Towards a Model-Driven Architecture for Autonomic Systems

Denis Gračanin, Shawn A. Bohner, Michael Hinchey
Virginia Tech
Department of Computer Science
7054 Haycock Road
Falls Church, VA 22043
USA
{gracanin,sbohner,mhinchey}@vt.edu

Abstract

Agent based systems and architectures provide a firm foundation for design and development of an autonomic system. The key challenge is the selection and efficient use of effective agent architecture. A model-driven approach accommodates the underlying architecture to automate, as much as possible, the development process. The Cognitive Agent Architecture (COUGAAR) is a distributed agent architecture that provides the primary components and an implementation platform for this research. COUGAAR has been developed primarily for very large-scale, distributed applications that are characterized by hierarchical task decompositions and as such is well suited for autonomic systems. We propose a framework for the agent-based, model-driven architecture for autonomic applications development. The framework consists of two main parts, General COUGAAR Application Model (GCAM) and General Domain Application Model (GDAM). Some COUGAAR related performance issues are also discussed.

1 1. Introduction

As society increasingly depends on software, the size and complexity of software systems continues to grow making them more difficult to understand and evolve. Manifest dependencies between critical elements of software now drive software architectures and increasingly influence the system architecture. Complexity and integration issues frequently dominate modern computing. To respond to the sheer volume of software and consequential complexity, the software community has increasingly embraced architecture principles. Software architecture provides a framework to understand dependencies that exist between the various components, connections, and configurations reflected in the requirements. Agent-based software architectures support autonomous systems and respond to their integration and process needs. These emergent technologies provide a reasonable basis for addressing complexity issues by separating concerns (integration, interoperability, decision support, and the like) and allowing agents to provide the necessary processing.

Autonomic Systems as a discipline is emerging to address these complex information and task-intensive situations. The task orientation coupled with intelligent agents provides a strategic and holistic environment for designing large and complex computer-based systems. These systems may support logistics management, battlefield management, supply-chain management to mention a few.

The Cognitive Agent Architecture (COUGAAR) is an open source, distributed agent architecture [1]. COUGAAR is the result of approximately eight years of development for the Defense Advanced Research Projects Agency (DARPA) under the Advanced Logistics Program (ALP) and the Ultra*Log program [2]. The primary focus of development has been on very large-scale, distributed applications that are characterized by hierarchical task decompositions, such as military logistics planning and execution. In addition, during the last four years, particular attention has been given to fault tolerance, scalability and security.

To give the reader a sense of this technology, this paper first provides a very brief, high-level overview of COUGAAR. We discuss its primary capabilities and some examples of its application. We then discuss the current Model-Driven Architecture work in extending this to better support autonomous systems development.

2 COUGAAR Overview

COUGAAR is an open-source Java-based agent infrastructure that arose from the DARPA Advanced Logistics and Ultra*Log programs. It is important to note that while COUGAAR has been developed primarily for large-scale, distributed applications, it is certainly not limited solely to these kinds of applications. Quantifying the amount of overhead introduced by the COUGAAR software is an important step in determining whether COUGAAR is a good fit for the development of a new application [6].

2.1 Agents

COUGAAR agents [2,3] are arranged into a society that collectively solves a problem or a class of problems. The society consists of one or more communities of agents that share the same functional purpose or organizational commonality. The same agent may belong to one or more of these communities.

A COUGAAR node is a single Java Virtual Machine (JVM) running on a single server that contains one or more agents. The node is a concept, not a class.

The society and communities are usually deployed across several nodes. Figure 1 shows an example of a society that is spread across three nodes.

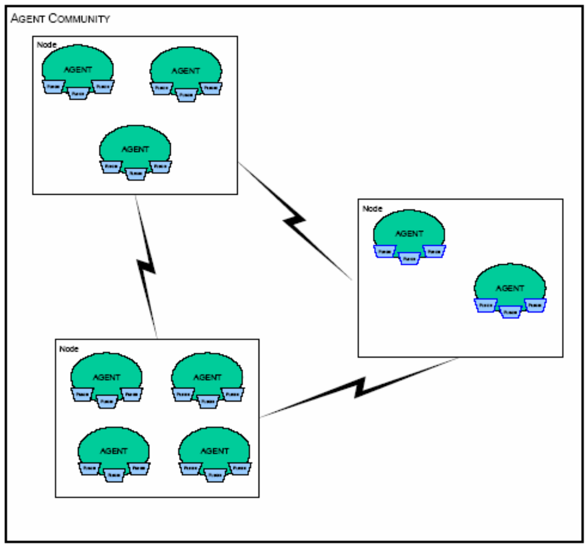

Figure 1: Example of COUGAAR society [2,3]

Agents on the same node compete for resources like CPU, memory, disk space, and network bandwidth.

An agent consists primarily of a blackboard and a set of plugins (Figure 2). The blackboard is essentially a container of objects that adheres to publish/subscribe semantics. A plugin implements a piece of the core business logic associated with a given agent. The agent is characterized by one or more plugins that are referentially uncoupled (i.e., they do not know about each other). Plugins publish objects, remove objects or publish changes to existing objects via the blackboard. Plugins also create subscriptions to be notified when objects are added, removed or changed in the blackboard.

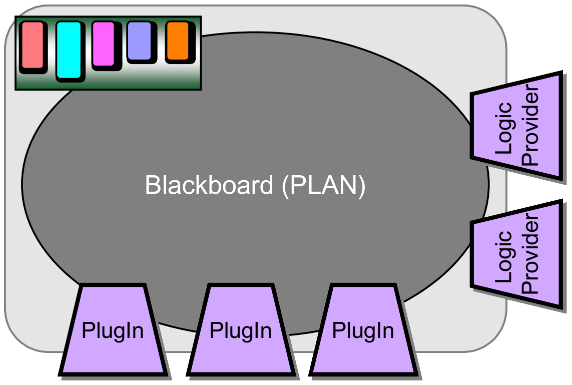

Figure 2: Agent internal structure [2]

2.2 Inter-agent communications

Agents collaborate with other agents, however the do not send messages directly to each other. Instead, a task is created. Each task creates an "information channel" flowing through the society for requirements passing down, and responses going back [2]. In order to send an object, A, to another agent, one must first associate A with the task. The problem is that only instances of the `Asset` class may be associated with a task. Thus, all multi-agent objects must be defined as assets. In other words, a multi-agent object must be an instance of a class that extends the `Asset` class.

One must then locate the agent to which to allocate the task. This is typically done by creating a subscription that examines the roles or property groups of organizations in the local blackboard. Once the proper organization is found, the task containing the object to be sent to the other agent is allocated to that organization by creating an allocation and publishing it to the blackboard. The COUGAAR communication infrastructure then ensures that the task is sent to the specified organization's and the specified agent's blackboard.

A relationship between two agents which can either be a superior/subordinate or customer/provider relationship. The superior/subordinate relationship supports long-standing orders where a superior gives high-level tasks to the subordinate which then performs the task and then report back aggregate and trend information back to the superior on a periodic basis. A customer/provider relationship on the other hand is for task-order services between agents on a peer-to-peer basic and may result in large scale discrete data flows between the agents.

2.3 Agent mobility

COUGAAR provides agent mobility service whereby an agent can move from one node to another through the process of serialization [10]. A special agent plugin provided by COUGAAR initiates the transfer by using the mobility services to suspend the agent, retrieve the agent state, serialize the state, and move the agent to the new node (using the messaging services and a special blackboard object). This object is received and then de-serialized, initialized, re-bind to required services, set the state, load and restart the agent. Once this process has completed successfully then an acknowledgement is sent back to the sending agent which is stopped, unloaded and eventually garbage collected. Services and other resources used by the agent are not moved and must be reestablished and rebound on the new node.

3 Performance Measurement

COUGAAR is designed as a workflow manager for hierarchical task decomposition problems. Tasks are decomposed into sub-tasks and distributed between the plugins, or even between agents on the same or other nodes.

COUGAAR provides facilities to measure and propagate performance metrics collected at all levels of the architecture [5,7]. This includes sensors which are simple and efficient modules that collect information at critical points in the system execution and typically maintains simple data structures such as statistics counters. These sensors are polled out-of-band from the main processing thread by clients who then bear the burden of information processing.

Performance measurements can affect the performance of the system [8] and different clients have different metrics and information delivery requirements. In order to provide varying Quality of Service (QoS) multiple channels are provided that provide different delivery times and quality of metrics.

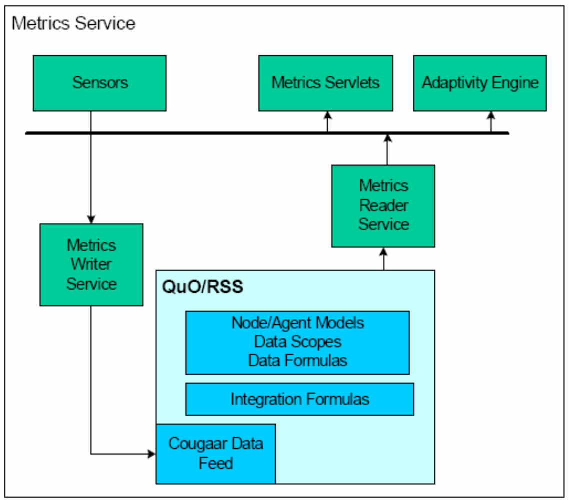

Figure 3: Metrics service

Figure 3 shows a schematic of the metrics service architecture [2]. Metrics writer and reader services provide sensor values to and read data from the QoS/RSS metrics data model. As values are collected this data model is updated in real-time and provides a comprehensive view of the society as a whole.

This data is used by metrics servlets that provide a web-based view, and the Adaptivity Engine which is used to adapt the execution of the agents in the society, and is explained in more detail below.

Platform specific measurements are taken by the computer system level instrumentation and are translated into device-independent values. These include CPU and memory usage statistics, and can be additionally aggregated across agents to provide community or node level statistics.

Measurements captured from the agent architecture level include blackboard metrics such as timestamps of published objects, counts of particular types of object, and the rate at which particular objects change, and message transport metrics such as message queue length, total number of bytes sent, and total messages sent.

3.1 Adaptive Control

COUGAAR provides adaptive control [9] through the Adaptivity Engine that makes use of the measurements collected by the metrics service. Each agent can have several modes of operation that provide increased QoS with increased resource consumption. The Adaptivity Engine can use the operating conditions indicated by the metrics to select the agent modes of operation. Figure 4 shows the adaptive control infrastructure based on the Adaptivity Engine. Each component publishes one or more operating modes. Each mode represents the allowed values for one of the degrees of freedom of component as well as the current value. The component in this case is a plugin and the degrees of freedom represent the tunable parameters used by that plugin. Metric Service (sensors) publishes conditions indicating the state of the system. Aggregation agents can also publish conditions that indicate the state of a collection of agents, a community, or the society as a whole.

The Operating Mode Policy Manager takes the conditions and operating modes and uses these to restrict the available Plays from the Playbook Manager. Each Play has a logical expression. The expressions are tested in succession to determine which Plays apply to the current conditions.

A Play specifies the restrictions and constraints on one or more operating modes and the conditions under which they are applied. A Play may specify a single value for an operating mode or it may give an ordered range of values that specify constraints on the operating mode value. This allows multiple plays to be specified that jointly control an operating mode. The constraints from all the applicable plays are combined by intersection to compute an effective value which is the minimum of the first range in the list.

Each agent contains an Adaptivity Engine that controls the agent and responds to the conditions and changes to the Playbook The component TechSpecs are combined with the system behavior to adapt the agent to optimize performance.

An example is given in Kleinmann [9] comprising of task generator agent and a provider agent that allocates tasks. The sensor conditions are the available CPU resources and the rate of arrival of new tasks at the provider agent. An operating mode tunes the allocation algorithm to control the quality of allocations which depends on the number of iterations that the algorithm can afford given the available CPU resources.

The playbook contains a play that uses the heuristic rule that if the incoming task rate is low and CPU resources are available then task allocation can be done more precisely using more iterations of the allocation algorithm. The play accomplishes this by dividing task rate by CPU availability and then mapping the result into the operating mode that determines the algorithm iterations. The selected plays in a playbook may not always result in the maximization of a performance measure.

There may be conditions affecting the performance that are not available to the Adaptivity Engine, or some assumptions of independence between different conditions and operating modes can cause some problems.

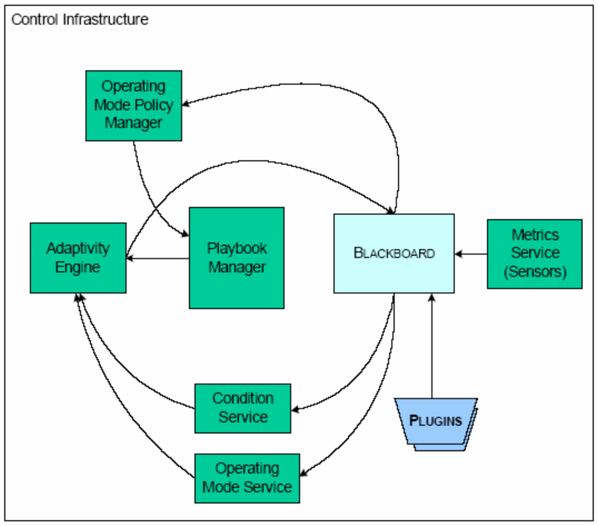

Figure 4: Adaptive control infrastructure [2]

The described performance characteristics and control infrastructure are used as a basis for efficient implementation of common application modules using the COUGAAR platform.

4 Proposed Framework

Figure 5 shows the proposed framework. The COUGAAR Model-Driven Architecture prescribes certain kinds of models to be used, how those models may be prepared and the relationships of the different kinds of models. It encompasses the key dependencies between software artifacts.

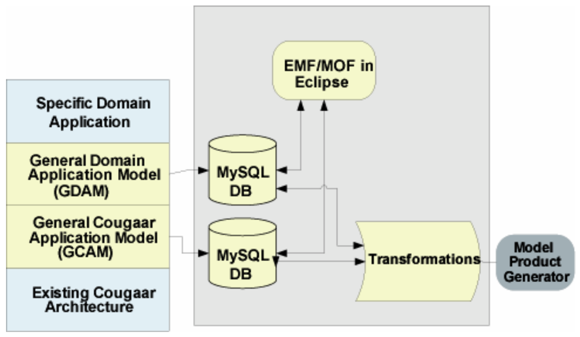
EMF/MOF in Eclipse
Specific Domain Application
General Domain Application Model (GDAM)
General Cougaar Application Model (GCAM)
Existing Cougaar Architecture
MySQL DB
MySQL DB
Transformations
Model Product Generator

Figure 5: Model driven architecture framework

The implementation platform is described using a General Cougar Application Model (GCAM) and is, therefore, based on the existing COUGAAR architecture [2]. The GCAM provides representation in its model of the COUGAAR basic constructs [2].

The core representation includes Agents, Communities, Societies, Plugins, Assets, Preferences, Knowledge Rules, Policies, Rules, Constraints, Events, Facts, Services, Service Providers, Tasks, Nodes, Subscriptions, Predicate, Messages, Directives, Logic Providers, Hosts, Domains, and Configuration.

General Domain Application Model (GDAM) builds upon the foundation of GCAM; therefore, the requirements and detailed designs collectively define the GDAM. For instance, the design may be captured through written requirements, use case analysis models, diagramming, etc. This specification must be sufficient to enable a successful proof-of-concept prototype.

GDAM provides representation in the model for general COUGAAR application concepts. The GDAM representations may include the following components:

- Generic GUI Plugin
- Planning Plugin
- General Business Processes and Rules
- General types of roles
- Common set of tasks
- Common types of relationships
- Communities of interest for information

The specific domain application requirements are translated into related GDAM and GCAM model components and stored in a corresponding database.

Transformations module contains representation/ description of how to transform model components into model products as well as key transformation rules governing the Model Product Generator.

The developed platform has to support different views into the model from some required viewpoints on the system. A viewpoint on a system is a technique for abstraction using a selected set of architectural concepts and structuring rules, in order to focus on particular concerns within the system. A view is the representation of the system from the perspective of a chosen viewpoint.

4.1 Initial Implementation

A fully functional model driven architecture approach for application development is a daunting task. We are approaching this problem in incremental fashion by creating a proof-of-concept prototype. Within a domain a selected set of application case studies (examples) has been selected to determine platform specific implementation issues. COUGAAR patterns, i.e. "combinations" of COUGAAR agents for a specific task, are being developed and used in the development of the proposed framework.

Figure 6 demonstrates "stacking" of the models. The COUGAAR architecture model provides a foundation for the GCAM, GDAM and domain models. The initial, "light-weight" implementation focuses on GDAM and its interface to domain requirements. As a consequence, domain requirements are directly mapped to the COUGAAR architecture which provides a foundation for the GCAM, GDAM and domain models.

A requirements is expressed in terms of GAM components which are the directly implemented in COUGAAR as a collection of agents. Those agents are added to the "run-time environment" which is monitored and maintained dynamically.

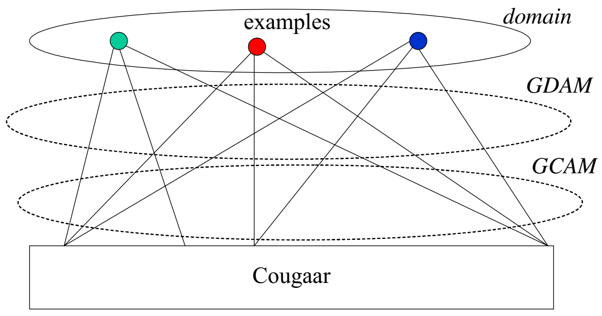

Figure 6: Model "stack"

The existing tools, such as CSMART [4], are designed "*... to provide an integrated toolset for building, running, monitoring, and analyzing all Cougaar societies, and for performing experiments on those societies by systematically varying their properties and comparing the resulting behaviors.*" They are static in nature and do not take advantage of dynamic capabilities of the COUGAAR architecture.

Dynamic construction and integration of the components, combined with the persistency support for COUGAAR applications, enables effective "recording" and "storage" of the developed application while taking into account the performance metrics discussed in Section 3.

The requirements are modeled in the computation independent viewpoint describing the situation in which the system will be used and the environment in which it will operate. The requirements should be traceable to the platform specific viewpoint constructs that implement them, and vice versa.

The next phase of implementation will determine the "line of separation" between GCAM and GDAM and the corresponding models. The outcome will be a "multi-faceted" model (Figure 7) with a programmer viewpoint that will conform to the framework described in Figure 5.

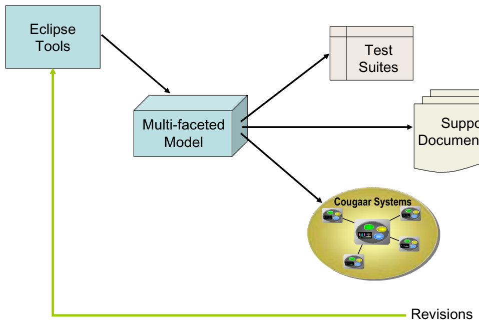

Figure 7: Implementation goals

The programmer viewpoint focuses on specifying in detail the functions and properties for the parts of the model created through the design process using the platform specific viewpoint. This viewpoint should support the ability to modify and create fine grained details within individual components of the models and trigger code and configuration file generation.

5 Conclusions

The COUGAAR architecture inherently supports an autonomic view of systems. Performance metrics and continuous analysis by agents allow the system to be self-aware, to recover from problems, or avoid other problems. This enables the system to be both self-healing and self-preserving. By adapting the existing metrics and adaptability mechanisms combined with the agent mobility a resource reservation protocol can be constructed that can be used to pre-allocate resources to optimize the performance of a society. This would require the addition of, and modifications, to services to the existing COUGAAR architecture. We are in the early stages of work funded by DARPA on developing a model-driven architecture which will support the agent-based architecture of COUGAAR, and which in turn can be used as the basis for system composition and automatic code-generation of Autonomic Systems. The outcome of this work should provide answers to the following questions:

- How do we build a model for large-scale distributed intelligent agent systems?
- How do we use the MDA approach on a mature architecture framework?
- How do we insure "stable" behavior?
- How do we leverage the library nature of the component approach to make each successive development faster and easier?

By answering those questions we can determine a closure on the model to understand what is missing, where there are design flaws and where there are inconsistencies in the domain logic.

Acknowledgment: This work has been supported, in part, by the DARPA STTR grant "AMIIE Phase II – Cougaar Model Driven Architecture Project," (Cougaar Software, Inc.) subcontract number CSI-2003-01.

6 References

1. BBN Technologies, "BBN Technologies," Retrieved December 4, 2003, from `http://www.bbn.com`.
2. BBN Technologies, "COUGAAR Architecture Document," Retrieved November 21, 2003, from `http://cougaar.org/docman/view.php/17/56/CAD_10_0.pdf`, February 1, 2003.
3. BBN Technologies, "COUGAAR Developers' Guide," Retrieved November 21, 2003, from `http://cougaar.org/docman/view.php/17/57/CDG_10_0.pdf`, February 1, 2003.
4. A. Connors, "QoS in COUGAAR and a proposed resource reservation protocol," CS 5204 course project report, Virginia Tech, Dec. 2003.
5. B. Hartman and V. Chougule, "An Introduction to COUGAAR and Beginning to Quantify its Overhead," CS 5204 course project report, Virginia Tech, Dec. 2003.
6. A. Helsinger, W. Ferguson, R. Lazarus, "Exploring Large-Scale, Distributed System Behavior with a Focus on Information Assurance," DISCEXII Proceedings, 2001.
7. A. Helsinger, R. Lazarus, Richard, W. Wright, and J. Zinky, "Tools and Techniques for Performance Measurement of Large Distributed Multiagent Systems," Proceedings of the Second International Joint Conference on Autonomous Agents and Multiagent Systems, ACM Press, New York, NY, 2003, pp. 843-850.
8. K. Kleinmann, R. Lazarus, and R. Tomlinson, "An Infrastructure for Adaptive Control of Multi-Agent Systems," IEEE KIMAS'03, October 2003.
9. M. Thome "Multi-Tier Communication Abstractions for Distributed Multi-Agent Systems," KIMAS'03 Proceedings, 2003.
10. BBN Technologies, "COUGAAR Society Monitoring, Analysis and Reporting Tool," Retrieved December 4, 2003, from `http://cougaar.org/docman/view.php/14/12/csmart-usersguide.pdf`.