ENHANCING AGENT CAPABILITY IN A LARGE SIMULATION SYSTEM

A THESIS SUBMITTED TO THE GRADUATE DIVISION OF THE UNIVERSITY OF HAWAI'I IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

IN

ELECTRICAL ENGINEERING

December 2005

By

Vengfai U

Thesis Committee:

Nancy Reed, Chairperson
Galen Sasaki
Tep Dobry
We certify that we have read this thesis and that, in our opinion, it is satisfactory in scope and quality as a thesis for the degree of Master of Science in Electrical Engineering.
ABSTRACT

This paper presents an enhanced agent model to improve the flexibility for creating different agent capabilities in large simulation systems. The simulation system we are improving is called RoboCup Rescue Simulation System, also known as RCRSS. The RCRSS is an environment simulator which simulates natural disasters such as earthquake. Currently the agent model in the RCRSS is not flexible to support mixed agent behaviors. In order to enhance agent capabilities, we designed a solution based on the enhanced agent mode which is specialized into different agent models with different behaviors that can be applied in the rescue simulation system. The solution extends the agent development framework called YabAPI to include the Helper Civilian model into the RCRSS. The aim is to simulate situations where agents with multiple roles who are not helpers can help in rescue. Performance enhancement should result in the population with Helper Civilian (HC) agents, which are configured to have partial or full capabilities, over to the population without any extra capabilities.

The experimental results show that the scenarios with a configured HC population consistently outperformed a pure civilian population. Performance increased by over 100 percent in terms of the overall score, the number of surviving agents, and the overall agent health condition. This enhanced agent behavior shows significant impact in rescue simulations. The range of agent behaviors increases enabling simulation of more complex scenarios than previously possible with more realism.
# TABLE OF CONTENTS

Abstract ........................................................................................................ iv
List of Figures ............................................................................................. vii
List of Tables ............................................................................................... viii

1. Introduction ............................................................................................ 1

2. RCRSS .................................................................................................... 3
   2.1. Background ..................................................................................... 3
   2.2. RCRSS Overview ......................................................................... 4
   2.3. World Model .................................................................................. 7
   2.4. Agents ............................................................................................. 9
   2.5. Communication Model .................................................................. 11
       2.5.1. Low-Level Communication ................................................ 12
       2.5.2. High-Level Communication ................................................. 14
   2.6. Simulation Process ........................................................................ 16
       2.6.1. Pre-Stage ............................................................................. 16
       2.6.2. On-Stage ............................................................................. 17
   2.7. Agent Development Framework .................................................. 19
       2.7.1. YabAPI ................................................................................. 19
       2.7.2. R_Civilian .......................................................................... 21
   2.8. Summary ...................................................................................... 24

3. Enhanced Agent Model – HelperCivilian ............................................. 25
   3.1. Background ................................................................................... 25
   3.2. Concept .......................................................................................... 27
   3.3. Design ........................................................................................... 27
   3.4. Type Installation ............................................................................ 27
   3.5. Unleashing Capabilities .................................................................. 34
   3.6. Configurability .............................................................................. 37
   3.7. Synchronization ............................................................................ 38
   3.8. Learning Infrastructure .................................................................. 41
   3.9. Default Behavior .......................................................................... 43
   3.10. GIS ............................................................................................... 46
   3.11. JGISEdit ....................................................................................... 46
   3.12. Summary .................................................................................... 47

4. Simulation ............................................................................................. 48
   4.1. Introduction ................................................................................... 48
   4.2. Process ........................................................................................... 48
       4.2.1. Planning ............................................................................... 49
       4.2.2. Execution ........................................................................... 53
   4.3. Method ........................................................................................... 56
   4.4. Score .............................................................................................. 57
5. Results ................................................................................................. 59
  5.1. Introduction .................................................................................. 59
  5.2. Results ....................................................................................... 59
  5.3. Analysis ...................................................................................... 68

6. Conclusion ......................................................................................... 69

References ............................................................................................. 72
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Architecture View of RoboCup Rescue Simulator</td>
<td>6</td>
</tr>
<tr>
<td>2.2</td>
<td>Class Hierarchy of Objects in the Disaster Space</td>
<td>8</td>
</tr>
<tr>
<td>2.3</td>
<td>Disaster Space</td>
<td>8</td>
</tr>
<tr>
<td>2.4</td>
<td>Building Objects</td>
<td>8</td>
</tr>
<tr>
<td>2.5</td>
<td>A Humanoid on a Road</td>
<td>8</td>
</tr>
<tr>
<td>2.6</td>
<td>Road Objects</td>
<td>9</td>
</tr>
<tr>
<td>2.7</td>
<td>Node and Edge Objects</td>
<td>9</td>
</tr>
<tr>
<td>2.8</td>
<td>Representative Points and Visual Range</td>
<td>9</td>
</tr>
<tr>
<td>2.9</td>
<td>Agent-to-Kernel Communication during Initialization</td>
<td>13</td>
</tr>
<tr>
<td>2.10</td>
<td>Agent-to-Kernel Communication during a Simulation Cycle</td>
<td>14</td>
</tr>
<tr>
<td>2.11</td>
<td>Agent-to-Agent Communication during Simulation</td>
<td>15</td>
</tr>
<tr>
<td>2.12</td>
<td>Architecture of YabAPI Framework</td>
<td>20</td>
</tr>
<tr>
<td>2.13</td>
<td>Architecture of R_Civilian</td>
<td>22</td>
</tr>
<tr>
<td>2.14</td>
<td>Decision Making Process of R_Civilian</td>
<td>23</td>
</tr>
<tr>
<td>3.1</td>
<td>RCRSS Package Organization</td>
<td>25</td>
</tr>
<tr>
<td>3.2</td>
<td>High Level Class relationship</td>
<td>28</td>
</tr>
<tr>
<td>3.3</td>
<td>Class Aggregation Association</td>
<td>28</td>
</tr>
<tr>
<td>3.4</td>
<td>Class Members of HelperCivilianAgent</td>
<td>29</td>
</tr>
<tr>
<td>4.1</td>
<td>The Editor Environment in JGISEdit</td>
<td>50</td>
</tr>
<tr>
<td>4.2</td>
<td>The Property Edit Panel of JGISEdit</td>
<td>51</td>
</tr>
<tr>
<td>4.3</td>
<td>The Execution of the RCRSS Simulation</td>
<td>53</td>
</tr>
<tr>
<td>4.4</td>
<td>Viewer’s presentation of RCRSS Simulation</td>
<td>54</td>
</tr>
<tr>
<td>4.5</td>
<td>Real-Time Scores Captured in Viewer’s Console</td>
<td>56</td>
</tr>
<tr>
<td>4.6</td>
<td>Agent Activities Captured in YabAPI’s Console</td>
<td>56</td>
</tr>
<tr>
<td>5.1</td>
<td>Official Score (V) versus Helper Civilian Population</td>
<td>60</td>
</tr>
<tr>
<td>5.2</td>
<td>Number of Surviving Agents (N) versus Helper Civilian Population</td>
<td>61</td>
</tr>
<tr>
<td>5.3</td>
<td>Agent Health Point Ratio (HP) versus Helper Civilian Population</td>
<td>62</td>
</tr>
<tr>
<td>5.4</td>
<td>Non-burned building Ratio (NB) versus Helper Civilian Population</td>
<td>63</td>
</tr>
<tr>
<td>5.5</td>
<td>Execution Time versus Helper Civilian Population</td>
<td>64</td>
</tr>
<tr>
<td>5.6</td>
<td>Mean Agent Health Point versus Helper Civilian Population</td>
<td>65</td>
</tr>
<tr>
<td>5.7</td>
<td>Standard Derivation of Agent Health Point versus Helper Civilian Population</td>
<td>65</td>
</tr>
<tr>
<td>5.8</td>
<td>Number of Rescue actions issued versus Helper Civilian Population</td>
<td>67</td>
</tr>
<tr>
<td>5.9</td>
<td>Number of Clear actions issued versus Helper Civilian Population</td>
<td>68</td>
</tr>
</tbody>
</table>
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1.</td>
<td>Summary of Agent Capabilities</td>
<td>11</td>
</tr>
<tr>
<td>2.2.</td>
<td>RCRSS Protocol Messages</td>
<td>16</td>
</tr>
<tr>
<td>2.3.</td>
<td>Modules and Messages Types Mapping</td>
<td>16</td>
</tr>
<tr>
<td>3.1.</td>
<td>List of Files Customized to Support the HelperCivilian Model</td>
<td>26</td>
</tr>
<tr>
<td>3.2.</td>
<td>Capabilities of RCR Agents</td>
<td>27</td>
</tr>
<tr>
<td>3.3.</td>
<td>Source Code of <code>commands</code> in the <code>misc</code> Sub-Simulator</td>
<td>35</td>
</tr>
<tr>
<td>3.4.</td>
<td>Source Code of Function <code>rescue</code></td>
<td>36</td>
</tr>
<tr>
<td>3.5.</td>
<td>Source Code of Function <code>clear</code></td>
<td>36</td>
</tr>
<tr>
<td>3.6.</td>
<td>Source Code of <code>commands</code> in the <code>fire</code> Sub-Simulator</td>
<td>37</td>
</tr>
<tr>
<td>4.1.</td>
<td>Configuration of Attribute <code>skills</code></td>
<td>50</td>
</tr>
<tr>
<td>4.2.</td>
<td>Color Representation in Viewer</td>
<td>55</td>
</tr>
<tr>
<td>5.1.</td>
<td>Configuration of the Simulation Setting</td>
<td>59</td>
</tr>
<tr>
<td>5.2.</td>
<td>Systems' Configuration</td>
<td>60</td>
</tr>
</tbody>
</table>
CHAPTER 1
INTRODUCTION

This thesis presents improvements to the versatility of a multi-agent simulation system with our modified agent development framework. Ideally the Agent Development Framework (ADF) is flexible and extendable, in which agent developers can simulate ad hoc situations with high realism and build agents rapidly. The objective is to create more realistic rescue simulations within the popular RoboCup Rescue Simulation System (RCRSS).

The RCRSS has a lot to offer to the community. Not only has it provided a common tool for researchers to study rescue strategies and multi-agent systems, but also it promotes the spirit of collaboration through annual competitions and exchange of ideas through message board and email subscription [RCR]. We believe that it will improve awareness and allow people to be better prepared in the event of a disaster. The RCRSS project has brought researchers around the world together to improve and strengthen the system. It was version 0.43 when we first worked with the simulator, and it has now advanced to version 0.48. We look forward to the possibility that our efforts could one day be deployed for saving lives.

Our new framework can be applied to similar distributed multi-agent simulation systems. Our idea is to generalize the base agent model and keep it simple so that complex behavioral models can be rapidly developed. The enhanced base agent model is called Helper Civilian (HC). We realized this new framework by extending YabAPI, the agent development framework a part of the RCRSS [RCR]. The YabAPI framework is
quite easy to use but limits the developers with a small set of agent models. To show the impact of our HC model, we implemented HC and tested them on the RCRSS. Our experiments show a gain in flexibility and expandability; as a result, the simulator becomes more powerful.

The RCRSS was chosen as the base platform for our experiments, and the new ADF should cooperate with the existing agents. With the new ADF, we want to simulate situations we could not do before. We will walk you through the experiments in the coming chapters.

We first start with a detailed summary of the RCRSS architecture to provide background information. Then we will describe the details of our ADF and reveal the nuts and bolts of how we designed the HC model based on the YabAPI agents and the RCRSS. We will compare the new to the original architecture. We will discuss the applications of the HC model in creating agent behavior and the readiness of the new architecture in supporting agent learning. The result and discussion chapters detail the methods used in the simulations and give impact analysis on the experiments. Finally, we will summarize the thesis and address future works.
CHAPTER 2
RCRSS

2.1. Background

The Robo-Cup Rescue (RCR) project is a research effort inspired by researchers who witnessed the tragedy of the Hanshin-Awaji Earthquake near Kobe, which killed and injured 6000 people and caused massive damage to Japan’s economy [RCR]. The main purpose of the project is to provide emergency decision support by integration of disaster information, prediction, planning, and human interface [RCR]. A diverse spectrum of possibilities of this technology will contribute to the creation of the safer social system in the future [RCR].

The RCR project has gained popularity and acceptance in research communities around the world. The researchers gather in annual contests to showcase their results and compete with each other in controlled robotics and simulation test-beds. Annual competitions are organized in software simulation and hardware robotics groups [RCR]. Having established popularity, the annual contests have drawn remarkable attention.

The goal of the annual RoboCupRescue Robot League competitions is to increase awareness of the challenges involved in urban search and rescue (USAR) applications,
provide objective evaluation of robotic implementations in representative environments, and promote collaboration between researchers [ISD].

The simulation test-bed is known as RoboCup Rescue Simulation System (RCRSS), a distributed multi-agent simulation system. Agent researchers use RCRSS to study rescue strategies, and fine tune agent communications, agent coordination and agent behaviors to achieve better performance [RCR]. Because the researchers share a common platform, their efforts can be evaluated against each other.

2.2. RCRSS Overview

This chapter covers the fundamental concepts of the RCRSS in system architecture, communication infrastructure, simulation model, agent organization, and agent development framework.

The RCRSS is a time-base distributed simulation system which can be hosted on one or multiple machines. All system activities are synchronous to simulation cycles. The system consists of these major components – Geographical Information System (GIS), Effect Simulators (misc, traffic, fire, blockage, and earthquake), Kernel, Viewer, Agent Development Frameworks (YabAPI, and R_Civilian), Agents (civilians, police, fire-fighters, medicals, and centers) and World Model [Tak01].

The architecture of the RCRSS in Figure 2.1 illustrates the abstraction of different layers and communication links across the system [Tak01]. The kernel integrates all components and is the core of the system, which manages all communications among
remote agents, RCR agents and sub-simulators. All messages directed from remote
agents or sub-simulators must first go to the kernel, which dispatches the messages to the
appropriate recipients [Tak01].

The GIS module contains repository of geographical properties about a virtual city
for the entire system. Sub-simulators (effect simulators) simulate effects such as
building collapse, traffic congestion, fire spread, and human activities, and each is
responsible for one type of natural phenomenon [Tak01]. The viewer is a graphical utility
to view and replay simulations. In the distributed environment, each RCR agent is
associated with individual instances of itself in each module within the system; the
instances refer the same agent entity; therefore, the states of each instance of the agent
should be identical. We refer the server-side agents as the instances which reside in the
kernel and the sub-simulators, and refer the remote agents as the instances which reside in
the agent development frameworks (YabAPI and R_Civilian).
The simulation is preset for an amount of time, in which all events and activities are logged. A simulation can be played back in the viewer. The simulation score is calculated by the viewer based on the criteria such as agent survival rate, agent health rate, and building destruction rate.

In the RCRSS there are two categories of human agents – rescuers and civilians. The sub-simulators generate fire and earthquake effects to add challenges to the human agents, while the human agents are to rescue injured civilian agents, stop fire spread, and help civilian evacuate within the given time period.

All human agents have the basic ability to sense, hear, say, and move; each type of rescue agent has its unique rescue capability. This agent model forces the developers to
collaborate multiple types of agents to perform in rescue operations by deploying some kind of strategies. However, the developers are left with no option to simulate complex ad hoc situations. An example ad hoc situation would be where the civilians help in rescues with partial capabilities; the current RCRSS does not support it.

The reason why we want to simulate these ad hoc situations is that we want to simulate certain reality scenarios, like civilians helping in rescue. By adding these reality scenarios, the new model will provide more flexibility and extended benefits to the RCRSS research community.

2.3. World Model

The world model is the data structure organized in a collection of Objects in a common hierarchy as shown in Figure 2.2 [Mor01]. It manages all entities in the virtual city including buildings, roads, connections, non-human agents and human agents. All human agents possess partial world models as to the kernel and the sub-simulators possess complete models. Each object in the model shares some common properties like position, shape, and ID. Individual agents have specialized properties. The world model uses Nodes to implement connections based on graph theory. Using that, the path between any two locations can be determined. The Nodes model building entrances and end-points of Roads; they also model the road-to-road and road-to-building conjunctions. The illustrations about Nodes are shown in Figure 2.3, 2.4, 2.6 and 2.7 [Mor01].
Figure 2.2. Class Hierarchy of Objects in the Disaster Space

(a) All objects are linked by their topological properties

(b) View of a disaster space

Figure 2.3. Disaster Space

Figure 2.4. Building Object

Figure 2.5. A Humanoid on a Road
Each object appears as two dimensional and each representative point (the dots in Figure 2.8) represent a location [Mor01]. The distance of any two objects is measured by their representative points. They are usually the centers of Buildings and Nodes or the midpoints of Roads. When a Humanoid is situated on a Building or a Node, its representative point becomes that of the object it is situated on; on a Road, its representative point becomes positionExtra away from the head of the Road, as illustrated in Figure 2.5 [Mor01].

2.4. Agents

The RCR agents exist in two forms - server-side agents and remote agents. Remote agents are the agents existed within an agent development framework (ADF)
such as YabAPI or R_Civilian; the remote agents implement the behavioral logic. Each remote agent relates to one server-side agent. The server-side agents are maintained by the kernel and effect simulators. The server-side agents and the remote agents together provide the facilities to synchronize their states among the system modules.

The agents are organized using the same hierarchy as in Figure 2.1 used by the world model. The human agents are the Humanoid; they are movable. The non-human agents are the center agents; they are non-movable; they model the communication centers. The Buildings, Roads, and Nodes are non-agents.

The agents defined in the RCRSS are Civilian, AmbulanceTeam, FireBrigade, PoliceForce, AmbulanceCenter, FireStation, and PoliceOffice. Human agents can be specialized into Civilian and Rescuer (AmbulanceTeam, FireBrigade, and PoliceForce). A friendlier way to remember the names for AmbulanceTeam, FireBrigade, and PoliceForce are paramedics, fire-fighters, and police, respectively. The center agents are AmbulanceCenter, FireStation, and PoliceOffice [Mor01].

The behavioral models are implemented by the remote agents; they act upon their behavioral models and interact with the kernel. All human agents have the basic ability to sense, hear, say, and move; the rescuer agents have their unique capabilities. The agents are designed to model human beings. They see by sense, hear voice by hear, speak by say, speak to a telecomm device by tell, and walk by move [Mor01]. Some of them are capable of rescue, load, unload, extinguish, and/or clear, and not all agents are capable of all the actions [Mor01].
The center agents broadcast incoming messages to every agent associated to their
groups. The rescuer agents call their associated centers to communicate with fellow
team members over long distance. The idea is similar to walkie-talkie system.

We summarized the capabilities for each type of RCR agents in Table 2.1 [Mor01].

<table>
<thead>
<tr>
<th>Type</th>
<th>Capabilities</th>
</tr>
</thead>
<tbody>
<tr>
<td>Civilian</td>
<td>Sense, Hear, Say, Move</td>
</tr>
<tr>
<td>Ambulance Team</td>
<td>Sense, Hear, Say, Tell, Move, Rescue, Load, Unload</td>
</tr>
<tr>
<td>Fire Brigade</td>
<td>Sense, Hear, Say, Tell, Move, Extinguish</td>
</tr>
<tr>
<td>Police Force</td>
<td>Sense, Hear, Say, Tell, Move, Clear</td>
</tr>
<tr>
<td>Ambulance Center</td>
<td>Sense, Hear, Say, Tell</td>
</tr>
<tr>
<td>Fire Station</td>
<td>Sense, Hear, Say, Tell</td>
</tr>
<tr>
<td>Police Office</td>
<td>Sense, Hear, Say, Tell</td>
</tr>
</tbody>
</table>

Although Civilian agents (R_Civilian) are currently programmed to have no rescue
skills, it should not be a fundamental limitation. Ideally, the civilians could be
“randomly” assigned to have some training in first-aid knowledge, to get them ready to
help in rescue and/or to clear debris.

2.5. Communication Model

The communication infrastructure is what makes the distributed system work.
We will cover the low-level communications - the handshaking mechanisms used in
connection setup, agent-to-kernel communication and agent-to-agent communication – as
well as the high-level communications among the remote agents. The communication is
based on the protocol defined in the RCRSS Protocol [Mor01].
2.5.1. **Low-Level Communication**

The low-level communications are the building blocks of the high-level communications or scenarios. The activities involving low-level communications can be summarized as the following:

- The agents initiate connections to the kernel
- The agents acknowledge a message from the kernel
- The kernel signals an error message
- The kernel requests agents for submitting action commands
- The agents submit action commands
- The kernel delivers voice messages to the remote agents

One of the most complicated scenarios is when an agent tries to connect to the kernel to initiate a connection. In this scenario a series of low-level communication activities are involved.

When a remote agent wants to initiate a connection, it first sends AK_CONNECT to the kernel. The kernel would either respond with KA_CONNECT_OK implicating the connection is successful, or KA_CONNECT_ERROR implicating a reject or failure.

When the remote agent gets the kernel’s attention, it processes the information in the message. If the message is KA_CONNECT_OK, the agent configures itself as its server-side counterpart and initializes its local world model with the information embedded in the message. If, however, the message directed from the kernel is KA_CONNECT_ERROR, the connection is failed to establish. The cause could be either an incompatible value was sent or the kernel server ran out of resources to create more agents [Mor01].
In the following, the remote agent must send AK_ACKNOWLEDGE to the kernel to acknowledge the reception of KA_CONNECT_OK that indicates all information is received correctly and the agent are successfully signed up to the system. Finally, the connection is established, and the mapping between the remote agent and the server-side agent is formed.

Figure 2.9 illustrates the scenario we just discussed of how the connection to the kernel is setup for the remote agent [Mor01]. Should the kernel receive all the acknowledgments, the simulation begins immediately.

![Figure 2.9. Agent-to-Kernel Communication during Initialization](image)

Amidst the simulation, the events are synchronous to the simulation cycles. The kernel signals the remote agents to submit action commands (or requests) by sending KA_SENSE to them. KA_SENSE contains updates to the local world model of the remote agents. The remote agents respond to the kernel with the commands they want to perform if there is any. We illustrated the scenario in Figure 2.10 [Mor01].

The kernel accepts only one request per agent in any simulation cycle and ignores excessive ones. These could produce glitches to the simulation. Think about the situations when the agents are sometimes too busy to respond within the cycle, and when
they sometimes send requests too fast or too many; the agents got their requests lost and their actions delayed. These glitches are caused by the nature of the time-base simulation of the RCRSS. The simulation cycle time has the effect to how often these glitches happen.

![Diagram](image)

**Figure 2.10. Agent-to-Kernel Communication during a Simulation Cycle**

### 2.5.2. High-Level Communication

Amidst the simulation, when a remote agent wants to talk to another remote agent, they could send AK\_SAY or AK\_TELL to the kernel, along with the message they want to express. For Civilian, they are permitted to send AK\_SAY. For rescuer agents, they can send both. The rescuer agents send AK\_TELL to communicate over long distance.

This form of messages is voice messages, and the remote agents “listen” to obtain the message. The voice messages would usually take multiple cycles to get to the recipients.

The kernel determines the recipients who hear the message. If AK\_SAY is used, the kernel would direct the message to the agents which are within the hearing distance of the speaker agent. If AK\_TELL is used, the kernel directs the message to the appropriate center agent, which then forwards the message to the associated group of agents. The recipient agents get the message by receiving KA\_HEAR sent by the kernel.
The commands together carry the high level communication as shown in Figure 2.11 [Mor01].

![Figure 2.11: Agent-to-Agent Communication during Simulation](image)

The tables in the following provide a broader view on what messages are defined and how the messages are used in the system. Table 2.2 summarizes the messages used in communication among the remote agents and the kernel [Mor01]. There are total of fifteen different messages: seven of them are used by the agents and four of them are used by the kernel. Table 2.3 summarizes the mappings between the message types and the modules that handle the messages [Mor01].
Table 2.2. RCRSS Protocol Messages

<table>
<thead>
<tr>
<th>Value</th>
<th>Header</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>To the kernel:</strong></td>
<td></td>
</tr>
<tr>
<td>0x10</td>
<td>AK CONNECT</td>
<td>To request for the connection to the kernel</td>
</tr>
<tr>
<td>0x11</td>
<td>AK ACKNOWLEDGE</td>
<td>To acknowledge for the KA CONNECT OK</td>
</tr>
<tr>
<td>0x81</td>
<td>AK MOVE</td>
<td>To submit the will to move to another position</td>
</tr>
<tr>
<td>0x88</td>
<td>AK RESCUE</td>
<td>To submit the will to rescue an humanoid</td>
</tr>
<tr>
<td>0x82</td>
<td>AK LOAD</td>
<td>To submit the will to load an humanoid</td>
</tr>
<tr>
<td>0x83</td>
<td>AK UNLOAD</td>
<td>To submit the will to unload an humanoid</td>
</tr>
<tr>
<td>0x86</td>
<td>AK EXTINGUISH</td>
<td>To submit the will to extinguish a fire</td>
</tr>
<tr>
<td>0x89</td>
<td>AK CLEAR</td>
<td>To submit the will to clear a blockade</td>
</tr>
<tr>
<td>0x80</td>
<td>AK REST</td>
<td>To submit the will to do nothing</td>
</tr>
<tr>
<td>0x84</td>
<td>AK SAY</td>
<td>To submit the will to say something</td>
</tr>
<tr>
<td>0x85</td>
<td>AK TELL</td>
<td>To submit the will to tell something</td>
</tr>
<tr>
<td></td>
<td><strong>From the kernel:</strong></td>
<td></td>
</tr>
<tr>
<td>0x50</td>
<td>KA CONNECT OK</td>
<td>To inform of the success of the connection</td>
</tr>
<tr>
<td>0x51</td>
<td>KA CONNECT ERROR</td>
<td>To inform of the failure of the connection</td>
</tr>
<tr>
<td>0x52</td>
<td>KA SENSE</td>
<td>To send vision information</td>
</tr>
<tr>
<td>0x55</td>
<td>KA HEAR</td>
<td>To send auditory information</td>
</tr>
</tbody>
</table>

Table 2.3. Modules and Message Types Mapping

<table>
<thead>
<tr>
<th>Module</th>
<th>Handling commands</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernel</td>
<td>AK SAY, AK TELL</td>
</tr>
<tr>
<td>Traffic sub-simulator</td>
<td>AK MOVE, AK LOAD, AK UNLOAD</td>
</tr>
<tr>
<td>Fire sub-simulator</td>
<td>AK EXTINGUISH</td>
</tr>
<tr>
<td>Misc sub-simulator</td>
<td>AK RESCUE, AK CLEAR</td>
</tr>
</tbody>
</table>

2.6. Simulation Process

The simulation process consists of two stages - pre-stage and on-stage. We will discuss the events happening behind the scene of the simulation in each stage.

2.6.1. Pre-Stage

In the pre-stage of the simulation, the system first initiates the GIS module. The GIS module reads the data files and constructs the world model. It reads the data about roads, buildings, relative locations, and entity connections in the world model.
After the GIS module is initiated, then the kernel and the sub-simulators start. The kernel connects to the GIS to retrieve the world model for the simulation. The kernel instantiates server-side agents and incorporates them into the world model. The viewer and the sub-simulators get initiated in the time being. The viewer also connects to the GIS to retrieve the world model for the display. When the sub-simulators, the viewer, the kernel, and the GIS module complete their initialization, R_Civilian and YabAPI initiate. R_Civilian creates mappings for Civilian agents, and YabAPI creates mappings for agents within the framework. Mappings between the remote agents and the server-side agents are formed. The viewer is preparing to animate the simulation and report scores as soon as the simulation starts.

Prior to the first simulation cycle, the following activities take place:

- The collapse simulator shakes the city
- The fire simulator places initial spots where to start fires
- The blockage simulator places road blocks and destructions
- The misc (stands for miscellaneous) simulator plans for human injuries and accidents
- After all initializations are in place, the kernel activates the agents.
- The simulation officially commences

2.6.2. On-Stage

Amidst the simulation, the kernel manages the following sequence to be executed during each cycle:
• The kernel sends signal to invite for requests to the remote agents and update their local world models.

• The remote agents submit requests, if any, to the kernel (however, only one request per agent will be accepted).

• The kernel dispatches the incoming requests to itself or appropriate sub-simulators for handling.

• The sub-simulators process the requests and update back to the kernel.

• The kernel then updates the server-side agents, the remote agents, and the viewer.

• The viewer then updates the score and animation.

• The simulation sets off for the next cycle.

The kernel interacts with the remote agents and the effect simulators, and it manages the order of the event sequence until the simulation finishes.

During the simulation, the remote agents act according to their own behavioral models. Some behavioral models evolve along the simulations, some are straightly based on rules, and some are hybrid. The agents are updated their view of the world model also known as local disaster-space or local world-model. An agent’s view contains partial world model and develops in the agent’s cognition along the lifetime of the simulation.

Most events are synchronous to the simulation cycles but some are not. This adds glitches to the simulation and affects the modules that operate by the cycle frequency.
The duration of a simulation cycle defines the periodical time frame (window) that the kernel uses to collect requests. If the window gets increased, the remote agents can process more tasks in an extended cycle. If the window gets decreased, the same agent task would take more cycles to complete and the effects get delayed. The time base simulation is sensitive to the window size.

A simulation cycle is not equivalent to one real second; the rate of simulation can be changed. It depends on the cycle time, the number of agents, the complexity of the world model, communication among the agents, and the processing power.

2.7. Agent Development Framework (ADF)

The RCRSS provides two agent development frameworks – YabAPI and R_Civilian. R_Civilian is being used to deploy the Civilian agents, and the YabAPI is used to deploy the rescuer agents and the center agents. They are the available frameworks for developers to specialize agent behaviors. They both are used to host for remote agents, to interact with the kernel using the RCRSS Protocol, and to specialize agent behaviors by the agent developers. However, they differ in architecture and technology, and these factors dictate the adaptability, flexibility, and extendibility of the frameworks.

2.7.1. YabAPI

The YabAPI becomes a dominant tool in the agent development as it is designed as a common tool for rapid agent development. YabAPI is developed in Java programming language and with object-oriented design methodology. The choice of using these technologies to implement YabAPI makes it very favorable for developers to adapt. The YabAPI provides a simple and effective architecture in supporting all
existing RCR agents. The architecture of YabAPI is shown in Figure 2.12 [Mor02].

YabAPI appears to be an attractive choice of ADF for incorporating our HelperCivilian model.

The YabAPI module consists of four major blocks - yab.io, yab.agent, yab.io.object, and yab.agent.object. The yab.io is responsible for implementing the RCRSS Protocol for communication between the kernel and the remote agents. The yab.agent provides the agent skeletons for developers to specialize agent behaviors. The yab.agent.object is responsible for the data structure used to describe the local world model (disaster space) for the agents. The yab.io.object is responsible for providing the hooks between yab.io and yab.agent.object blocks.

Figure 2.12. Architecture of YabAPI Framework
In this architecture, IO and hook implementations are not exposed to agent developers who use YabAPI for specializing agent behaviors. The developers need not know any details about the communication protocol when using YabAPI. YabAPI has provided a rich set of utilities for various queries and agent commands needed for development.

The developers need only to create a new class based on anyone of the abstract classes in the yab.agent block, and then override the methods act and hear of the new class. The method act is called by the framework every cycle, and it defines the actions of the agent and behavior. The method hear is called by the framework when a voice message is “heard” by the agent. The agent uses the voice message as event, and the agent may ignore or respond to the event. A voice message is “heard” by the agent when another agent is communicating with the agent or another agent is talking in the neighborhood. The agent developers are responsible for implementing these two functions to specialize the agent behavior [Mor02].

2.7.2. R_Civilian

The R_Civilian is an agent development framework that utilities a decisions-making module called parallel scenario problem solver engine (PS2) to determine a Civilian agent’s behavior and responses to a situation [Shin00]. Figure 2.13 illustrates the architecture of R_Civilian [Shin00].
Each time PS2 evaluates a set of behavior rules applicable to the current state of the agent. Each rule defines the response to an event in a situation, and a collection of rules is grouped in an entity called *posit*. Posit stands for Part of Situation. A *posit* represents a portion of an agent's mental situation and describes the situational behaviors in *s-expression* or so called *lisp* programming language. A *posit* is analogous to an entity in a logic state diagram. Figure 2.14 illustrates the decision process architecture of R_Civilian [Shin00].

![Figure 2.13. Architecture of R_Civilian](image-url)
This framework is specialized to address two key characteristics of a normal civilian’s behavior – 1) a normal civilian is likely to change his/her interests or goals on different situations; and 2) a normal civilian is likely to have one or multiple goals at any given situation [Shin00], given the assumption that a Civilian agent should have multiple short-term goals and routine responses to the environment, instead of having deep thinking process committed to a single goal [Shin00].

R_Civilian is so specialized in Civilian behavior that using it for specializing agent behaviors for other agents would be ineffective, considering using lisp as the development language and the lack of support for other agent types. The overhead to extend this framework to support HelperCivilian is very high. R_Civilian does not appear to be an attractive ADF for implementing HelperCivilian model.
2.8. Summary

We covered the fundamental concepts of the RCRSS in system architecture, communication infrastructure, simulation model, agent organization, and agent simulation framework. We evaluated the agent development frameworks, and YabAPI shows up as the attractive platform to use for incorporating HelperCivilian. The conversion work should be manageable with YabAPI given a concisely defined architecture and user-friendly implementation technologies. The next chapter will discuss about the implementation of the HelperCivilian model based on the YabAPI framework.
CHAPTER 3
ENHANCED AGENT MODEL – HELPERCIVILIAN

3.1. Background

The HelperCivilian model was designed and implemented to work within the YabAPI and RCRSS. This model is generalized and simple to use for developers to rapidly develop new agent behaviors for the RCRSS. It is a powerful tool allowing the agents to be configurable with their capabilities, making it possible to change behaviors at design time and at run-time. With HelperCivilian, we can develop ad hoc agent behaviors we could not do before. Figure 3.1 shows the organization of the RCRSS package. The modules are developed in Java and C++ programming languages. To incorporate HelperCivilian into the existing architecture, we studied each module and identified the program elements that need to be modified which include the object definitions and certain system behaviors.

Figure 3.1. RCRSS Package Organization

The design and implementation of this HelperCivilian model represent our effort in improving this rescue simulation system. We pinpointed more than 39 files in our modification list in Table 3.1.

In this chapter we will cover the issues on - how to register the new agent type to the RCRSS; how to instantiate the agents; how to synchronize the states of the
HelperCivilian agents; how to enable HelperCivilian with additional skills; how it is possible for the agents to learn new skills from other agents; how we customized the GIS and JGISEdit modules to support HelperCivilian.

Table 3.1. List of Files Customized to Support the HelperCivilian Model

<table>
<thead>
<tr>
<th>File</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>objdef.h</td>
<td>\rcr\program\blockadessimulator\</td>
</tr>
<tr>
<td>objdef.h</td>
<td>\rcr\program\firesimulator\</td>
</tr>
<tr>
<td>main.cxx</td>
<td>\rcr\program\gis\</td>
</tr>
<tr>
<td>objdef.h</td>
<td>\rcr\program\gis\</td>
</tr>
<tr>
<td>basic.hxx</td>
<td>\rcr\program\kernel\</td>
</tr>
<tr>
<td>HelperCivilian.inl</td>
<td>\rcr\program\kernel\</td>
</tr>
<tr>
<td>objdef.enum.h</td>
<td>\rcr\program\kernel\</td>
</tr>
<tr>
<td>objdef.h</td>
<td>\rcr\program\kernel\</td>
</tr>
<tr>
<td>RescueSystem.cxx</td>
<td>\rcr\program\kernel\</td>
</tr>
<tr>
<td>System.cxx</td>
<td>\rcr\program\kernel\</td>
</tr>
<tr>
<td>main.cxx</td>
<td>\rcr\program\miscsimulator\</td>
</tr>
<tr>
<td>misc_objdef.h</td>
<td>\rcr\program\miscsimulator\</td>
</tr>
<tr>
<td>objdef.h</td>
<td>\rcr\program\miscsimulator\</td>
</tr>
<tr>
<td>basic.hxx</td>
<td>\rcr\program\R_Civilian\librescue\</td>
</tr>
<tr>
<td>objdef.enum.h</td>
<td>\rcr\program\R_Civilian\librescue\</td>
</tr>
<tr>
<td>objdef.h</td>
<td>\rcr\program\R_Civilian\librescue\</td>
</tr>
<tr>
<td>HelperCivilian.java</td>
<td>\rcr\program\traffic_morimoto\traffic\object\</td>
</tr>
<tr>
<td>PropertyConstants.java</td>
<td>\rcr\program\traffic_morimoto\traffic\object\</td>
</tr>
<tr>
<td>TypeConstants.java</td>
<td>\rcr\program\traffic_morimoto\traffic\object\</td>
</tr>
<tr>
<td>Viewer.java</td>
<td>\rcr\program\traffic_morimoto\traffic\</td>
</tr>
<tr>
<td>WorldModel.java</td>
<td>\rcr\program\traffic_morimoto\traffic\</td>
</tr>
<tr>
<td>HelperCivilian.java</td>
<td>\rcr\program\viewer-0.12\src\viewer\object\</td>
</tr>
<tr>
<td>PropertyConstants.java</td>
<td>\rcr\program\viewer-0.12\src\viewer\object\</td>
</tr>
<tr>
<td>TypeConstants.java</td>
<td>\rcr\program\viewer-0.12\src\viewer\object\</td>
</tr>
<tr>
<td>Viewer.java</td>
<td>\rcr\program\viewer-0.12\src\viewer\</td>
</tr>
<tr>
<td>WorldModel.java</td>
<td>\rcr\program\viewer-0.12\src\viewer\</td>
</tr>
<tr>
<td>HelperCivilianAgent.java</td>
<td>\rcr\program\yabapi\sample\src\sample\</td>
</tr>
<tr>
<td>Main.java</td>
<td>\rcr\program\yabapi\sample\src\sample\</td>
</tr>
<tr>
<td>AbstractHelperCivilianAgent.java</td>
<td>\rcr\program\yabapi\yab\src\yab\agent\</td>
</tr>
<tr>
<td>DisasterSpace.java</td>
<td>\rcr\program\yabapi\yab\src\yab\agent\</td>
</tr>
<tr>
<td>HelperCivilian.java</td>
<td>\rcr\program\yabapi\yab\src\yab\agent\object\</td>
</tr>
<tr>
<td>RCRObject.java</td>
<td>\rcr\program\yabapi\yab\src\yab\agent\object\</td>
</tr>
<tr>
<td>BaseHelperCivilian.java</td>
<td>\rcr\program\yabapi\yab\src\yab\object\</td>
</tr>
<tr>
<td>BaseRCRObject.java</td>
<td>\rcr\program\yabapi\yab\src\yab\object\</td>
</tr>
<tr>
<td>ProtocolConstants.java</td>
<td>\rcr\program\yabapi\yab\src\yab\</td>
</tr>
<tr>
<td>RCRSSProtocol.java</td>
<td>\rcr\program\yabapi\yab\src\yab\</td>
</tr>
</tbody>
</table>
3.2. Concept

The HelperCivilian is the generalized form of Civilian, PoliceForce and Ambulance models. The HC agents can be trained or non-trained. Non-trained HC agents are like the Civilian agents; they manage to evacuate in disasters. The fully trained HC agents are more powerful; in addition to the basic capabilities such as Sense, Hear, Say, and Move, they are capable of Rescue and Clear. Ideally, we want the HC model to support all possible available skills in the RCRSS. As for demonstration purpose, we simplified the design. Table 3.2 summarizes the capabilities of the agents in our extended platform.

Table 3.2. Capabilities of RCR Agents

<table>
<thead>
<tr>
<th>Type</th>
<th>Capabilities</th>
</tr>
</thead>
<tbody>
<tr>
<td>Helper Civilian</td>
<td>Sense, Hear, Tell, Move, Rescue, Clear</td>
</tr>
<tr>
<td>Civilian</td>
<td>Sense, Hear, Say, Move</td>
</tr>
<tr>
<td>Ambulance Team</td>
<td>Sense, Hear, Say, Tell, Move, Rescue, Load, Unload</td>
</tr>
<tr>
<td>Fire Brigade</td>
<td>Sense, Hear, Say, Tell, Move, Extinguish</td>
</tr>
<tr>
<td>Police Force</td>
<td>Sense, Hear, Say, Tell, Move, Clear</td>
</tr>
<tr>
<td>Ambulance Center</td>
<td>Sense, Hear, Say, Tell</td>
</tr>
<tr>
<td>Fire Station</td>
<td>Sense, Hear, Say, Tell</td>
</tr>
<tr>
<td>Police Office</td>
<td>Sense, Hear, Say, Tell</td>
</tr>
</tbody>
</table>

3.3. Design

We designed the HelperCivilian as a derived class of HumanoidAgent. It gains the human identity through the inheritance. We present the design in the UML diagrams as shown in Figure 3.2, 3.3, and 3.4.

In Figure 3.2, the class diagram models the inheritance between HumanoidAgent, AbstractHelperCivilianAgent and HelperCivilianAgent in a high level view. In Figure 3.4, the class diagram discloses further details including the class attributes, the methods,
the inheritance, and the association relationships. HelperCivilianAgent inherited the properties from HumanoidAgent; it specialized the methods *act* and *hear*, and used a special variable called *skill*, implemented in the BaseHelperCivilian class, to make capabilities configurable. In Figure 3.3, the class diagram models the aggregation in which each Agent has one DisasterSpace, and the DisasterSpace can have zero to many RCRObjec objects. The class diagrams depict the HelpCivilian model and describe the relationship among HelperCivilian and related entities in the system.

![Figure 3.2. High-Level Class Relationship](image)

![Figure 3.3. Class Aggregation Associations](image)
3.4. Type Installation

HelperCivilian was developed in C++ and Java programming languages. The files that have been customized to support the HelperCivilian were listed in Table 3.1.

The kernel, the blockades simulator, the fire simulator, the R_Civilian, the misc simulator, and the GIS modules require the C++ version of the HelperCivilian definition. Their agent definitions are organized in objdef.h, objdef.enum.h, and basic.hxx, and HelperCivilian.inl. Some modules may require more files.

The objdef.h was modified to include HelperCivilian.inl, to integrate with the HelperCivilian definition. The misc sub-simulator uses an extra file misc_objdef.h. The objdef.enum.h defines all property IDs and agent type IDs used in the system including
TYPE_HELPERCIVILIAN the type ID for HelperCivilian and

PROPERTY_RESCUE_SKILLS_MATRIX the property ID for the attribute skill.

const TypeId TYPE_HELPERCIVILIAN = 237;
const PropertyId PROPERTY_RESCUE_SKILLS_MATRIX = 1000;

In basic.hxx, it defines the enumeration constants, message type IDs and the primitive data types referenced across the system. The enumeration constant AGENTTYPE_HELPERCIVILIAN is the index to the array agentTypeToTypeId which maps the index to the agent type ID TYPE_HELPERCIVILIAN. The order of the enumeration constants are required to match with the order the type IDs arranged in the agentTypeToTypeId array.

```cpp
enum {
    AGENTTYPE_CIVILIAN,
    AGENTTYPE_FIRE_COMPANY,
    AGENTTYPE_FIRE_STATION,
    AGENTTYPE_AMBULANCE_TEAM,
    AGENTTYPE_AMBULANCE_CENTER,
    AGENTTYPE_POLICE_FORCE,
    AGENTTYPE_POLICE_OFFICE,
    AGENTTYPE_HELPERCIVILIAN,
    AGENTTYPE_MAIZ,
};

const TypeId agentTypeToTypeId[] = {
    TYPE_CIVILIAN,
    TYPE_FIRE_COMPANY,
    TYPE_FIRE_STATION,
    TYPE_AMBULANCE_TEAM,
    TYPE_AMBULANCE_CENTER,
    TYPE_POLICE_FORCE,
    TYPE_POLICE_OFFICE,
    TYPE_HELPERCIVILIAN,
};
```
In objdef.h, the method ObjectPool::newObject is responsible for instantiating all agent objects. We modified the function to support instantiation for HelperCivilian agents.

```
inline Object* ObjectPool::newObject(const TypeId& type) {
    switch(type) {
        // omitted for readability
        case TYPE_HELPERCIVILIAN:
            result = new HelperCivilian();
            break;
        // omitted for readability
    }
}
```

The traffic module, the viewer module, and the YabAPI module require the Java version of the agent definition.

The setup of HelperCivilian is very similar for the traffic module and the viewer. Their HelperCivilian definition is organized in HelperCivilian.java, TypeConstants.java, and PropertyConstants.java.

The class definition and the type ID of HelperCivilian are defined in HelperCivilian.java and TypeConstants.java, respectively. The property ID of the attribute skill is defined in PropertyConstants.java. The function WorldModel::newRescueObject, in WorldModel.java, is responsible for instantiating the agents.
```
private RescueObject newRescueObject(int type, int id) {
    RescueObject obj;
    switch (type) {
    case TYPE_WORLD: obj = new World(id); break;
    case TYPE_RIVER: obj = new River(id); break;
    case TYPE_RIVER_NODE: obj = new RiverNode(id); break;
    case TYPE_ROAD: obj = new Road(id); break;
    case TYPE_NODE: obj = new Node(id); break;
    case TYPE_BUILDING: obj = new Building(id); break;
    case TYPE_AMBULANCE_CENTER: obj = new AmbulanceCenter(id); break;
    case TYPE_FIRE_STATION: obj = new FireStation(id); break;
    case TYPE_POLICE_OFFICE: obj = new PoliceOffice(id); break;
    case TYPE_REFUGE: obj = new Refuge(id); break;
    case TYPE_CIVILIAN: obj = new Civilian(id); break;
    case TYPE_HELPERCIVILIAN: obj = new HelperCivilian(id); break;
    case TYPE_AMBULANCE_TEAM: obj = new AmbulanceTeam(id); break;
    case TYPE_FIRE_BRIGADE: obj = new FireBrigade(id); break;
    case TYPE_POLICE_FORCE: obj = new PoliceForce(id); break;
    case TYPE_CAR: obj = new Car(id); break;
    default: Util.myassert(false, "illegal object type" + type); throw new Error();
    }
    return obj;
}
```

For the YabAPI module, HelperCivilian's definition is organized in

- `HelperCivilianAgent.java`
- `AbstractHelperCivilianAgent.java`
- `HelperCivilian.java`
- `BaseHelperCivilian.java`
- `ProtocolConstants.java`

They are scattered in the following locations:

```bash
\program\yabapi\sample\src\sample\HelperCivilianAgent.java
\program\yabapi\yab\src\yab\agent\AbstractHelperCivilianAgent.java
\program\yabapi\yab\src\yab\agent\object\HelperCivilian.java
\program\yabapi\yab\src\yab\io\object\BaseHelperCivilian.java
\program\yabapi\yab\src\yab\io\ProtocolConstants.java
```

As mentioned in Chapter 2, the YabAPI consists of four packages. We must be sure that each package is setup for HelperCivilian.

The `ProtocolConstants.java` defines the property ID for the attribute `skills`. The `HelperCivilian.java`, `BaseHelperCivilian.java`, `AbstractHelperCivilianAgent.java` are
created for the following packages - yab.agent.object, yab.io, and yab.agent, respectively.

In HelperCivilianAgent.java, it defines the class HelperCivilianAgent, which is the base
class of the HelperCivilian model and defines with default behavior model.

HelperCivilianAgent is a derived class of AbstractHelperCivilianAgent; ideally, we
would like HelperCivilianAgent to be a member of the yab.agent package. Developers
should always create a new class based on class HelperCivilianAgent whenever they take
advantage of the HelperCivilian model.

```java
public final class Main implements ProtocolConstants {
    // omitted for readability
    private static final int CV = 0, FB = 1, FS = 2, AT = 3, AC = 4, PF = 5, PO = 6, HC = 7, NUM_TYPE = 8;
    // omitted for readability
    private static Agent connectAgent(int type) {
        switch (type) {
            default: throw new Error();
            case CV: return new CivilianAgent(m_address, m_port);
            case HC: return new HelperCivilianAgent(m_address, m_port);
            case FB: return new FireBrigadeAgent(m_address, m_port);
            case FS: return new FireStationAgent(m_address, m_port);
            case AT: return new AmbulanceTeamAgent(m_address, m_port);
            case AC: return new AmbulanceCenterAgent(m_address, m_port);
            case PF: return new PoliceForceAgent(m_address, m_port);
            case PO: return new PoliceOfficeAgent(m_address, m_port);
        }
    }
}
```

In "program\yabapi\sample\src\sample\main.java", the function connectAgent is
responsible for instantiating the YabAPI agents. The constant HC is required to be
defined for instantiating the HelperCivilian agents. It is likely that the developers create
a custom class based on HelperCivilianAgent. In order to instantiate their agents, they
need to modify connectAgent so that it can direct to the appropriate HelperCivilian agent
class.
3.5. Unleashing Capabilities

The kernel and sub-simulators administer the agent activities in the RCRSS. Their functions are to make sure that each agent is engaged in its own set of authorized activities. The sub-simulators define a function called commands to authorize and execute agent activities.

HelperCivilian is capable of sense, say, move, and hear, inherited from Humanoid. The extra skills of rescue and clear can only be obtained by changing the way the misc sub-simulator administers agent activities.

The misc simulator is programmed to administer the rescue, clear, move, load, unload, and stretch activities. In "misc\main.cxx" we modified the two functions called rescue and clear to grant HelperCivilian permissions to extra capabilities.

Referring Table 2.3, the kernel is programmed to direct rescue and clear requests to misc; load and unload requests to traffic simulator. The misc module only administers rescue and clear activities. The kernel controls where the messages go in the system.

In "misc\main.cxx", the function commands checks permissions for the activities the misc simulator administers. The definitions of the functions commands, rescue and clear, are provided in Table 3.3, 3.4, and 3.5, for better understanding on how the misc sub-simulator works and how we granted additional skills to HelperCivilian.
Table 3.3. Source Code of commands in the mise sub-simulator

```c
void commands(S32& time, Header type, Input& input, ObjectPool& pool, Config& config) {
    Id sender;
    while(sender = input.get(), sender != 0) {
        S32 size = input.get();
        Input::Cursor start = input.cursor();

        Object* o = pool.get(sender);
        MovingObject* agent = dynamic_cast<MovingObject*>(o);
        if(agent != 0) {
            switch(type) {
                case AK_LOAD:
                    load(time, agent, input, pool, config);
                    break;
                case AK_UNLOAD:
                    unload(time, agent, input, pool);
                    break;
                case AK_RESCUE:
                    rescue(time, agent, input, pool, config);
                    break;
                case AK_STRETCH:
                    stretch(time, agent, input, pool, config);
                    break;
                case AK_MOVE:
                    move(time, agent, input, pool, config);
                    break;
                case AK_CLEAR:
                    clear(time, agent, input, pool, config);
                    break;
            }
        }

        input.setCursor(start);
        input.skip(size);
    }
}
```
Table 3.4. Source Code of Function rescue

```c
void rescue(S32& time, MovingObject* agent, Input& input, ObjectPool& pool, Config& config) {
    AmbulanceTeam* am = dynamic_cast<AmbulanceTeam*>(agent);
    HelperCivilian* hc = dynamic_cast<HelperCivilian*>(agent);
    Humanoid* a = NULL;

    if (am != 0)
        a = am;
    else if (hc != 0)
        a = hc;

    if(a != 0 && a->buriedness() == 0) {
        Id id = input.get();
        Humanoid* target = dynamic_cast<Humanoid*>(pool.get(id));
        // omitted
    }
}
```

Table 3.5. Source Code of Function clear

```c
void clear(S32& time, MovingObject* agent, Input& input, ObjectPool& pool, Config& config) {
    PoliceForce* police = dynamic_cast<PoliceForce*>(agent);
    HelperCivilian* hc = dynamic_cast<HelperCivilian*>(agent);
    Humanoid* a = NULL;

    if (police != 0)
        a = police;
    else if (hc != 0)
        a = hc;

    if(a != 0 && a->buriedness() == 0) {
        Id id = input.get();
        Road* target = dynamic_cast<Road*>(pool.get(id));
        // omitted
    }
}
```

The fire sub-simulator is responsible for administering the extinguish activities among the agents. It authorized the fire-fighter agents to extinguish fires. HelperCivilian was not designed to extinguish fires such that we need not modify this module. In "firesimulator\main.cxx", the commands function, which checks permission for
extinguish activities, is provided in Table 3.6, to give better idea about the fire sub-simulator.

Table 3.6. Source Code of commands in the fire sub-simulator

```
void commands(S32& time, Header type, Input& input, RescueObjectPool& pool, Config& config)
{
    Id sender;
    while(sender = input.get(), sender != 0) {
        S32 size = input.get();
        Input::Cursor start = input.cursor();
        Object* o = pool.get(sender);
        FireBrigade* agent = dynamic_cast<FireBrigade*>(o);
        if(agent != 0) {
            switch(type) {
                case AK_EXTINGUISH:
                    extinguish(time, agent, input, pool, config);
                    break;
            }
        }
    }
}
```

3.6. Configurability

The HC was designed to support new configurations – capable of 1) neither rescue nor clear capabilities, 2) either rescue or clear capability, 3) both rescue and clear capabilities.

The flexibility is powered by the attribute skills, a 4-byte integer we created in the HelperCivilian definition, used to keep track of what skills an agent has. The attribute is used in bit-level. The bits in the attribute are read and written using bitwise computation. Counting from the right, the first bit is RESCUE bit, and the second bit is CLEAR bit. They are the only bits that are currently in use. A defined bit position stores a value indicating the availability of a particular skill to the agent. To determine an agent’s capabilities, we apply the attribute with CLEAR_MASK and RESCUE_MASK. If the
bitwise computation results in a non-zero value, the agent is capable of the skill being queried; otherwise, it is not.

In `BaseHelperCivilian.java`, class `BaseHelperCivilian` defines attribute `skills` as integer. We defined `skills` as integer across the system. Developers may implement this attribute using other type of data structure. However, they would have to apply the change uniformly to all modules. We provide two user-friendly methods `isRescueDoable` and `isClearDoable`. With the two functions available, the developers can avoid using bitwise computation in the development.

```java
public class BaseHelperCivilian extends BaseCivilian {
    public BaseHelperCivilian (int id) { super(id); }

    private int skills = 0;
    final static int RESCUE_MASK = 1;
    final static int CLEAR_MASK = 2;

    private void setRescueSkills(int skills) { this.skills = skills; }
    public boolean isRescueDoable() { return (skills & RESCUE_MASK) != 0; }
    public boolean isClearDoable() { return (skills & CLEAR_MASK) != 0; }

    public void setProperty(int type, int[] value) {
        switch (type) {
            default: super.setProperty(type, value); break;
            case PROPERTY_RESCUE_SKILLS_MATRIX: setRescueSkills(value[0]); break;
        }
    }
}
```

3.7. Synchronization

The RCRSS is a distributed system. Each agent attribute presents in both the remote agent and the server side agent. When an attribute is changed on one side, the other side must mirror the new value to maintain synchronization. The synchronization facility is built in the definition on both sides.
In the YabAPI module, the framework synchronizes the state of a remote agent through the interface `setProperty`, a function defined in each agent in the `yab.io` package. The interface is called by the framework whenever the kernel requests an update to the remote agent. The interface `setProperty` for `HelperCivilian` is defined in the class `BaseHelperCivilian` in `BaseHelperCivilian.java`.

The server-side agent implements two methods – `output` and `input` – to implement agent-wise synchronization. The method `input` is the interface used by the kernel when the state of the server-side agent needs to be updated. The kernel updates the agent when a sub-simulator determines a new state for the agent. The method `output` is an interface used by the kernel when the state of the agent located in other modules, such as YabAPI module, needs to be updated. The interface `output` is called usually after the kernel calls the interface `input` as part of the routine to synchronize the agent across the system. When it is called, the interface `output` on the server-side sends a signal to all remote modules regarding the change to the state of the agent. On receiving this signal at a remote side module, the YabAPI module, for example, responds to this event and calls the interface `setProperty` of the corresponding remote agent. For class `HelperCivilian`, the interface `output` and `input` are defined in class `HelperCivilian` in `HelperCivilian.inl`.

In `HelperCivilian.inl`, another attribute called `m_rescueSkillsMatrixUpdate` is defined, which is used to record the time when the attribute `skill` was last updated. Any request to update the agent must be compared against the timestamp stored in `m_rescueSkillsMatrixUpdated`; obsolete and duplicate requests will be rejected.

The RCRSS utilizes these synchronization interfaces `output`, `input`, and `setProperty` to synchronize the state of a RCR agent in the distributed environment. For
your information, some modules, such as Viewer and Traffic sub-simulator, may name the interface `setProperty` as `input` instead; they are the same. In the next section, we will discuss how this synchronization facility is extended to support agent learning.

```cpp
class HelperCivilian : public Civilian {
public:
    // omitted for readability
private:
    S32 m_skills;
    S32 m_rescueSkillsMatrixUpdated;
    // omitted for readability

public:
    // omitted for readability
virtual int output(Output& buffer, PropertyId property, S32 time) const {
    int result = 0;
    switch(property) {
        // omitted for readability
        case PROPERTY::RESCUE_SKILLS_MATRIX:
            if(m_rescueSkillsMatrixUpdated >= time) {
                buffer.put(property);
                buffer.put(m_skills);
                result = 1;
            }
            break;
    }
    return result;
}

virtual void input(S32 time, ObjectPool& pool, Input& buffer, PropertyId property) {
    switch(property) {
        default:
            Civilian::input(time, pool, buffer, property);
            break;
        case PROPERTY::RESCUE_SKILLS_MATRIX:
            S32 value = buffer.get();
            setSkills(value, time);
            break;
    }
    // omitted for readability
};
```
3.8. Learning Infrastructure

In the RCRSS, the agents either obtain new information by learning from the neighboring agents or from the limited local world model. Agent learning in the original architecture is limited to exchange in knowledge and must be conducted in a conversation among agents. Using the extended infrastructure, it is possible that the agents can learn knowledge as well as new skills among themselves through conversations. It is a powerful feature that the agents can gain extra skills from other agents. With this feature, it is possible to simulate the scenarios, such as agents conduct first-aid training or CPR training in disasters, which is something we could not do before.

The learning infrastructure is based on the synchronization infrastructure developed for the HelperCivilian model. In "kernel\basic.hxx", we defined a new message `ID AK_SKILLS_LEARNED`, which is used in a request, through which the HC agent requests the kernel to approve the desired configuration. The requester must include the desired configuration in the request, and may include supporting information, such as relevant trainer's endorsement. The kernel decides whether to approve or reject the requests, based on the dialogues between the trainer and the trainee agents, the applicability of the desired configuration, and/or any evidence supporting the qualification. The kernel is the only authority which sets the state of the attribute skill for the HC agents. The approval process and the agent dialogues implementing the skill-transfer scenarios are left open for future works.

The method `System::processMessage` in the "kernel\System.cxx" processes the AK_SKILLS_LEARNED messages initiated by the remote agents. By default, the kernel approves any configuration and calls the function `m_objectPool.restructure`, which writes
the new configuration to the server-side agent and synchronizes the system. The commented lines in the function are left for customization. The method System::processMessage is quite an important function, with which the kernel communicates with the remote agents, sub-simulators, GIS, and the viewer. It is the function that the kernel uses to process all incoming messages.

```cpp
bool System::processMessage(const LongUDPSocket& from) {
    // omitted for readability
    case AK_SKILLS_LEARNED:
    // omitted for readability
    HelperCivilian *o = dynamic_cast<HelperCivilian *>(m_objectPool.get(sender»;
    if (o != NULL) {
        // if the kernel approves the new skills to the requester agent
        // this part is left for customization
        m_objectPool.restructure(m_time, m_inputBuffer);
    }
    // omitted for readability
}
```

The following is a sample dialogue between agent A (trainer) and agent C (trainee) in a learning scenario. The dialogue is labeled 1 to 8 in the order of the events, along with the agent identity and the oral message.

1. A: "Want to learn a skill?"
2. C: "Yes"
3. A: "I'm teaching <X>"
4. C: "I'm learning <X>"
5. A: "I'm teaching <Y>"
6. C: "I'm learning <Y>"
7. C: "Skill <X> learned"
8. C: "Skill <Y> learned"

When agent C is expressing "Skill <X> learned" or "Skill <Y> learned", the agent sends a request to the kernel for approving the desired skill. Agent C may attach the endorsement issued by the trainer along with the request. The kernel may approve or reject the request after evaluated the endorsement.
3.9. Default Behavior

We provided the default behavior for the HelperCivilian agents in an agent class called HelperCivilianAgent which is defined in HelperCivilianAgent.java. The default behavior is developed for the testing purpose to validate the implementation of the HelperCivilian agent model. We will discuss the overview of the default behavioral in the following.

In the HelperCivilianAgent model, the agents can be trained or non-trained. They are programmed to identify threats, to rescue, to retreat, to search for victims, to evacuate in disasters and to look for shelters. Applying rescue and clear capabilities on a single agent is something we could not do before with the original framework.

The specialized agent HC model is similar to real civilians. The agents evacuate in disasters and look for shelters. Some of them are trained with first-aid knowledge and some are not. The trained agents use their capabilities to rescue the injured, remove roadblocks, and search for victims.

By default, the HC agents are programmed to evacuate and look for shelters known as "refuges". They are programmed to say "What_a_relief!" when they arrive at the shelters. They are programmed to say "help_me" to signal for help and repeat that every 1 to 5 simulation cycles; they stop calling for help when they are free. When their health point drops below 2000 of 10000 point maximum, they are programmed to retreat from current activities and leave for shelters. The trained agents are programmed to clear road blocks, rescue injured agents, and search for the agents that are calling for help, given the condition that the agents maintain a health point above 2000. On the other hand, the non-trained HC agents are programmed to evacuate and find shelters. Once the
HC agents arrive at the shelter; they are programmed to rest until the simulation ends. The HC and the Civilian agents are programmed to say the same expressions in all situations. Potentially, the HC agents could respond to Civilian agents, and the former could possibly attract attention from the agents, which are programmed to respond to Civilian.

As mentioned earlier in Chapter 2, each YabAPI base agent uses a method called act, which is invoked by the framework every simulation cycle. The method act implements the behavior of the agents. The following is the definition of the method act of the specialized HC agent, HelperCivilianAgent.
public class HelperCivilianAgent extends AbstractHelperCivilianAgent {
    // omitted for readability
    protected void act() throws ActionCommandException {
        if (isDead) throw new ActionCommandException();

        if (mlpos() instanceof Refuge) {
            if (iSafe == 0) {
                iSafe++;
                say("What_a_relief!"); // mimic behavior of R_Civilian
                rest();
            }
            throw new ActionCommandException();
        }

        if (self().hp() == 0) {
            s_numDead++;
            isDead = true;
            throw new ActionCommandException();
        }

        // Full Health is 10000;
        // If HC health is less than 20%, agent should go to refuge immediately
        bWeak = self().hp() < 2000;
        // If Weak, drop victim list and retreat
        if (bWeak) lstAgentsNeedHelp.clear();

        if (self().buriedness() > 0) {
            if (say_freq++ >= say_wait) {
                say("help_me"); // mimic behavior of R_Civilian
                say_freq = 0;
                say_wait = (int) (Math.random() * 5);
            }
            throw new ActionCommandException();
        }

        List hmons = POSITION_PRP.eq(mlpos() and(IS_ALIVE_CND).extract(world.humanoids);
        hmons.remove(self());
        List refuges = IS_NOT_BURNING_CND.extract(world.refuges);
        if (!bWeak) rescue(hmons); // should rescue living humans only
        clearHere(); // HC clears road block if sees any; won't actively look for blocks
        cleanVisitedLocation(); // clear visited locations in the search history

        if (!bWeak) {
            if (!lstAgentsNeedHelp.isEmpty()) {
                moveToHumans(lstAgentsNeedHelp); // approach agents who need help
                searchTheInjured();
            }
        }

        move(refuges); // go to a safe location (refuge)
    }
}
// omitted for readability
3.10. GIS

The GIS module is responsible for reading the setup files (including the gisini.txt) and constructing the world model based on them. The GIS module calls the function `makeWorld` in "gismain.cxx" to build the world model. We expanded the data structure used by `makeWorld` to support HelperCivilian.

```c
void makeWorld(Config& config, ObjectPool& pool, int number, const char* mapdir, const char* gisini, S32 tank_quantity_maximum)
```

This function creates roads, buildings, and nodes, and identifies the connections among them; then it creates the motionless agents and the human agents, identifies whether the agent is on a road, in a building or on a node for each of them, and establishes the connections accordingly.

After all the connections are figured out and the objects are instantiated, the GIS module waits for the kernel to connect to it. The kernel retrieves a copy of the finished world model when it connects to the GIS, and uses the information to initialize the modules that connect to it.

3.11. JGISEdit

JGISEdit is the editor we used to develop the world model and generate the setup files for the simulation, including the gisini.txt [La02]. The gisini.txt defines the locations and configurable properties of the agents in the world model. We modified the editor to support HelperCivilian. We will reveal more about JGISEdit in Chapter 4.
3.12. Summary

This new infrastructure is our initial effort to improve the features of the RCRSS. The extended RCRSS allows new behaviors to be simulated and supports agent learning via skills transfer. We look forward to more iterative improvements and expansion in future works. The next chapter will discuss the simulations we developed to demonstrate the RCR agents using the new features.
CHAPTER 4
SIMULATION

4.1. Introduction

In this chapter, we walk through on how we constructed, executed, and verified the simulations based on the extended architecture. We break down the simulation process into Planning and Execution, and will discuss on each of them. Through developing the scenarios with the extended framework, we were able to verify the implementation. We verified the implementation by analyzing the log messages and observing the simulation activities.

We enhanced the viewer and YabAPI modules to print real-time scores and agent activities to their consoles, respectively. This information is useful for analyzing agent activities and simulation performance, and can be looked up on the fly. We automated the simulation execution in a single user command; automation allows us to go back to check results and run analysis. Toward the end of this chapter, we will discuss the score evaluation method used in the RCRSS and how we break down the scores into greater detail for our evaluation.

4.2. Process

The process of developing a RCRSS simulation can be broken down into Planning and Execution.
4.2.1. Planning

In Planning, we prepare a setting for the simulation. A simulation setting consists of the layout (buildings, roads, and connections); the placements of fire spots, police offices, fire-stations, shelters (hospitals), call center agents, and human agents (civilians, fire-fighters, paramedics, police officers, and helper-civilian agents); and initial configurations for each agent. We developed our setting using a simulation planner called JGISEdit, which is a graphical front-end application with which agent developer creates and edits the elements of a virtual city in a graphical editor environment [La02]. The planner leverages the complicated scripts generation, so that the developers can have a better focus on the design.

For demonstration, it is desirable and convincing to show a moderately complex layout with a reasonable number of agent activities. We chose the Kobe map, which comes with the RCRSS package version 0.44, as it has good details and complexity. The setup files of Kobe are located in “/maps/Kobe”. They describe the buildings, roads, connection points, seismic intensity and ground acceleration distributions, which are organized in building.bin, road.bin, node.bin, galpolydata.dat and shindopolydata.dat, respectively, as mentioned earlier [Mor1]. The configuration of Kobe used in the demonstration is shown in Chapter 5.

In Figure 4.1, the simulation setting is shown in the JGISEdit environment. The Civilian and HC agents appear as human icons; the HC agents are distinguished by the “HC” symbols located on the right chin; the rescuer agents appear in police car, fire-truck, and ambulance icons; the center agents and shelters appear with their first initials in the icons; the fire spots appears in the fire icons.
In Figure 4.2, the Property Edit panel is shown for the selected HC agent. In the panel, the attribute *skills* was set to 3, meaning that the agent is capable of both *rescue* and *clear* capabilities. The attribute *skills* can be set in any one of the configurations listed in Table 4.1.

**Table 4.1. Configuration of attribute *skills***

<table>
<thead>
<tr>
<th>Skills</th>
<th>Rescue</th>
<th>Clear</th>
<th>Equivalent To</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>No</td>
<td>No</td>
<td>Civilian</td>
</tr>
<tr>
<td>1</td>
<td>Yes</td>
<td>No</td>
<td>Medical Tech</td>
</tr>
<tr>
<td>2</td>
<td>No</td>
<td>Yes</td>
<td>Police</td>
</tr>
<tr>
<td>3</td>
<td>Yes</td>
<td>Yes</td>
<td>Medical Tech/Police</td>
</tr>
</tbody>
</table>
When the configuration is complete, we would save to finalize the setting. The save command creates the setup files in a designated directory. The configurations about the agents and the virtual city are kept in separate files. The agent configuration is written to \textit{gisini.txt} in text format; each agent is defined using a preset syntax.

With the addition of Helper Civilian, a new syntax was added in \textit{gisini.txt} for declaring HC agents. The new syntax takes 9 parameters instead of 8, and is based on the syntax used to declare traditional agents. The extra parameter is required to support the attribute \textit{skills} in a HC-type declaration. Therefore, the value shown in the Property
Edit panel in Figure 4.2 should appear as the extra parameter in the agent declaration.

With the use of JGISEdit, we can go back to modify the setting anytime. To reopen a setting, we would use the Open command and point it to *building.bin*; the editor will be able to load the complete setting.

The simulation parameters such as simulation cycles, cycle resolution, and rules are defined in *config.txt*, located in the “\boot” directory. Our simulations are based on the original *config.txt*. Once a simulation setting is ready, we move forward to execute the simulation [Morl].
4.2.2. Execution

In Figure 4.3, the execution of the RCRSS Simulation is depicted. The simulation starts immediately once all the modules are initialized.

In Execution, we initiate the simulation by calling the shell script `all.sh`, located in the "/boot" directory. Invoking `all.sh` in turn invokes other supporting scripts; it is responsible for initiating the modules in the proper order. The execution created the windows in the user environment as shown in Figure 4.3.

The GIS module reads the setup files and builds the world model. The remaining modules start initiating and connect to the system. The simulation starts immediately once all the modules are initialized.
In Figure 4.4, the viewer displays the simulation activities that based on the new model. We customized the viewer to show specific captions and colors in the way we desire. The color representation in the viewer is listed in Table 4.2. The human agents represent in solid circles within which the first letter of the associated type is printed for identification. The crosses represent the blocked roads. A building is painted in yellowish-green when it is on fire; the level of yellowish-green appears on the building reflects its level of fire damage. When a human agent is flashing and appears in bold, the agent is performing either a *rescue* or *clear* action.
Table 4.2. Color Representation in Viewer

<table>
<thead>
<tr>
<th>Entity</th>
<th>Color</th>
</tr>
</thead>
<tbody>
<tr>
<td>Civilian / Helper Civilian</td>
<td>Pink</td>
</tr>
<tr>
<td>Police Officer</td>
<td>Blue</td>
</tr>
<tr>
<td>Ambulance (Paramedics)</td>
<td>White</td>
</tr>
<tr>
<td>Fire-Fighter</td>
<td>Cyan</td>
</tr>
<tr>
<td>Dead agent</td>
<td>Black</td>
</tr>
<tr>
<td>Building</td>
<td>Grey</td>
</tr>
<tr>
<td>Building on Fire</td>
<td>Yellowish-green</td>
</tr>
<tr>
<td>Shelter (Hospital)</td>
<td>Blue</td>
</tr>
<tr>
<td>Center</td>
<td>Lawn green</td>
</tr>
</tbody>
</table>

The viewer is an important module as it is responsible for reporting the scores and animating the scenarios. The viewer calculates the scores based on the Evaluation Rule 2003-05 defined in the RCR document [RCRSL]. We will discuss the method that calculates the scores in Section 4.4. The system logs all events in rescue.log and action.log. These logs are used for replaying the simulation using the offline mode of the viewer [Mor1].

The consoles of the Viewer and YabAPI modules, respectively, printed the real-time scores and agent activities, as shown in Figure 4.5 and 4.6, which were later written to viewer.txt and yabapi.txt, respectively. The real-time elements, which are text base, can be lookup immediately and are useful for evaluation, verification, and debug purposes.

To prevent any loss of data by accident, the log files must be archived prior to another simulation begins. In our automated environment, the following files were scheduled to be archived after each run - rescue.log, action.log, gisini.txt, viewer.txt and yabapi.txt.
Civilian, 4.3. road segments and 730 buildings. We simulated the scenarios by varying the

4.3. Method

We created a virtual environment setting which made up of 100 HelperCivilian, 1
Civilian, 10 Police, 10 Fire-Fighter, and 5 Paramedics agents in a virtual city with 820
road segments and 730 buildings. We simulated the scenarios by varying the
distributions between civilians (skills = 0) and trained civilians, which can be in any of the following configurations (skills = Rescue (R), Clear (C), or Rescue & Clear combined (RC)) among the overall HC population while keeping everything else constant. The distributions for trained HC agents started from 0 and increased to 100 percent in the step of 20; in each simulation only one type of configuration is applied uniformly to the trained HC agents and was repeated for the configuration Rescue, Clear, and combination of Rescue & Clear. The agents were randomly selected to become the trained agents.

4.4. Score

Simulation performance in the RCRSS is evaluated by the scores based on the Evaluation Rule 2003-05 as shown in the following [RCRSL]:

<table>
<thead>
<tr>
<th>Evaluation Rule 2003-05: V = (N + HP) * SQRT (NB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Where,</td>
</tr>
<tr>
<td>N = number of surviving agents</td>
</tr>
<tr>
<td>HP = HPfin / HPini</td>
</tr>
<tr>
<td>NB = NBfin / NBini</td>
</tr>
<tr>
<td>HPini = total agent health point at time = 0</td>
</tr>
<tr>
<td>HPfin = total agent health point at time = T</td>
</tr>
<tr>
<td>NBini = total area of non-burned buildings at time = 0</td>
</tr>
<tr>
<td>NBfin = total area of non-burned buildings at time = T</td>
</tr>
</tbody>
</table>

The score (V) depends on the number of living agents (N), the total agent health point ratio (HP), and the non-burned buildings ratio (NB). The component HP is weighed negligibly with respect to N and NB. To researchers who are interested in collecting results on the health condition or evacuation efficiency of overall population,
the score $V$ is not necessarily a good indicator to them, as $V$ is weighed significantly by
the effectiveness of the fire-fighter agents. To promote RCRSS as a common tool to
study multi-agent problems, we customized the viewer console to print $V$, $N$, $HP$, and $NB$
per simulation cycle, in order to provide more information to researchers with different
interests and perspectives.
CHAPTER 5
RESULTS

5.1. Introduction
This chapter presents the simulation results and analysis.

5.2. Results
Each of our simulations takes 300 simulation cycles to complete. The simulations used are based on the virtual environment setting shown in Table 5.1, and executed on the systems shown in Table 5.2.

Table 5.1. Configuration of the Simulation Setting

<table>
<thead>
<tr>
<th>Element Type</th>
<th>Element Count (N = Node, B = Building, R = Road)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Road segments</td>
<td>820</td>
</tr>
<tr>
<td>Node</td>
<td>765</td>
</tr>
<tr>
<td>Building</td>
<td>730</td>
</tr>
<tr>
<td>AmbulanceCenter</td>
<td>1</td>
</tr>
<tr>
<td>PoliceOffice</td>
<td>1</td>
</tr>
<tr>
<td>FireStation</td>
<td>1</td>
</tr>
<tr>
<td>Refuge</td>
<td>7</td>
</tr>
<tr>
<td>Civilian</td>
<td>1 (N:1,B:0,R:0)</td>
</tr>
<tr>
<td>HelperCivilian</td>
<td>100 (N:2,B:86,R:12)</td>
</tr>
<tr>
<td>AmbulanceTeam</td>
<td>5 (N:5,B:0,R:0)</td>
</tr>
<tr>
<td>FireBrigade</td>
<td>10 (N:10,B:0,R:0)</td>
</tr>
<tr>
<td>PoliceForce</td>
<td>10 (N:9,B:0,R:1)</td>
</tr>
<tr>
<td>FirePoint</td>
<td>4</td>
</tr>
</tbody>
</table>
Table 5.2. Systems Configuration

<table>
<thead>
<tr>
<th>Processor</th>
<th>Intel Celeron M 1.3Ghz</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory</td>
<td>512MB RAM</td>
</tr>
<tr>
<td>Operating Systems</td>
<td>Redhat Linux 9.0</td>
</tr>
<tr>
<td>RCRSS</td>
<td>Version 0.44</td>
</tr>
</tbody>
</table>

We created the figures based on the simulation results with respect to the official score (V), the number of surviving agents (N), the agent health point ratio (HP), the non-burned buildings ratio (NB), execution times, the mean agent health point, the standard deviation of agent health point, the number of rescue commands issued, and the number of clear commands issued.

Figure 5.1, 5.2, and 5.3 show experimental results inline with expectations. The agent population configured with Rescue and Clear (RC) combined capabilities outperforms the ones configured with Rescue (R) or Clear (C) capability alone. They all outperforms pure civilian populations (skills = 0) with no extra capabilities.

Figure 5.1. Official Score (V) versus Helper Civilian Population
In Figure 5.1, as the population is 100 percent trained, the population with Rescue and Clear combined capabilities scored 70; with Rescue capability or Clear capability alone, the population scored 45; the pure civilian population scored nearly 30. The Rescue, Clear, and Rescue & Clear combined configurations all show significant improvement in official score $V$ over pure civilian population; the pure RC population remarkably demonstrates over 130 percent gain in performance.

![Figure 5.2. Number of Surviving Agents (N) versus Helper Civilian Population](image)

In Figure 5.2, as the population is 100 percent trained, the number of survivors increased to roughly 110 in the Rescue & Clear combined configuration, 70 each in Rescue and Clear configurations, compared to 50 in the pure civilian population. They all show significant improvement in the number of survivors (N) over pure civilian population; the pure RC population remarkably shows 120 percent gain.
In Figure 5.3, the agent health point ratio (HP) shows consistent gain with configuration Rescue, Clear, and Rescue & Clear combined. Although HP is not a significant factor in the Evaluation Rule, it is important to include HP in the results as it gives significant insight when the score (V) and the number of living agents (N) cannot provide enough information on performance analysis.

As the population is 100 percent trained, the health point ratio is roughly 65 percent in the Rescue & Clear combined configuration, 40 percent each in Rescue and Clear configuration, compared to 30 percent in the pure civilian population. They all show significant improvement in agent health point ratio (HP) over pure civilian population; the pure RC population remarkably shows 115 percent gain.
In Figure 5.4, the total non-burned building ratio (NB) shows that the non-destructed area remains at 40 percent firmly as the trained population increased to 100 percent. The non-destructed area remains at 26 percent and 40 percent; the variation is largely due to the randomness in the nature of the sub-simulator. For most cases, the non-destructed area remains firmly at 40 percent. It shows that with the presence of the extra skills, the fire destruction is kept lower in the Clear and Rescue & Clear combined configurations. This is because the trained agents help remove road blocks to free the fire-fighters to reach more burning areas. In general, the NB ratio improves by 60 percent when the population is 100 percent trained.
The execution times for each simulation were measured, as shown in Figure 5.5. By observation, the execution time varies from 600 to 1000 seconds, which depends on the complexity of the simulation setting, the number of agents, the volume of exchanged messages, and the available hardware resources. The result shows that RC configured population caused the simulation to run longer. The extended period is resulted from additional complexity in agent behavior. The execution times in the Rescue and Clear configurations do not show obvious gain over the pure civilian configuration.
Figure 5.6. Mean Agent Health Point Ratio versus Helper Civilian Population

Figure 5.7. Standard Deviation of Agent Health Point Ratio versus Helper Civilian Population
In Figure 5.6 and 5.7, we measure the health point ratio per agent in terms of mean and standard deviation.

In Figure 5.6, the population with the Rescue and the Clear configurations saturated at 0.4 in the health point per agent; increase in the trained population does not create further gain. On the other hand, the combined skills in the RC configuration produce a compound effect in the increase in agent health point ratio. Higher health condition contributes to higher survival rate. The pure RC configured population results in 65 percent in agent health point ratio, which is over 115 percent gain, and that is equivalent to the HP in Figure 5.3. In fact Figure 5.3 and 5.6 are measuring the same indicator.

In Figure 5.7, the standard deviation measures the variance in the health point ratio per agent at 0.44 in Rescue and Clear configurations, and the variance in the RC configuration is shown to be converging sharply as the trained population increases. It suggested that variance can be further improved in a larger population with the RC configuration but not with the Rescue and Clear configurations. Since Figure 5.3 and 5.6 are measuring the same HP, Figure 5.7 is measuring the standard deviation of HP.
In Figure 5.8 and 5.9, we measure the number of Rescue and Clear commands issued in a simulation. We want to quantify the rescue effort in terms of the number of road blocks being cleared and the number of agents being rescued in a simulation. These two factors provide another perspective on analyzing simulation performance.

Figure 5.8 shows that the Clear configured population results with the least in the number of Rescue commands issued followed by the Rescue configured population which records the next highest, and the RC configured population which records the highest. The Clear configured population and the pure civilian population did not result in additional Rescue commands issued.
Figure 5.9 shows that the Rescue configured population results with the least in the number of 'Clear' commands issued, followed by the Clear configured population which records the next highest number, and the RC configured population which records the highest. The Rescue configured population and the pure civilian population did not result in additional 'Clear' commands issued.

5.3. Analysis

Inline with our expectation, the RC configured population results in the highest number of survivors (N), the least number of injuries (HP), and the highest in the official score (V). The Rescue, Clear, and RC configured populations result in the higher in almost every aspect in the RCRSS evaluation (V) over the pure civilian population in the our environment. As shown, the performance gain is significant and impact is large on simulation results.
CHAPTER 6
CONCLUSION

In this chapter we will summarize the thesis and discuss the potential future work in this research effort.

This paper presented the enhanced agent model with the aim of improving the flexibility for creating mixed agent behaviors in a large rescue simulation system. This framework is based on the idea of a generalized agent model called Helper Civilian which is specialized into agent models with different behaviors that can be applied in the rescue simulation system. With this new model, we improved a popular rescue simulation system called RCRSS to become more powerful.

We first started with a brief coverage on RCR history, followed by a detailed description of the RCRSS architecture to provide background information. Then we discussed the details about the HC model that based on the YabAPI agents and the RCRSS, the readiness of the new architecture in supporting agent learning, and the difference between the new and the original architecture. In the demonstration, the agents based on the HelperCivilianAgent class were able to perform activities such as rescue and clear capabilities that they could not do before with the original RCRSS. We verified the implementation of the HC model experimentally and showed the impact of the HC agents to the simulations through the demonstration.
We also discussed other issues such as the evaluation rule that determines the simulation performance, and the methods we used to collect our simulation results.

The extended architecture allows new behaviors to be simulated and supports agent learning through skill training. We look forward to future improvements and expansion through future work, including implementation of different agent dialogues, agent collaboration scenarios, and effective tools to define new configurations to the HC agents in skill training events. Currently we enabled partial capabilities to HelperCivilianAgent agents; we look forward to enabling all available capabilities to HelperCivilianAgent in the future. The Helper Civilian model does not yet support multiple levels for each agent skill; we leave that for future improvement.

We aimed to simulate situations like civilians helping with rescues and expected a performance enhancement as a result with HC agents having partial or full capabilities over the population without any extra capabilities. The results show that the scenarios such as “civilians helping in rescue” were successfully simulated, and the RC configured population outperformed the pure civilian population by 130 percent in the overall score (V), 120 percent in the number of survivor agents (N), and 115 percent in the overall agent health point. The enhanced agent behavior shows significant impact in rescue simulations. Our
results demonstrated increased flexibility and power in RCRSS simulations. With our generalized agent model, we can simulate scenarios that were not possible before and achieve a higher realism.
References


