Multi-Agent System Simulator

CISC475/675 Fall 2014 Team 6
Mike Boch
Benjamin Gotthold
Steven Noyes
Varun Sharma
Stefan Zimmerman

Change Log

- v0.5 11/14/2014 - In this release we continued to work on the requirements document correcting spelling, fixing mistakes and updating the verification plan. We also worked on writing new tests to increase the branch and overall instruction coverage to help improve the over all accuracy. We have worked on incorporating all the requirements mentioned in SRS document in the tasktree and the simulationstate module. We have also started updating the Javadoc for the project.
- v0.4 10/31/2014 - In this release we continued to work on the requirements document making images clearer, correcting spelling, and fixing formatting mistakes. We incorporated comprehensive coverage testing and created a verification plan to help improve our software accuracy. We were able to link the main modules of our program to provide the first runnable iteration where we created a simple external agent to assist in the running of a basic simulation.
- v0.3 10/17/2014 - Fixed several errors in the document in terms of spelling and grammar. Updated the USES diagrams of the system and the tasktree package. Added detailed descriptions of the cTAEMS grammar and the task tree. Gave more detail on each stakeholder. Updated the ERD to reflect their proper language. Added descriptions of the interface that the Agents must adhere too. Added sample messages of each type for the Agents. Added an example task tree to show how a small scale simulation would run. Removed the cost attribute from the task tree. Updated the description of each module and package. Added a test verification plan. Converted most applicable images to the SVG format. Removed references to the Event Engine as it is no longer needed. Changed Event to Node as it fits the task tree design better. Added more detail to some of the Core Features. Added more terms to the Definition of Concepts.
- v0.2 10/03/1014 - This release focused on clarifying the project requirements and incorporating some basic design ideas. We worked to correct errors in the document pointed out by our client and instructor and added some additional content and figures. The most notable change is the addition of Chapter 5, Detailed System Design, where we began describing the transition from requirements to actual code.
- v0.1 09/19/2014 - Initial Document Release

Team Information

This project, including the content of this document, have been designed by University of Delaware students Mike Boch, Ben Gotthold, Steven Noyes, Varun Sharma, and Stefan Zimmerman. To view the current release of this project please visit our project page at http://cisc475-6.cis.udel.edu .

Table of Contents

List of Figures

Chapter 1

Introduction

As technology advances, software is fast becoming the most economical way to test complex solutions to real world problems. Software testing allows a user to run unlimited iterations with complete control over environment variables. Our Multi-Agent System Simulator (MASS) will provide an environment for Dr. Decker and other researchers to test software Agents allowing them to design more accurate solutions to real world problems faster than ever before.

1.1 Document Purpose

The purpose of this Software Requirements Specification (SRS) is to serve as a statement of understanding between the users of the proposed product and the software developers of the product. The Software Requirements Specification is defined using a subset of the Unified Modeling Language (UML), an Entity-Relationship Diagram (ERD) describing all of the objects/entities along with their attributes, relations, and Data Flow Diagram (DFD).

1.2 Scope

We will develop a general-purpose command line program called Multi-Agent System Simulator (MASS). The MASS will take a Simulation File Input (SFI) and a Configuration File Input (CFI) to produce an environment in which user defined software Agents can connect and interact to solve problems. Upon completion the MASS will output the results of the problem Simulation as well as produce a Log File Output (LFO) detailing the events that occurred within the Simulation. The SFI will be a cTAEMS file describing the problem domain that the MASS will use to construct the Simulation environment. The CFI will be a plain text file that contains settings relating to the execution of the MASS but are independent of a given SFI. Agents, which are independent user-defined programs, will connect to the MASS through TCP socket connections before a simulation begins. While running, the MASS allows inter-Agent communication and logs statistics such as the number of messages sent by each Agent. Upon completion the MASS will output the results of the simulation in terms of Quality and Duration. It is important to note that the MASS does not restrict the design of the Agents themselves but requires them to adhere to a connection interface so that communication is possible.

1.3 Acronyms and Abbreviations

This section contains definitions of acronyms and abbreviations used in this document.

- **MASS**: Multi-Agent System Simulator
- **SFI**: Simulation File Input
- **CFI**: Configuration File Input
- **LFO**: Log File Output
- **UML**: Unified Modeling Language
- **DFD**: Data Flow Diagram
- **TCP:** Transmission Control Protocol
- **JSON**: JavaScript Object Notation
- **SRS**: Software Requirements Specification
- **ERD**: Entity-Relationship Diagram
- **QAF**: Quality Accumulation Function

Chapter 2

General Description

The product is meant to serve as a common platform for academic and research oriented activities in the area of Multi-Agent Simulation. The following sections describe the high level view of the system and establish its context. These sections do not state specific requirements but make the specific requirements easier to understand.

2.1 Stakeholders

The stakeholders of the Multi-Agent System Simulator are classified into the following categories.

- **Dr. Decker**: Needs a platform to test his software agents in a way that can help advance his research and lead to new innovations. He can be contacted through email and met with in person readily.
- **Dr. Siegel**: Needs a challenging project for his 475 students that is able to be completed over the course of the semester. The project should condone the use of software tools to test implementations and track changes. He can be contacted through email or Skype and can be met with in person readily.
- **Researchers**: Researchers might want to use this software for their research to design agents and simulations when testing new ideas.
- **CIS faculty**: Needs an interactive tool for students to make the task of learning more interesting.
- **Developers**: Developers are responsible for the designing, testing, configuring, and upgrading of the subsystems.
- **Support**: Support personnel are responsible for the maintenance of the system, the software, the installation of subsystems, and configuration changes.

Chapter 3

General System Requirements

The MASS is designed as a domain independent central process where all domain knowledge is obtained from the SFI and CFI. To prevent any interference or assumption, the MASS and the Agents run as separate independent processes. The cTAEMS model supplies the internal knowledge used to build a Task Tree from which Agents will select available Methods for execution. This coupled with the easy configuration mechanism, logging, reporting and statistical analysis makes the simulator a good platform for research and evaluation of Multi-Agent Systems.

3.1 System Features

The following list offers a brief outline and description of the main features and functionalities of the MASS. The features are split into two major categories: core features and optional features. Core features are essential to the application's operation, whereas optional features are preferred but not required. For a better understanding of the basic features of the system see Figure 3.1.

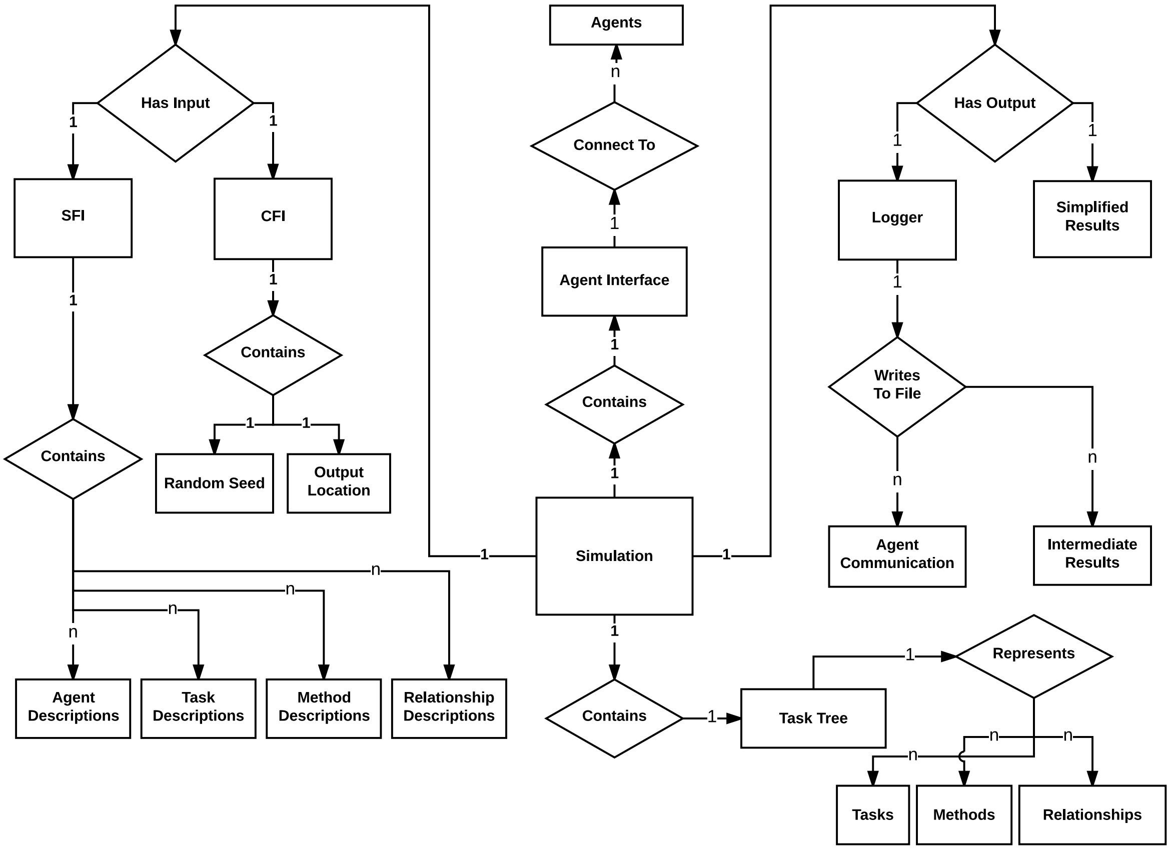

Fig. 3.1: ERD diagram for basic components of MASS.

Core Features

1. **Running The MASS**: The MASS will run via the command line inside of a BASH script that takes the SFI and optionally the CFI as inputs.
2. **Simulation File Input (SFI)**: The SFI is a cTAEMS file.
   (a) The system is only required to support the use of the AND, OR, and SUM QAFs of the cTAEMS grammar.
   (b) Methods and Tasks may only have Qualities and Durations. Costs will not be implemented.
   (c) The grammar of the cTAEMS file is in Figures 3.2 and 3.3(items in all capitals are variable).
   QAF can be q_and, q_sum, or q_or.
   ENABLES_DISABLES can be enables or disables.
   FACILITATES_HINDERS can be facilitates or hinders.

```
;; Comment. Sample CTAEMS file.
(spec_version "VERSION_NUMBER")
;; Agent specification
(spec_agent
	(label AGENT1_NAME)
)
(spec_agent
	(label AGENT2_NAME)
)
;; More Agents as needed
;; Tasks
(spec_task_group
	(label TASK_GROUP1_NAME)
	(subtasks SUBTASK1_NAME SUBTASK2_NAME) ;; More Subtasks as needed
	(qaf QAF)
)
(spec_task
	(label SUBTASK1_NAME)
	(earliest_start_time EARLIEST_START)
	(deadline DEADLINE)
	(subtasks METHOD1_NAME) ;; More Subtasks or Methods as needed
	(qaf QAF)
)
(spec_task
	(label SUBTASK2_NAME)
	(earliest_start_time EARLIEST_START)
	(deadline DEADLINE)
	(subtasks METHOD2_NAME)
	(qaf QAF)
)
(spec_method
	(label METHOD1_NAME)
	(agent AGENT1_NAME)
	(outcomes
		(METHOD1_NAME_OUTCOME_0
			(density DENSITY)
			(quality_distribution QUALITY1 PROBABILITY1 QUALITY2 PROBABLITY2) ;; More
Qualities as needed
			(duration_distribution DURATION1 PROBABILITY1 DURATION2 PROBABILITY2) ;;
More Durations as needed
	)
)
(spec_method
	(label METHOD2_NAME)
	(agent AGENT2_NAME)
	(outcomes
		(METHOD2_NAME_OUTCOME_0
			(density DENSITY)
```

Fig. 3.2: cTAEMS File Grammar part 1

```
                              (quality_distribution QUALITY1 PROBABILITY1 QUALITY2 PROBABLITY2) ;; More
Qualities as needed
                              (duration_distribution DURATION1 PROBABILITY1 DURATION2 PROBABILITY2) ;;
More Durations as needed
          )
)
;; More Methods, Tasks, and Task Groups as needed
;; Relationships
(spec_ENABLES_DISABLES
          (label RELATIONSHIP1_NAME)
          (from TASK1_NAME)
          (to TASK2_NAME)
)
(spec_FACILITATES_HINDERS
          (label RELATIONSHIP2_NAME)
          (from TASK1_NAME)
          (to METHOD2_NAME)
          (quality NEW_QUALITY)
          (duration NEW_DURATION)
)
;; More Relationships as needed
```

Fig. 3.3: cTAEMS File Grammar part 2

3. **Configuration File Input (CFI)**: The program by default will look for a plain text configuration file in the current directory. Optionally, a user can specify the location of this CFI.

   (a) If no configuration file is found the system will make one in the current directory with default values.

   (b) The CFI will contain a random number seed, output file destination, the length (in milliseconds) of each Tick, and the port for the Simulator to listen on.

   (c) Not all items in the CFI need to be specified. The system will resort to default values for the missing items.

   (d) A sample CFI is shown in Figure 3.4.

config.ini

```
seed=8237428342734
outputDestination=output.log
tickLength=1000
port=9876
```

Fig. 3.4: Example CFI

4. **Agent Facing Interface**: The system will provide a consistent interface of which all Agents must implement in order to connect to the internal communication model.

   (a) All communication between the Agents and the MASS will be represented in the JSON format.

   (b) The system must handle any Agent architecture in compliance with the Agent Interface (by using the correct JSON format of messages) and refuse connection to any Agent in violation of this. If an Agent does use the correct format for JSON messages, the Simulator will terminate.

   (c) It is the job of the agent to choose from the available Methods and report to the Simulation when starting and event.

   (d) The Simulator will tell an agent when it's Method has finished.

   (e) Figures 3.5-3.13 describe the format of each Message type.

```
{
        "MessageType":"AgentRegistrationMessage",
        "Message":{
                "MsgSender":"SENDERNAME",
                "MsgDest":"DESTNAME"
        }
}
```

Fig. 3.5: Format of AgentRegistration Message

```
{
        "MessageType":"AskMethodStatusMessage",
        "Message":
        {
                "MsgSender":"SENDERNAME",
                "MsgDest":"DESTNAME",
                "MethodName":"METHODNAME"
        }
}
```

Fig. 3.6: Format of AskMethodStatus Message

```
{
        "MessageType":"ConfirmMethodStartMessage",
        "Message":
        {
                "MsgSender":"SENDERNAME",
                "MsgDest":"DESTNAME",
                "MethodName":"METHODNAME",
                "Started":BOOLEAN
        }
}
```

Fig. 3.7: Format of ConfirmMethodStart Message

```
NewNodeMessage

{
        "MessageType":"NewNodeMessage",
        "Message":
        {
                "MsgSender":"SENDERNAME",
                "MsgDest":"DESTNAME",
                "Node":
                {
                        "NodeType":"Task_OR_Method",
                        "NodeName":"NODENAME",
                        "Relationships":
                        [
                                {
                                        "RelationshipName":"RELATIONSHIPNAME",
                                        "Source":"SOURCENAME",
                                        "Destination":"DESTINATIONNAME",

        "RelationshipType":"Disables_OR_Enables_OR_Facilitates_OR_Hinders",
                                        if Facilitates OR Hinders
                                                "NewQuality":DOUBLE,
                                                "NewDuration":INTEGER
                                },
                                ...
                        ]
                        if Task
                                "QAF":"QAFVALUE",
                                "EarliestStartTime":INTEGER,
                                "Deadline":INTEGER,
                                "SubTasks":
                                [
                                        {NODE},
                                        ...
                                ]
                        if Method
                                "Quality":DOUBLE,
                                "Duration":INTEGER,
                }
        }
}
```

Fig. 3.8: Format of NewNode Message

```
{
	"MessageType":"NotifyMethodCompletedMessage",
	"Message":
	{
		"MsgSender":"SENDERNAME",
		"MsgDest":"DESTNAME",
		"MethodName":"METHODNAME",
		"Quality":DOUBLE,
		"Duration":INTEGER
	}
}
```

Fig. 3.9: Format of NotifyMethodCompleted Message

```
{
	"MessageType":"NotifyMethodStatusMessage",
	"Message":
	{
		"MsgSender":"SENDERNAME",
		"MsgDest":"DESTNAME",
		"MethodName":"METHODNAME",
		"Started":BOOLEAN,
		"Completed":BOOLEAN,
		"Enabled":BOOLEAN
	}
}
```

Fig. 3.10: Format of NotifyMethodStatus Message

```
{
	"MessageType":"NotifyRelationshipActivationMessage",
	"Message":
	{
		"MsgSender":"SENDERNAME",
		"MsgDest":"DESTNAME",
		"RelationshipName":"RELATIONSHIPNAME"
	}
}
```

Fig. 3.11: Format of NotifyRelationshipActivation Message

```
{
	"MessageType":"SetRandomSeedMessage",
	"Message":
	{
		"MsgSender":"SENDERNAME",
		"MsgDest":"DESTNAME",
		"Seed":LONG
	}
}
```

Fig. 3.12: Format of SetRandomSeed Message

```
{
        "MessageType":"StartMethodMessage",
        "Message":
        {
                "MsgSender":"SENDERNAME",
                "MsgDest":"DESTNAME",
                "MethodName:"METHODNAME"
        }
}
```

Fig. 3.13: Format of StartMethod Message

5. **Agent Communication**: Agents must have the ability to communicate throughout the Simulation.

   (a) An Agent can send a message intended for another Agent, or it can send one directly to the Simulator. Either way, the message will pass through the Simulator so that it may log the communication patterns of the Agents.

6. **Task Tree**: A Simulation will consist of a Task Tree that keeps track of what Methods are able to be executed at any given time.

   (a) The Task Tree consists of Tasks and Methods. Tasks can have Nodes as children while Methods are the leaves of the tree. The total quality of a Task Group is determined from the head of that group's Task Tree.

   (b) When a Method is finished, it's quality will propagate up through the tree based on its parents' QAFs.

   (c) An example Task Tree showing three Tasks, four Methods, and two Relationships is shown in Figure 3.14.

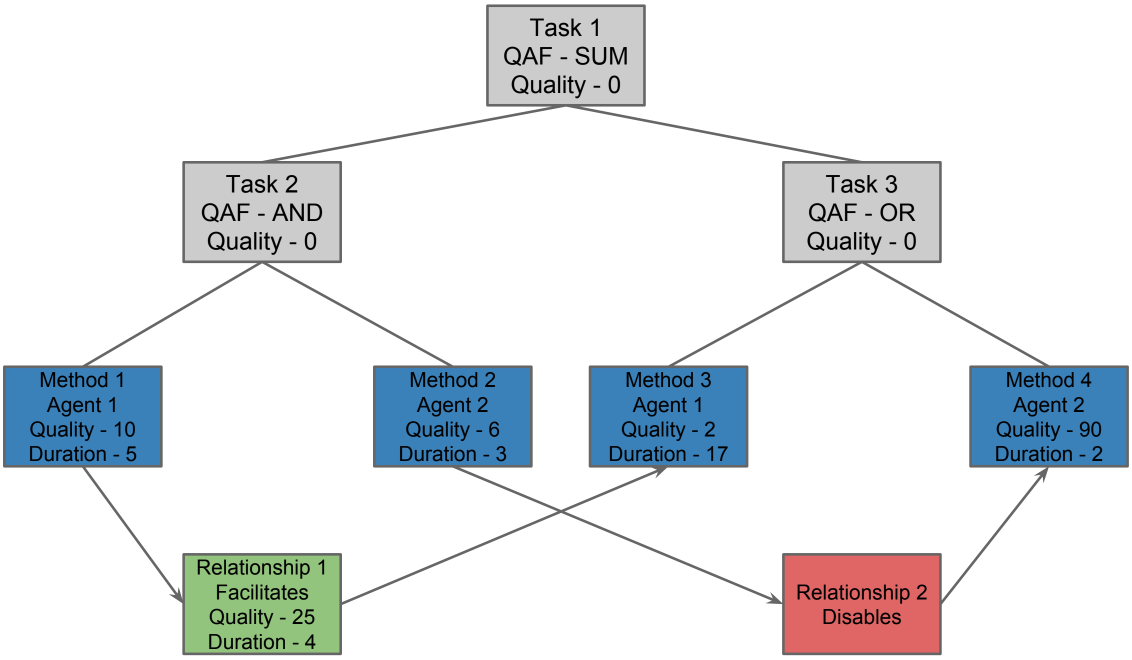

Fig. 3.14: Example Task Tree with Relationships

7. **Creating a Simulation**: A Simulation must be repeatable.

   (a) Any probability distributions must be computed using the seed value obtained from the CFI as the SFI is being parsed.

   (b) Agents must be given the seed value obtained from the CFI in case they use any random calculations in their logic.

   (c) Agents are given an initial set of Nodes that are visible to them at the start of the Simulation. Visibility is defined as follows:

   An Agent can see its assigned Methods.

   An Agent can see all Tasks that are direct ancestors of its Methods.

   An Agent can see all Relationships that have one the Agent's visible Nodes as the source/target.

   An Agent can see the Nodes that are the sources/targets of a relationship with one of that Agent's visible Relationships.

   (d) When any Node or Relationship is visible to an Agent, that Agent knows:

   The name of the Node or Relationship

   The names of all other Agents that can see that Node

   (e) When a Method is visible to an Agent, that Agent knows:

   The Method's quality

   The Method's duration

(f) When a Task is visible to an Agent, that Agent knows:

The Task's QAF

The Task's (visible) Subtasks

(g) When a Relationship is visible to an Agent, that Agent knows:

The type of the Relationship

The source and target of the Relationship

The new quality and duration if the Relationship is of Facilitates or Hinders type

8. **Running a Simulation**: The MASS must keep track of Nodes.
   (a) The MASS must know what Methods are able to be executed at any given time.
   (b) The MASS must know what Methods are being executed at any given time.
   (c) The MASS must know what Methods have been executed at any given time.
   (d) The MASS must know what the Qualities of all Tasks are at any given time.
   (e) The MASS must know which Tasks have been completed at any given time.
   (f) The MASS must know when the Simulation is finished. This is achieved when every Task Group has no Methods that are both enabled and unfinished.
9. **Simulation Output**: The system will produce an output detailing the overall Quality and Duration of the entire Simulation.
   (a) The output should occur on the command line.
10. **Log File Output (LFO)**: The system will produce a detailed log file including the intermediate Quality and Cost between every Task and Method.
    (a) The LFO will contain a transcript of Agent communication
    (b) The LFO will contain compiled statistics on Agent communication frequency.
    (c) The LFO will contain the intermediate and final Quality, Cost, and Duration that results from completing an Event in the Task Tree.

Additional Features

1. **Graphical Representation of Task Tree**: The Task Tree is a hierarchical tree-like structure representing Nodes and the Relationships between them.
   (a) The system may represent this structure as a graphical model to help understand the Simulation results better.

2. **cTAEMS Grammar Support**: The system may go beyond the logical AND, OR, and SUM operations and include additional implementations within the cTAEMS grammar.

3. **Agent Behavior**: The System may support the ability of Agents to stop, pause, or resume Methods if such actions are applicable based on the Simulation specifications.

Chapter 4

Other non-functional requirements

4.1 Performance requirements

All Agents involved with the model are connected to the simulator using sockets. The Agents themselves are independent processes, which could run on physically different machines. Note that the simulator does not control the Agents' activities, it merely allocates time slices and records the Events performed by the Agent during the time slice. This makes the performance of the MASS independent of an Agent's performance. The system shall function in real-time, however, the effect of network load or maximum limit on socket connections can only be published after testing.

4.2 Maintainability

The standardized design and implementation documents will be provided in order to maintain the system. All changes will be documented. A standard architecture will be applied and therefore allowing for quick evolution of the software to adapt to possible situations in the future.

4.3 Software Quality Attributes

The user interface of the Multi-Agent System Simulator is to be designed with usability as the first priority. The system will be presented and organized in a manner that is both visually appealing and easy for the user to navigate. To ensure reliability and correctness, there will be zero tolerance for errors in the Simulation environment.

Chapter 5

Detailed System Description

5.1 Design overview

The system is broken into five major packages called application, input, output, simulation, and tasktree. Figure 5.1 shows the relationship between these components.

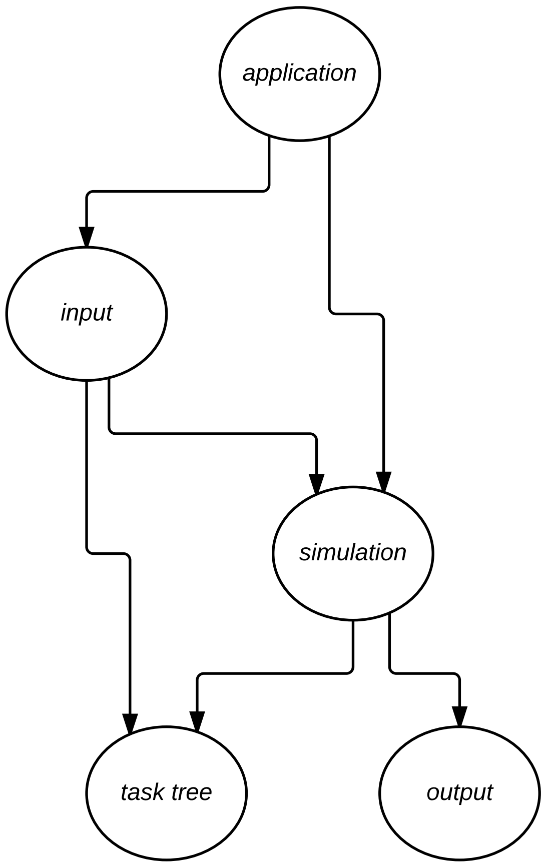

Fig. 5.1: USES diagram for packages within MASS.

5.2 Package Overview

1. **application**

    - Description: The application package is responsible for initializing the other packages and linking everything together.
    - Contains: The application package contains module AISim.
    - Uses: This package uses the input, and simulation packages.

2. **input**

    - Description: The input package is responsible for reading the initial data input and building a Task Tree to represent this data.
    - Contains: The input package contains three modules which include Parser, ConfigurationData, and InputData.
    - Uses: This package uses the tasktree, and simulation packages.

3. **simulation**

    - Description : The simulation package is responsible for connecting and communicating with agents, managing the Task Tree, and advancing the clock. Figure 5.2 represents the USES diagram for the package simulation.
    - Contains : This package contains the modules Simulator, Agent, Message and ServerThread.
    - Uses : This package uses the tasktree and output packages.

4. **tasktree**

    - Description : The tasktree package is responsible for representing the Task Tree in a hierarchical manner that is easy for the simulation package to update and distribute.
    - Contains : This package contains the modules Node, NodeRelationship, and Distribution.
    - Uses : This package does not use any packages.

5. **output**

    - Description: The output package is responsible for printing data after the simulation has completed. It is also responsible for logging detailed data to a file during the course of the simulation.
    - Contains: The output package contains a module Logger.
    - Uses: This package does not use any packages.

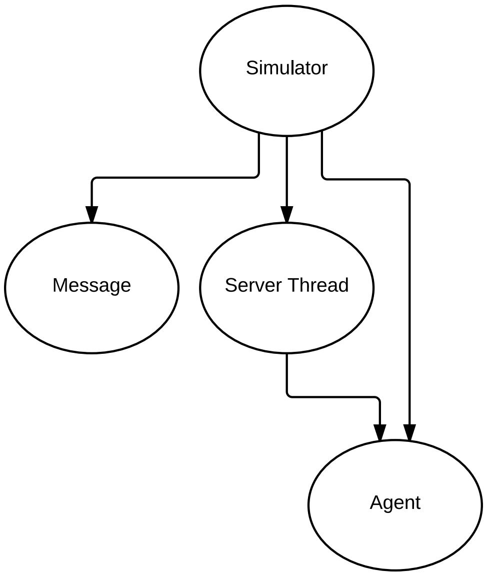

Fig. 5.2: Uses diagram for the package simulation

5.3 Detailed Package Design

1. **Application** The application package is responsible for initializing the modules within the input and simulation packages in order to begin the Simulation.

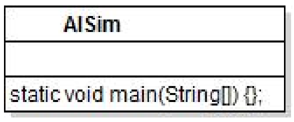

Fig. 5.3: UML diagram for input package

2. **Input** Figure 5.3 represents the UML class diagram for input package. This package is responsible for reading and parsing the cTAEMS SFI and the plain text CFI. This packages contains an abstract parser which serves as a template for the InputParser and the ConfigurationParser.

   - **InputParser**: The InputParser is responsible for reading the SFI which contains definitions for Agents, Nodes, and Relationships that make up the Simulation. The first step of the parse involves reading the desired cTAEMS file into a string by implementing Java's BufferedReader class. Second, the string is split into tokens such as Agent, Task, Task Group, Method, and Relationship. These tokens are further parsed in order to analyse their various components. For example, a Task could be broken up into a label, list of

Subtasks and QAF which are instantiated into the corresponding Task class. This data is used to build an instance of the TaskTree class which is then stored in the InputData class which will be passed to the simulation.

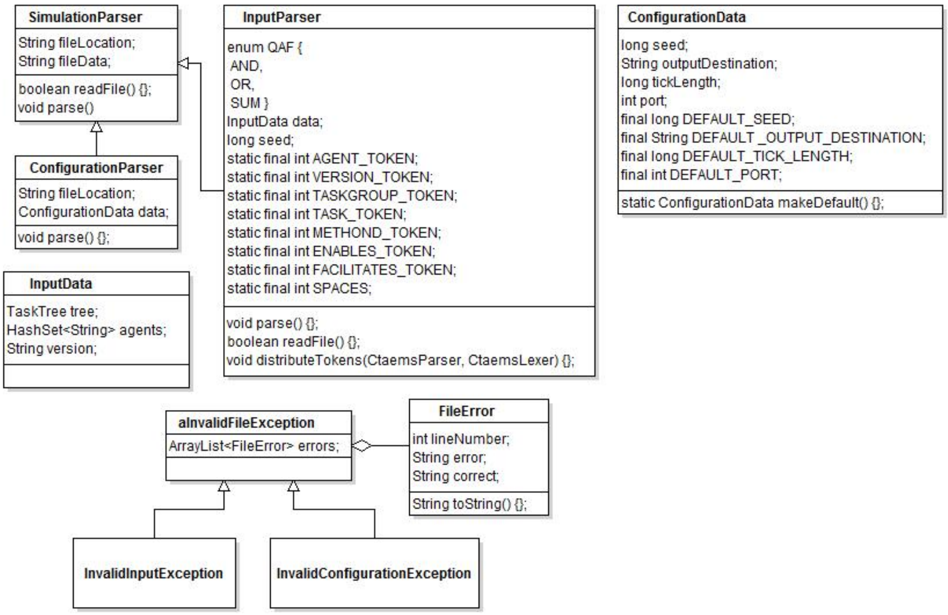

Fig. 5.4: UML diagram for input package

- **ConfigurationParser**: The configuration parser reads and parses the CFI to determine the random seed used throughout the Simulation. The Configuration parser will use Java's BufferedReader class to read in the file, with the first line representing the seed, the second the input location, the third output destination, the fourth the length of a tick, and finally the communication port. This data is written to the ConfigurationData class which is sent to the simulator.

- **Exceptions**: Within the input package there are three exceptions which are used to report a problem encountered during a parse. These exceptions are: InvalidFileException, InvalidInputException, and InvalidConfigurationException. InvalidInput and InvalidConfiguration both inherit from InavlidFile.

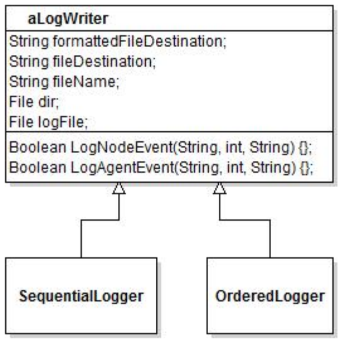

Fig. 5.5: UML diagram for output package

3. **Output** Figure 5.4 represents the UML diagram for output package. The output package is responsible for logging data after the simulation has completed. It contains a Logger class which upon completion will create the LFO document.

    - **Logger**: The Logger class is responsible for producing a transcript of Agent communication, statistics on Agent communication frequency, and the intermediate and final Quality and Duration that results from completing a Node in the Task Tree. Data will be passed to the Logger by the modules within the simulation package as the simulation advances. Upon completion of a simulation, the Logger will compile statistics on agent communication and produce the LFO containing the previously mentioned data.

4. **Simulation** Figure 5.5 represents the UML diagram for simulation package. The simulation package is responsible for handling all of the communications with Agents as well as updating the Task Tree architecture to model the current state of the simulation.

    - **Simulator**: This class contains all of the information about the simulation such as the random number seed, the Task Tree instance, the list of currently connected Agents, the Logger object, the list of currently running Methods, and the ServerCommunicateThread. This class is the core timekeeper of the simulation.

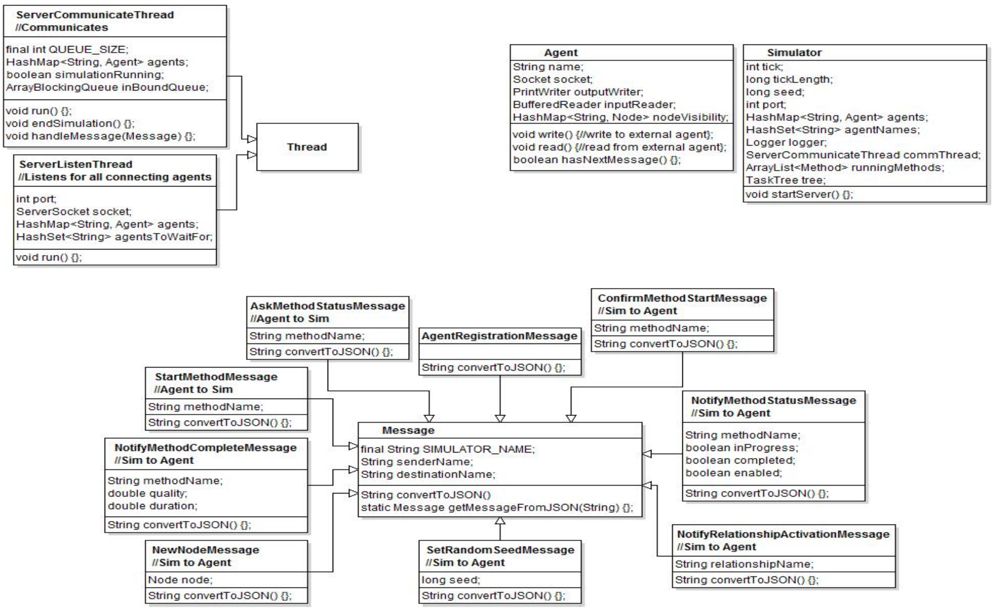

Fig. 5.6: UML diagram for Simulation package

- **Agent**: This class is used to encapsulate all the relevant information about an Agent such as its name, and HashMap of visible Nodes. This class also contains a PrintWriter and BufferedReader that are used to communicate with the external Agent program that it represents.

- **ServerListenThread**: This Thread will listen for new Agents that are trying to connect to the simulation. If an Agent connects with a valid name then this thread packages the Agent's information into an Agent object so that it can be accessed and communicated with throughout the simulation.

- **ServerCommunicateThread**: This Thread will create and process all Messages coming in from and going out to all Agents in the simulation. It contains a queue of received messages that the Simulator will dump and process at the start of each tick.

- **Message**: This class contains the base information for a Message such as the senderName, and the destinationName. There are many subtypes of Messages listed below. It also has a factory method to allow the creation of a new Message from a received JSON string.

  (i) StartMethodMessage: Tells the Simulator that the Agent wants to start a Method.

(ii) ConfirmMethodStartMessage: Tells an Agent that they have successfully started a Method.

(iii) AskMethodStatusMessage: Ask the Simulator for the status of a currently running Method.

(iv) NotifyMethodStatusMessage: Tells an Agent the status of a currently running Method.

(v) NotifyMethodCompleteMessage: Tells an Agent that their Method has completed and what the result is.

(vi) NewNodeMessage: The Simulator tells the Agent about it's initial visible nodes.

(vii) SetRandomSeedMessage: Tells an Agent to set their random number seed to a specified number.

(viii) NotifyRelationshipActivationMessage: Tells an Agent that a NodeRelationship was activated and what the result is.

(ix) AgentRegistrationMessage: Ask the Simulator to register this Agent into Simulation.

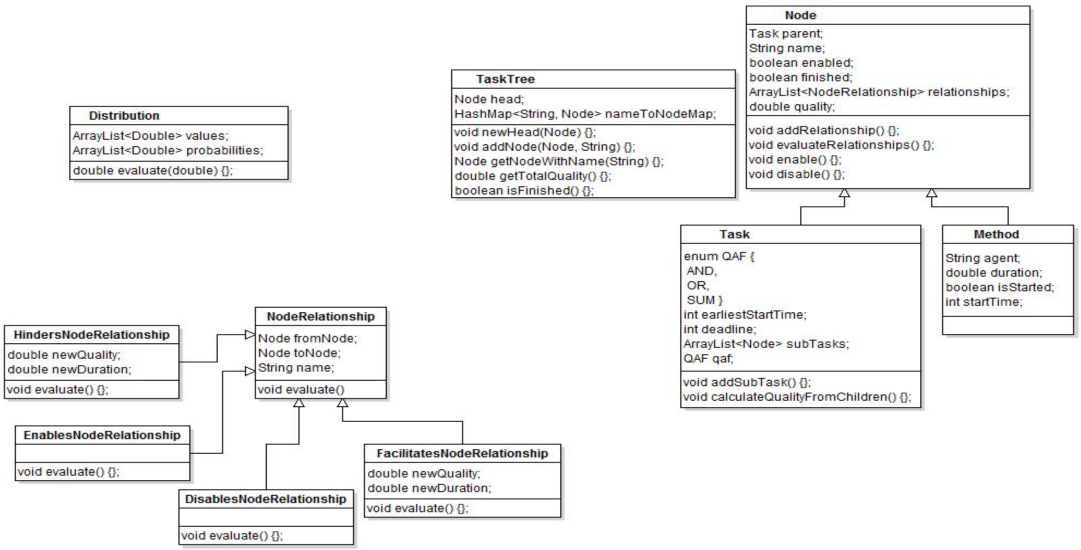

Fig. 5.7: UML diagram for tasktree package

5. **Tasktree** This package provides a framework for building a Task Tree. It is used by the input package to construct a programmatic representation of the SFI including Task, Methods, and Node relations.

- **TaskTree**: A wrapper class for the whole Task Tree. Contains a reference to the head Node of the tree. Also contains some methods to allow easily information gathering about the tree.
- **Node**: The base Node class that contains all the information about a Node of the tasktree such as Quality, name, this Nodes parent Node, a list of NodeRelationships from this Node, whether or not the Node is enabled, and whether or not the Node is finished.

- **Task**: A Task is a type of Node whose completion is defined by its Subtasks. A Task's completion quality is the result of it's QAF being run. More Quality options can be added as needed. A Subtask can either be another Task which would have more Subtasks or a Method which has no Subtasks.

- **Method**: A Method is a Node that can be completed by a specific Agent. The Method contains a Duration which is the amount of time it takes to complete the Method. Methods can have their Duration and Quality modified by the FacilitatesNodeRelationship and HindersNodeRelationship.

- **NodeRelationships**: There are many NodeRelationships that represent the affect that the completion of one Node has on another. The types of NodeRelations are below.

  (i) HindersNodeRelationship: Completing one Node decreases the Quality and increases the Duration of a Method.

  (ii) FacilitatesNodeRelationhip: Completing one Node increases the Quality and decreases the Duration of a Method.

  (iii) EnablesNodeRelationship: The completion of the first Node enables the starting of the second Node

  (iv) DisablesNodeRelationship: The completion of the first Node disables the starting of the second Node.
- **Distribution** A Distribution is used to generate Qualities and Durations for the Nodes. It is given a probability distribution specified in the cTAEMS file and a random number. The Distribution then calculates which value it should return.

Chapter 6

Verification Plan

6.1 Verification Plan

The verification plan and testing is the process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements. We write a test prior to implementing a requirement in order to cover the requirement. Post implementation of the requirement we write additional tests to cover any statements which were not covered by the previous tests. The testing strategy for the Multi-Agent System Simulator is a combination between black box and white box testing.

1. **Black Box Testing:** The objective of black-box testing is to verify the functionality of the Simulator. Here, each module is treated as a 'black-box', whose internals cannot be seen. We examine the specification of each module by defining different input scenarios that would result in different behaviour. We use the input and the output packages and perform tests to verify the desired output and the state of the simulator.

2. **White Box Testing:** For the purpose of white box testing we will be making use of test coverage tools to measure the statement coverage, branch coverage and missed lines. Each module and subsequent files related to the module will generate these statistics. We test the application module, input module, simulation module, task tree module and the output module which forms the overall structure of the Multi-Agent System Simulator. For each of these modules we aim at achieving 95-100

   - **Unit Testing:** Unit tests are performed during the implementation process on individual units of the source code. For unit testing, all the test cases shall be implemented using the JUnit Test Suite. For each test case, there shall be an expected output and an actual output. If the actual output matches the expected output, the test case shall pass; otherwise, it shall fail. In table 6.1 and 6.2 we represent all the tests performed so far.

Table 6.1: Test Status Report

Test ID	Test Class	Test Name	Description	Status
T1	ConfigurationDataTest	testMakeDefault, basicConstructorTest	Tests the configuration file data and generation of a default configuration file.	Success
T2	ConfigurationParserTest	testParse	Test if the data is parsed in accordance to the ConfigurationParser	Success
T3	FileErrorTest	testToSpring, testFileError, testExceptions	Test for correctness for the input file.	Success
T4	FileExceptionsTest	testExceptions	Test the input and configuration files for handling exceptions.	Success
T5	InputParserTest	basicConstructorTest, testParse, exceptionTest, testNotNull	Test for valid input data and exception handling	Error
T6	OrderedLoggerTest	testformattFileDestinationOne	Test the log events for all tasks and methods	Success
T7	SequentialLoggerTest	testformattFileDestinationOne, testformattFileDestinationOne	Test the format of the generated logger file.	Success
T8	AgentRegistrationMessageTest	toJSONTest	If the AgentRegistrationMessage is in accordance with JSON structure.	Success
T9	AskMethodStatusMessageTest	toJSONTest	If the AskMethodStatusMessage is in accordance with JSON structure.	Success
T10	ConfirmMethodStartMessageTest	toJSONTest	Checks status for start method message.	Success
T11	MessageTest	getMessageFromJSONUnknownTest, getMessageFromJSONTest	Test for all message exchanges.	Success

Table 6.2: Test Status Report

Test ID	Test Class	Test Name	Description	Status
T12	NextTickMessageTest	toJSONTest	Returns the status for method NextTickMessage	Success
T13	NotifyMethodCompletedMessageTest	toJSONTest	Returns the status for method on competition.	Success
T14	NotifyMethodStatusMessageTest	toJSONTest	Returns the method status.	Success
T15	NotifyRelationshipActivationMessageTest	toJSONTest	Returns the relationship and the sender.	Success
T16	ServerCommunicateThreadTest	testRun, testHandleMessage, testBadHandleMessage, tesBadRun toJSONTest	Check if messages are passsed from server to agent using sockets.	Success
T17	ServerListenThr- ead-Test	testBadRunSocketFail, testBadRunCreateFail, testBadRunNullName, testBadRunReadFail, testBadRunAcceptFail	Check if messages are passed from agents to server using sockets.	Success
T18	SetRandomSeedMessageTest	toJSONTest	Test the message communication for random seed.	Success
T19	StartMethodMessageTest	toJSONTest	Checks if method initializes	Success
T20	AgentTest	testAgentName, testTaskTree, testSocket	Test the socket connection and the new messages from the generated task tree.	Success
T21	SimulatorTest	testSimOneAgent, testSimFinishesTreeDone, testAgentDoMethod	Test the message exchanges in simulator for multiple agents for given input data.	Success
T22	DisablesNodeRelationshipTest	testEvaluate	Tests for the node relationship for methods.	Success
T23	DistributionTest	testEvaluate	Test to generate quality and duration for nodes.	Success

Table 6.3: Test Status Report

Test ID	Test Class	Test Name	Description	Status
T24	EnablesNodeRelationshipTest	testEvaluate	Test enables relationship for nodes.	Success
T25	FacilitatesNodeRelationshipTest	testEvaluate	Tests facilitates relationship for nodes.	Success
T26	HindersNodeRelationshipTest	testEvaluate	Tests hinders relationship for nodes.	Success
T27	MethodTest	testMethodGetSet	Tests if method returns all the attributes.	Success
T28	TaskTest	testCalculateQualityFromChildren, testCheckIsFinished, testSetQAF	Tests task tree status for a given set of tasks.	Success
T29	TaskTreeTest	testActivateAllRelationships, testGetTotalQuality, testGetTotalQuality, testAddNode, testComputeEarliestStartAndDeadline, testTaskTree, testFindVisibleToAgents, testNewHead, testIsFinished	Tests generation of task tree and nodes based on the input task and methods.	Success
T30	InputDataTest	basicConstructorTest		Success
T31	EndSimulationMessageTest	toJSONTest	Checks method return message (endSimulationMessage)	Success
T32	InitialTreeMessageTest	testToJSON	Checks the return value for a given task using the task tree	Success
T33	StartSimulationMessageTest	toJSONTest	Checks if method initializes and returns message StartSimulationMessage	Success

- **Integration Testing:** In this testing strategy we aim at testing combined parts of the application to determine if they function correctly together. We use a Bottom-up Integration approach where we first perform unit testing, followed by tests of progressively higher-level combinations of modules. Figure 6.1 shows the component modules that form the MASS application. In order to decide the number of integration tests we consider the following two criterion's: Check that all data exchanged across an interface agrees with the data structure specifications and confirm that all the control flows have been implemented.

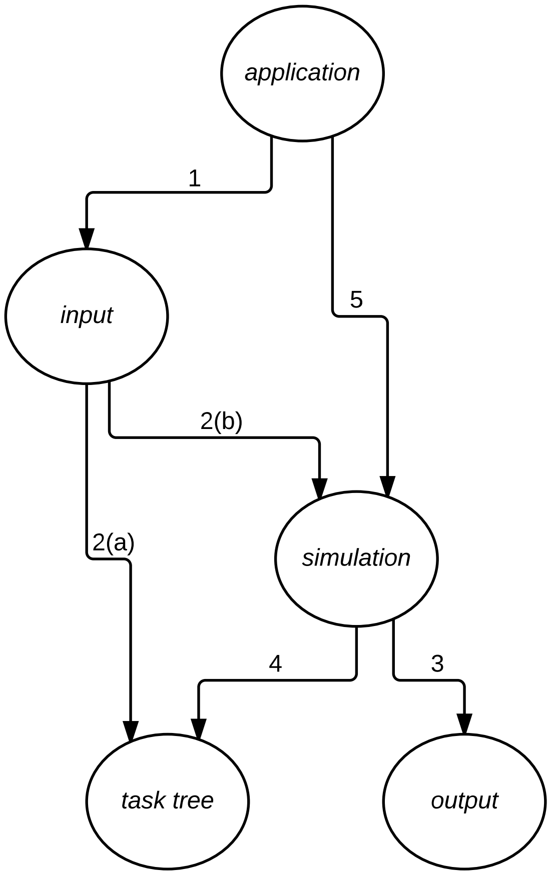

Fig. 6.1: Overview of the USES diagram for packages within MASS.

– **Input Module:** The input module takes in two types of flies: Configuration File Input (CFI) and Simulation File Input (SFI). We classify this input data into a set of equivalence classes. For any given error, input data sets in the same equivalence class will produce the same error.

  Equivalence Classes for Configuration File Input (CFI):
  Boundary Class: No Configuration File: We classify this scenario as a

boundary class and not as an Illegal class as this would not lead to the termination of the simulation. If no configuration file is submitted, the system will make one in the current directory with default values.

Illegal Class: Wrongly Formatted Configuration File: We classify this scenario as an illegal class as this would result in the termination of the simulation.

Nominal Class: Correct Configuration File: We classify this scenario as a nominal class. A correctly formatted configuration file wouldn't cause an immediate failure of the simulation.

Equivalence classes for Simulation File Input (SFI):
Boundary Class: We don't support a boundary class in the Simulation File Input. We employ a strict format standard for the Simulation File Input.

Illegal Class: Wrongly Formatted Configuration File: We classify this scenario as an illegal class as this would result in the termination of the simulation. Any Simulation File Input (SFI) is considered to be wrongly formatted if it is unable to parse correctly.

Nominal Class: Correct Simulation File Input (SFI): We classify this scenario as a nominal class. Any correctly formatted Simulation File Input (SFI) would not cause a failure of the simulation.

- **Output Module:** The output module takes in all the Agent communication transcriptions along with other statistical parameters and produce a Log File Output(LFO).

  Equivalence Classes for Log File Output(LFO):

  Boundary Class: We don't support a boundary class in the Log File Output. We employ a strict format standard for the Log File Output.

  Illegal Class: In the event that an agent transcription does not reach the logger we classify this scenario as an illegal class.

  Nominal Class: If all the transcriptions and statistics are recorded and filed by the Logger into the Log File Output we classify this scenario as a nominal class.

- **Simulation Module:** The simulation module takes the parsed input files, and is responsible for the communication process with and between agents. Several unit tests have been performed on this module to insure its integrity. However we do intend to perform integration tests on the simulation package as well.

- **System Testing:** After Integration testing we test the system as a whole. Once all the components are integrated, the application as a whole is tested rigorously to see that it meets quality standards.
- **User Acceptance Testing:** After all the Unite, Integration and System tests are performed we will perform acceptance tests as per the clients requests to determine if the requirements of a specification are met.

6.2 Testing Results

1. **Sprint 3:**

   After sprint 3 there are a total of 24 tests with 1 error, 0 skipped, 1 failures and a success rate of 91.6%. The most significant packages are comunication, simulatestate, tasktree, input and output, however we had only performed tests on communication, tasktree and output packages and had achieved instruction coverage of 51%, 71% and 64%, for the respective packages. The branch coverage for communication package is 47%, tasktree package is 76%, and output package is 54%. We had achieved an overall instruction coverage of 47% and a branch coverage of 42%

2. **Sprint 4:**

   After sprint 4 there are a total of 56 tests with 2 error, 0 skipped, 0 failures and a success rate of 96.43%. The most significant packages are comunication, simulatestate, tasktree, input and output and have achieved instruction coverage of 100%, 99%, 93%, 95% and 97% for the respective packages. The branch coverage for communication package is 81%, simulatestate package is 88%, tasktree package is 93%, input package is 89% and output package is 92%. The main application package is not tested in sprint 4, but will be done before sprint 5. We have achieved an overall instruction coverage of 97% and a branch coverage of 88% .

3. **Sprint 5:**

   After sprint 5 there are a total of 64 tests with 1 error, 0 skipped, 0 failures and a success rate of 98.44%. The most significant packages are communication, simulatestate, tasktree, input and output and have achieved instruction coverage of 97%, 99%, 99%, 97% and 92% for the respective packages. The branch coverage for communication package is 82%, simulatestate package is 85%, tasktree package is 94%, input package is 92% and output package is 89%. The thread creation and utilization does tend to vary the coverage but is not significant an alternate way to get a higher coverage is to create integration tests for the modules using threads. The main application package is not tested in sprint 4, but will be done before sprint 5. We have achieved an overall instruction coverage of 97% and a branch coverage of 88% .

Chapter 7

Conclusion

Overall, a comprehensive list of requirements has been created. Due to the explorative and innovative character of the project, it is rather a broad and deep insight in the examined area of Multi-Agent System Simulator. Thus, it should facilitate further focusing on project aims and guide to successful case studies and prototypes. The latter will be used to identify new or still overseen demands.

7.1 Glossary

- **Multi-Agent System**: A project that includes two or more independent software Agents working to complete a common goal.
- **Multi-Agent System Simulator (MASS)**: The specific computer program being outlined in this document.
- **Simulation**: A generic instance in which the MASS is running on a given input.
- **Agent**: A user defined software program that can communicate with the Simulation through a TCP socket connection.
- **cTAEMS**: A derivative of the TAEMS language that will be employed for specifying task domains.
- **Task**: A high level goal within the system.
- **Subtask**: A Task that is the child of another Task.
- **Method**: An action executed by an Agent in order to complete a Task or Subtask. Each Method is assigned a Duration and Quality for completion. It is also assigned to only one Agent that is allowed to execute it.
- **Node**: An individual Method or Task.
- **Task Group**: A high level grouping of Tasks that share a similar structure or goal.
- **Quality**: A numeric value used to measure the degree of satisfaction resulting from a particular Node.
- **Duration**: A numeric value used to measure the time it takes to complete an Node.
- **Quality Accumulation Function (QAF)**: A properties of Tasks that is the logic behind how a Task is deemed completed. Also determines how the Quality of a Task is calculated.
- **AND**: A QAF where all Subtasks of a Task must be completed for that Task to be completed. Quality is the minimum of all Subtasks' Qualities.
- **OR**: A QAF where at least one Subtask of a Task must be completed for that Task to be completed. Quality is the maximum of all Subtasks' Qualities.
- **SUM**: A QAF where at least one Subtask of a Task must be completed for that Task to be completed. Quality is the sum of all Subtasks' Qualities.

- **Relationship**: A linking between two Nodes. This linking causes the completion of one Node to directly affect the state of another.
- **Enables**: A relationship where a Node can allow the execution of another Node.
- **Disables**: A relationship where a Node can disallow the execution of another Node.
- **Hinders**: A relationship where a Node can negatively affect the Quality, and Duration of a Method. It is important to note that only a Method can be on the receiving end of this relationship.
- **Facilitates**: A relationship where a Node can positively affect the Quality, and Duration of a Method. It is important to note that only a Method can be on the receiving end of this relationship.
- **Task Tree**: A way to represent Nodes and the Relationships between them.
- **Tick**: A one unit advancement of an internal counter used as a clock.