Master of Software Engineering Portfolio

By

Kyle Hill

B. S., Kansas State University, 2007

MSE, Kansas State University, 2011

A PORTFOLIO

submitted in partial fulfillment of the requirements for the degree

MASTER OF SOFTWARE ENGINEERING

Department of Computing and Information Sciences
College of Engineering

KANSAS STATE UNIVERSITY

Manhattan, KS

2011

Approved by:

[Signature]
Major Professor
Dr. Scott DeLoach
0 Abstract

The GMoDS Based Runtime Agent Role Interpreter is an agent software architecture that takes an assigned goal and role as input and then "interprets" the role by mapping the role's high-level capabilities to a to a set of low-level agent capabilities. For this architecture, the role is defined by a GMoDS role-level goal model, and a goal-capability map. The role-level goal model is decomposed into a goal tree whose leaves are mapped to role capabilities by the goal-capability map. This agent architecture will be demonstrated using the CROS simulator.

Designing agents using a generic role interpreter gives system designers great freedom and makes multiagent systems more flexible. By using a role interpreter, agents do not need to have roles defined at design time; rather, they can be assigned at runtime as the organization sees fit. This allows the organization to make optimizations at runtime to blend redundant roles together, or to decompose complex roles into multiple simple ones.

Adding role interpreters to agents increases system modularity by cleanly separating role definitions from the agent architectures that execute them. Agents are simply given a formal role definition that they execute. In our system, roles are defined using GMoDS goal models, allowing us to make use of its established execution semantics. By defining roles using GMoDS, we can decompose role behavior into low-level functions that can be mapped directly to agent capabilities using a goal capability map.
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>ii</td>
</tr>
<tr>
<td>Chapter 1 - Vision Document</td>
<td>4</td>
</tr>
<tr>
<td>Chapter 2 - Project Plan</td>
<td>13</td>
</tr>
<tr>
<td>Chapter 3 - Software Quality Assurance Plan</td>
<td>21</td>
</tr>
<tr>
<td>Chapter 4 - Architectural Design</td>
<td>25</td>
</tr>
<tr>
<td>Chapter 5 - Formal Specification</td>
<td>38</td>
</tr>
<tr>
<td>Chapter 6 - Technical Inspection Checklist</td>
<td>43</td>
</tr>
<tr>
<td>Chapter 7 - Test Plan</td>
<td>45</td>
</tr>
<tr>
<td>Chapter 8 - Component Design</td>
<td>51</td>
</tr>
<tr>
<td>Chapter 9 - XML Schema</td>
<td>64</td>
</tr>
<tr>
<td>Chapter 10 - Assessment Evaluation</td>
<td>65</td>
</tr>
<tr>
<td>Chapter 11 - User Manual</td>
<td>67</td>
</tr>
<tr>
<td>Chapter 12 - Project Evaluation</td>
<td>76</td>
</tr>
<tr>
<td>References</td>
<td>82</td>
</tr>
</tbody>
</table>
1 Chapter 1 - Vision Document

1.1 Introduction

This document provides an overview of a Goal Model for Dynamic Systems (GMoDS)-based Runtime Agent Role Interpreter. This interpreter serves as the basis for an agent architecture in an Organizational Model for Adaptive Computational Systems (OMACS) multiagent system. This document will describe the critical use cases and requirements for this role interpreter, an agent architecture based on it, as well as a simple Wumpi World simulation to demonstrate this architecture.

1.2 Motivation

Designing agents using a generic role interpreter gives system designers great freedom and makes multiagent systems more flexible. By using a role interpreter, agents do not need to have roles defined at design time; rather, they can be assigned at runtime as the organization sees fit. This allows the organization to make optimizations at runtime to blend redundant roles together, or to decompose complex roles into multiple simple ones.

Developing generic agents based on a sophisticated role interpretation engine increases system modularity by cleanly separating role definitions from the agent architectures that execute them. Agents are simply assigned formal role definitions to execute. In our system, roles are defined using GMoDS goal models, allowing us to make use of its established execution semantics. Agents execute their goal model by systematically following GMoDS conventions; decomposing role behavior into a set of interrelated goals into low-level functions that can be mapped directly to agent capabilities using a goal capability map.

1.3 Terms and Definitions

1.3.1 OMACS

Organization Model for Adaptive Complex Systems is a multiagent systems architecture that defines a system in terms of an organization consisting of goals, roles, agents, capabilities, and the relationships between these entities.

1.3.2 GMoDS

The Goal Model for Dynamic Systems provides a formal definition and decomposition of system goals and the relationships between them. GMoDS provides a framework for executing a goal model within an organization.
1.3.3 Goal

A Goal is an OMACS entity that defines what the system should do. For our purposes, goals will be formally defined using GMoDS.

1.3.4 Role

A Role is an OMACS entity that defines algorithms and behaviors to achieve some goal. Roles are assigned to achieve a specific system goal at runtime.

1.3.5 Agent

An Agent is an OMACS entity that actually interacts with the system environment while playing its role to achieve some goal. Our Role Interpreter will run inside an agent instance, providing most of the agent architecture.

1.3.6 Capability

A Capability is an OMACS entity that denotes some function or ability that an agent can possess and that a role can require. Capabilities, along with other organizational policies, can be used to restrict the assignment of agents to roles.

1.3.7 RLGM

A Role Level Goal Model [3] (RLGM) is a GMoDS goal model that defines role behavior. Role Level Goal Models decompose role tasks into low-level goals that can be mapped directly to agent capabilities at runtime.

1.3.8 GCM

A Goal Capability Map [3] (GCM) provides a mapping of role level goals to agent capabilities. Our system will use a Goal Capability Map to associate leaf goals with specific agent capabilities.

1.3.9 Wumpi World

Wumpi World is simple OMACS simulation software for testing and validating agent architectures.

1.4 Project Overview
The GMoDS-based Runtime Agent Role Interpreter will be integrated into an agent’s architecture. The agent role interpreter will take a role and a goal assignment as input. The input role will be defined by a GMoDS role level goal model (RLGM), and a goal capability map (GCM). The RLGM will decompose role behavior into a tree structure where the leaf goals can essentially be treated as function calls. These low-level goals are mapped to agent capabilities using the role’s GCM. This map provides a simple one-to-one mapping of abstract, low-level role behaviors to concrete agent capabilities.

At runtime, OMACS will assign agents to play roles to achieve goals. At this time, an agent instance will be provided with a goal instance and a role instance. The agent will execute its role following GMoDS semantics: The interpreter will non-deterministically select a goal from the set of active goals. It will then directly call that goal’s associated capability, with the specified parameters, by looking it up in the goal-capability map.

1.5 Project Goal

This project will provide an implementation of a GMoDS-based Runtime Agent Role Interpreter. This interpreter will make up the bulk of an individual agent’s architecture. To demonstrate that the Role Interpreter works correctly, this project will also provide a full agent architecture based on it, as well as a simple Wumpi World simulation.

1.6 System Context
The GMoDS-based Runtime Agent Role Interpreter will make up a large part of an individual agent’s architecture. At runtime, OMACS will assign a role and goal to an agent. That agent will play the assigned role to achieve the assigned goal. The agent must be capable of playing the assigned role, and the assigned role must achieve the assigned goal.
1.7 Requirements Specification

1.7.1 Critical Use Cases

Figure 3. Critical Use Cases

1.7.2 Use Case 1: Receive Assignment

**Description:** This use case describes the assignment of roles and goals to agents.

**Includes:** Interpret Role Level Goal Model, Interpret Goal Capability Map, Validate Capabilities

**Pre-Conditions:** The role must be able to achieve the assigned goal.

**Details:** The agent hosting the role interpreter is assigned a role and a goal by OMACS. The agent will play the assigned role by executing its associated RLGM to achieve the assigned goal.

**Post Conditions:** The agent is ready to play the assigned role to achieve its goal.

**Specific Requirements:**

1.7.2.1 SR1.1 [Critical Requirement]
The system shall accept a GMoDS RLGM and GCM as input representing the role it will play.
1.7.2.2 SR1.2 [Critical Requirement]
The system shall accept a GMoDS goal instance specification as input representing the
goal it will achieve. The goal instance specification shall include goal type information
and values for each attribute.

1.7.2.3 SR1.3
The system shall accept the RLGM input according to the Goal Model XML schema used
by the agentTool3 software.

1.7.2.4 SR1.4
The system shall accept the GCM input according to an XML schema that will be used
by a future version of the agentTool3 software. The GCM schema definition does not
exist yet and it will be provided as part of this project.

1.7.3 Use Case 2: Interpret Role Level Goal Model

Description: This use case describes the parsing of a role’s Role Level Goal Model
Pre-Conditions: The given Role Level Goal Model is associated with the role assigned
to this agent.
Details: OMACS provides a Role Level Goal Model as part of role assignment. This
RLGM is a GMoDS goal model that describes the runtime behavior of the associated
role.
Post Conditions: The agent is ready to execute the RLGM.
Specific Requirements:

1.7.3.1 SR 2.1 [Critical Requirement]
The system shall parse the input RLGM into a goal tree structure using the GMoDS
parser.

1.7.4 Use Case 3: Interpret Goal Capability Map

Description: This use case describes the parsing of a role’s Goal Capability Map
Pre-Conditions: The given Goal Capability Map is associated with the role assigned to
this agent. The agent possesses the capabilities required by the assigned role.
Details: OMACS provides a Goal Capability Map as part of role assignment. The GCM
provides a one-to-one mapping between RLGM leaf goals and a role’s abstract
capabilities. The agent’s concrete capabilities map directly to the role’s abstract
capabilities at assignment.
Post Conditions: The agent can map RLGM leaf goals to specific capabilities.
Specific Requirements:
1.7.4.1 SR3.1 [Critical Requirement]
The system shall parse the input GCM into a map structure with leaf-level goals serving as keys, and concrete agent capabilities serving as values.

1.7.5 Use Case 4: Validate Capabilities

Description: This use case describes the verification that agents perform when assigned a role.
Pre-Conditions: The agent has already been assigned a role. The agent’s assigned role can achieve the given goal.
Details: OMACS should only assign roles that this agent is able to play. However, we should perform validation of an agent’s capabilities at assignment time to make the system more robust in case the organizational does not have accurate knowledge of the agent’s current capabilities, or if a capability has failed.
Post Conditions: The agent is ready to execute the RLGM.
Specific Requirements:

1.7.5.1 SR4.1 [Critical Requirement]
The system shall read the set of required capabilities out of the assigned role. These capabilities will be the union of all capabilities discovered by looking up all leaf-level goals in the role’s RLGM.

1.7.5.2 SR4.2 [Critical Requirement]
The system shall iterate over all required capabilities from the role and check to make sure that each capability is in the set of capabilities possessed by the agent.

1.7.5.3 SR4.3
If the system does not possess a required capability, it shall inform the organization so that it can take appropriate action, such as assigning a new role to this agent.

1.7.6 Use Case 5: Execute Role Level Goal Model

Description: This use case describes how the role interpreter executes the RLGM to achieve the assigned goal.
Includes: Perform Actions
Pre-Conditions: The agent is playing a role in pursuit of a goal. The agent’s role has been defined using both an RLGM and a GCM.
Details: RLGM execution takes place according to GMoDS semantics: The interpreter will non-deterministically select a goal from the role’s set of active goals. It will then directly call that goal’s associated capability, with the specified parameters, by looking it up in the GCM associated with the role. The GCM provides a one-to-one mapping between RLGM leaf goals and a role’s abstract capabilities. The role’s abstract capabilities are mapped to the agent’s concrete capabilities.
**Post Conditions:** Either the assigned goal is achieved, obviated, or the agent failed to achieve it.

**Specific Requirements:**

1.7.6.1 SR5.1 [Critical Requirement]
The system shall execute the RLGM according to GMoDS semantics. It will first select a goal from the RLGM’s set of active goals and pursue it.

1.7.6.2 SR5.2 [Critical Requirement]
If the current goal is a leaf goal, it shall be looked up in the GCM, and the action is performed.

1.7.6.3 SR5.3 [Critical Requirement]
If the current goal is not a leaf goal, then one of its children shall become the active goal.

1.7.7 Use Case 6: Perform Actions

**Description:** This use case describes the actions performed by the agent running the role interpreter while playing its role in pursuit of its goal.

**Pre-Conditions:** The agent is playing a role in pursuit of a goal. The agent’s role has been defined using both an RLGM and a GCM.

**Details:** As an agent operates, it performs actions to help it achieve the goal associated with the role it is playing. During execution, the role interpreter will call role level goals that are mapped to agent capabilities through the GCM. These capabilities represent either hardware or software functions that this agent possesses. These capabilities act as effectors on the agent’s environment, modifying it in some way that helps the agent achieve its goal.

**Post Conditions:** Either the assigned goal is achieved, obviated, or the agent failed to achieve it.

**Specific Requirements:**

1.7.7.1 SR6.1 [Critical Requirement]
The agent shall perform a requested action by making use of its capabilities. These effectors shall modify the environment to help the agent achieve its goal.

1.7.7.2 SR6.2
If an agent’s capability fails while performing an action, it shall report the failure to the organization so that appropriate actions can be taken.

1.7.8 Use Case 7: Report Status

**Description:** This use case describes the reporting the agent performs after attempting to achieve its goal.

**Pre-Conditions:** The agent attempted to achieve its goal by playing its assigned role.
Details: The agent can achieve its goal, the goal can become obsolete, or the agent can fail to achieve its goal due to capability failure, or some other external factor. This status needs to be reported back to OMACS so that new assignments can be given.

Post Conditions: The status of the agent’s goal is reported back to OMACS.

Specific Requirements:

1.7.8.1 SR7.1 [Critical Requirement]
The agent shall report the status of its goal back to the organization after attempting to complete its task.

1.7.8.2 SR7.2
The agent’s status can be one of the following: Goal Achieved, New Goal Triggered, Goal Failure, or Goal Obviated.

1.8 Assumptions
- The user has JRE 1.6 available at runtime.
- The user has the Wumpi World simulator software installed. The version used for this project will be CIS-844-Fall-2010 from the projects.cis.ksu.edu CVS repository.

1.9 Constraints
- Agents in this system will not attempt to play Roles that are not defined by a Role Level Goal Model.
- All Agents will be provided a Goal Capability Map as part of Role assignment.
- Only a simple demonstration of this architecture will be provided using Wumpi World. A more complex simulation using the MACR Cooperative Robotic Organization Simulator (CROS) will not be provided.
- No role blending or decomposition will be attempted in this implementation.

1.10 Environment
- The agent role interpreter will be written in the Java language and it will be compiled using JDK 1.6.
- The agent role interpreter will be developed using the Eclipse IDE.
- The agent role interpreter will be tested using the Wumpi World simulator on the Windows platform using Windows 7.
- The version used for this project will be CIS-844-Fall-2010 from the projects.cis.ksu.edu CVS repository.
2 Chapter 2 - Project Plan

2.1 Introduction

This document serves as the initial project plan for the GMoDS-based Runtime Agent Role Interpreter project. It outlines the basic work breakdown structure, provides a cost estimate using the COCOMO estimation model, and describes the activities to be performed during the elaboration phase of this project.

2.2 Work Breakdown Structure

This project will follow a basic iterative process consisting of three phases: Inception, Elaboration, and Production. Within each phase, requirements gathering, design, and construction activities will occur. The first phase will be completed during the fall of 2010, while the other two phases will be completed during the summer of 2011. Figure 1 below shows a basic work breakdown structure in the form of a Gantt chart.

![Figure 1 – Basic Project Gantt Chart](image)

2.2.1 Inception Phase

During the inception phase of this project, a vision document, a project plan, a Software Quality Assurance (SQA) plan, and an initial prototype will be created. The vision document will provide a project overview and detail the critical software requirements. The project plan will detail the work to be accomplished during each phase of the project. The SQA plan will outline documentation that will be generated as part of the software
project, as well as provide an overview of the standards and conventions to be followed during the development process. Finally, an initial software prototype will be created to demonstrate the feasibility of the project. These work products will be presented to the supervisory committee once the initial prototype has been completed.

The inception phase will end when these deliverables have been approved by the supervisory committee.

2.2.2 Elaboration Phase

During the elaboration phase of this project, the original vision document and project plan documents will be revised according to input received during construction of the initial prototype and from committee members during the project presentation. The complete design of the project architecture will be provided in the form of UML diagrams. In addition, at least one component of the project will be specified using a formal specification methodology. During this phase, a software test plan will also be created to document the project’s testing process. Additionally, two technical inspectors will review the software architecture and provide feedback. Finally, an executable architecture prototype will be constructed and presented to the supervisory committee.

The elaboration phase will end when these documents and the executable architecture prototype have been approved by the supervisory committee.

2.2.3 Production Phase

During the production phase of this project, low-level component design will be completed, and the final software deliverables will be constructed. In addition, each component will be unit tested and the final project will undergo integration testing according to the test plan provided during the elaboration phase. Once testing is completed, a user manual will be written and the final project evaluation will take place.

The production phase will end when the final project deliverables has been presented and approved by the supervisory committee.

2.3 Cost Estimate

2.3.1 COCOMO

The COCOMO software cost estimation model, created by Barry Boehm in 1981, is used to estimate the cost of this project in terms of both effort and time. The GMoDS-based Runtime Agent Role Interpreter will be a stand-alone software component of average complexity. The software will be implemented by a single developer with good experience and with somewhat flexible requirements. Based on the software
classifications provided by the COCOMO model, this project is considered organic. According to this classification, the effort and time estimations are provided by the following equations:

\[
\text{Effort (staff-months)} = 3.2 \times \text{EAF} \times \text{(Size)}^{1.05}
\]

\[
\text{Time (in months)} = 2.5 \times (\text{Effort})^{0.38}
\]

The Size parameter is measured as the estimated number of significant lines of code, in thousands, (KSLOC). Figure 2 provides a size estimate, in significant lines of code, for each component of the system. Based on this table, the size estimate for this project is 4.75 KSLOC.

<table>
<thead>
<tr>
<th>Component</th>
<th>SLOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Launcher</td>
<td>50</td>
</tr>
<tr>
<td>Agents</td>
<td>700</td>
</tr>
<tr>
<td>Capabilities</td>
<td>1000</td>
</tr>
<tr>
<td>Mapping and Navigation</td>
<td>1600</td>
</tr>
<tr>
<td>Goals</td>
<td>100</td>
</tr>
<tr>
<td>Roles</td>
<td>300</td>
</tr>
<tr>
<td>Role Interpreter</td>
<td>1000</td>
</tr>
<tr>
<td><strong>Total:</strong></td>
<td><strong>4750</strong></td>
</tr>
</tbody>
</table>

The EAF parameter is the product of 15 different adjustment factors listed in Figure 3 below. These factors are classified as one of Very Low, Low, Nominal, High, Very High, and Extra High. A particular adjustment factor’s classification determines where within the given range that factor’s value lays.

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Effort Adjustment Factor</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>RELY</td>
<td>Required Reliability</td>
<td>0.75 - 1.40</td>
</tr>
<tr>
<td>DATA</td>
<td>Database Size</td>
<td>0.94 - 1.16</td>
</tr>
<tr>
<td>CPLX</td>
<td>Product Complexity</td>
<td>0.70 - 1.65</td>
</tr>
<tr>
<td>TIME</td>
<td>Execution Time Constraint</td>
<td>1.00 - 1.66</td>
</tr>
<tr>
<td>STOR</td>
<td>Main Storage Constraint</td>
<td>1.00 - 1.56</td>
</tr>
<tr>
<td>VIRT</td>
<td>Virtual Machine Volatility</td>
<td>0.87 - 1.30</td>
</tr>
<tr>
<td>TURN</td>
<td>Computer Turnaround Time</td>
<td>0.87 - 1.15</td>
</tr>
<tr>
<td>ACAP</td>
<td>Analyst Capability</td>
<td>0.71 - 1.46</td>
</tr>
<tr>
<td>AEXP</td>
<td>Applications Experience</td>
<td>0.82 - 1.29</td>
</tr>
<tr>
<td>PCAP</td>
<td>Programmer Capability</td>
<td>0.70 - 1.42</td>
</tr>
<tr>
<td>VEXP</td>
<td>Virtual Machine Experience</td>
<td>0.90 - 1.21</td>
</tr>
<tr>
<td>LEXP</td>
<td>Language Experience</td>
<td>0.95 - 1.14</td>
</tr>
<tr>
<td>MODP</td>
<td>Use of Modern Practices</td>
<td>0.82 - 1.24</td>
</tr>
<tr>
<td>TOOL</td>
<td>Use of Software Tools</td>
<td>0.83 - 1.24</td>
</tr>
<tr>
<td>SCED</td>
<td>Required Development Schedule</td>
<td>1.10 - 1.23</td>
</tr>
</tbody>
</table>

Figure 3 – COCOMO Effort Adjustment Factors
Based upon the COCOMO documentation and experience with similar projects, the following adjustment factor classifications were chosen.

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Classification</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>RELY</td>
<td>Low</td>
<td>0.88</td>
</tr>
<tr>
<td>DATA</td>
<td>Nominal</td>
<td>1.00</td>
</tr>
<tr>
<td>CPLX</td>
<td>High</td>
<td>1.15</td>
</tr>
<tr>
<td>TIME</td>
<td>Nominal</td>
<td>1.00</td>
</tr>
<tr>
<td>STOR</td>
<td>Nominal</td>
<td>1.00</td>
</tr>
<tr>
<td>VIRT</td>
<td>Low</td>
<td>0.87</td>
</tr>
<tr>
<td>TURN</td>
<td>Low</td>
<td>0.87</td>
</tr>
<tr>
<td>ACAP</td>
<td>Low</td>
<td>1.19</td>
</tr>
<tr>
<td>AEXP</td>
<td>Nominal</td>
<td>1.00</td>
</tr>
<tr>
<td>PCAP</td>
<td>High</td>
<td>0.86</td>
</tr>
<tr>
<td>VEXP</td>
<td>High</td>
<td>0.90</td>
</tr>
<tr>
<td>LEXP</td>
<td>High</td>
<td>0.95</td>
</tr>
<tr>
<td>MODP</td>
<td>High</td>
<td>0.82</td>
</tr>
<tr>
<td>TOOL</td>
<td>Nominal</td>
<td>1.00</td>
</tr>
<tr>
<td>SCED</td>
<td>Nominal</td>
<td>1.00</td>
</tr>
</tbody>
</table>

Figure 4 – Selected COCOMO Effort Adjustment Factors

In general, this project has low hardware constraints and reliability requirements. This is a somewhat complex product, as it will be used as an agent architecture within a multiagent system. However, to offset this, a small and experienced development team consisting of myself will handle development of the system. I am confident in my abilities to program within the chosen environment.

Based on these selected adjustment factors, the EAF value is 0.55. Thus, the Effort value evaluates to the following:

\[
\text{Effort} = 3.2 \times 0.55 \times (4.75)^{1.05} = 9.04 \text{ staff-months}
\]

Time, therefore evaluates to the following:

\[
\text{Time} = 2.5 \times (9.04)^{0.38} = 5.77 \text{ months}
\]

Based on my projected timeline for this project, 5.77 months seems like reasonable value, if a little on the high side.

### 2.4 Architecture Elaboration Plan

The following tasks will be completed during the elaboration phase of this project.
2.4.1 Revise Vision Document

The original vision document will be revised to provide a more complete representation of the system use cases and requirements. These changes will be based on knowledge gained during the initial prototype construction, as well as from feedback received from committee members during the first project presentation. The revised vision document will be submitted to the major professor for approval.

2.4.2 Revise Project Plan

The original project plan will be updated to reflect any schedule changes that have taken place during the course of the inception phase. Additional feedback provided by committee members will be incorporated as well. The revised project plan will be submitted to the major professor for approval.

2.4.3 Create Formal Specification

The Role Interpreter module will be formally modeled in OCL/USE and the pre- and post-conditions of the major GoalCapabilityMap and RoleLevelGoalModel classes will be formally specified. This formal specification will be submitted to the major professor for approval.

2.4.4 Create Architectural Design

A complete architectural design will be created using UML diagrams to the component interface level. This design will be reviewed by the project technical inspectors and be submitted to the major professor for approval.

2.4.5 Create Test Plan

A test plan will be created that contains tests to show that the final software deliverables meet the requirements specified in the project vision document. This plan will include unit and integration testing criteria for all critical use cases. This plan will also include a set of test data to simulate real usage scenarios. This test plan will be submitted to the major professor for approval.

2.4.6 Conduct Technical Inspection

The project’s architectural design will be inspected by Shylaja Chippa and Mike Fraka. An inspection checklist will be developed and provided to the technical inspectors. Any feedback provided by the inspectors will be reviewed and incorporated into the project’s work products. The inspection checklist and reviewer feedback will be submitted to the major professor for approval.
2.4.7 Create Executable Architecture Prototype

An executable architecture prototype will be constructed to demonstrate that the provided architecture is suitable to satisfy all critical use cases outlined in the project vision document. This architectural prototype will be demonstrated to the supervisory committee for comment and approval.

2.5 Implementation Plan

2.5.1 Deliverables

The following deliverables will be delivered during the production phase of this project.

2.5.2 Action Items

Any corrective actions or additional action items identified during the second presentation will be performed.

2.5.3 Technical Inspection Letters

The project’s architectural design will be inspected by Shylaja Chippa and Mike Fraka. Any feedback provided by the inspectors will be reviewed and incorporated into the project’s work products. The students will provide letters stating that they have reviewed the project’s design and that no major faults were found.

2.5.4 Component Design Document

A detailed design of each system component will be provided, including interface documentation, pre- and post-conditions. Class diagrams, State charts, and Sequence diagrams will also be provided.

2.5.5 User Manual

A user manual will be provided, detailing how to start the demonstration system, what command-line parameters are accepted, and what the expected behavior of the system is.

2.5.6 Source Code
Source code for the final project implementation will be provided in a .zip archive. The source code is also housed in the CIS projects CVS server at cvs.projects.cis.ksu.edu.

2.5.7 Assessment Evaluation

An evaluation of the results of project testing will be provided. This evaluation will document how the system performs on each test case outlined in Test Plan 1.0. This evaluation will include reliability and failure estimates.

2.5.8 Project Evaluation

A project review will be provided that give details about the delivered product and the process used to create it. The process review will contain cost estimate reviews, technical review usefulness, and methodology evaluation. The product review will state whether the delivered system meets its functional requirements, as outlined in the Vision Document 2.0.

2.5.9 References

A detailed bibliography of any works cited or used in the project will be provided.

2.6 Work Breakdown Structure

<table>
<thead>
<tr>
<th>Deliverable</th>
<th>Task</th>
<th>Completion Criteria</th>
<th>Time</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source Code</td>
<td>Develop Launcher Component</td>
<td>Executable Code</td>
<td>6/24</td>
<td>0.5 days</td>
</tr>
<tr>
<td></td>
<td>Develop Agent Component</td>
<td>Executable Code</td>
<td>6/24-6/25</td>
<td>2 days</td>
</tr>
<tr>
<td></td>
<td>Develop Capability Component</td>
<td>Executable Code</td>
<td>6/26-6/27</td>
<td>2 days</td>
</tr>
<tr>
<td></td>
<td>Develop Mapping Component</td>
<td>Executable Code</td>
<td>6/28-7/1</td>
<td>3 days</td>
</tr>
<tr>
<td></td>
<td>Develop Role Component</td>
<td>Executable Code</td>
<td>7/1</td>
<td>0.5 days</td>
</tr>
<tr>
<td></td>
<td>Develop Goal Component</td>
<td>Executable Code</td>
<td>7/2</td>
<td>0.5 days</td>
</tr>
<tr>
<td></td>
<td>Develop Role</td>
<td>Executable Code</td>
<td>7/2-7/7</td>
<td>5 days</td>
</tr>
<tr>
<td>Component Design</td>
<td>Create UML Documents</td>
<td>All Components Diagrammed</td>
<td>7/8</td>
<td>1 day</td>
</tr>
<tr>
<td>------------------</td>
<td>----------------------</td>
<td>---------------------------</td>
<td>-----</td>
<td>------</td>
</tr>
<tr>
<td></td>
<td>Create Design Document</td>
<td>Approved by Major Professor</td>
<td>7/9-7/10</td>
<td>2 days</td>
</tr>
<tr>
<td>Testing/Assessment Evaluation</td>
<td>Develop Test Cases</td>
<td>Test Cases Written for All Use Cases</td>
<td>7/11-7/12</td>
<td>2 days</td>
</tr>
<tr>
<td></td>
<td>Evaluate Tests</td>
<td>All Test Cases Evaluated</td>
<td>7/12</td>
<td>0.5 days</td>
</tr>
<tr>
<td>User Manual</td>
<td>Write User Manual</td>
<td>Approved by Major Professor</td>
<td>7/13-7/14</td>
<td>2 days</td>
</tr>
<tr>
<td>References</td>
<td>Write References Document</td>
<td>Approved by Major Professor</td>
<td>7/15</td>
<td>0.5 days</td>
</tr>
<tr>
<td>Technical Inspection Letters</td>
<td>Send and Receive Inspection Letters</td>
<td>Approved by Major Professor</td>
<td>7/15</td>
<td>0.5 days</td>
</tr>
<tr>
<td>Project Evaluation</td>
<td>Evaluate Product and Process</td>
<td>Evaluation Completed</td>
<td>7/16-7/17</td>
<td>2 days</td>
</tr>
<tr>
<td></td>
<td>Complete Portfolio</td>
<td>Approved by Major Professor</td>
<td>7/18-7/25</td>
<td>7 days</td>
</tr>
</tbody>
</table>
3  Chapter 3 - Software Quality Assurance Plan

3.1 Purpose
This document serves as the Software Quality Assurance (SQA) plan for the GMoDS-based Runtime Agent Role Interpreter. This project is the Master of Software Engineering final project for Kyle Hill.

3.2 Management

3.2.1 Management Organization
The GMoDS-based Runtime Agent Role Interpreter project will have the following management organization.

3.2.2 Supervisory Committee
- Dr. Scott DeLoach
- Dr. Gurdip Singh
- Dr. William Hsu

3.2.3 Major Professor
- Dr. Scott DeLoach

3.2.4 Developer
- Kyle Hill

3.2.4.1 Technical Inspectors
- Mike Fraka
- Shylaja Chippa

3.3 Tasks
All project tasks are scheduled and defined within the Project Plan.

3.4 Roles and Responsibility

3.4.1 Supervisory Committee
The supervisory committee will be responsible for attending all three project presentations. Each project presentation will be presented by the developer. The
committee will provide guidance and feedback to the developer at the conclusion of each presentation.

3.4.2 Major Professor

The major professor will be responsible for monitoring and guiding the developer throughout the project.

3.4.3 Developer

The developer will be responsible for creating all project documentation and deliverables. In addition, the developer will present a summary of the state of all project documentation and deliverables at the conclusion of each development phase.

3.4.4 Technical Inspectors

The technical inspectors will be responsible for conducting a formal review of the project’s architectural design. They will review the design according to a review checklist provided by the developer. The technical inspectors will then return a letter of inspection to the major professor and developer.

3.5 Documentation

All project documentation will be published at http://people.cis.ksu.edu/~kylhill/.

3.5.1 Purpose

The project documentation will serve as the primary source of information about the project. It will also provide information about the current state of the project at any given time.

3.5.2 Minimum Documentation Requirement

The following documents will be produced during each phase of development:

Phase 1
- Time Log
- Vision Document 1.0
- Project Plan
- SQA Plan
- Presentation 1
Phase 2
- Time Log
- Vision Document 2.0
- Project Plan 2.0
- Architectural Design 1.0
- Formal Requirements Specification
- Technical Inspection Checklist
- Test Plan
- Presentation 2

Phase 3
- Time Log
- Component Design 1.0
- Technical Inspection Letters
- User Manual
- Project Evaluation
- Presentation 3

3.6 Standards, Practices, Conventions, and Metrics

The GMoDS-based Runtime Agent Role Interpreter project documentation will follow the standards outlined in the IEEE Software Quality Assurance standards [2][3]. In addition, the project source code will conform to the Java 6 platform conventions and guidelines [4].

3.7 Software Reviews

At the conclusion of each of the three project phases, the developer will present all documents and deliverables to the project committee for review and comment. In addition, a formal technical review of the project architecture will be conducted by the technical inspectors.

3.8 Test

A detailed project test plan will be provided during the second phase of the project. This test plan will outline detailed unit and integration tests to be performed on the GMoDS-based Runtime Agent Role Interpreter

3.9 Problem Reporting and Corrective Action
The developer will be responsible for requesting guidance from the major professor throughout the project. Any discrepancies or concerns the major professor identifies will be communicated to the developer. In addition, the supervisory committee will provide recommendations for corrective actions at the conclusion of each project phase. Finally, technical inspectors will be responsible for reporting any architectural problems they identify during their inspection. The developer will be responsible for implementing any suggestions provided by the project members.

3.10 Tools, Techniques, and Methodologies

The project will be implemented in the Java 6 programming language. The project source code will conform to its conventions and guidelines [4]. All project documentation will be prepared using Microsoft Office 2007, specifically Word, Excel, PowerPoint, Visio, and Project will be used. All software development and testing will take place within the Eclipse 3.6 IDE. All UML documents will be produced using the Argo UML diagramming tool.

3.11 Media Control

All project source code will be kept under version control by the Multiagent and Cooperative Robotics Laboratory (MACR) Content Versioning System (CVS) repository. In addition, a copy of the source code will always be stored locally on the developer’s computer. The final software deliverables and source code will be published to the project site: http://people.cis.ksu.edu/~kylhill/.

3.12 Record Collection, Maintenance, and Retention

All project deliverables and documentation will be stored locally on the developer’s computer, as well as published to the project site: http://people.cis.ksu.edu/~kylhill/.

3.13 Risk Management

The developer will be responsible for identifying project risks and bringing them to the attention of the major professor. The major professor will then provide guidance on how to mitigate the risk, or will assist the student in finding someone with the required expertise who can provide guidance to the developer. Any project risks that are not reported to the major professor will be identified at one of the three project presentations that will be held at the end of each phase.
4 Chapter 4 - Architectural Design

4.1 Introduction

This document provides system design information for the GMoDS-based Runtime Agent Role Interpreter. This interpreter serves as the basis for an agent architecture in an OMACS multiagent system. This document details the component design and interface specification. In addition, it provides a high-level overview of the entire system’s static design. It also provides mid-level design details for each component. However, a full interface specification for each component is not provided. Finally, component interaction during role execution is specified via a sequence diagram.

4.2 Architecture

The overall system architecture is constrained by the existing OMACS and GMoDS frameworks into which the role interpreter and its example agent architecture must fit. The system can be decomposed into four major components: The Agent Architecture, the Capability definitions, the Role Interpreter itself, and finally the OMACS Role Adapters. The core of the system consists of the Role Interpreter and its three constituent parts: The RoleLevelGoalModel, the GoalCapabilityMap, and the RoleInterpreter itself. In addition, an example agent architecture is provided to demonstrate the viability of the Role Interpreter in the WumpiWorld test environment. The remainder of the system provides the components necessary for interaction with the rest of the GMoDS and OMACS frameworks.
4.3 Component Design

Figure 1 – Component Diagram

4.4 Component Interface Specification

Only the public interface exported and used by the Role Interpreter is defined in detail. While other components, such as the Agent Architecture and Capability definitions provide other public interfaces, these are less interesting as they are largely defined by OMACS and GMoDS frameworks with which they interact. In addition, those modules only exist to demonstrate the viability of the core of the system: The Role Interpreter.

GoalCapabilityMap

<table>
<thead>
<tr>
<th>Signature</th>
<th>addMapping( s : String, m : Method, c : Capability )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Adds the given name, method, capability entry to the map</td>
</tr>
<tr>
<td>Pre-conditions</td>
<td>The given string, method, and capability are not null.</td>
</tr>
<tr>
<td>Post-conditions</td>
<td>A new mapping of the given string to method, capability pair has been added to the database</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Signature</th>
<th>invoke( g : String, i : InstanceParameters ) : Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Invokes the given goal name (capability method name) on the object that</td>
</tr>
</tbody>
</table>
maps to the given goal name in the map.

| Pre-conditions | The given string is not null. A mapping whose key matches the given name that takes the given InstanceParameters exists within the map. If no method is found a NoSuchMethodException is thrown. |
| Post-conditions | The method that maps to the given name and formal parameters has been called with the given actual parameters. |

**RoleLevelGoalModel**

| Signature | event( g : InstanceGoal<InstanceParameters>, s : SpecificationEvent ) : InstanceChanges |
| Purpose | Fires the given SpecificationEvent from the given InstanceGoal |
| Pre-conditions | The InstanceGoal and SpecificationEvent are not null |
| Post-conditions | The given event has been fired and the InstanceTree has been updated to reflect the event |

| Signature | getEventsToFire( g : SpecificationGoal, r : Object) : Set<SpecificationEvent> |
| Purpose | Returns a set of SpecificationEvents from the given SpecificationGoal and method invocation return object. This set is the set of events that should be fired based on the invocation return value. |
| Pre-conditions | The given SpecificationGoal is not null |
| Post-conditions | The returned set contains all events that should be fired |

| Signature | getNextInstanceGoal() : InstanceGoal<InstanceParameters> |
| Purpose | Returns a leaf-level InstanceGoal from the set of active InstanceGoals whose preconditions have been met. |
| Pre-conditions | None |
| Post-conditions | A leaf-level InstanceGoal from the set of active InstanceGoals whose preconditions have been met has been returned. |

| Signature | hasActiveInstanceGoals() : Boolean |
| Purpose | Returns true if a call to getNextInstanceGoal would return a non-null value. |
| Pre-conditions | None |
| Post-conditions | True has been returned if there is an active, leaf instance goal to pursue, false otherwise. |

| Signature | reset(i : InstanceParameters) |
| Purpose | Resets the InstanceTree back to its default state. |
| Pre-conditions | None |
| Post-conditions | The InstanceTree has been reset back to its default state. |
### RoleInterpreter

<table>
<thead>
<tr>
<th>Signature</th>
<th><code>execute( a : GaaAgent, g : InstanceGoal&lt;InstanceParameters&gt; )</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Executes this interpreter’s RLGM using the given agent in pursuit of the given top-level goal.</td>
</tr>
<tr>
<td>Pre-conditions</td>
<td>The agent and goal are not null. The given goal is a top-level organizational goal.</td>
</tr>
<tr>
<td>Post-conditions</td>
<td>Exactly one leaf-level active goal from the set of current active goals in the RLGM’s instance tree has been executed and the resulting events from that execution have been fired (updating the RLGM’s instance tree for the next call to execute).</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Signature</th>
<th><code>isDone() : boolean</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Returns true if there are no active instance goals in the RLGM to execute.</td>
</tr>
<tr>
<td>Pre-conditions</td>
<td>None</td>
</tr>
<tr>
<td>Post-conditions</td>
<td>True is returned if there are no active instance goals in the RLGM to execute, false otherwise.</td>
</tr>
</tbody>
</table>

### GaaRole

<table>
<thead>
<tr>
<th>Signature</th>
<th><code>getRoleLevelGoalModel() : RoleLevelGoalModel</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Returns the RoleLevelGoalModel object that defines this Role</td>
</tr>
<tr>
<td>Pre-conditions</td>
<td>None</td>
</tr>
<tr>
<td>Post-conditions</td>
<td>A new RoleLevelGoalModel object that defines this Role has been returned. The RoleLevelGoalModel InstanceTree has been reset back to its initial state.</td>
</tr>
</tbody>
</table>

### GaaAgent

<table>
<thead>
<tr>
<th>Signature</th>
<th><code>getGoalCapabilityMap() : GoalCapabilityMap</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Returns the GoalCapabilityMap that contains goal name to capability method mappings for this Agent’s capabilities.</td>
</tr>
<tr>
<td>Pre-conditions</td>
<td>None</td>
</tr>
<tr>
<td>Post-conditions</td>
<td>The GoalCapabilityMap that contains goal name to capability method mappings for this Agent’s capabilities has been returned.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Signature</th>
<th><code>registerCapability( c : GaaCapability )</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Registers the given capability with this agent’s GoalCapabilityMap</td>
</tr>
<tr>
<td>Pre-conditions</td>
<td>The given capability is not null.</td>
</tr>
<tr>
<td>Post-conditions</td>
<td>The given capability has been registered with this agent’s GoalCapabilityMap.</td>
</tr>
</tbody>
</table>

### GaaCapability
### 4.5 System Analysis

![Diagram]

**Figure 2 – Analysis Class Diagram**

This analysis class diagram captures the basic relations between the core Role Interpreter classes and the rest of the system. The RoleInterpreterImpl contains internal references to the RoleLevelGoalModelImpl and the GoalCapabilityMapImpl. The RoleLevelGoalModelImpl is a delegate to the GMoDS GoalTree, which represents both an InstanceTree and a SpecificationTree. The GoalCapabilityMapImpl is a wrapper around a Java map of String goal names to CapabilityMapEntry objects, which are a simple tuple of a Method and its owning Capability object.
4.6 High-Level Design

Figure 3 – High-Level Class Diagram

This high-level class diagram captures the basic relations and concepts of the entire system: The Role Interpreter, the agent architecture, and the rest of the code required to interface with the OMACS and GMoDS frameworks. The gray classes are existing classes that are used by the system, but not actually provided by it. The GaaRoleImpl is a simple adapter between the OMACS AbstractRolePlan and the RoleInterpreter by associating RoleLevelGoalModels with Roles.

The AbstractGaaAgent serves as the basis for an agent architecture that demonstrates the viability of the Role Interpreter. The AbstractGaaAgent contains a lot of code that is specific to the WumpiWorld demonstration. However, it also contains code to bootstrap the GoalCapabilityMap by registering the agent’s GaaCapability objects with the GoalCapabilityMap. This map is provided to the RoleInterpreter during goal execution. The GaaCapability provides a thin wrapper around standard capabilities that allows for method registration with the GoalCapabilityMap. The rest of the system has already been described with the analysis class diagram.
4.7 Mid-Level Design

4.7.1 Agents

The class diagram for the agent component is shown in Figure 4. The class consists of three classes: The GaaAgent, the GoldDigger, and the HunterKiller. Most of this code is specific to the WumpiWorld demonstration. Agents serve as platforms for the deployment of capabilities into the system. They are responsible for actually playing the roles they are assigned. In this system, all role and capability assignments are done statically at compile time. There is no real reason for this other than to make the demonstration system as simple as possible. The GoldDigger class adds capabilities that are required to perform the FetchGold and ReturnGold roles. Similarly, the HunterKiller class adds capabilities for the KillWumpi role.
4.7.2 Capabilities

The class diagram from the Capability component is shown in Figure 5. This component is almost completely specific to the agent architecture demonstration in Wumpi World. This component contains a number of capabilities that the agents possess and make use of to achieve their assigned roles. Each public capability method is mapped to a leaf-level goal in the Role Level Goal Model through an agent’s GoalCapabilityMap.

The abstract parent class GaaCapability requires child classes to implement the registerMethods() method to provide a mapping from goal name to capability method. The OmacsInterface capability is somewhat special. This capability contains methods that are needed to interact with the rest of the OMACS system. Currently, it only contains the createGoal() method that will instantiate a new instance goal when called.

Figure 5 – Capability Package Class Diagram
4.7.3 Role Interpreter

The Role Interpreter component is shown in Figure 6. The details of this component have already been given in previous sections. This diagram shows the same relationships between classes and provides information about methods and attributes of those classes.
4.7.4 Roles

The Role component class diagram is shown in Figure 7. This component’s responsibility is to provide an adapter between the Role representation used by OMACS in WumpiWorld, and the Role Interpreter. The parent class, GaaRole provides methods to return a statically defined role priority (to make the agent architecture as simple as possible) and to return the RoleLevelGoalModel that defines the Role’s behavior for the RoleInterpreter.

The RoleLevelGoalModel is defined by a GMoDS Goal Model XML file that is created by the AgentTool3 graphical editor. This XML file is read in at runtime by the RoleLevelGoalModel class. This allows for users of the system to customize its behavior by simply editing the XML file in the graphical editor.

4.7.5 Component Interaction

This system contains a large number of interactions between various components. Startup behavior, top-level goal creation, capability-specific interactions, and role execution are all complex and vital for the agent architecture demonstration. However, the most interesting sequence is the role execution sequence. This sequence is the heart of the system and defines how agent assignments are transformed into actions that are executed through capabilities to accomplish system goals. This sequence’s behavior is independent of the agent architecture and scenario in which it is deployed.
4.8 Role Execution Sequence

Figure 8 – Role Execution Sequence Diagram

Figure 8 shows the interactions required between various objects in order to execute a role using the Role Interpreter.

1. The GaaAgent starts its main execution loop by creating new assignment tasks for itself based on the initial system conditions and any sensor data that is immediately available. This self-task assignment is not strictly required, any other method for assigning tasks to an agent would be sufficient. However, this method keeps the agent architecture as simple as possible.
2. Next the agent selects the “best” task to work on based on some built heuristics. Again, the details of this are specific to the agent architecture. In this system, the highest priority task is selected. If two tasks share the same priority, then the one whose goal is the “closest” is chosen first.
3. The agent then gets the plan associated with the assignment’s role. This causes a lazy instantiation of a RoleInterpreter and its associated RoleLevelGoalModel.
4. Then, while the plan is not done, execute is called repeatedly by the agent
5. The RoleInterpreter gets an active leaf-level instance goal from the instance tree to execute.
6. The RoleInterpreter also gets the GoalCapabilityMap from the calling agent.
7. Next, the RoleInterpreter actually invokes the capability that is associated with the active goal using the GoalCapabilityMap.
8. The RoleInterpreter calls into the RoleLevelGoalModel with the return value of the capability call to determine what events to fire to cause the correct changes in the instance tree.
9. Then, for each event that is to be fired, the RoleInterpreter actually triggers the event in the instance tree.
10. Finally, based on the return value from the capability method, the current goal is either marked as achieved or failed.

4.9 Role Models

4.9.1 Area Searcher

4.9.2 Gold Fetcher
4.9.3 Gold Returner

- **Goal**: 0 ReturnGold
  - loc : LocationData

- **Goal**: 1 setDestinationLocation
  - loc : LocationData

- **Goal**: 2 isAtDestinationLocation
  - true() -> **Goal**: 3 move
  - false() -> **Goal**: 4 readSensors
  - triggers() -> **Goal**: 5 putGoldIntoBin

4.9.4 Hunter-Killer

- **Goal**: 0 KillWumpi
  - loc : LocationData

- **Goal**: 1 setTarget
  - loc : LocationData

- **Goal**: 2 SetupFiringLocation
  - triggers() -> **Goal**: 3 isWumpiAtLocation
  - loc : LocationData

- **Goal**: 4 isAtDestinationLocation
  - true() -> **Goal**: 5 fire
  - loc : LocationData

- **Goal**: 2.1 hasFiringLocation
- **Goal**: 2.2 getNextFiringLocation
  - triggers(firingLoc)
5 Chapter 5 - Formal Specification

-----------------------------------------------------------------------------
-- GMoDS Based Agent Architecture
--
-- This file contains a formal specification of the invariants maintained by
-- the getEventsToFire method from the RoleLevelGoalModel
--
-- File:   GMoDSAgentArchitecture.use
-- Author: Kyle Hill
-- Date:   June 20, 2011
-----------------------------------------------------------------------------

model GMoDSAgentArchitecture

-----------------------------------------------------------------------------
-- Classes
-----------------------------------------------------------------------------
class Object
end

-----------------------------------------------------------------------------
-- GAA Classes
-----------------------------------------------------------------------------
class RoleInterpreter
end
class RoleLevelGoalModel
operations
getEventsToFire(g : ParameterizedSpecificationGoal, r : Boolean) :
Set(SpecificationEvent)
getNextInstanceGoal() : InstanceGoal
end
class GoalCapabilityMap
operations
addMapping(s : String, e : CapabilityMapEntry)
invoke(g : ParameterizedSpecificationGoal, p : InstanceParameters)
end
class CapabilityMapEntry
attributes
   id : String
end
class Method
attributes
   name : String
   return : Object
end
class Capability
end

-----------------------------------------------------------------------------
-- GMoDS Classes
-----------------------------------------------------------------------------
class GoalTree
end
class SpecificationTree
end

class SpecificationEvent
attributes
  id : String
end

class ParameterizedSpecificationGoal
attributes
  id : String
  isLeaf : Boolean
end

class SpecificationParameters
end

class SpecificationParameter
attributes
  key : String
end

class InstanceTree
end

class InstanceGoal
attributes
  id : String
end

class InstanceParameters
end

class InstanceParameter
attributes
  key : String
  value : Object
end

-------------------------------------------------------------------------------
-- Associations
-------------------------------------------------------------------------------

-- GAA Associations

association RLGM between
  RoleInterpreter[1]
  RoleLevelGoalModel[1] role rlgm
end

association RLGMGoalTree between
  RoleLevelGoalModel[1]
  GoalTree[1] role goalTree
end

association GCM between
  RoleInterpreter[1]
  GoalCapabilityMap[1] role gcm
end

association MapEntries between
GoalCapabilityMap[1]
   CapabilityMapEntry[0..*] role entries
end

association EntryMethod between
   CapabilityMapEntry[0..*]
   Method[1] role method
end

association EntryCapability between
   CapabilityMapEntry[0..*]
   Capability[1] role capability
end

association MethodParams between
   Method[1] role method1
   Object[0..*] role params
end

-- GMoDS Associations
association SpecTree between
   GoalTree[1]
   SpecificationTree[1] role specTree
end

association SpecGoals between
   SpecificationTree[1]
   ParameterizedSpecificationGoal[1..*] role goals
end

association SpecGoalParam between
   ParameterizedSpecificationGoal[1]
   SpecificationParameters[0..1] role param
end

association SpecEvents between
   SpecificationTree[1]
   SpecificationEvent[0..*] role events
end

association SpecEventParams between
   SpecificationEvent[1]
   SpecificationParameters[0..1] role param
end

association SpecParams between
   SpecificationParameters[1]
   SpecificationParameter[0..*] role params
end

association InstTree between
   GoalTree[1]
   InstanceTree[1] role instTree
end

association ActiveInstGoals between
   InstanceTree[1]
   InstanceGoal[0..*] role activeGoals
end

association InstGoalParam between


InstanceGoal[1]
InstanceParameters[0..1] role param
end

association InstParams between
  InstanceParameters[1]
  InstanceParameter[0..*] role params
end

-------------------------------------------------------------------------------
-- Constratints
-------------------------------------------------------------------------------

context RoleLevelGoalModel::getEventsToFire(g : ParameterizedSpecificationGoal, r : Boolean) : Set(SpecificationEvent)

-- The given specification goal must exist within the specification tree and be
-- unique
pre GoalInTree:
  goalTree.specTree.goals->select(id = g.id)->size() = 1

-- If the return value is not boolean, then all goal model specified events are
-- returned
post NoBoolReturnsAll:
  r.isUndefined() implies goalTree.specTree.events = result

-- If the return value is true, then all the "true" prefixed events are returned.
-- If no events are prefixed with "true", then all unconditional events are returned.
-- Otherwise, if the return value is false, then all the "false" prefixed events are returned.
-- If no events are prefixed with "false", then all unconditional events are returned.
post CorrectEventsReturned:
  let allEvents   : Set(SpecificationEvent) = goalTree.specTree.events in
  let trueEvents  : Set(SpecificationEvent) = allEvents->select(id.toLower().substring(1, 4) = 'true') in
  let falseEvents : Set(SpecificationEvent) = allEvents->select(id.toLower().substring(1, 5) = 'false') in
  let unconEvents : Set(SpecificationEvent) = allEvents - trueEvents - falseEvents in
  if r then
    if trueEvents->isEmpty() then
      result = unconEvents
    else
      result = trueEvents
    endif
  else
    if falseEvents->isEmpty() then
      result = unconEvents
    else
      result = falseEvents
    endif
  endif

context RoleLevelGoalModel::getNextInstanceGoal() : InstanceGoal

-- The returned instance goal is either null, or it is in the set of active
-- goals and it is a leaf goal
post NullOrActiveLeaf:
    result.isUndefined() or
    (goalTree.instTree.activeGoals->includes(result) and
     goalTree.specTree.goals->select(id = result.id and isLeaf)->notEmpty())

-- GoalCapabilityMap Constraints
-----------------------------------------------------------------------------
context GoalCapabilityMap::addMapping(s : String, e : CapabilityMapEntry)
-- The given mapping contains no null values
pre NoNulls:
    not s.isUndefined() and
    not e.isUndefined() and
    not e.method.isUndefined() and
    not e.capability.isUndefined()

-- The entry has been added to the mapping
post EntryAdded:
    entries = entries@pre->including(e) and
    e.id = s

call GoalCapabilityMap::invoke(g : ParameterizedSpecificationGoal, p : InstanceParameters)
-- The given goal is not undefined
pre NoNulls:
    not g.isUndefined()

-- A mapping is already present in the GCM for this goal
pre MappingExists:
    entries->exists(id = g.id)

-- The specification parameters and instance parameters are null, or
-- There exists an instance parameter for each specification parameter of the goal
pre FormalParamsMatch:
    let specificationParams = g.param.params in
    let instanceParams = p.params in
    ((g.param.isUndefined() or specificationParams->isEmpty()) and
     (p.isUndefined() or instanceParams->isEmpty())) or
    specificationParams->forall(fp : SpecificationParameter | instanceParams->exists(ap : InstanceParameter | fp.key = ap.key))
6 Chapter 6 - Technical Inspection Checklist

6.1 Introduction
This document provides a list of items in the GMoDS-based Runtime Agent Role Interpreter project that are to be inspected. The technical inspectors should follow this document performing their inspection to make sure that all major areas of the project have been covered. The technical inspection will be performed to ensure the correctness and consistency of the architectural design and formal design documents.

6.2 Items to be Inspected
The following items will be inspected for correctness and consistency:
1. System Architecture Design
   a. Component Design (Section 2.1)
   b. Component Interface Specification (Section 2.2)
   c. Analysis Class Diagram (Section 2.3)
   d. High-Level Class Diagram (Section 2.4)
   e. Agent Architecture (Section 3.1)
   f. Capabilities (Section 3.2)
   g. Role Interpreter (Section 3.3)
   h. Roles (Section 3.4)
   i. Component Interaction (Section 4.1)
2. Formal Specification

6.3 Technical Inspectors
- Mike Fraka
- Shylaja Chippa

6.4 Technical Inspection Checklist

<table>
<thead>
<tr>
<th>Inspection Item</th>
<th>Pass/Fail/Partial</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>The symbols used in class diagrams conform to UML 2.0.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>The symbols used in sequence diagrams conform to UML 2.0.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>The symbols used in component diagrams conform to UML 2.0.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>The Component Interface</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Specification in Section 2.2 clearly defines the major public operations on the system.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td></td>
<td></td>
</tr>
<tr>
<td>The Analysis Class Diagram in Figure 2 clearly describes the high-level relationships between model elements.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>The High-Level Class Diagram in Figure 3 correctly shows how this system interfaces with the OMACS and GMoDS frameworks.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>The provided descriptions for high-level components are clear and accurate.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>The provided component diagrams are clear and accurate.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>The provided component descriptions are clear and accurate.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>The description of the Role Execution Sequence is accurate and correct.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>The provided USE/OCL model classes, attributes, and relations are consistent with the high-level class model in Figure 3.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>The RoleLevelGoalModel constraints are correctly specified and understandable.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>The GoalCapabilityMap constraints are correctly specified and understandable.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Table 1 – Technical Inspection Checklist**
7 Chapter 7 - Test Plan

7.1 Test Plan Identifier

GAA-TestPlan-1.0

7.2 Introduction

This document describes the testing that will be performed on the GMoDS-based Runtime Agent Role Interpreter. This project provides a generic Role Interpreter that allows GMoDS models to define role plans. These models are read in at runtime and then executed by mapping leaf-level goals to concrete capability methods. This system consists of three basic parts: The Role Interpreter, the Demonstration Agent Architecture, and the interface required to fit into the existing OMACS and GMoDS frameworks. This testing will be performed in accordance with the guidelines listed in the Project Plan 2.0 and the Software Quality Assurance Plan 1.0.

7.3 Test Items

The following items will be tested. The requirements specification for these items is provided in Vision Document 2.0, and the architectural design for each item is provided in System Architecture Design 1.0:

- Role Interpreter Framework
  - RoleInterpreter
  - GoalCapabilityMap
  - RoleLevelGoalModel
  - GaaRole
  - GaaCapability
  - GaaAgent
  - OmacsInterface

- Demonstration Agent Architecture
  - GoldDigger
  - HunterKiller
  - AdvancedBazooka
  - AdvancedGoldGrabber
  - AdvancedMovement
  - AdvancedSensors
  - AreaSearcherRole
  - GoldFetcherRole
  - GoldReturnerRole
7.4 Features to Be Tested
The following items, outlined in Vision Document 2.0, will be tested.

7.4.1 SR1.1
The system shall accept a GMoDS RLGM and GCM as input representing the role it will play.

7.4.2 SR1.2
The system shall accept a GMoDS goal specification as input representing the goal it will achieve. The goal specification shall include goal type information and values for each attribute.

7.4.3 SR1.3
The system shall accept the RLGM input according to the Goal Model XML schema used by the agentTool3 software.

7.4.4 SR1.4
The system shall accept the GCM input according to an XML schema that will be used by a future version of the agentTool3 software. The GCM schema definition does not exist yet and it will be provided as part of this project.

7.4.5 SR2.1
The system shall parse the input RLGM into a goal tree structure using the GMoDS parser.

7.4.6 SR3.1
The system shall parse the input GCM into a map structure with leaf-level goals serving as keys, and concrete agent capabilities serving as values.

7.4.7 SR4.1
The system shall read the set of required capabilities out of the assigned role. These capabilities will be the union of all capabilities discovered by looking up all leaf-level goals in the role’s RLGM.
7.4.8 SR4.2
The system shall iterate over all required capabilities from the role and check to make sure that each capability is in the set of capabilities possessed by the agent.

7.4.9 SR5.1
The system shall execute the RLGM according to GMoDS semantics. It will first select a goal from the RLGM’s set of active goals and pursue it.

7.4.10 SR5.2
If the current goal is a leaf goal, it shall be looked up in the GCM, and the action is performed.

7.4.11 SR6.1
The agent shall perform a requested action by making use of its capabilities. These effectors shall modify the environment to help the agent achieve its goal.

7.4.12 SR7.1
The agent shall report the status of its goal back to the organization after attempting to complete its task.

7.5 Features Not to Be Tested

7.5.1 SR4.3
If the system does not possess a required capability, it shall inform the organization so that it can take appropriate action, such as assigning a new role to this agent.

The demonstration architecture has the set of capabilities hardcoded in agents at compile time. There is also no organization to report the problem to. This has been done purposefully to simplify the demonstration architecture, as the primary purpose of this project is to implement the Role Interpreter.

7.5.2 SR5.3
If the current goal is not a leaf goal, then one of its children shall become the active goal.

This is an inherent property of the GMoDS model. There is nothing here to test.
7.5.3 SR6.2
If an agent’s capability fails while performing an action, it shall report the failure to the organization so that appropriate actions can be taken.

In the demonstration architecture, capabilities cannot fail. There is also no organization to report the problem to. This has been done purposefully to simplify the demonstration architecture, as the primary purpose of this project is to implement the Role Interpreter.

7.5.4 SR7.2
The agent’s status can be one of the following: Goal Achieved, New Goal Triggered, Goal Failure, or Goal Obviated.

This is an inherent property of the GMoDS model. There is nothing here to test.

7.6 Approach
Only functional black box testing will be performed on this project. A WumpiWorld scenario has been prepared and it will be executed by the demonstration agent architecture. This architecture internally makes use of the Role Interpreter on four different roles defined using GMoDS models in the agentTool3 XML format. In addition, a GCM exists for each of five different capability classes.

Since successful execution of any one of these GMoDS models requires the successful execution of all system requirements, we will run the system through a series of increasingly more complex WumpiWorld scenarios and provide scoring and completion requirements for each scenario.

The WumpiWorld scenario is scored to reflect how well the agents in the environment accomplish their assigned roles. The fewer capabilities used and messages sent will result in a higher score. Additionally, if agents are able to avoid traps, kill Wumpi adversaries, and recover gold, their score will be higher. In general, the higher the score, the better the agents were able to perform their roles.

7.7 Item Pass/Fail Criteria
Tests will be considered passed if they meet the specification requirement outlined in Vision Document 2.0, otherwise, they are considered to have failed.
7.8 Suspension Criteria and Resumption Requirements

7.8.1 Suspension Criteria
If a test case fails, then testing will be suspended for all features that depend upon that test case. The failed test case will be logged, along with a description of the failure. Other test cases that do not depend upon the failed feature will continue.

7.8.2 Resumption Criteria
For failed test cases, testing will resume once the defect that caused the original failure has been resolved.

7.9 Test Deliverables

7.9.1 Test Log
A test log will be produced, documenting all test cases and if the test case passed or failed. If a test case fails, then a description of the failure will also be logged and a corrective action will be taken, if appropriate.

7.10 Testing Tasks

7.10.1 Test Case 1

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Features Tested</th>
<th>Passing Criteria</th>
</tr>
</thead>
<tbody>
<tr>
<td>HunterKiller.xml</td>
<td>All</td>
<td>At least three Wumpi are killed. No agent falls into a pit or is killed by Wumpi. Minimum score: 19000.</td>
</tr>
</tbody>
</table>

7.10.2 Test Case 2

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Features Tested</th>
<th>Passing Criteria</th>
</tr>
</thead>
<tbody>
<tr>
<td>HunterKillerTeam.xml</td>
<td>All</td>
<td>At least five Wumpi are killed. No agent falls into a pit or is killed by Wumpi. Minimum score: 23000.</td>
</tr>
</tbody>
</table>

7.10.3 Test Case 3

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Features Tested</th>
<th>Passing Criteria</th>
</tr>
</thead>
<tbody>
<tr>
<td>GoldDigger.xml</td>
<td>All</td>
<td>At least two pieces of gold are successfully recovered. No agent falls into a pit or is killed by Wumpi. Minimum score: -2000.</td>
</tr>
</tbody>
</table>

7.10.4 Test Case 4

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Features Tested</th>
<th>Passing Criteria</th>
</tr>
</thead>
<tbody>
<tr>
<td>GoldDiggerHunterKillerTeam.xml</td>
<td>All</td>
<td>At least three pieces of gold are</td>
</tr>
</tbody>
</table>
successfully recovered and three Wumpi are killed. No agent falls into a pit or is killed by Wumpi. Minimum score: 22000.

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Features Tested</th>
<th>Passing Criteria</th>
</tr>
</thead>
<tbody>
<tr>
<td>WumpiWorld.xml</td>
<td>All</td>
<td>At least four pieces of gold are successfully recovered and five Wumpi are killed. No agent falls into a pit or is killed by Wumpi. Minimum score: 25000.</td>
</tr>
</tbody>
</table>

### 7.11 Environment Needs

The Role Interpreter and demonstration agent architecture will be tested on a Windows 7 computer running the Java JDK 1.6. The demonstration architecture will be executed within the GMoDS and OMACS frameworks defined by the CIS-844-Fall-2010 WumpiWorld archive from the projects.cis.ksu.edu CVS repository.
Chapter 8 - Component Design

8.1 Introduction

This document provides component design information for the GMoDS-based Runtime Agent Role Interpreter. This interpreter serves as the basis for an agent architecture in an OMACS multiagent system. This document details component-level detailed design information and interface specifications. In addition, this document provides a full description of the method, attributes, and other properties of all system classes.

8.2 Architecture

The overall system architecture is constrained by the existing OMACS and GMoDS frameworks into which the role interpreter and its example agent architecture must fit. The system can be decomposed into four major components: The Agent Architecture, the Capability definitions, the Role Interpreter itself, and finally the OMACS Role Adapters. The core of the system consists of the Role Interpreter and its three constituent parts: The RoleLevelGoalModel, the GoalCapabilityMap, and the RoleInterpreter itself. In addition, an example agent architecture is provided to demonstrate the viability of the Role Interpreter in the WumpiWorld test environment. The remainder of the system provides the components necessary for interaction with the rest of the GMoDS and OMACS frameworks.
Figure 1 – High-Level Class Diagram

Figure 1 shows a high-level class diagram of the most important components of the system. Each component will be described in detail later in this document. This diagram serves to illustrate how the various pieces of the system fit together. This system is designed to integrate seamlessly with the existing OMACS and GMoDS frameworks. The RoleInterpreter itself extends directly from the ExecutionPlan class provided by OMACS. Similarly, the RoleLevelGoalModel internally delegates its public API to a GoalTree provided by GMoDS. The capabilities that are used in the GoalCapabilityMap are special versions of AbstractCapabilityActions provided by OMACS. The only difference is that these classes have annotated parameters so that name information is available at runtime for GMoDS parameter to Java parameter mapping.

8.3 Component Design

The following sections provide a detailed description of the component-level design for each major component of the system. This description includes a detailed class diagram of all class methods and attributes, a description of each class, and an interface definition for the major operations of each package.
8.4 Agents

8.4.1 Description

The agent package consists of three primary classes: AbstractGaaAgent, GoldDigger, and HunterKiller. The public interface exported to the rest of the system is declared in the GaaAgent interface. Most of this code is specific to the WumpiWorld demonstration. Agents serve as platforms for the deployment of capabilities into the system. They are responsible for actually playing the roles they are assigned. In this system, all role and capability assignments are done statically at compile time. There is no real reason for this other than to make the demonstration system as simple as possible. Future implementations of this system could include a more robust organization that would allow assignments to be made dynamically based on agent capabilities and other environmental factors.
8.4.2 Class Diagram

Figure 2 – Agents Package Class Diagram

8.4.3 Class Descriptions

8.4.3.1 AbstractGaaAgent

The AbstractGaaAgent serves as the common base class for all agent implementations. It also serves as the primary class of the demonstration agent architecture. It is responsible for constructing and housing the GoalCapabilityMap for the agent, and for actually invoking the RoleInterpreter with the agent’s assigned Role.

In this system, tasks are self-assigned by the agents. The AbstractGaaAgent creates tasks for the agent by interpreting shared map data. These tasks are added to the set of active tasks for the agent to consider performing. Tasks that have been achieved or failed by the
agent are removed from consideration. When the agent goes to perform its next task, it chooses the “best” task to work on selecting the one with the highest priority whose goal destination is closest to the agent’s current location.

8.4.3.2 HunterKiller

The HunterKiller agent is responsible for searching the map for Wumpi to kill. It contains special code to instantiate new KillWumpi goals whenever a new, unclaimed Wumpi is discovered on the map. This is the only agent type who can play the KillWumpiRole.

8.4.3.3 GoldDigger

The GoldDigger agent is responsible for searching the map for gold, retrieving it, and then returning it to the bin at the start location on the map. It contains special code to instantiate new FetchGold and ReturnGold goals whenever new, unclaimed gold is discovered on the map, or when gold has been picked up. This is the only agent type who can play the FetchGoldRole and the ReturnGoldRole.

8.4.4 Interface Specification

GaaAgent

<table>
<thead>
<tr>
<th>Signature</th>
<th>getGoalCapabilityMap() : GoalCapabilityMap</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Returns the GoalCapabilityMap that contains goal name to capability method mappings for this Agent’s capabilities.</td>
</tr>
<tr>
<td>Pre-conditions</td>
<td>None</td>
</tr>
<tr>
<td>Post-conditions</td>
<td>The GoalCapabilityMap that contains goal name to capability method mappings for this Agent’s capabilities has been returned.</td>
</tr>
</tbody>
</table>

8.5 Capabilities

8.5.1 Description

This component is almost completely specific to the agent architecture demonstration in Wumpi World. This component contains a number of capabilities that the agents possess and make use of to achieve their assigned roles. Each public capability method is mapped to a leaf-level goal in the RoleLevelGoalModel through an agent’s GoalCapabilityMap. Each public capability method in this component is annotated with a custom @Name annotation on each parameter. This annotation contains the name of the parameter and allows the GoalCapabilityMap to intelligently map GMoDS instance parameters to Java method parameters by matching arguments with the same name.
The interface GaaCapability does not export any public methods, but simply serves as an indication that implementing classes have had their public methods appropriately annotated. Future versions of this system could use these annotations to hook into the Java compiler and provide compile-time verification that all parameters have been correctly annotated and that the annotations match the expected name.

### 8.5.2 Class Diagram

Figure 3 – Capabilities Package Class Diagram

### 8.5.3 Class Descriptions

#### 8.5.3.1 AdvancedBazooka

The AdvancedBazooka capability extends the existing Bazooka capability with targeting capabilities and ammo counting. When a new target is set on the capability, a set of prioritized firing locations is generated. The agent should navigate to each of these, in turn, and try to fire upon the target Wumpi. At the first targeting location the agent can fire from, the agent should take the shot and then sense to see if the target Wumpi has been killed.
8.5.3.2 AdvancedCommunication

The AdvancedCommunication capability extends the existing CommunicationsImpl capability. No new abilities are added through this extension, but all public methods are annotated so that they can be executed through the GoalCapabilityMap. This capability can send or receive messages on specific channels, or send and receive broadcast messages. It is used by this system to synchronize map information between agents.

8.5.3.3 AdvancedGoldGrabber

The AdvancedGoldGrabber capability extends the existing GoldGrabber capability. It does not add any new abilities, but simply annotates the GoldGrabber’s public methods so that they can be executed through the GoalCapabilityMap. This capability can grab gold, report if it has gold, and release gold.

8.5.3.4 AdvancedMovement

The AdvancedMovement capability wraps the more primitive RobotMovement and GPSImpl and extends them with advanced mapping and route planning capabilities. Together with the AdvancedSensors, these capabilities make up the core of the capabilities used by the demonstration agent architecture. AdvancedMovement is used by every Role to determine the agent’s current location, set goal movement locations, plan routes between locations, and then actually execute those routes.

8.5.3.5 AdvancedSensors

The AdvancedSensors capability internally wraps the Breeze, Smell, and Sparkle sensors. It also encapsulates the sonar capability. It is responsible for aggregating sensor input from a variety of sources, interpreting it, and then storing it in the agent’s map database. It is also responsible for sharing this map information with other agents, and merging their map data with the agent’s map data.

8.5.3.6 OmacsInterface

The OmacsInterface is a special capability that allows RoleLevelGoalModels to communicate with the rest of the OMGCS system. It currently only allows for the construction and assignment of a single ReturnGoldGoal. However, in the future it could be extended to provide generic OMGCS interface capabilities, such as role assignment, failure reporting, and reorganization. All of these tasks could be executed through existing RoleLevelGoalModels.
8.6 Role Interpreter

8.6.1 Description

The Role Interpreter component is the heart of the system. It is responsible for parsing the role plan definitions, constructing a runtime GoalTree, executing that GoalTree through the GoalCapabilityMap, and then reporting the status of the role execution to the rest of the system. The module consists of four major classes: The RoleLevelGoalModelImpl, the RoleInterpreterImpl, the GoalCapabilityMapImpl, and the GoalCapabilityMapFactory. The package exports the RoleLevelGoalModel, RoleInterpreter, and GoalCapabilityMap interfaces to the rest of the system.

At startup the system constructs a GoalCapabilityMap for each Agent instance using the GoalCapabilityMapFactory to parse the mapping XML. When the agent goes to execute its assigned role, it will lazily initialize a RoleInterpreter, which, in turn, will lazily initialize a new RoleLevelGoalModel by parsing the goal definition using the GMoDS parser. The agent then repeatedly calls execute() on the RoleInterpreter. Each time execute is called, an active goal from the RoleLevelGoalModel is invoked through the GoalCapabilityMap. This causes the capability method that is mapped to the given goal name to be executed through the Java reflection API.

8.6.2 Class Diagram
8.6.3 Class Descriptions

8.6.3.1 RoleLevelGoalModelImpl

The RoleLevelGoalModelImpl is a façade for the GMoDS GoalTree. It exposes a specialized set of methods that are used by the RoleInterpreter to manipulate the GoalTree during Role execution.

The getNextInstanceGoal() method is used to return an arbitrary leaf-level goal from the set of active goals. This is used by the RoleInterpreter to decided which goal should be invoked from its execute method.

The getEventsToFire() method is one of the most important methods in the entire system, as it allows conditional execution to take place. It uses the return value from recently invoked goal, along with the set of outgoing events from a given goal and decided which ones should be fired. The exact semantics of this method are defined in the formal specification.

8.6.3.2 RoleInterpreterImpl

The RoleInterpreterImpl brings the Role’s RoleLevelGoalModel and the Agent’s GoalCapabilityMap together to actually perform Role execution. It is also responsible for reporting the status of the top-level goal that the agent is attempting to achieve by playing this Role.

8.6.3.3 GoalCapabilityMapImpl

The GoalCapabilityMapImpl is a mapping of goal names to CapabilityMapEntry objects. It essentially provides a mapping from GMoDS goals to actual Java methods. When the invoke() method is called with a given InstanceGoal, the GoalCapabilityMap is used to map that InstanceGoal to a specific method on a specific capability object. In addition, the GoalCapabilityMap takes care of matching up the InstanceParameters of an InstanceGoal with the formal parameters required by the method it is trying to invoke.

8.6.3.4 CapabilityMapEntry

The GoalCapabilityMapEntry is a simple tuple object that represents a Method-Object pair in the GoalCapabilityMap. These entries are not really used outside of the GoalCapabilityMapImpl.

8.6.3.5 GoalCapabilityMapFactory
The GoalCapabilityMapFactory class is a factory class that is responsible for constructing a new GoalCapabilityMap from a given XML file and a list of capability objects. It takes care of parsing the XML file, matching capability classes with runtime objects, and determining which methods on those objects should be invoked by which GMoDS goal names. This XML file is formally defined in the GoalCapabilityMap.xsd file.

8.6.4 Interface Specification

GoalCapabilityMap

<table>
<thead>
<tr>
<th>Signature</th>
<th>addMapping(s : String, m : Method, c : Object)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Adds the given name, method, capability entry to the map</td>
</tr>
<tr>
<td>Pre-conditions</td>
<td>The given string, method, and capability are not null.</td>
</tr>
<tr>
<td>Post-conditions</td>
<td>A new mapping of the given string to method, capability pair has been added to the database</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Signature</th>
<th>invoke(g : InstanceGoal) : Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Invokes the given InstanceGoal (method) on the object that maps to the given goal’s name in the map.</td>
</tr>
<tr>
<td>Pre-conditions</td>
<td>The given InstanceGoal is not null. A mapping whose key matches the InstanceGoal’s name that takes the given InstanceGoal’s InstanceParameters exists within the map. If no method is found a NoSuchMethodException is thrown.</td>
</tr>
<tr>
<td>Post-conditions</td>
<td>The method that maps to the given name and formal parameters has been called with the given actual parameters.</td>
</tr>
</tbody>
</table>

RoleLevelGoalModel

<table>
<thead>
<tr>
<th>Signature</th>
<th>event(g : InstanceGoal&lt;InstanceParameters&gt;, s : SpecificationEvent) : InstanceChanges</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Fires the given SpecificationEvent from the given InstanceGoal</td>
</tr>
<tr>
<td>Pre-conditions</td>
<td>The InstanceGoal and SpecificationEvent are not null</td>
</tr>
<tr>
<td>Post-conditions</td>
<td>The given event has been fired and the InstanceTree has been updated to reflect the event</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Signature</th>
<th>getEventsToFire(g : SpecificationGoal, r : Object) : Set&lt;SpecificationEvent&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Returns a set of SpecificationEvents from the given SpecificationGoal and method invocation return object. This set is the set of events that should be fired based on the invocation return value.</td>
</tr>
<tr>
<td>Pre-conditions</td>
<td>The given SpecificationGoal is not null</td>
</tr>
<tr>
<td>Post-conditions</td>
<td>The returned set contains all events that should be fired</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Signature</th>
<th>getNextInstanceGoal() : InstanceGoal&lt;InstanceParameters&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Returns a leaf-level InstanceGoal from the set of active InstanceGoals whose</td>
</tr>
<tr>
<td>Pre-conditions</td>
<td>None</td>
</tr>
<tr>
<td>----------------</td>
<td>------</td>
</tr>
<tr>
<td>Post-conditions</td>
<td>A leaf-level InstanceGoal from the set of active InstanceGoals whose preconditions have been met has been returned.</td>
</tr>
</tbody>
</table>

**Signature**

<table>
<thead>
<tr>
<th>getRootStatus() : Status</th>
</tr>
</thead>
</table>

**Purpose**

Returns the execution status of the root goal

**Pre-conditions**

None

**Post-conditions**

ACTIVE is returned if the root goal is still being pursued. ACHIEVED is returned if the root goal is achieved. FAILED is returned if the root goal was failed to be achieved.

**Signature**

<table>
<thead>
<tr>
<th>hasActiveInstanceGoals() : Boolean</th>
</tr>
</thead>
</table>

**Purpose**

Returns true if a call to getNextInstanceGoal would return a non-null value.

**Pre-conditions**

None

**Post-conditions**

True has been returned if there is an active, leaf instance goal to pursue, false otherwise.

**Signature**

<table>
<thead>
<tr>
<th>reset(i : InstanceParameters)</th>
</tr>
</thead>
</table>

**Purpose**

Resets the InstanceTree back to its default state.

**Pre-conditions**

None

**Post-conditions**

A leaf-level InstanceGoal from the set of active InstanceGoals whose preconditions have been met has been returned.

---

### RoleInterpreter

<table>
<thead>
<tr>
<th>Signature</th>
<th>getGoalStatus()</th>
</tr>
</thead>
</table>

**Purpose**

Returns the execution status of the root goal

**Pre-conditions**

The agent and goal are not null. The given goal is a top-level organizational goal.

**Post-conditions**

ACTIVE is returned if the root goal is still being pursued. ACHIEVED is returned if the root goal is achieved. FAILED is returned if the root goal was failed to be achieved.

**Signature**

<table>
<thead>
<tr>
<th>getRoleLevelGoalModel() : RoleLevelGoalModel</th>
</tr>
</thead>
</table>

**Purpose**

Returns the RoleLevelGoalModel associated with this interpreter

**Pre-conditions**

None

**Post-conditions**

The RoleLevelGoalModel associated with this interpreter has been returned

---

### 8.7 Roles

#### 8.7.1 Description
The Role component’s responsibility is to provide an adapter between the Role representation used by OMACS in WumpiWorld, and the Role Interpreter. The parent class, 
AbstractGaaRole defines a method to return a statically defined role priority (to make the agent architecture as simple as possible). The public interface exported by this component, the GaaRole, provides a method to return the RoleLevelGoalModel associated with a Role.

Roles encapsulate the behavior an agent should perform while pursuing its assigned goal. They are parameterized by instance goals and their instance parameters. In our system, role behavior is entirely defined by the RoleLevelGoalModel. No additional code is used to specify behaviors.

The RoleLevelGoalModel is defined by a GMoDS Goal Model XML file that is created by the AgentTool3 graphical editor. This XML file is read in at runtime by the RoleLevelGoalModel class. This allows for users of the system to customize its behavior by simply editing the XML file in the graphical editor.

### 8.7.2 Class Diagram

![Role Package Class Diagram](image)

**Figure 5 – Role Package Class Diagram**

### 8.7.3 Class Descriptions

**AbstractGaaRole** – The AbstractGaaRole is the base role. It defines the method that is used to create a static priority mapping between roles for the demonstration architecture. In addition, it contains code to lazily initialize the RoleInterpreter whenever the getPlan() method is called. This method will call the abstract getRoleLevelGoalModel() method to
lazily initialize the RoleLevelGoalModel that is associated with the concrete Role class that has been instantiated.

**AreaSearcherRole** – The AreaSearcherRole is the lowest priority role in the system. It is defined by the AreaSearcherRole.goal model. It is responsible for navigating the agent to a previously unsearched location and then searching it for Wumpi and Gold.

**GoldFetcherRole** – The GoldFetcherRole is the second highest priority role in the system. It is defined by the GoldFetcherRole.goal model. It is responsible for navigating the agent to a location that contains gold and picking it up. This role will instantiate a new ReturnGold goal if gold is successfully picked up.

**GoldReturnerRole** – The GoldReturnerRole is the highest priority role in the system. It is defined by the GoldReturnerRole.goal model. It is responsible for navigating the agent back to the gold drop off location, the bin. It is also responsible for making sure that the agent actually releases the gold into the bin successfully.

**HunterKillerRole** – The HunterKiller role is the third highest priority role in the system. It is defined by the HunterKillerRole.goal model. It is responsible for generating a set of firing locations for the given target Wumpi and then making sure that the agent actually fires at the Wumpi once a suitable firing location is reached.

### 8.7.4 Interface Specification

**GaaRole**

<table>
<thead>
<tr>
<th>Signature</th>
<th>getRoleLevelGoalModel() : RoleLevelGoalModel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Returns the RoleLevelGoalModel object that defines this Role</td>
</tr>
<tr>
<td>Pre-conditions</td>
<td>None</td>
</tr>
<tr>
<td>Post-conditions</td>
<td>A new RoleLevelGoalModel object that defines this Role has been returned. The RoleLevelGoalModel InstanceTree has been reset back to its initial state.</td>
</tr>
</tbody>
</table>
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="Parameter">
    <xs:complexType>
      <xs:attribute name="package" type="xs:string" use="required"/>
      <xs:attribute name="name" type="xs:string" use="required"/>
      <xs:attribute name="class" type="xs:string" use="required"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="Method">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="Parameter" minOccurs="0" maxOccurs="unbounded"/>
        <xs:attribute name="goal_name" type="xs:string" use="required"/>
        <xs:attribute name="declared_name" type="xs:string" use="required"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="GoalCapabilityMap">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="Capability" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="Capability">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="Method" maxOccurs="unbounded"/>
        <xs:attribute name="package" type="xs:string" use="required"/>
        <xs:attribute name="class" type="xs:string" use="required"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
10 Chapter 10 - Assessment Evaluation

10.1 Introduction

This document details the results of black box testing performed on the GMoDS-based Runtime Agent Role Interpreter as outlined in Test Plan 1.0. Since successful execution of any one of these test cases requires the successful execution of all system requirements, this assessment covers the entire GMoDS-based Runtime Agent Role Interpreter implementation.

10.2 Test Case Result Summary

<table>
<thead>
<tr>
<th>#</th>
<th>Scenario</th>
<th>Scenario Description</th>
<th>Score</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>HunterKiller.xml</td>
<td>Single HunterKiller</td>
<td>19551</td>
<td>Pass</td>
</tr>
<tr>
<td>2</td>
<td>HunterKillerTeam.xml</td>
<td>Pair of HunterKiller</td>
<td>23650</td>
<td>Pass</td>
</tr>
<tr>
<td>3</td>
<td>GoldDigger.xml</td>
<td>Single GoldDigger</td>
<td>-1876</td>
<td>Pass</td>
</tr>
<tr>
<td>4</td>
<td>GoldDiggerHunterKillerTeam.xml</td>
<td>HunterKiller and GoldDigger</td>
<td>22400</td>
<td>Pass</td>
</tr>
<tr>
<td>5</td>
<td>WumpiWorld.xml</td>
<td>3 HunterKiller, 2 GoldDiggers</td>
<td>32198</td>
<td>Pass</td>
</tr>
</tbody>
</table>

10.3 Test Case Result Details

10.3.1 Test Case 1

This scenario executed as expected. The single HunterKiller agent was able to eliminate five out of six of the Wumpi in the world without falling into a pit or being killed by a Wumpi. However, due to the lack of GoldDigger agents, no gold could be recovered. The agent was able to detect the sixth one, but due to a lack of ammunition, was unable to eliminate the target. This test passed with a score of 19551.

10.3.2 Test Case 2

This scenario executed as expected. The two HunterKiller agents were able to eliminate all Wumpi from the world without falling into a pit or being killed by the Wumpi. However, due to the lack of GoldDigger agents, no gold could be recovered. This test passed with a score of 23650.

10.3.3 Test Case 3
This test was passed successfully. The single agent was able to only recover three out of the six possible pieces of Gold in the world. Due to the conservative exploration algorithm the agent employs, it will never take risks while searching for gold. Since no HunterKillers were present to take out any of the Wumpi, a large portion of the map was inaccessible to the agent. This test passed with a score of -1876.

10.3.4 Test Case 4

This test passed successfully. The team of the HunterKiller and the GoldDigger was able to eliminate five out of six Wumpi in the world and recover four out of six pieces of Gold. Due to a limited amount of ammunition not all Wumpi could be killed and the gold the last Wumpi defended could not be recovered. This test passed with a score of 22400.

10.3.5 Test Case 5

This final test scenario passed successfully. This scenario represented the full final scenario presented in CIS844 for the fall of 2010. All Wumpi in the world were killed and all gold was recovered. The final score of the test was 32198, which was more than 3000 points better than my final score on the project, 29035. This demonstration shows that agents designed using a generic role interpreter can perform as well as, or better than handcrafted agents at the same task. This is probably due to the higher level of abstraction it allows system designers to work at when defining agent behavior.

10.4 Problems Encountered

No real problems were encountered while testing the system. No new bugs, regressions, or previously unknown limitations were uncovered.

10.5 Summary

All test cases outlined in Test Plan 1.0 passed successfully. The GMoDS-based Runtime Agent Role Interpreter is ready for release.
11 Chapter 11 - User Manual

11.1 Introduction

This document serves a user manual for the GMoDS-based Runtime Agent Role Interpreter. It contains information about how to obtain the interpreter’s source code and executable binary. It also contains information about how to setup and run the Agent Architecture Demo that shows the Role Interpreter in action. Finally, it contains information about how to design your own Role Level Goal Models, Capabilities, and Goal Capability Maps.

11.2 Installation

11.2.1 Prerequisites

The GMoDS-based Runtime Agent Role Interpreter requires the following prerequisites in order to run:

- Java Runtime Environment 6.0 or higher

11.2.2 Required Files

To obtain the GMoDS-based Runtime Agent Role Interpreter, download the final GAA source archive from [http://people.cis.ksu.edu/~kylhill/phase_3/gaa.zip](http://people.cis.ksu.edu/~kylhill/phase_3/gaa.zip). The only required files from within the archive are gaa.jar, and the contents of the configs, models, and scenarios directories. The other files within the archive are the project source code and documentation.

Alternatively, the GAA project and all of its dependencies can be checked out from the K-State CIS projects CVS server at projects.cis.ksu.edu/cvsroot/gaa. The setup and use of this CVS repository are outside the scope of this document.

11.2.3 Setup

Extract the final GAA source archive, gaa.zip, to the desired location on your computer. No other installation steps are required.

11.3 Running the Agent Architecture Demo

The GMoDS-based Runtime Agent Role Interpreter contains an agent architecture demonstration in addition to its basic Role Interpreter functionality. To run this
demonstration, execute the following command on a command line from the directory where you extracted the project archive:

```
java -jar gaa.jar scenarios/WumpiWorldFull.xml configs/GoalCapabilityMap.xml
```

**Figure 1 – Sample Demonstration Invocation**

The first parameter to gaa.jar provides the path to the CROS environment description XML file. This file provides information describing the environment the agents will exist within, as well as their starting locations and other parameters. Several sample scenarios have been provided within the scenarios subdirectory.

The second parameter to gaa.jar provides the path the Goal Capability Map description XML file. This file provides information to map specification goal and parameter names from Role Level Goal Models to actual capability method calls at runtime. Instructions for producing your own Role Level Goal Models and Goal Capability Maps are given in section 4.

**Figure 2 – Agent Architecture Demonstration**

A window similar to the one shown in Figure 2 should appear after invoking the given command. This window is the standard a CROS environment display for the Wumpi World scenario. This scenario was given as a final project to students in CIS-844 during the fall of 2010. The goal of this scenario is to gather all of the gold in the world and kill all of the Wumpi monsters that guard it. While carrying out these tasks, agents, represented by the robot icons in the lower-left must avoid falling into pit traps or being
attacked by the Wumpi monsters. Agents are scored based on how efficiently they search the map, how many Wumpi are killed, and how much gold is retrieved.

![GoldDigger 1](image)

**Figure 3 – Agent Map Display**

Several other windows, similar to the one shown in Figure 3, should show up as well. These windows show what each agent thinks the world around them looks like. The key for this map is given in Figure 4.

<table>
<thead>
<tr>
<th>Tile Color</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>White</td>
<td>Searched Tile, Nothing Special</td>
</tr>
<tr>
<td>Light Gray</td>
<td>Unsearched Tile</td>
</tr>
<tr>
<td>Dark Gray</td>
<td>Unknown Obstacle</td>
</tr>
<tr>
<td>Black</td>
<td>Wall</td>
</tr>
<tr>
<td>Red</td>
<td>Wumpi</td>
</tr>
<tr>
<td>Light Orange</td>
<td>Possible Pit</td>
</tr>
<tr>
<td>Dark Orange</td>
<td>Possible Wumpi Danger</td>
</tr>
<tr>
<td>Yellow</td>
<td>Possible Gold</td>
</tr>
<tr>
<td>Green</td>
<td>Claimed Goal Tile</td>
</tr>
<tr>
<td>Blue</td>
<td>Agent, Active</td>
</tr>
<tr>
<td>Magenta</td>
<td>Agent, All Goals Complete</td>
</tr>
</tbody>
</table>

**Figure 4 – Agent Map Display Key**

All agent activity within this scenario is controlled by the Role Level Goal Models supplied within the models subdirectory of the archive. These goal models provide the
instructions that agents are to carry out as they pursue their assigned goals. The mappings of these goals to agent capability methods are defined by the Goal Capability Map. The design and construction of these models is discussed in section 4.

11.4 Designing Models

The GMoDS-based Runtime Agent Role Interpreter is designed to modular and extensible. To achieve that aim, new Role Level Goal Models, Capabilities, and Goal Capability Maps can be supplied by users to enable them to use the Role Interpreter in any multiagent system that they choose.

11.5 Creating New Role Level Goal Models

New Role Level Goal Models can be created to specify new behavior for agents to perform while playing the role the model defines. Role Level Goal Models are simply regular GMoDS Goal Models whose root goal is a leaf-level goal in the top-level system Goal Model. The leaf-level goals of a Role Level Goal Model map directly to capability methods that the agent possesses. Role Level Goal Models can be constructed using the agentTool3 Diagram Editor just like any other GMoDS goal model.

Role Level Goal Models are no different from standard GMoDS Goal Models, except for the following restrictions:

1. The number and names of leaf-goal specification parameters must match with number and names of formal parameters of the methods to which they are to be mapped. If they do not match, runtime exceptions will be thrown when the Goal Capability Map is parsed at startup.

2. Special “true”- and “false”- prefixed event names must be used if conditional execution and, by extension, iteration, is desired. These events only possess special meaning if they originate from a leaf-level goal whose mapped method returns a boolean value. If they are used on a non-boolean method, then they have no special meaning. If the method returns true at runtime, only the true-prefixed events are executed. Likewise, if the method returns false at runtime, only the false-prefixed events are executed.

3. Only one instance goal in a “triggers-cycle” may be active at a time. This is due to the way GMoDS determines how specification goals are completed. A custom patch was applied to GMoDS for this project to allow execute of “triggers-cycles” under most cases. This only means that Role Level Goal Models cannot support parallelism within a Role if conditional execution is also required. Additional work on GMoDS could remove this restriction.
4. At most, only one new specification parameter may appear in an event leaving a given goal. It is assumed that this specification parameter represents the return value from the invoked method. Any and all previous instance and inherited parameters can be passed along as part of the event without any restrictions. There is no real way for the system to deterministically provide values for more than one new specification parameter. If this behavior is required, then leaf-level goals should be split up into smaller ones that only generate a single new event parameter when executed.

11.6 Existing Capabilities

The following existing capabilities are present in the system and may be used within Role Level Goal Models and mapped to by Goal Capability Maps.

11.6.1 Advanced Bazooka

The AdvancedBazooka capability extends the existing Bazooka capability with targeting capabilities and ammo counting.

<table>
<thead>
<tr>
<th>Available Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>fire(loc : LocationData) : boolean</td>
</tr>
<tr>
<td>getNextFiringLocation() : LocationData</td>
</tr>
<tr>
<td>hasAmmo() : boolean</td>
</tr>
<tr>
<td>hasFiringLocation() : boolean</td>
</tr>
<tr>
<td>setTarget(loc : LocationData)</td>
</tr>
</tbody>
</table>

*Figure 5 – Advanced Bazooka Methods*

11.6.2 Advanced Communication

The AdvancedCommunication capability extends the existing CommunicationsImpl capability. This capability can send or receive messages on specific channels, or send and receive broadcast messages.

<table>
<thead>
<tr>
<th>Available Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>broadcast(channelID : String, content : Object) : boolean</td>
</tr>
<tr>
<td>broadcastIncludeSelf(channelID : String, content : Object) : boolean</td>
</tr>
<tr>
<td>receive() : Boolean</td>
</tr>
<tr>
<td>removeChannel(channelID : String) : boolean</td>
</tr>
<tr>
<td>send(agentID : UniqueIdentifier, channelID : String, content : Object) : Boolean</td>
</tr>
</tbody>
</table>

*Figure 6 – Advanced Communication Methods*
11.6.2 Advanced Gold Grabber

The AdvancedGoldGrabber capability extends the existing GoldGrabber capability. This capability can grab gold, report if it has gold, and release gold.

<table>
<thead>
<tr>
<th>Available Methods</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>grab() : boolean</td>
<td></td>
</tr>
<tr>
<td>putGoldIntoBin() : boolean</td>
<td></td>
</tr>
<tr>
<td>hasGold() : boolean</td>
<td></td>
</tr>
</tbody>
</table>

Figure 7 – Advanced Gold Grabber Methods

11.6.3 Advanced Movement

The AdvancedMovement capability wraps the more primitive RobotMovement and GPSImpl and extends them with advanced mapping and route planning capabilities. AdvancedMovement is used to determine the agent’s current location, set goal movement locations, plan routes between locations, and then actually execute those routes.

<table>
<thead>
<tr>
<th>Available Methods</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>getCurrentLocation() : LocationData</td>
<td></td>
</tr>
<tr>
<td>getDestinationLocation() : LocationData</td>
<td></td>
</tr>
<tr>
<td>getOldLocation() : LocationData</td>
<td></td>
</tr>
<tr>
<td>isAtDestinationLocation() : boolean</td>
<td></td>
</tr>
<tr>
<td>move() : boolean</td>
<td></td>
</tr>
<tr>
<td>setDestinationLocation(loc : LocationData)</td>
<td></td>
</tr>
</tbody>
</table>

Figure 8 – Advanced Movement Methods

11.6.4 Advanced Sensors

The AdvancedSensors capability internally wraps the Breeze, Smell, and Sparkle sensors. It also encapsulates the sonar capability. It is responsible for aggregating sensor input from a variety of sources, interpreting it, and then storing it in the agent’s map database. It is also responsible for sharing this map information with other agents, and merging their map data with the agent’s map data.

<table>
<thead>
<tr>
<th>Available Methods</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>isGoldAtLocation(loc : LocationData) : boolean</td>
<td></td>
</tr>
<tr>
<td>isLocationSearchable(loc : LocationData) : boolean</td>
<td></td>
</tr>
<tr>
<td>isWumpiAtLocation(loc : LocationData) : boolean</td>
<td></td>
</tr>
<tr>
<td>isAtDestinationLocation() : boolean</td>
<td></td>
</tr>
<tr>
<td>readOthersMaps()</td>
<td></td>
</tr>
<tr>
<td>readSensors() : boolean</td>
<td></td>
</tr>
<tr>
<td>sendOthersMaps()</td>
<td></td>
</tr>
<tr>
<td>setLocationUnsearchable(loc : LocationData)</td>
<td></td>
</tr>
</tbody>
</table>

Figure 9 – Advanced Sensors Methods
11.6.5 OMACS Interface

The OmacsInterface is a special capability that allows Role Level Goal Models to communicate with the rest of the OMACS system. It currently only allows for the construction and assignment of a single ReturnGoldGoal. However, in the future it could be extended to provide generic OMACS interface capabilities, such as role assignment, failure reporting, and reorganization.

<table>
<thead>
<tr>
<th>Available Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>createReturnGoldGoal()</td>
</tr>
</tbody>
</table>

Figure 10 – OMACS Interface Methods

11.7 Creating New Capabilities

New capabilities can be added to the system very easily. To make a capability that can be used within a Role Level Goal Model and Goal Capability Map, simply state that the capability implements the empty GaaCapability interface and then annotate all parameters of all methods that you wish to export to the Goal Capability Map with the custom @Name annotation. The name you give the parameter is the name that must be called out in the Goal Capability Map. As sample annotated method signature is provided below for reference:

```java
public final boolean addChannel(@Name("channelID") final String channelID, @Name("channel") final CommunicationChannel channel);
```

Figure 11 – Example Method Annotation

11.8 Creating New Goal Capability Maps

Creating new Goal Capability Maps is simple and straightforward. When you have finished creating a new Role Level Goal Model and any new capability classes, check all leaf-level goals and see if they already contain a mapping to capability method. If they do not, then you will need to provide such a mapping. If any mappings are missing, a runtime exception will be thrown indicating which mappings need to be established.

The sample XML file GoalCapabilityMap.xml provides mappings for all existing capabilities to goal names used by the Role Level Goals Models in within the project models subdirectory. If you wish to use different leaf-level goal names to map to existing capability methods, either rename the existing mapping’s goal_name attribute, or provide a new mapping with a different goal_name attribute. Similarly, if you create a new capability class, you will need to construct mappings for each public method that you wish to use from your RoleLevelGoalModels.
11.8.1 Goal Capability Map Schema

For easy reference, the Goal Capability Map Schema is provided below to aid in the construction of valid mappings. To check your Goal Capability Map, simply validate your XML file with the schema provided below. This will guarantee that the file will at least be parsed correctly when given the Role Interpreter.

Figure 13 – GoalCapabilityMap

Figure 14 – Capability

Figure 15 – Method

Figure 16 – Parameter
Figure 17 – The Goal Capability Map Schema
12 Chapter 12 - Project Evaluation

12.1 Introduction

This document provides a list of problems encountered, lessons learned, and possible future work for the GMoDS-based Runtime Agent Role Interpreter project. This document also summarizes the number of lines of code in the project as well as various scheduling statistics.

12.2 Problems Encountered

This section provides a summary of some problems encountered during the project.

12.2.1 Learning Existing OMACS and GMoDS Frameworks

It took me a lot longer than I expected to learn and understand the OMACS and GMoDS frameworks. I was learning multiagent systems concepts around the same time as I started this project, so I had to devote a lot of time early on to learning basic system concepts and then mapping them code I read within the frameworks. This made for slow development and design tasks at the start of the project.

12.2.2 Determining Project Requirements

It took quite a while to nail down the exact requirements for this project. I did not get a lot of face-to-face time with my advisor since I am a distance education student. This made for many discussions back and forth over e-mail, adding to the latency when coming up with my initial design and requirements documents.

12.2.3 Time Management

I had many problems with time management on this project. It was hard for me to stay motivated to work on the various tasks required for this project at a consistent rate. Since I am working full time and am a student part time, it was hard for me to juggle the various priorities in my life. I think that this project would have been much easier had I been able to devote myself to it full time.

12.3 Estimates

This section compares the original estimates in lines of code and total hours of work to the actual totals to provide feedback.
12.3.1 Source Lines of Code

<table>
<thead>
<tr>
<th>Phase</th>
<th>SLOC Estimate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Phase 1</td>
<td>5000</td>
</tr>
<tr>
<td>Phase 2</td>
<td>4750</td>
</tr>
<tr>
<td>Phase 3</td>
<td>5135 (Actual)</td>
</tr>
</tbody>
</table>

Table 1 – Estimates of SLOC by phase

Table 1 shows the total project SLOC estimate by phase. The original estimate of 5000 lines was decreased in the second phase because I did not think an XML parser needed to be implemented. The actual size grew by about 400 lines in the final phase. The final figure of 5135 SLOC is remarkably close to the original 5000 SLOC estimate.

12.3.2 Project Duration

<table>
<thead>
<tr>
<th>Phase</th>
<th>Estimated End Date</th>
<th>Actual End Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Phase 1</td>
<td>December 8, 2010</td>
<td>December 8, 2010</td>
</tr>
<tr>
<td>Phase 2</td>
<td>March 14, 2010</td>
<td>June 23, 2011</td>
</tr>
<tr>
<td>Phase 3</td>
<td>May 13, 2010</td>
<td>July 25, 2011</td>
</tr>
</tbody>
</table>

Table 2 – Estimated and actual end dates by phase

Table 2 shows the estimated versus actual end dates for each phase. For the first phase, I knew the end date of the phase because I had already scheduled the presentation by the time I had made the estimate. I greatly underestimated how much work I needed to do in the second and third phases of the project. Due to time pressures from home, work, and other classes, I was not able to devote as much time as I wanted to this project and I had to extend the project out into the summer intersession. I was able to take time off work in the summer and I was able to get a large amount of work done in a short amount of time.
Figure 1 – Percent of time spent in each phase

Figure 2 – Percent of time for each task in phase 1

Figure 3 – Percent of time for each task in phase 2
Figures 2 through 4 provide a breakdown, by phase, of the amount of time spent on various tasks throughout the project. At the beginning of the project, I spent a lot more time reading and fleshing out requirements. Near the end of the project, I spent a lot more time testing, finalizing development tasks, and writing documents.

<table>
<thead>
<tr>
<th>Phase</th>
<th>Development</th>
<th>Writing</th>
<th>Designing</th>
<th>Testing</th>
<th>Requirements</th>
<th>Reading</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Phase 1</td>
<td>47</td>
<td>36.5</td>
<td>0</td>
<td>0</td>
<td>11.25</td>
<td>17</td>
<td>111.75</td>
</tr>
<tr>
<td>Phase 2</td>
<td>30</td>
<td>30</td>
<td>9</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>69</td>
</tr>
<tr>
<td>Phase 3</td>
<td>18</td>
<td>50</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>71</td>
</tr>
<tr>
<td>Total</td>
<td>95</td>
<td>116.5</td>
<td>9</td>
<td>3</td>
<td>11.25</td>
<td>17</td>
<td>251.75</td>
</tr>
</tbody>
</table>

Table 3 – Time (in hours) for each task by phase
Figure 5 is a graphical representation of the data contained within Table 3. In total, I spent 252 hours working on this project. Nearly half of that time was spent writing documents and another large portion of the time was spent developing the actual software deliverables. I was surprised to see how little time was spent on design, testing, and requirements gathering. Additionally, I was surprised at how much time was spent writing documentation.

12.4 Lessons Learned

12.4.1 Computability Theory and Expressiveness

I learned a lot about Computability Theory and Expressiveness concepts, such as Turing Completeness while working on this project. In order to determine the limits of agent behavior specified by Role Level Goal Models, I had to read a lot about these theoretical concepts.

12.4.2 Java Reflection API

I learned a lot about the runtime Java Reflection API while implementing this project. This powerful interface allows developers to introspectively discover what classes are available to a system, and then determine what methods exist within those classes. This allowed me to programmatically establish goal-capability mappings at runtime without having to know anything about the capabilities agents intended to use beforehand.

12.4.3 Programming Language Concepts

I learned a lot about various programming language concepts such as the difference between actual and formal parameters and exactly how method overloading works.
These concepts were crucial for me to understand when using the Java Reflection API. In addition, I learned a lot when mapping goal and event parameters to actual method parameters.

12.5 Future Work

All features and use cases described in the Vision Document were successfully implemented in the project. However, during the course of the project, several areas were identified for possible future enhancements.

12.5.1 Expand the OMACS Interface Capability

The current OMACS interface capability is rather weak. This capability is intended to provide all the OMACS interactions that a system could need at runtime within a capability that is callable from Role Level Goal Models. Unfortunately, the current capability only allows the creation of hard-coded goals to return Gold to a predefined location. It would be nice to define a more robust and reusable interface for a variety of goal creation and obviation tasks.

12.5.2 Make a More Complex Agent Architecture Demo

The current agent architecture demo makes use of predefined agent-role-goal assignments. It also does not have any overall organization that controls the assignment of agents to tasks. There is a lot of room for making a more robust system that makes use of an organization to coordinate these activities.

12.5.3 Remove GMoDS Limitations

A limitation within GMoDS prevents more than one active goal from existing within a “triggers-loop” at one time. I have provided a small patch to GMoDS that makes it possible to actually execute triggers-loops, but it would be nice to make this patch more robust so that it allows more than one active goal at a time. It would also be nice if this patch could be improved so that it is accepted into the official GMoDS framework.
13 References


