often are, accomplished by means of software agents. The value and relevance of agents in this role has been discussed in several previous works including [30, 13, 25, 1, 14, 15] and will thus be taken as an assumption for the purposes of this paper.

Our approach is focused on examining the utility of an agent-oriented modeling language when applied to problems in the autonomic computing domain. Agent-based modeling is a powerful and flexible means of modeling complex, distributed, interconnected and interacting systems and as self-* principles can be enacted through the behaviors of autonomous agents, we expect that a well-specified modeling language should be intrinsically capable of capturing them both visually and logically.

In this paper we therefore demonstrate the suitability of the Agent Modeling Language [27, 6], created by the authors as a comprehensive and versatile extension to UML 2.0 for modeling agent-based systems. Throughout the course of this paper we will demonstrate how the core features of AML can be applied in this respect, although due to space limitations a comprehensive description of AML abstract syntax, semantics, and notation is not provided (for details see [5]).

AML is a semi-formal visual modeling language for specifying, modeling and documenting systems that incorporate concepts drawn from multi-agent systems theory. It is designed to address the specific qualities offered by multi-agent systems (MAS) that are difficult, or impossible, to model with more traditional modeling languages such as UML 2.0. AML can also be applied to other domains such as business systems, social systems, robotics, and of course, autonomic systems. In general, AML can be used whenever it is suitable or useful to build models that (1) consist of a number of autonomous, concurrent and/or asynchronous (possibly proactive) entities, (2) comprise entities that are able to observe and/or interact with their environment, (3) make use of complex interactions and aggregated services, (4) employ social structures, and (5) capture mental characteristics of systems and/or their parts.

Although not alone in this field (others include Gaia [31], AALAADIN [11], AOR [29], AUML [2, 16, 17]), INGENIAS [21], MaSE [9], MAS CommonKADS [12]), MESSAGE [10], OPM/MAS [24], PASSI [8, 7], Prometheus [20], TAO [23], TROPOS [3]) we consider AML to be one of the most advanced and comprehensive languages available for this purpose[1].

From the outset, the development of the language has been driven by the extant need for a ready-to-use, versatile, and highly expressive modeling language suitable for development of com-

[1] The paper cannot and also does not intend to provide a detailed analysis of the alternative approaches to model MAS and how they compare to AML. Such a comparative analysis can be found in [4].

mercial software solutions based on multi-agent technologies. In order to achieve this, AML has been designed to address and satisfy the most significant deficiencies of the currently available MAS oriented modeling languages, which often are:

- insufficiently documented and/or specified,
- using proprietary and/or non-intuitive modeling constructs,
- aimed at modeling only a limited set of MAS aspects,
- applicable only to a specific theory, application domain, MAS architecture, or technology,
- mutually incompatible in terms of concepts, metamodel, and notation, or
- insufficiently supported by Computer-Aided Software Engineering (CASE) tools.

To overcome the above, AML (1) incorporates and unifies the most significant concepts from the broadest set[2] of existing multi-agent theories, abstract models, modeling languages, and methodologies, (2) extending them where necessary to account for aspects thus far covered insufficiently, inappropriately or not at all, and (3) assembling the entirety into a consistent framework specified (to the maximum possible extent) as a *conservative extension of UML*[3] (for details on AML definition see Section 2).

This gives AML the features which distinguish it from the others as the language which:

- is built on proven technical foundations,
- integrates best practices from agent-oriented software engineering (AOSE) and object-oriented software engineering (OOSE) domains,
- is well specified and documented,
- is internally consistent from the conceptual, semantic and syntactic perspectives,
- is versatile and easy to extend,
- is independent of any particular theory, software development process or implementation environment, and
- is supported by CASE tools.

AML supports all of the typical architectural and behavioral concepts associated with multi-agent systems, including MAS entities, communicative interactions, observations and effecting interactions, behavior abstraction and decomposition, social aspects, goal-oriented reasoning, services, ontologies, deployment and mobility. As software agents are capable of assuming the task of operational management and coordination of autonomic elements, it is therefore a logical assumption that AML can be applied to model these and, by extension, the broader characteristics of autonomic systems.

During the course of this paper we introduce many of these key elements of AML and demonstrate their direct relevance to modeling aspects of autonomic systems through a series of associated models, each addressing an aspect of the IBM Unity architecture [25]. We have selected this example due to its level of recognition with domain practitioners and as it offers the opportunity to demonstrate how the application of AML can clarify and enhance system design. As such, AML offers a means of accurately representing models of autonomic systems and applications in a visually and logically consistent manner.

[2] A list of sources can be found in a previous paper [27].

[3] A conservative extension of UML is a strict extension of UML which retains the standard UML semantics in unaltered form[28].

As a key component of this, we demonstrate the use of AML to model how autonomic elements of the Unity architecture collaborate to bring about self-healing recovery from policy repository failures [25]. A similar approach to the use of software agents for managing clusters of autonomic computational elements is described in Baldassari et al. [1].

The remainder of this paper is structured as follows: Section 2 presents a concise definition of the AML language and the available extensibility mechanisms. Section 3 explains the relevant fundamental AML entities, their features and how they can be used to model autonomic elements. Sections 4, 5, 6, 7, 8 and 9 then detail an AML approach to modeling various aspects of the selected exemplar autonomic system.

2. AML language definition

AML is built upon the Unified Modeling Language (UML) 2.0 Superstructure [19], augmenting it with several new modeling concepts appropriate for capturing the typical features of multi-agent systems (see Fig. 1).

The main advantages of this approach are (1) reuse of the well-defined, commonly used concepts of UML, (2) reuse of metamodel extensions and UML profiles for specifying and extending UML-based languages, and (3) ease of incorporation into existing UML-based CASE tools.

The abstract syntax, semantics and notation of the language are defined at the *AML Metamodel and Notation* level. The *AML Metamodel* is further structured into two main packages: *AML Kernel* and *UML Extension for AML.*

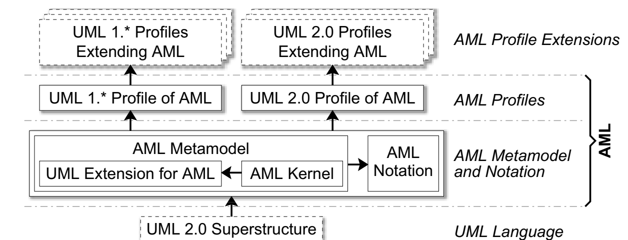

Figure 1. Levels of AML definition.

The *AML Kernel* is a conservative[4] extension of UML 2.0, comprising specification of all the AML modeling elements. It is logically structured into several packages, each of which contains specification of modeling elements dedicated for modeling specific aspect of MAS.

The *UML Extension for AML* package adds some meta-properties and structural constraints to the standard UML elements. It is thus a non-conservative extension of UML, and therefore an optional part of the language. However, the extensions contained within are simple and can be easily implemented in most existing UML-based CASE tools.

Upon the AML Metamodel and Notation two UML profiles of AML are specified: *UML 1.* Profile for AML* (based on UML 1.*) and *UML 2.0 Profile for AML* (based on UML 2.0). The primary objective of these profiles is to enable implementation of AML into existing UML 1.* and UML 2.0 based CASE tools, respectively.

2.1 Extensibility of AML

AML is designed to encompass a broad set of relevant theories and modeling approaches, it being essentially impossible to cover

[4] A conservative extension of UML is an extension of UML which retains the standard UML semantics in unaltered form [28].

all inclusively. In those cases where AML is insufficient, several mechanisms can be used to extend or customize it as required:

Metamodel extension offers first-class extensibility (as defined by MOF [18]) of the AML metamodel and notation.

AML profile extension offers the possibility to adapt AML for a given domain, platform or development method by means of UML Profiles, without the need to modify the underlying AML Metamodel and Notation.

Concrete model extension allows to employ alternative MAS modeling approaches as complementary specifications to the AML model.

3. AML modeling of autonomic elements

In general, *entities* are objects that can exist independently of others. In order to maximize reuse and comprehensibility of the metamodel AML defines several auxiliary abstract metamodeling concepts called *semi-entities* and their types. Semi-entity types are specialized UML classes used to specify coherent set of features, logically grouped according to particular aspects of MASs. They are used to specify features of other types of modeling elements.

3.1 AML semi-entities

AML defines the following semi-entities:

Behaviored semi-entities represent elements which can own capabilities, observe and/or effect their environment by means of perceptors and effectors, provide and use services, and can be decomposed into behavior fragments.

Socialized semi-entities represent elements which can form societies, can participate in social relationships and can own social properties.

Mental semi-entities represent elements which can be characterized in terms of their mental attitudes, e.g. which information they believe in, what are their objectives, needs, motivations, desires, what goal(s) they are committed to, when and how a particular goal is to be achieved, which plan to execute, etc.

3.2 AML fundamental entities

The fundamental entities that compose MASs are: agents, resources, and environments. AML therefore defines three modeling concepts, which can be used to model the above mentioned fundamental entities at both type and instance levels:

Agent type is used to specify the type of agents, i.e. self contained entities that are capable of interactions, observations and autonomous behavior within their environment.

Resource type is used to model the type of resources within the system, i.e. physical or informational entities with which the main concern is their availability (in terms of its quantity, access rights, conditions of usage/consumption, etc.).

Environment type is used to model the type of a system's inner environment[5], i.e. the logical or physical surroundings of entities which provide conditions under which the entities exist and function.

In AML, all the aforementioned entity types are specialized UML classes, and thus can utilize all the features defined for UML classes, i.e. can be instantiated, can own structural and behavioral features, behaviors, can be structured into parts and ports, participate in interactions, can participate in various kinds of relationships (e.g. associations, generalizations, dependencies), etc. The instances of the entity types (called entities) can be modeled by means of UML instance specifications classified according to the corresponding types.

[5] *Inner environment* is that part of an entity's environment that is contained within the boundaries of the system.

Furthermore, all the AML fundamental entity types inherit features of behaviored semi-entities, and in addition to these, agent and environment types are also socialized and mental semi-entities.

3.3 Modeling autonomic elements

Fundamental patterns of modeling the core architectural concepts of autonomic systems are shown in Fig. 2. The figure identifies three possible means of modeling an autonomic element with AML, drawn from the basic architecture of an autonomic element defined in [13]. Fig. 2(a) shows the basic model of an agent type representing an autonomic element where it is not necessary to explicitly identify an associated managed entity. Fig. 2(b) shows the case of an agent type acting as an autonomic manager, managing zero or more managed entities represented by AML resource types. The autonomic element in this instance is represented as an AML environment type[6]. Fig. 2(c) is similar to the previous case with the exception that the managed element is represented by an AML entity role type (see Section 4.3), which indicates that the role of the managed element can be played by zero or more entities. By extension this implies that the role can also be played by another autonomic element or other organization of autonomic elements, allowing composite hierarchies of embedded elements to be modeled. Finally, Fig. 2(d) is an alternative notation of Fig. 2(c) with identical semantics.

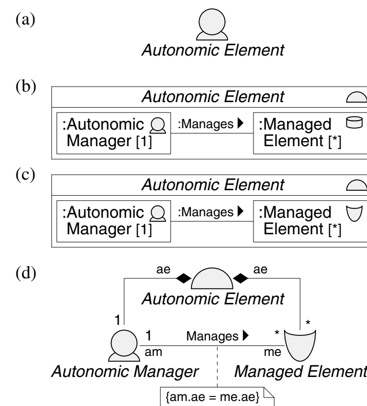

Figure 2. Modeling autonomic elements.

4. Modeling social aspects

As with MASs, autonomic systems are commonly perceived as systems comprised of a number of autonomous entities, situated in a common environment, and interacting with each other in order that the desired functionality and properties of the systems can emerge. These properties are not always derivable or representable solely on the basis of properties and capabilities of individual entities, but are usually given also by their mutual relationships, interactions, coordination mechanisms, social attitudes, etc. Such aspects are in MAS theory commonly referred to as *social aspects*.

From this perspective the following aspects of MAS are commonly considered in models (for details see [6]):

- *Social structure* concerning mainly with the identification of societies which can evolve within the system, specification of their properties, structure, identification of comprised roles,

[6] Specialized AML environment types called organization unit types can be used instead as explained later in Section 4.

individual entities that can participate in such societies, what roles they can play, their mutual relationships, etc.

- *Social behavior* covering such phenomena as (1) social dynamics (i.e. temporal relationships and causality of social events such as the formation and abolition of societies, the entrance and withdrawal of an entity to or from a society, acquisition, disposal and change of a role played by an entity, modification of properties of a society or its members, etc.), (2) norms (i.e. rules or standards of behavior shared by members of a society), (3) social interactions (i.e. how individuals and/or societies interact with others in order to exchange information, coordinate their activities, etc.), and (4) social activities of individual entities and societies (e.g. how they change their attitudes, roles they play, social relationships), etc.
- *Social attitudes* addressing the individual and/or common tendencies (usually expressed in terms of motivations, policies, goals, intentions, beliefs, commitments, etc.) to anything of a social value.

In this section the focus is on modeling the social structure of multi-agent systems. Our evidence is that the same concepts can be used to model architectural features of autonomic systems. AML modeling constructs which can be used to model social behavior and social attitudes are outlined in the subsequent sections.

To accommodate the special needs of modeling social structure, AML utilizes the concepts of: organization units, social relationships, entity roles, and role properties.

4.1 Organization units

Organization unit types are specialized environment types, and thus inherit features of behaviored, socialized and mental semi-entity types. They are used to specify the type of societies that can evolve within the system from both external and internal perspectives.

From an *external perspective*, organization units represent coherent autonomous entities, which can be characterized in terms of their mental and social attitudes, can perform behavior, participate in different kinds of (social) relationships, can observe and interact with their environment, offer and use services, play roles, etc. Their properties and behavior are both (1) emergent properties and behavior of all their constituents, their mutual relationships, observations and interactions, and (2) the features and behavior of organization units themselves.

For modeling organization units from external perspectives, in addition to features defined for UML classes (structural and behavioral features, owned behaviors, relationships, etc.), also all the features of behaviored, socialized, and mental semi-entities can be utilized.

From an *internal perspective*, organization units are types of environment that specify the social arrangements of entities in terms of structures, interactions, roles, constraints, norms, etc.

For this purpose organization unit types usually utilize the possibilities inherited from UML structured classifier, and model their internal structure by contained parts and connectors, in combination with entity role types used as types of the parts.

An example of an organization unit used to model the Unity clustering of policy repositories in the context of self-healing behavior, as described in [25], is shown in Fig. 4 and elaborated in Sect. 5. Another example, Fig. 3, utilizes an organization unit to model a Unity autonomic element *application environment* (for detailed description of the example see Sect. 4.3).

4.2 Social relationships

Social relationship is a particular type of connection between social entities related to or having dealings with one another. For modeling such relationships, AML defines a special type of UML property, called social property. The social property can be used either in the form of an owned social attribute, or as the end of a social association, and can specify its social role kind[7].

The Fig. 4 shows that `Resource Arbiter` is superordinate (indicated by the filled triangles at the social association ends) to `Cluster`, `Cluster Element` and `Sentinel` (indicated by the hollow triangles at the social association ends), that `Sentinel` is superordinate to `Cluster Element`, and that `Cluster Elements` are peers to each other (indicated by the half-filled triangles placed at both ends of the social association).

4.3 Roles and role properties

Roles are used to define a normative behavioral repertoire of entities, and thus provide the basic building blocks of MAS societies. For modeling roles, AML provides *entity role type*, a specialized behaviored, socialized and mental semi-entity type. Entity role types are used to model abstractions of coherent sets of features, capabilities, behaviors, observations, relationships, participation in interactions, and services offered or required by entities participating in a particular context. Each entity role type should be realized by a specific implementation possessed by an entity that can play that entity role type. An instance of an entity role type is called entity role and exists only while some behavioral entity plays it.

For modeling the ability of an entity to play an entity role type, AML provides *role properties*. Role property is a specialized UML property, used to specify that an instance of its owner (i.e. a behavioral entity) can play one or several roles of a particular entity role type. The role property can be used either in the form of a role attribute or as the end of a play association.

One entity can at each time play several entity roles. These entity roles can be of the same as well as of different types. The multiplicity defined for a role property constrains the number of entity roles of given type the particular entity can play concurrently. Additional constraints which govern the playing of entity roles can be specified by UML constraints.

To allow explicit manipulation of entity roles in UML activities and state machines, AML defines a set of actions for entity role creation and disposal, particularly the *create role* and *dispose role* actions.

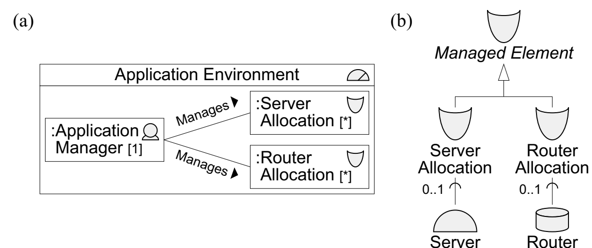

Figure 3. Application Environment specification.

An example in Fig. 3(a) demonstrates the application of the pattern from Fig. 2(c) to model a Unity element. The Unity application environment, as described in [25], is modeled as an organization unit `Application Environment` consisting of one autonomic manager (modeled by an agent of type `Application Manager`) and two types of managed elements (modeled by entity role types `Server Allocation` and `Router Allocation`. The possibility of an allocation of a `Server` autonomic element (modeled as an AML environment) to an application environment is

[7] AML predefines *peer*, *subordinate* and *superordinate* social role kinds, but this set can be extended as required.

modeled by the play association between the Server and Server Allocation in Fig. 3(b). Similarly the Router and Router Allocation are modeled.

5. Modeling Unity self-healing architecture

As described in [25] the purpose of a self-healing system is to provide reliability and data integrity in systems that may be subject to failure. In Unity this is approached by organizing policy repositories into clusters within which they replicate their state consisting of policies and subscriptions. This guarantees that all of the state data of a failed policy repository is retained by the surviving cluster members. By reassigning the state data temporarily to a surviving member, the system will continue to operate without interruption.

Fig. 4 describes the architecture features of the Unity self-healing mechanism. In the example, the cluster is modeled as organization unit type Cluster comprising a set of Cluster Element entity role types which can be played by Policy Repositories. The ability of a policy repository to play a Cluster Element entity role type is expressed by the play association. This indirection allows (1) abstraction of the behavioral and structural features of the policy repositories that are common to the role of being a cluster member, (2) possible reuse of this clustering pattern for elements other than policy repositories, and (3) to model the dynamics of participating in a cluster (i.e. to make use of create and dispose role actions).

In operation, when a resource arbiter (modeled by agent type Resource Arbiter) deploys a policy repository it is supplied with its intended role (modeled by the Policy Provider entity role type), identifier of the cluster that it should join, and other necessary information. As the policy repository initializes, it uses the registry to contact already registered members of the cluster and thus join the cluster. Whenever one of the policy repositories receives changes to its policy set or subscriptions, the changes are communicated throughout the remainder of the cluster.

The resource arbiter also contracts a sentinel (modeled by the Sentinel agent type) to monitor the policy repository (expressed by a perceives dependency). If this sentinel determines that a policy repository has failed, it notifies the resource arbiter which in turn will select one of the live policy repositories to take over the role of the failed repository and notify all cluster members of the reassignment. Then the resource arbiter examines the available hosts and selects one on which to deploy a replacement policy repository. Deployment is made via the OSContainer (modeled by OSContainer agent type) on the target host. Upon initialization, the new policy repository joins the cluster, retrieves a copy of the current cluster policies and subscriptions and takes over the role of the failed repository.

The model further expresses that the Resource Arbiter is a superordinate of all other autonomic elements shown by the super-sub social relationships. Also shown is the Policy Provider role which abstracts the structural behavior features of a policy provider concerning the provision of the Policy Subscription service. This service allows other autonomic elements of the system to subscribe and be notified of policy changes managed by the policy provider.

6. Modeling interactions

To support modeling of interactions AML provides a number of UML extensions, which can be logically subdivided into: (1) generic extensions to UML interactions, (2) speech act based extensions to UML interactions, (3) observations and effecting interactions, and (4) services.

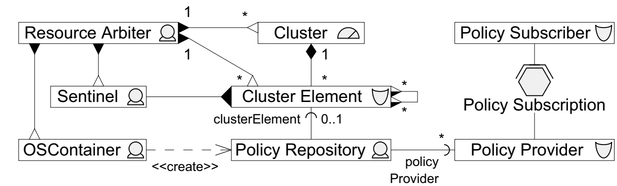

Figure 4. Cluster structure.

6.1 Generic extensions to UML interactions

Generic extensions to UML interactions provide means to model: (1) interactions between groups of entities (multi-message and multi-lifeline), (2) dynamic change of object's attributes to express changes in internal structure of organization units, social relationships, or played entity roles, etc., induced by interactions (attribute change), (3) modeling of messages and signals not explicitly associated with the invocation of corresponding methods and receptions (decoupled message), (4) mechanisms for modification of interaction roles of entities (not necessarily entity roles) induced by interactions (subset and join dependencies), and (5) modeling the actions of dispatch and reception of decoupled messages in activities (send and decoupled message actions, and associated triggers).

Multi-message is a specialized UML message which is used to model a particular communication between (unlike UML message) multiple participants, i.e. multiple senders and/or multiple receivers.

Multi-lifeline is a specialized UML lifeline, used to represent (unlike UML lifeline) multiple participants in interactions.

Decoupled message is a specialized multi-message used to model the asynchronous dispatch and reception of a message payload without (unlike UML message) explicit specification of the behavior invoked on the side of the receiver. The decision of which behavior should be invoked when the decoupled message is received is up to the receiver, which allows it to preserve its autonomy in processing messages.

Attribute change is a specialized UML interaction fragment used to model the change of attribute values (state) of interacting entities induced by the interaction. Attribute change thus enables the expression of addition, removal, or modification of attribute values, and also to express the added attribute values by sub-lifelines. The most likely utilization of attribute change is in modeling the dynamic change of entity roles played by behavioral entities represented by lifelines in interactions, and the modeling of entity interactions with respect to the played entity roles (i.e. each sub-lifeline representing a played entity role can be used to model interaction of its player with respect to this entity role).

Subset is a specialized UML dependency between event occurrences owned by two distinct (superset and subset) lifelines used to specify that since the event occurrence on the superset lifeline, some of the instances it represents (specified by the corresponding selector) are also represented by another, the subset lifeline.

Similarly, the *join* dependency is also a specialized UML dependency between two event occurrences on lifelines (subset and union ones), used to specify that a subset of instances, which have been until the subset event occurrence represented by the subset lifeline, is after the union event occurrence represented by the union lifeline. Thus the union lifeline after the union event occurrence, represents the union of the instances it has been representing before, and the instances specified by the join dependency. *Send decoupled message action* is a specialized UML send object action used to model the action of dispatching a decoupled message, and *accept decoupled message action* is a specialized UML accept event action used

to model reception of a decoupled message action that meets the conditions specified by the associated decoupled message trigger.

The model in Fig. 5 shows the interaction of Unity autonomic elements during the creation and initialization of the policy repository and the process of it joining a cluster (as described in Section 5.

Initially the `Resource Arbiter` sends a `createPolicyRepository` message to the `OSContainer` to create a new `Policy Repository`. Once created a `creationConfirmation` message is returned to the `Resource Arbiter`. The new `Policy Repository` then registers itself with the `Registry` and starts to play the role of `clusterElement`. After this two parallel courses of interaction take place. In the first, the `Resource Arbiter` sends a `monitor` message to the `Sentinel` requesting it to monitor the newly created `Policy Repository`. In the second, the `Policy Repository` requests the `Registry` for the list of existing cluster members in order to announce to each of them that it has joined the cluster and has started to play the role of `policyProvider`.

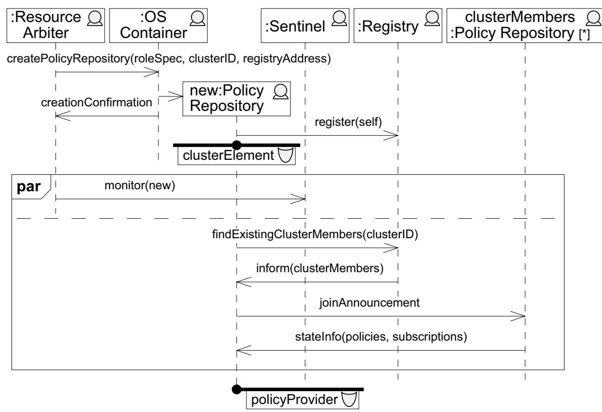

Figure 5. Deployment of a policy repository.

6.2 Speech act specific extensions to UML interactions

Speech act specific extensions to UML interactions comprise modeling of speech-acts (communication message), speech act based interactions (communicative interactions), patterns of interactions (interaction protocols), and modeling the actions of dispatch and reception of speech-act based messages in activities (send and accept communicative message actions, and associated triggers).

Communication message is a specialized decoupled message used to model communicative acts of speech act based communication within *communicative interactions* (a specialized UML interaction) with the possibility of explicit specification of the message performative and payload. Both the communication message and communicative interaction can also specify the agent communication and content languages, ontology and payload encoding used.

Interaction protocol is a parametrized communicative interaction template used to model reusable templates of communicative interactions.

6.3 Observations and effecting interactions

AML provides several mechanisms for modeling observations and effecting interactions in order to (1) allow modeling of the ability of an entity to observe and/or to bring about an effect on others (perceptors and effectors), (2) specify what observation and effecting interactions the entity is capable of (perceptor and effector types and perceiving and effecting acts), (3) specify what entities can observe and/or effect others (perceives and effects dependencies), and (4) explicitly model the actions of observations and effecting interactions in activities (percept and effect actions).

Observations are modeled in AML as the ability of an entity to perceive the state of (or to receive a signal from) an observed object by means of *perceptors*, which are specialized UML ports. *Perceptor types* are used to specify (by means of owned *perceiving acts*) the observations an owner of a perceptor of that type can make.

Perceiving acts are specialized UML operations which can be owned by perceptor types and thus used to specify what perceptions their owners, or perceptors of a given type, can perform.

The specification of which entities can observe others, is modeled by a *perceives* dependency. For modeling behavioral aspects of observations, AML provides a specialized *percept action*. This is demonstrated in Fig. 4 where the `perceives` dependency is used to model that a `Sentinel` observes `Policy Repositories`.

Different aspects of effecting interactions are modeled analogously, by means of *effectors*, *effector types*, *effecting acts*, *effects* dependencies, and *effect actions*.

6.4 Services

The AML support for modeling services comprises (1) the means to specify the functionality of a service and the way in which a service can be accessed (service specification and service protocol), (2) the means to specify which entities provide/use services (service provision, service usage, and serviced property), and (if applicable) by what means (serviced port).

A *service* is a coherent block of functionality provided by a behaviored semi-entity, called service provider, that can be accessed by other behaviored semi-entities (which can be either external or internal parts of the service provider), called service clients.

Service specification is used to specify a service by means of owned service protocols, i.e. specialized interaction protocols extended with the ability to specify two mandatory, disjoint and nonempty sets of (not bound) parameters, particularly: provider and client template parameters.

The *provider template parameters* of all contained service protocols specify the set of the template parameters that must be bound by the service providers, and the *client template parameters* of all contained service protocols specify the set of template parameters that must be bound by the service clients. Binding of these complementary sets of template parameters specifies the features of the particular service provision/usage which are dependent on its providers and clients.

Service provision/*usage* are specialized dependencies used to model provision/use of a service by particular entities, together with the binding of template parameters that are declared to be bound by service providers/clients. Fig. 4 gives an example of a `Policy Subscription` service that is provided by the `Policy Provider` entity role type to all `Policy Subscribers`.

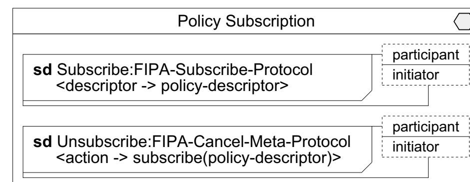

Figure 6. Example of service specification.

Fig. 6 shows a specification of the `Policy Subscription` service defined as a collection of two service protocols. The `Subscribe` service protocol is used by autonomic elements to subscribe to notification of changes to policies maintained by policy providers.

It is based on the standard FIPA-Subscribe-Protocol[8] [26] and binds the descriptor parameter (content of the subscribe message) to the description of the policy to modification of which the initiator subscribes (policy-description). The participant parameter of the FIPA-Subscribe- Protocol is mapped to a service provider and the initiator parameter to a service client. Similarly, the Unsubscribe service protocol is based on FIPA-Cancel-Meta-Protocol and its purpose is to void a subscription. Therefore service protocol maps the action (content of the cancel message) to subscribe(policy-description), i.e. to the action that is no longer intended.

7. Modeling capabilities and behavior

AML extends the capacity of UML to abstract and decompose behavior by another two modeling elements: capability and behavior fragment.

Capability is an abstract specification of a behavior which allows reasoning about and operations on that specification. Technically, a capability represents a unification of the common specification properties of UML's behavioral features and behaviors expressed in terms of their inputs, outputs, pre- and post-conditions.

Behavior fragment is a specialized behaviored semi-entity type used to model a coherent re-usable fragment of behavior and related structural and behavioral features. It enables the (possibly recursive) decomposition of a complex behavior into simpler and (possibly) concurrently executable fragments, as well as the dynamic modification of an entity's behavior in run-time. The decomposition of a behavior of an entity is modeled by owned aggregate attributes of the corresponding behavior fragment type.

Fig. 7(a) shows a decomposition of the application manager (as shown in [25]) into a structure of behavior fragments (e.g. Demand Forecaster and Utility Calculator) and local data stores (e.g. Rt) modeled as properties.

Fig. 7(b) shows the behavior fragment Demand Forecaster described in terms of its own capabilities (dataSeriesAnalysis and userBehaviorPatternRecognition).

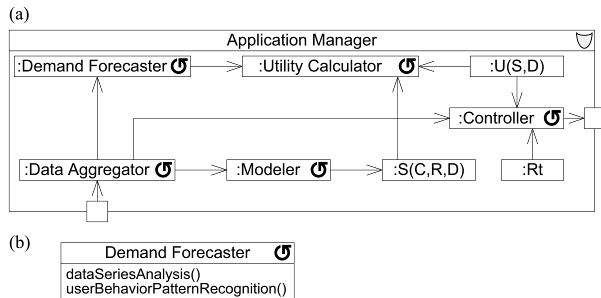

Figure 7. Inside an Application Manager.

Fig. 8 models the policy repository failure recovery scenario as described in Section 5 by means of an activity diagram. When a Sentinel observes the failure of a policy repository it sends a notification message to the Resource Arbiter which then selects an appropriate substitute for the failed repository (Select substitute action). Then, two courses of activity start in parallel. In one, the selected substitute takes over the role of the failed policy repository (Become substitute create role action). In the other, the Resource Arbiter first notifies the remaining cluster members of the substitution (Notify of substitution send decoupled message action) and then selects the host on which to deploy a replacement policy repository (New PR). The New PR is deployed and upon initialization, gets the cluster state data from the other members of the cluster (Get cluster state data action). At this point the two parallel courses join, the substitution stops (Stop substitution dispose role action) and the New PR starts to play the role of policy repository which failed (Replace failed PR create role action).

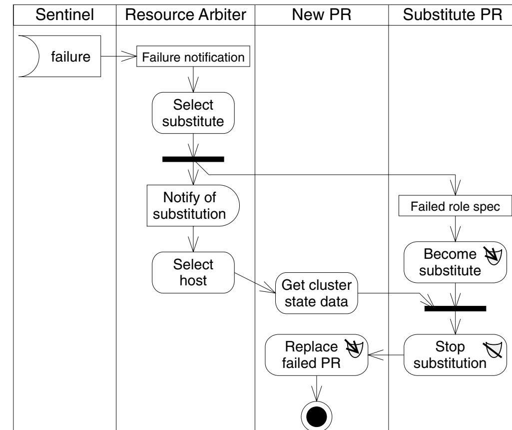

Figure 8. Policy repository failure recovery.

8. Modeling MAS deployment and mobility

The means provided by AML to support the modeling of deployment and mobility comprise: (1) the support for modeling the physical infrastructure onto which entities are deployed (agent execution environment), (2) what entities can occur on which nodes of the physical infrastructure and what is the relationship of deployed entities to those nodes (hosting property), (3) how entities can get to a particular node of the physical infrastructure (move and clone dependencies), and (4) what can cause the entity's movement or cloning throughout the physical infrastructure (move and clone actions).

Agent execution environment type is a specialized UML execution environment used to model types of execution environments within which entities can run. While it is a behaviored semi-entity type, it can explicitly, for example, also specify a set of services that the deployed entities can use or should provide at run time.

Agent execution environment can also own *hosting properties*, which are used to classify the entities which can be hosted by the owning agent execution environment. The hosting property's *hosting kind* specifies the relation of the referred entity type to its owning agent execution environment (i.e. either *resident* or *visitor*).

Hosting association is a specialized UML association used to specify hosting property in the form of an association end.

Move is a specialized UML dependency between two hosting properties used to specify that the entities represented by the source hosting property can be moved to the instances of the agent execution environments owning the destination hosting property. The *clone* dependency is used similarly.

Move and *clone actions* are specialized UML add structural feature actions used to model actions that cause movement or cloning of an entity from one agent execution environment to another. Both the actions thus specify: (1) which entity is being moved or cloned,

[8] The AML specification of the interaction protocol can be modeled in a similar way to the specification of the FIPA-Query-Protocol presented in [5].

(2) the destination agent execution environment instance where the entity is being moved or cloned to, and (3) the hosting property where the moved or cloned entity is being placed.

9. Modeling mental aspects

To model mental attitudes, AML provides: goals, beliefs, plans, contribution relationships, mental properties and associations, mental constraints, and commit/cancel goal actions.

Goal is a specialized UML class used to model goals, i.e. conditions or states of affairs with which the main concern is their achievement or maintenance. Goals can thus be used to represent objectives, needs, motivations, desires, etc.

Belief is a specialized UML class used to model a state of affairs, proposition or other information relevant to the system and its mental model.

The attitude of a mental semi-entity to a belief or commitment to a goal is modeled by the belief or the goal instance being held in a slot of the corresponding *mental property* (owned by the mental semi-entity, or a mental association relating the belief or the goal to the mental semi-entity).

Plan is a specialized UML activity used to model predefined plans or fragments of behavior from which the plans can be composed.

Mental constraint is a specialized UML constraint used to specify properties of owning beliefs, goals and plans which can be used within the reasoning processes of mental semi-entities. Supported kinds of mental constraints are pre- and post-conditions, commit conditions, cancel conditions and invariants.

Contribution is a specialized UML relationship used to model logical relationships between goals, beliefs, plans and their mental constraints. The manner in which the specified mental constraint (e.g. post-condition) of the contributor influences the specified mental constraint kind of the beneficiary (e.g. pre-condition). The degree of the contribution can also be specified.

Responsibility is a specialized UML realization used to model a relation between belief, goal and plan (called responsibility objects) and elements (called responsibility subjects) that are obligated to accomplish (or to contribute to the accomplishment of) those beliefs, goals, or plans (e.g. modification of beliefs, achievement or maintenance of Goals, realization of Plans, etc.).

Actions to model commitments to and de-commitments from goals within activities are also provided.

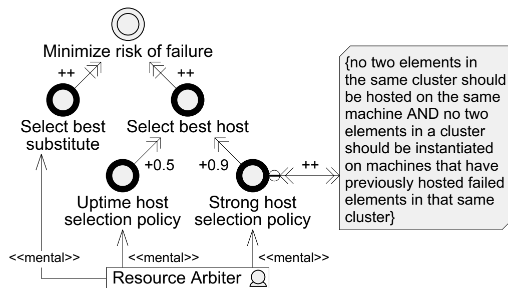

Figure 9. Example of a mental model.

Fig. 9 shows an example of a snapshot of the mental model of a `Resource Arbiter`[9]. One of the primary goals of the resource arbiter is to minimize the risk of system failure (modeled by undecideable goal `Minimize risk of failure`). This goal is decomposed into two sub-goals, both of which are necessary to the achievement of the composite one (expressed by the necessary contribution relationships between the goals). The first is to select the best temporary substitute based on longest uptime (`Select best substitute`), and the second is to select the best host by achieving one of two further sub-goals. The first of these sub-goals (`Strong host selection policy`) is stronger (as expressed with the contribution relationship) and states that no two elements in the same cluster should be hosted on the same machine and no two elements in a cluster should be instantiated on machines that have previously hosted failed elements in that cluster. This is expressed by the necessary and sufficient contribution between the shown belief and the postcondition of the goal. If the above described goal cannot be achieved a weaker alternative specified by the `Uptime host selection policy` is applied.

[9] In order to better demonstrate the capability of AML to model mental aspects we have added additional policies to the Unity architecture.

10. Conclusion

The intention of this paper is to illustrate the effective application of the Agent Modeling Language to the modeling of both agent-specific and standard (non-agent specific) aspects of autonomic systems. This has been achieved through the explanation of the most significant features of AML in terms of the autonomic elements embodied in the Unity architecture and how they collaborate to bring about self-healing recovery from policy repository failures [25].

AML has already been demonstrated, through commercial application [22], to be a powerful means of modeling systems containing autonomous software entities. We have now found, through this work, that AML is also well suited to the modeling of autonomic systems and can intrinsically model aspects of self-* principles especially enacted through the behaviors of autonomous agents. In fact, modeling systems in this manner not only helps to clarify and simplify the analysis and design aspect of creating agent and autonomic systems, it also helps to identify where flaws and problems may lie in less well-formed and/or complex system designs.

The AML specification is quite comprehensive and thus cannot be presented in its entirety in a paper of this length, nor was that our intention. Our aim was to demonstrate the utility of AML when applied to a selected problem in the autonomic computing domain. We believe that the examples provided give a reasonable flavor of AML and the set of constructs it offers for modeling applications embodying and/or exhibiting characteristics of multi-agent systems and autonomic systems.

References

[1] J. Baldassari, C. Kopec, E. Leshay, W. Truszkowski, and D. Finkel. Autonomic cluster management system (ACMS): A demonstration of autonomic principles at work. In *Proceedings of Workshop on the Engineering of Computer-Based Systems*, pages 512–518, 2005.

[2] B. Bauer, J. Muller, and J. Odell. Agent UML: A formalism for specifying multiagent interaction. In P. Ciancarini and M. Wooldridge, editors, *Agent-Oriented Software Engineering*, pages 91–103. Springer-Verlag, Berlin, 2001.

[3] P. Bresciani, P. Giorgini, F. Giunchiglia, J. Mylopoulos, and A. Perini. TROPOS: An agent-oriented software development methodology. *Autonomous Agents and Multi-Agent Systems*, 2(3):203–236, 2004.

[4] R. Cervenka. *Modeling Multi-Agent Systems*. PhD thesis, Comenius University in Bratislava, Faculty of Mathematics, Physics and Informatics, 2005.

[5] R. Cervenka and I. Trencansky. Agent Modeling Language: Language specification. Version 0.9. Technical report, Whitestein Technologies, 2004.

[6] R. Cervenka, I. Trencansky, and M. Calisti. Modeling social aspects of multi-agent systems: The AML approach. In J. Muller and F. Zambonelli, editors, *Agent-Oriented Software Engineering VI:*

6th International Workshop, AOSE 2005, Lecture Notes in Computer Science 3950, pages 28–39. Springer-Verlag, February 2006.

[7] M. Cossentino and C. Potts. A CASE tool supported methodology for the design of multi-agent systems. In *Proceedings of the 2002 International Conference on Software Engineering Research and Practice (SERP02)*, Las Vegas, NV, USA, 2002.

[8] M. Cossentino, L. Sabatucci, and A. Chella. A possible approach to the development of robotic multi-agent systems. In *IEEE/WIC Conference on Intelligent Agent Technology (IAT'03)*, pages 539–544, Halifax, Canada, 2003.

[9] S. DeLoach, M. Wood, and C. H. Sparkman. Multiagent systems engineering. *International Journal of Software Engineering and Knowledge Engineering*, 11(3):231–258, 2001.

[10] R. Evans, P. Kearny, J. Stark, G. Caire, F. Garijo, J. Gomez-Sanz, F. Leal, P. Chainho, and P. Massonet. MESSAGE: Methodology for engineering systems of software agents. Technical Report P907, EURESCOM, 2001.

[11] J. Ferber and O. Gutknecht. A meta-model for the analysis and design of organizations in multi-agent systems. In *3rd Int. Conference on Multi-Agent Systems (ICMAS'98)*, pages 128–135. IEEE Computer Society, 1998.

[12] C. Iglesias, M. Garijo, J. Gonzalez, and J. Velasco. Analysis and design of multiagent systems using MAS-CommonKADS. In M. Singh, A. Rao, and M. Wooldridge, editors, *Intelligent Agents IV (LNAI Vol. 1365)*, volume 1365, pages 313–326. Springer-Verlag, 1998.

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

[14] Z. Li and M. Parashar. Rudder: A rule-based multi-agent infrastructure for supporting autonomic grid applications. In *Proceedings of ICAC'04*, pages 278–279, 2004.

[15] M. Mamei and F. Zambonelli. Self-maintaining overlay data structures for autonomic distributed computing. In *Proceedings of ICAC'05*, pages 376–377, 2005.

[16] J. Odell, H. Parunak, and B. Bauer. Extending UML for agents. In G. Wagner, Y. Lesperance, and E. Yu, editors, *Proceedings of the Agent-Oriented Information Systems Workshop at the 17th National conference on Artificial Intelligence*, pages 3–17, Austin, Texas, 2000.

[17] J. Odell, H. Parunak, M. Fleischer, and S. Brueckner. Modeling agents and their environment. In F. Giunchiglia, J. Odell, and G. Weiss, editors, *Agent-Oriented Software Engineering III: Third International Workshop, AOSE 2002*, pages 16–31. Springer-Verlag, Berlin, 2002.

[18] OMG. Meta Object Facility (MOF) specification. Version 1.4, formal/2002-04-03, april 2002.

[19] OMG. Unified Modeling Language: Superstructure version 2.0. ptc/03-08-02, 2003.

[20] L. Padgham and M. Winikoff. Prometheus: A methodology for developing intelligent agents. In F. Giunchiglia, J. Odell, and G. Weiss, editors, *Agent-Oriented Software Engineering III: Third International Workshop, AOSE 2002*, pages 174–185. Springer-Verlag, Berlin, 2002.

[21] J. Pavon and J. Gomez-Sanz. Agent oriented software engineering with INGENIAS. In V. Marik, J. Muller, and M. Pechoucek, editors, *Multi-Agent Systems and Applications III: 3rd International Central and Eastern European Conference on Multi-Agent Systems CEEMAS 2003*, pages 394–403. Springer Verlag, 2003.

[22] G. Rimassa, M. Calisti, and M. Kernland. Living Systems Technology Suite. *Whitestein Series: Software Agent-Based Applications, Platforms and Development Kits*, 2005.

[23] V. Silva, A. Garcia, A. Brandao, C. Chavez, C. Lucena, and P. Alencar. Taming agents and objects in software engineering. In A. Garcia, C. Lucena, J. Castro, A. Omicini, and F. Zambonelli, editors, *Software Engineering for Large-Scale Multi-Agent Systems: Research Issues and Practical Applications*, volume LNCS 2603, pages 1–25. Springer-Verlag, Berlin, 2003.

[24] A. Sturm, D. Dori, and O. Shehory. Single-model method for specifying multi-agent systems. In *Proceedings of the Second International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 121–128. ACM Press, New York, NY, 2003.

[25] G. Tesauro, D. Chess, W. Walsh, R. Das, A. Segal, I. Whalley, J. Kephart, and S. White. A multi-agent systems approach to autonomic computing. In *Proceedings of AAMAS'04*, pages 464–471, 2004.

[26] The Foundation for Intelligent Physical Agents. FIPA Specifications Repository. URL: http://www.fipa.org/repository/index.html, 2004.

[27] I. Trencansky and R. Cervenka. Agent Modelling Language (AML): A comprehensive approach to modelling MAS. *Informatica*, 29(4):391–400, 2005.

[28] W. Turski and T. Maibaum. *The Specification of Computer Programs*. Addison-Wesley, London, 1987.

[29] G. Wagner. The Agent-Object-Relationship metamodel: Towards a unified view of state and behavior. *Information Systems*, 28(5):475–504, 2003.

[30] T. D. Wolf and T. Holvoet. Towards autonomic computing: Agent-based modelling, dynamical systems analysis, and decentralised control. In *Proceedings of the First Workshop on Autonomic Computing Principles and Architectures*, pages 10–18, 2003.

[31] F. Zambonelli, N. Jennings, and M. Wooldridge. Developing multiagent systems: The Gaia methodology. *ACM Transactions on Software Engineering and Methodology*, 12(3):317–370, 2003.