2007-12

Little by little does the trick design and construction of a discrete event agent-based simulation framework

Matsopoulos, Alexandros

Monterey, California. Naval Postgraduate School

NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

LITTLE BY LITTLE DOES THE TRICK:
DESIGN AND CONSTRUCTION OF A DISCRETE EVENT AGENT-BASED SIMULATION FRAMEWORK

by

Alexandros Matsopoulos

December 2007

Thesis Advisor:	Paul J. Sánchez
Second Reader:	Arnold H. Buss

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.			
1. AGENCY USE ONLY ***(Leave blank)***	**2. REPORT DATE** December 2007	**3. REPORT TYPE AND DATES COVERED** Master's Thesis	
4. TITLE AND SUBTITLE Little by Little Does the Trick: Design and Construction of a Discrete Event Agent-Based Simulation Framework			**5. FUNDING NUMBERS**
6. AUTHOR(S) Alexandros Matsopoulos			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000			**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A			**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.			**12b. DISTRIBUTION CODE**

13. ABSTRACT (maximum 200 words)

Simulation is one of the most widely used techniques in operations research. In the military context, agent-based simulations have been extensively used by defense agencies worldwide. Despite the numerous disadvantages and limitations associated with time-stepping, most of the combat-oriented agent-based simulation models are time-step implementations. The Discrete Event Scheduling (DES) paradigm, on the other hand, is free of these disadvantages and limitations.

The scope of this thesis is to design and implement a library of reusable software components that will facilitate building combat-oriented agent-based simulation models by extending the Simkit DES toolkit. We describe our design of what an agent-based DES implementation framework should look like. We show that the extensive use of Java interfaces allows the user to implement different models and scenarios without being constrained by pre-built components. We also enhance Simkit's existing Sensing model by introducing a Situational Awareness model and a Behavioral model. Finally, we build a small agent-based model using the component architecture to demonstrate the library's functionality.

14. SUBJECT TERMS Agent Based Model, Agent Based Simulation, Discrete Event Simulation, Design of Experiments, MANA, Pythagoras, Simkit, Java			**15. NUMBER OF PAGES** 77
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UU

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited.

LITTLE BY LITTLE DOES THE TRICK: DESIGN AND CONSTRUCTION OF A DISCRETE EVENT AGENT-BASED SIMULATION FRAMEWORK

Alexandros Matsopoulos
Captain, Hellenic Air Force
B.S., Hellenic Air Force Academy, 1996

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN OPERATIONS RESEARCH

from the

NAVAL POSTGRADUATE SCHOOL
December 2007

Author: Alexandros Matsopoulos

Approved by: Paul J. Sánchez
Thesis Advisor

Arnold H. Buss
Second Reader

James N. Eagle
Chairman, Department of Operations Research

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Simulation is one of the most widely used techniques in operations research. In the military context, agent-based simulations have been extensively used by defense agencies worldwide. Despite the numerous disadvantages and limitations associated with time-stepping, most of the combat-oriented agent-based simulation models are time-step implementations. The Discrete Event Scheduling (DES) paradigm, on the other hand, is free of these disadvantages and limitations.

The scope of this thesis is to design and implement a library of reusable software components that will facilitate building combat-oriented agent-based simulation models by extending the Simkit DES toolkit. We describe our design of what an agent-based DES implementation framework should look like. We show that the extensive use of Java interfaces allows the user to implement different models and scenarios without being constrained by pre-built components. We also enhance Simkit's existing Sensing model by introducing a Situational Awareness model and a Behavioral model. Finally, we build a small agent-based model using the component architecture to demonstrate the library's functionality.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

EXECUTIVE SUMMARY

Agent based simulations (ABSs), such as those that are used to represent combat, have traditionally been modeled using a time-step approach. The time-step approach, however convenient it may seem in the modeling of aspects like movement and sensing, is associated with many problems and limitations. Discrete Event Simulation (DES) is free of the disadvantages of the time-step approach. Modelers, however, have overlooked the potential to implement the DES paradigm in the creation of ABSs. Contrary to the popular belief that DES cannot easily model movement and sensing, work by Buss and Sánchez [2005] has demonstrated that such a representation is not only feasible, but also desirable from several standpoints.

Simkit is a DES software package, written in the Java programming language. To write a DES in Simkit, however, one need only master the Java basics. Many NPS students who use Simkit for their theses, including the author, are not experienced programmers. We believe that it is a good practice for the analyst to be knowledgeable about the inner mechanisms and assumptions of the tool upon which his or her research is based. A simulation model should not be an opaque box to the analyst. We think this is a strong argument in favor of using a tool such as Simkit rather than one of the plethora of off-the-shelf simulation packages available.

The most important element of a combat-oriented ABS is the agent itself. Movement and sensing are among the agent's most fundamental modeling aspects. Simkit provides the tools for designing entities that move and sense. In this thesis, we have complemented these aspects of Simkit by creating a versatile and extensible framework, upon which new agent behaviors and capabilities could be built. We have also built a small-scale model, which we used to conduct a simple experiment to demonstrate the functionality of our components.

Our simple search model demonstrates that DES can be used to create models that include thousands of agents. Furthermore, the use of DES yields much shorter run times. A typical single simulation run, including approximately 1000 agents packed in a

relatively small area, moving randomly, detecting and classifying each other, took less than one minute to complete on a 2.0 GHz, 2 GB RAM MacBook. This is approximately 32 times faster than the same scenario implemented in MANA.

Our model is a first attempt to use Java programming and existing Simkit components to create Agent-Based Simulations within the Discrete Event paradigm. Despite the numerous off-the-shelf simulation packages that are available, we strongly advocate the creation of a library consisting of ABS components, developed and maintained by users. This thesis prototypes an architectural design which is generalizable, reusable, and extensible. We have created an initial set of model elements that demonstrate the approach we are advocating, and anticipate future growth and enrichment of these components. Although there is a long distance to be covered before our agents can be a part of large-scale, combat-oriented simulations, incremental improvements should be able to eventually bridge this distance.

ACKNOWLEDGMENTS

Dedicated to my wonderful wife, Effrosyni, to my sons, Victor and Fotios, and to the Holy Theotokos and Virgin Mary: “Υπεραγία Θεοτόκε σώσον ημάς.” (“Most Holy Theotokos, save us.”)

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

Untutored courage is useless against educated bullets.

- George S. Patton

A. BACKGROUND

Simulation is one of the most widely used techniques in operations research and management science and, by all indications, its popularity has been increasing over the years [Law & Kelton, 1982]. Its intuitive appeal arises from its ability to mimic what happens in a real system or what might happen in a hypothetical situation, in a simple and cost-effective way. In the military context, multi-agent simulations have been extensively used by defense agencies worldwide to analyze both battlefield scenarios and non-combat operations, such as peace-keeping or logistics support.

According to Thomas Aquinas, an agent is an entity capable of election or choice [Rocha, 1999]. We define an "agent" as a virtual (simulated) entity that can perceive its environment (in a partial way), act autonomously, use its skills to pursue its goals and tendencies, and communicate with other entities. The agent will usually be defined within a multi-agent (or agent-based) system that contains an abstract environment, agents acting within that environment, relations between all the agents, a set of operations that the agents can perform, and the changes in the environment over time and due to these actions [Ferber, 1999].

Multi-agent simulation systems – referred to as agent-based simulations or ABSs hereafter – have many advantages when it comes to modeling a combat environment as opposed to traditional simulating techniques. They furnish a very natural and intuitive way of modeling combat; they provide the framework for the creation of autonomous, self-governing entities, thus decentralizing the decision making process. They provide the framework for the studying of interactions between the simulated entities and they allow for the building of high complexity models using relatively simple components.

B. TIME ADVANCE MECHANISMS

Simulation models in general, and agent-based models in particular, can be categorized according to their time-advance mechanism. Historically, two principal approaches have been widely used for advancing simulated time: *next-event time advance* and *fixed-increment time advance*. In the next-event time advance approach, which is used widely for Discrete Event simulation, time advances directly to the next "significant" event. Contrast this with the fixed-increment time advance, also referred to as time-step approach, where time advances in predetermined increments, even during periods that are characterized by lack of significant activity. This is one of the reasons that the fixed-increment time advance mechanism is notorious for eating up a lot of computer time [Law & Kelton, 1982].

Due to the increasing popularity of agent-based simulation in defense research and analysis, numerous software packages have been developed and are currently being used in simulating combat environments. Despite the fact that the time-step approach has several known limitations and disadvantages, almost all of these software packages are using it. Some of the most common problems include:

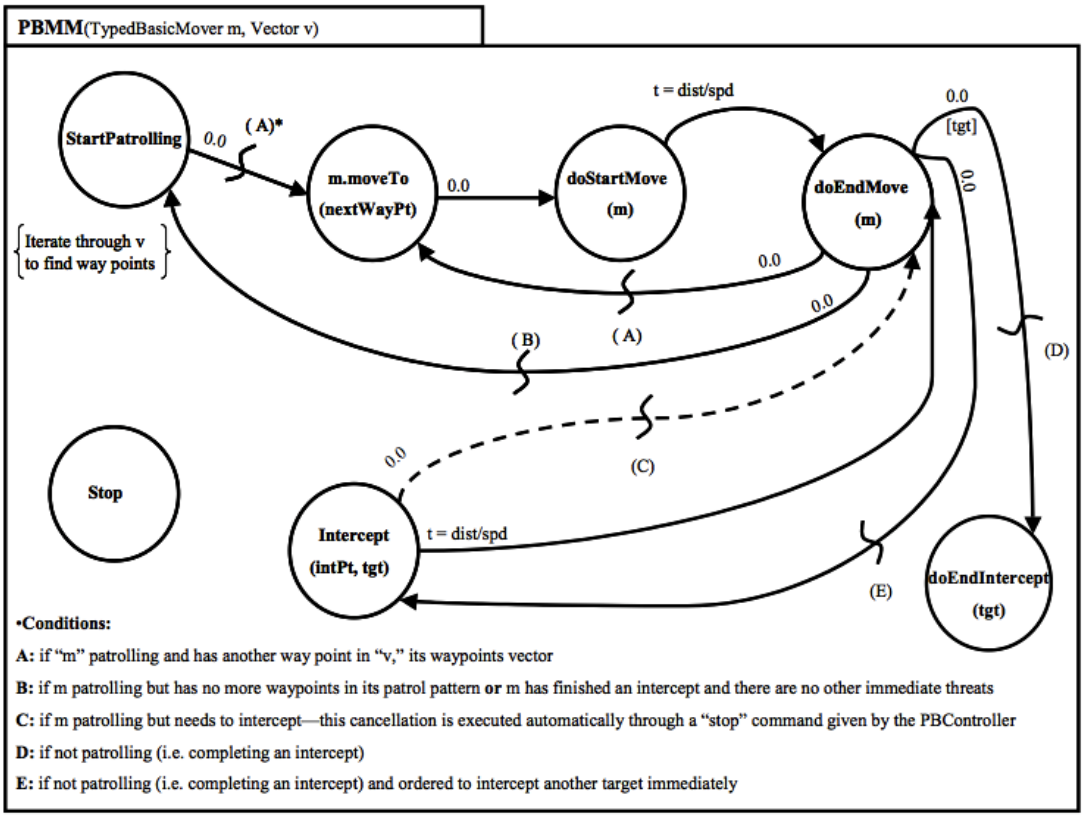
1. Computational inefficiency: time-step simulations have to check the position and state of all agents with respect to every other agent in the model in every time increment, regardless as to whether something "important" happens. This can be alleviated somewhat by subdividing the model into sectors, and only checking interactions between entities in adjacent sectors, but doing so adds artificial constraints and complexity to the model. Whether the model is sectorized or not, lots of computational power is wasted in useless calculations.
2. Scalability: time-step simulations check the relationships between pairs of agents in every time-step. This means that the computational power needed increases quadratically with the number of agents in the model. As a result, time-step models are often incapable of modeling realistic scenarios due to computing resource limitations.
3. The occurrence of artifacts: time-step simulations are known to present artifacts inherently related to the time-step approach. For example, agents may seem to aimlessly oscillate around a certain position. Another example of a model artifact is when two events, which in the real world occur at distinct times, both get "rounded" to the same time-step. The modeler must then make an artificial decision about what order to perform

> the actions in, which adds complexity to the model and increases the likelihood that the choice will not conform to the real-world circumstances.

Discrete Event Simulation (DES) is free of the limitations of the time-step approach. DES models usually run much faster than their time-step counterparts, allowing for more replications in a given time window; they scale better, which allows for much larger experiments; and they are free of the artifacts from which time-step simulations suffer. However, most of the commercial DES packages available are business-, manufacturing-, and logistics-oriented, and are unsuitable for modeling battlefield scenarios.

Simkit is a freely available package for building DES, developed by Professor A. H. Buss at the Naval Postgraduate School. Simkit represents DES using the Event Graph paradigm, first described by Schruben in 1983 [Buss, 2001]. The Event Graph is a simple, yet extremely powerful construct that graphically portrays DES models in a language-independent manner. Event Graphs provide a natural way to represent DES models and, indeed, are the only such constructs that directly capture the event-driven nature of these models [Buss & Sanchez, 2002] (Figure 1).

Combat simulations based on Simkit have historically been "single-use" models written for a specific scenario, despite the potential for model and component reuse. We believe that what is missing is a library of Simkit-based reusable components that will allow the user to easily build and experiment on a wide variety of Combat-oriented simulation models.



Mover Manager for Patrol Boat (Childs, 2002)

Figure 1. Example of an Event Graph. (From: Brutzman et al., 2004.)

C. SCOPE

The scope of this thesis is to design and implement a library of reusable software components that will extend Simkit for building combat-oriented agent-based simulation models. In Chapter II, we will briefly present the state-of-the-art simulation packages currently used in agent-based military modeling and note where the time-step approach causes difficulties. The DES methodology will be presented in Chapter III. We will show that, contrary to popular belief, modeling movement and sensing is not incompatible with DES. In Chapter IV, we will describe our design of what an agent-based DES implementation framework should look like. We will show that the extensive use of Java interfaces allows the user to implement different models and scenarios without being constrained by pre-built components. We also enhance the Sensing model

already implemented by Simkit by introducing a Situational Awareness model and a Behavioral model. Finally, we will build one small agent-based model using the component architecture to demonstrate the library's functionality.

THIS PAGE INTENTIONALLY LEFT BLANK

II. TIME-STEP MODELS OVERVIEW

The only reason for time is so that everything doesn't happen at once.

- Albert Einstein

A. INTRODUCTION

Since the beginning of the 20th century, several efforts have been made to analyze combat by the use of simple, yet intelligent, mathematical models. These models utilize mathematical methods (such as calculus or probability theory) to obtain an analytic solution to the questions of interest [Law, 2007]. Models of military encounters, however, are very complex systems and incorporate numerous intangible elements. Their outcomes are determined by the interplay of many factors such as equipment, communications and tactics, and the interactions between the combatants themselves. Simple mathematical models, like the Lanchester or Hughes' Salvo equations, cannot capture the effects of these interactions.

During the last decades, various Agent-Based Simulations (ABS) have been developed to address the problem of modeling battlefield interactions. Most of these models have one thing in common: they follow the time-step approach, thus inheriting the problems associated with it. ABSs like MANA and Pythagoras, both extensively used by NPS students, belong to this category. Our objective, on the other hand, is to build an ABS free of the problems of time-stepping. We will present a short overview of MANA and Pythagoras along with their merits and disadvantages, so that we effectively manifest our goals.

B. MANA [MCINTOSH ET AL., 2006]

MANA is an acronym for Map Aware Non-Uniform Automata. It was first released by the New Zealand Defence and Technology Agency (DTA) in 1999. The current version, 4.0, has been used since 2006. MANA falls into a subset of Agent-based models called Cellular Automaton models, of which Conway's "Game of Life" is the

best-known example. Cellular automata are dynamic systems of entities whose behavior is determined wholly in terms of local relations [Ferber, 1999]. The word "Automaton," derived from the Greek word "αυτόματος" meaning "acting on one's own will," denotes an autonomous entity driven by its own impulses and desires.

1. The Roots of MANA

The roots of MANA development lie in the pioneering work of Andy Ilachinsky and his ISAAC and EINSTein models. Andy Ilachinsky believed that it was possible to represent combat as a non-homogeneous system of entities (agents), with varying behaviors, which adapt to their environment and learn from their experiences (a Complex Adaptive System). He also showed that interactions generated between very simple sets of agent behaviors can produce an astonishingly wide range of outcomes. This appealing feature is in accordance with the Occam's razor principle: "entia non sunt multiplicanda praeter necessitatem."[1] According to the MANA developers: "…if even simple rules produce complicated and unpredictable behavior, then what hope is there of relating observed behavior back to specific aspects of a complicated rule set?"

2. Basic Characteristics of MANA

One of the main problems of using simulation to model real-world systems is that writing the computer program to execute them can be an arduous task indeed [Law, 2007]. This can be overcome by the use of specialized software that provides most of the tools required to "build" a simulation model. The nature of modern warfare, however, is so complex and chaotic that it makes it very hard to provide a one-design-fits-all pattern for the combatants' behaviors. One would have to create a specific set of rules for each type of scenario, an often tedious and time-consuming task. One of the main advantages of models like Ilachinsky's ISAAC/EINSTein is that new scenarios can be set up quite easily and quickly, using a simple set of agent parameters.

[1] "Entities should not be multiplied beyond necessity."

a. Decision-Making

MANA is based and expands on ISAAC/EINSTein. The basic element of a MANA model is the "squad," comprising several agents. A MANA agent is described by a set of user-defined parameters: personality weightings, move constraints (triggered by emerging behaviors), sensing characteristics and weapons' effects and, finally, movement constraints imposed by terrain features and/or randomness. The user is only required to adjust the parameter values. The agents, then, are set free to act "instinctively." Thus, the squad is not guided by a predetermined set of rules or by a leader entity within the squad. In MANA, the task of decision making is distributed among the simulated entities themselves.

This decentralization of the decision-making process does not guarantee that the agents will always act in an "optimal" fashion. On the contrary, MANA agents are expected to make "mistakes" and exhibit behaviors that were hard to predict in the first place. This approach, however, allows the user to quickly create, test and explore different behavioral patterns and identify the most efficient amongst them. It also allows the analysis of the consequences that different types of "mistakes" produce.

b. Movement Sensing and Weapons

Action in MANA takes place within a 200 x 200 grid. Each agent occupies one cell. Movement is governed by personality weightings assigned to each agent and the agent's relative position with respect to other agents. An agent may, for instance, be given the tendency to move toward enemy agents rather than friendly ones or vice-versa (Figure 1).

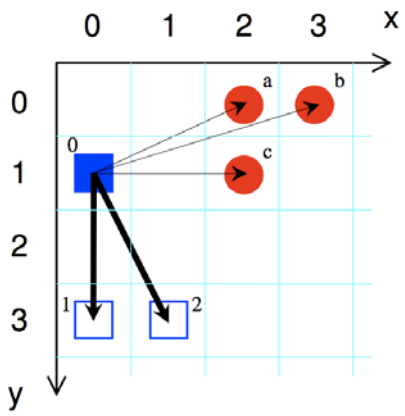


Figure 2. Movement in MANA. (From: McIntosh et al., 2006.)

MANA uses a basic "cookie-cutter" model to represent sensors and sensing. Detection events occur based on a simple probability calculation (roll of a die) within the sensor's range. There are two types of sensing models: Simple and Advanced. In the Simple sensing model, sensors are only described by their detection and classification range (Figure 2). In the Advanced mode, a range table can be set up to represent the degradation of the sensing capabilities with distance. Furthermore, in the Advanced mode, the user may limit detection events to occur only within a section of the circle defining the sensor's range. This requires the agent to "look around" in order to detect other agents. Weapons ranges and "hit" events are represented in an analogous way (Figure 3).

Figure 3. A Simple Sensor in MANA. (From: McIntosh et al., 2006.)

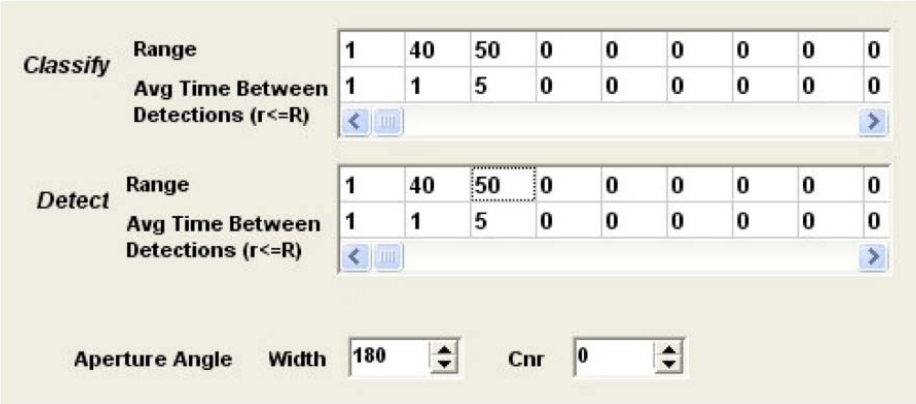


Figure 4. An Advanced Sensor in MANA. (From: McIntosh et al., 2006.)

c. Situational Awareness and Communications

Working with programs like JANUS and CAEn helped DTA's analysts realize that however detailed and rigorous these models were in aspects such as weapons' effects, they were lacking flexibility in effectively representing key characteristics of combat, such as command and control (C2), situational awareness (SA) and the informational edge that enhanced sensors provide. Pre-designed behaviors that these programs furnished were bounding the user in modeling these aspects of combat. Military technology evolves very quickly and current information models may rapidly turn obsolete. The user should be free to model new, and even not yet implemented, advancements in an effortless and arbitrary fashion. With that goal in mind, MANA developers introduced concepts that distinguished it from other agent-based models of the time.

MANA represents Situational Awareness (SA) by a map, shared by the members of each squad. The main idea is that all the information that a squad member collects is put up on the map. As a shared resource, it thus becomes accessible by all squad members. The SA map works in several ways: it represents a rudimentary form of collective memory; it makes a first step in reproducing an agent's basic cognitive processes by introducing a delay between target detection by an agent's sensor and its

positioning on the squad map; and finally, it represents communications within the squad, whereby agents share information as one would expect in a cohesively working unit (Figure 4).

MANA uses the SA map concept to model communications as well. Information that other squads obtain is combined in a dedicated map, called the squad's Inorganic map. Communication links are established by choosing which squads are allowed to exchange information. A dedicated set of parameters is used to simulate the quality of communications. These parameters represent delays in the flow of information, the likelihood that information gets lost en-route to its destination, the rate over time at which information flows through communication links, and whether all available information should be sent or only part of it (e.g., enemy contacts only). Furthermore, additional personality weightings allow the agents to respond to distant contacts that have not yet been detected by their own sensors.

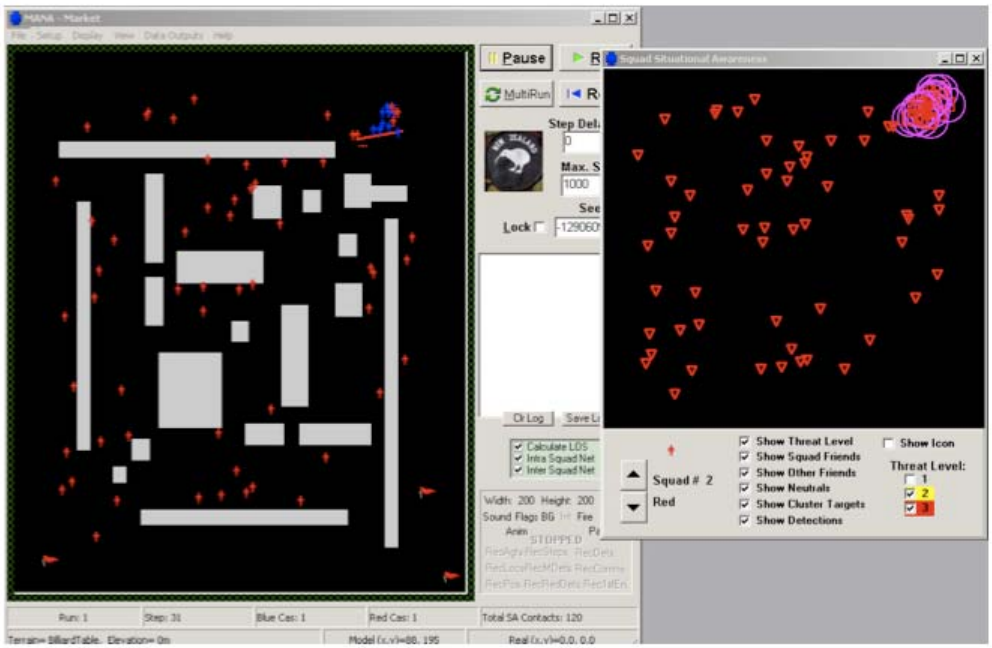


Figure 5. The Situational Awareness (SA) map. (From: McIntosh et al., 2006.)

C. PYTHAGORAS [BITINAS ET AL., 2003]

"Pythagoras is an ABS environment, originally developed to support Project Albert, a U.S. Marine Corps-sponsored international initiative that focused on human factors in military combat and non-combat situations" [Henscheid et al., 2006]. Pythagoras is similar to MANA in many ways: terrain is represented as a rectangular grid, movement is controlled by desires inherent to the agents, and emphasis has been given to the easy setup and assessment of various scenarios. What distinguishes Pythagoras from other ABSs, though, is the use of "soft rules" to establish individuality between agents and an elaborate color scheme to establish many gradations of sidedness.

1. Soft Rules

We are living in a relative world; individuality and subjectiveness characterize all human cognitive processes. Therefore, similar stimuli can generate diametrically different reactions in different individuals. Pythagoras was created to explore the effects of homogeneity (or non-homogeneity) in a group of agents, be it a group of villagers or a special-forces platoon.

When an agent is instantiated at the beginning of a scenario run, it selects the parameters describing its behavior from an appropriately chosen random distribution. Adjusting the spread of the distribution, the user can control the homogeneity of the group to which the agent belongs. Thus, a group can be created as very homogeneous (e.g., well trained, disciplined military forces), or quite heterogeneous (e.g., a crowd of civilians), or some value in between. The concept is illustrated in Figure 5.

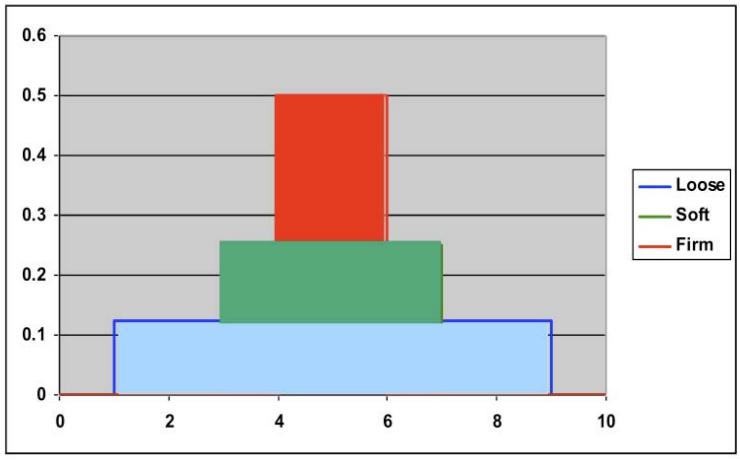


Figure 6. Three degrees of individuality. (From: Henscheid et al., 2006.)

The three distributions illustrated in Figure 5 have the same midpoint. They differ considerably, however, in spread. The values on the x-axis are quantitative representations of agent behaviors. They might represent, for instance, the minimum number of enemies from which an agent would retreat. The behavior value with which the agent is initialized is found along the x-axis, and its corresponding probability is the y-axis value. Agents choosing their behaviors from the red distribution will make a firmer, more homogeneous group, whereas agents choosing from the blue distribution will form a softer, more heterogeneous one.

2. Dynamic Sidedness

In Pythagoras, affiliation among the agents is represented using an RGB (Red/Green/Blue) color scheme. Unlike other ABSs though, not just one but any combination of the three basic colors may be used. Agents in Pythagoras are thus characterized by their combination of greenness, blueness and redness. Each of these three properties can take any integer value from 0 to 255 inclusive (corresponding to standard color-monitor settings the Java programming language uses to control image colors), yielding more than 16 million gradations. An example using both blue and green to establish an agent's affiliation is illustrated in Figure 6.

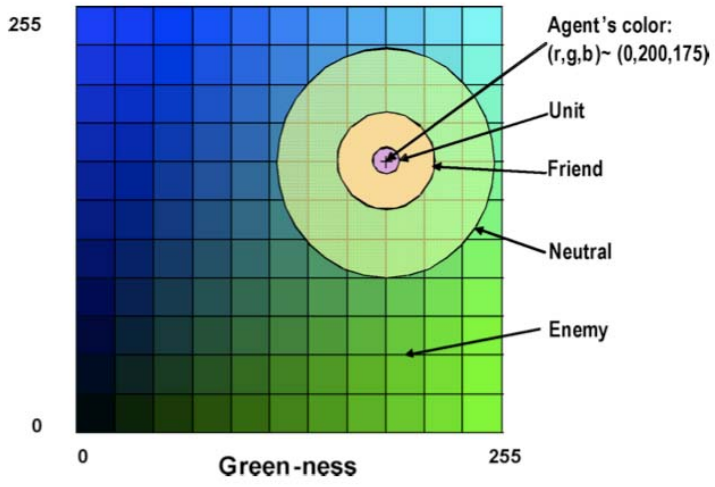


Figure 7. Two-color sidedness used to establish two-dimensional affiliation. (From: Henscheid et al., 2006)

Agents with similar color (as measured by either the difference in absolute value or the root sum square of the differences of the active colors) are considered to be members of the same unit. Those whose colors are close are considered to be friends. Those whose colors are far away are considered to be enemies. Colors between enemy and friendly agents are neutrals. This scheme allows for multiple affiliations within a single scenario. Colors can also be used to encode other characteristics of agents. Examples of other uses include: establishing command hierarchies among the agents (by using different shades of blue, for instance), or assigning colors to properties such as fear or morale.

3. Problems Related to the Time-Step Approach

MANA and Pythagoras are two simulation packages representative of the state-of-the-art in agent-based military modeling. Along with the early ISAAC/EINSTein models, they introduced concepts that changed the landscape of simulation-assisted scenario analysis in the Western world. One of the characteristics that these models have in common though, the use of the time-step approach for advancing simulation time, proves to be an Achilles' heel in some of their implementations.

In simulation there are, in general, two approaches to advancing simulated time: the time-step approach and the discrete-event approach. Simulations that use the time-step approach advance the simulation clock in increments of exactly Δt time units. After each time-step, they check whether one or more events should have occurred within the previous time increment; if there are any, they are considered to occur at the end of the time increment [Law & Kelton, 1982]. On the other hand, discrete-event simulations advance the clock directly to the next "significant" event.

The choice of the time advance mechanism for a simulation usually relies upon the situation modeled. Financial-planning simulations, for example, call for a fixed time increment because plans are commonly made on a daily, weekly, monthly or annual basis [Watson et al., 1989]. When it comes to simulating a complicated battlefield environment, though, selecting a fixed-increment time advance mechanism may create challenges hard to fathom.

Watching the animated screens of simulations like MANA and Pythagoras, for instance, one may notice a seemingly random jitter around an average position, which characterizes many agents' movements. This modeling artifact, typical of time-stepping, can be explained by the use of an example. Consider a soldier on a battlefield adhering to the following rule: "Move towards the enemy side; if you detect an enemy, "join" at least three friendly soldiers, then move toward the enemy," with "join" translating to "come within a certain distance." Now suppose that moving toward the enemy side, our soldier detects an enemy force. Because he does not want to make a hero of himself (and because he is designed to do so, of course), he turns to find his fellows. In the next time-step, as he moves toward the friendly side, he loses track of the enemy. Because he sees no enemies, he turns back toward the enemy side. In the next time step he detects the enemy force (again), but no friendly forces are close enough, so he turns to join his fellow soldiers, and so on and so forth. Although unrealistic for an actual situation, and rather unpleasant to the eye, this phenomenon is not considered to create any serious repercussions regarding the model's general behavior. There are, however, more important considerations regarding the use of time-stepping.

One of the most serious implications comes from the lack of efficiency in the use of computational power. Time-step models will check for the occurrence of events at the end of every time increment, even if no event was to occur, unnecessarily slowing down the simulation run. This characteristic, ingrained in time-step models like MANA and Pythagoras, affects their usability in multiple ways. One obvious drawback is that experiments using time-step models usually take longer to complete, so that the advantage gained by the quick simulation setup may well be lost at run-time. Another implication comes from the fact that time-step simulations have to recompute, at every time-step, all pairwise associations between agents. If n is the total number of agents in the model, the number of all possible pairwise associations between them will be proportional to $n \cdot n$, or n^2. Thus, at the end of every time-step, the simulation has to perform on the order of n^2 operations to determine the states of all the agents. Because of this, run time increases quadratically with the number of agents involved. Because of the increased time required for a single run, large scale simulation experiments get harder to perform. The use of an increased number of agents may slow down the experiment so much that users are often forced to decrease the scale of the model, just to get results within an acceptable time frame.

One way of solving the problem of increased run time and limited scalability of time-step simulations is by increasing the size of the time-step. This approach, however, poses new challenges. After each update of the clock, the simulation checks whether any events should have occurred during the previous interval of length Δt. The larger the time-step is, of course, the more events are likely to occur. The model considers these events to happen at the end of the interval, *all at once* [Law, 2007].

Because the model treats non-simultaneous events as if they were simultaneous, the modeler has to figure out a way of determining which events should be processed first. The higher the complexity of the simulation, though, the harder this task is going to be and the more errors are likely to occur. As a result, the size of the time-step may influence the simulation output. R. Hillestad et al., examining the differences in combat outcomes predicted by models of different resolutions, concluded that "changing the time

step, changes the battle outcome dramatically. [In] a corps or theater model, it might be convenient to have long time steps in order to keep run time down; this could be dangerous."

Non-monotonicities in the output space of many combat simulations have also been attributed to time stepping. J. A. Dewar et al., exploring a deterministic, time-stepped combat model, observed that the outcome exhibited a non-monotonic behavior with respect to the initial conditions. The model consisted of two opposing forces fighting until a stopping criterion was reached. In many cases, an increase in the initial size of one force would result in an outcome less favorable for the same force, for no obvious reason. Dewar et al. concluded that the choice of time-step had a significant role in the final outcome. They noted that "time step granularity [was] a particular concern.". "[If] the time step taken [was] too large, non-monotonicities [could] occur." They also pointed out that "the choice of the time-step [was] independent of the situation being modeled and [depended] only on the mathematical behavior of the model." [Dewar et al., 1996]

Early combat simulation developers used the time-step method to break-up time and action into small, manageable pieces. By doing so, however, they introduced new problems to already complicated (chaotic, sometimes,) systems. In simple, small-scale models this approach posed no significant limitations. As models grew larger and more complicated, though, the problems related to the use of time stepping became more apparent. Limitations of the time step-approach become limitations of the models. We believe that modeling conventions should not come between the modeler and his representation of the world.

III. DISCRETE EVENT METHODOLOGY AND AGENT-BASED SIMULATION

> There is an appointed time for everything, and a time for every affair under the heavens.
>
> - Ecclesiastes, 3:1

A. THE DISCRETE EVENT METHODOLOGY

In order to describe a system at any particular time, we need to define a set of variables relative to the objectives of study [Law, 2007]. We define this set of variables as the system's "state." DES models are identified by three important characteristics regarding their state variables. First, they are usually stochastic in nature; some, or all, of their state variables are generated by random distributions. Second, they are dynamic; their system states evolve over time. Third, and most important of all, they are event-driven; any changes in the state variables are associated with events that occur at discrete time instances only [Leemis et al., 2006]. In fact, an event is formally defined to be a point in time at which the system state changes.

The mechanism responsible for advancing simulation time in DESs is the Pending Events List (PEL) [Banks et al., 1996]. The PEL stores all pending events, always keeping "on top" of the list the event scheduled to occur at a time closest to the current (simulated) time. It ensures, thus, that everything takes place in the correct chronological order. Sometimes the model requires that more than one event be scheduled to occur at the exact same simulated time. The PEL allows this to happen. Concurrent events, though, must be prioritized by the use of secondary tie-breaking rules. When an event occurs, by convention, all state changes associated with that event are performed; next, all further events are scheduled; finally, the event notice is removed from the PEL. When and if the PEL becomes empty, the simulation terminates. Note that because of the terminating condition, the simulation will not run unless at least one event is found in the PEL [Buss, 2001].

The PEL principles are not always easy to visualize in terms of events and state variables. The difficulty of representing a DES model increases dramatically with the size and the complexity of the model. In 1983, Lee W. Schruben proposed the Event Graph methodology to portray DES models graphically in a language-independent manner [Buss and Sánchez, 2002]. The Event Graph methodology provides one of the most natural and elegant ways to represent a DES.

Figure 8 shows a rudimentary event graph. Events are represented by single vertices in the graph, illustrated here as circles. Any state changes associated with a particular event are usually placed below the event vertex, inside curly braces ("{}"). The scheduling relationship between events is represented by a directed edge between the vertices. The edge originates at the event that conducts the scheduling, and terminates at the event to be scheduled. If the edge is a dotted line instead of a solid one, the event to which the edge points gets cancelled. The delay between the scheduling event and the actual occurrence of the scheduled event is represented by a number, in simulated time units, placed above the edge's tail. If no number appears above the edge's tail, by convention the scheduled event occurs with zero delay. Sometimes it is required that the scheduling of an event must not take place unless a certain condition be satisfied. The condition is a Boolean function of the state, located next to a tilde-shaped line that bisects the scheduling edge. Event Graph notation can also provide representations of arguments being passed to events, and a way of prioritizing concurrent events (Figure 9).

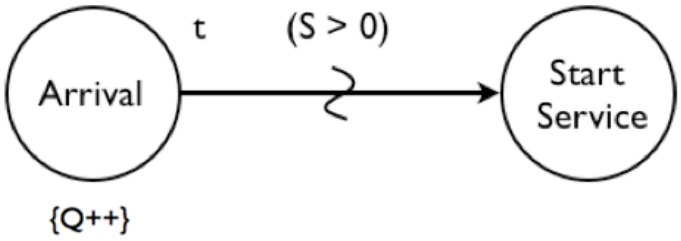


Figure 8. A rudimentary event graph. (After: Buss, 2001.)

The Event Graph in Figure 10 is an example of a Multiple Server Queue. The system state is defined by two variables: Q – the number of customers in queue; and S – the number of available servers. The occurrence of an "Arrival" event increases the number of customers in queue by one, and puts a "Start Service" event in the PEL, *provided* that there are available servers. Note that the "Arrival" event also schedules *itself*, with a delay of t_a. Once a "Start Service" event occurs, both S and Q are decreased by one, and an "End Service" event is scheduled with a delay of t_s. The "End Service" event increases the number of available servers by one and schedules a "Start Service" event *if* there are any customers in the queue.

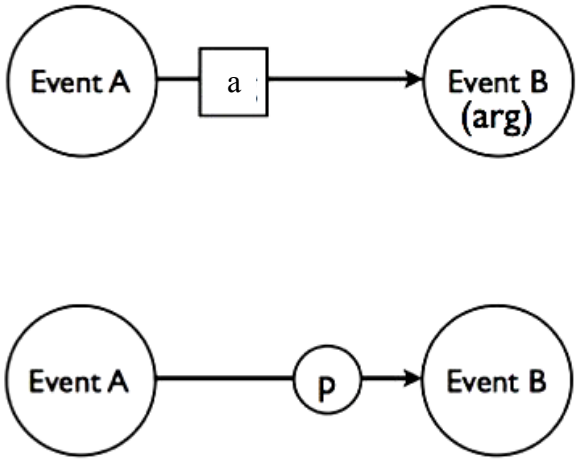


Figure 9. Special cases of Event Graph notation. (After: Buss, 2001.)

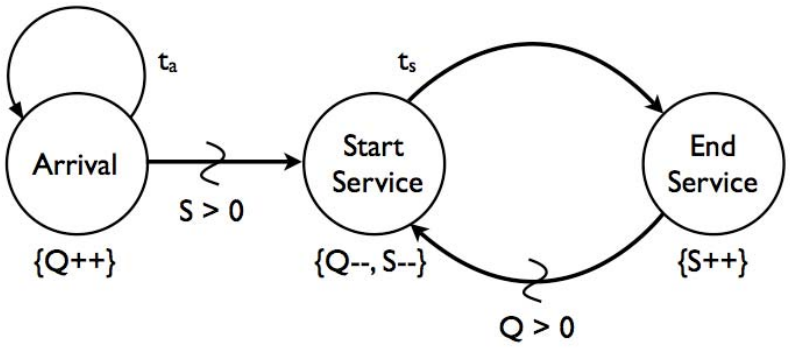


Figure 10. Event graph of a Multiple Server Queue. (After: Buss, 2001.)

The principles of DES and the Event List methodology are much harder to implement when representing a battlefield environment. A model involving agents that move and sense is far more complex than a simple customer-server system. The difficulty in representing movement comes, mainly, from the fact that an agent's position can constantly change. The intuitive approach would be to compute the agent's position and any associated state changes, at regular time increments. Time stepping is, however, associated with various modeling difficulties and artifacts described in Chapter II. A. H. Buss and P. J. Sánchez showed that the use of the DES paradigm to model moving and sensing is not only possible, but also desirable, from several standpoints [Buss and Sánchez, 2005].

B. MOVEMENT AND SENSING IN DES [BUSS AND SÁNCHEZ, 2005]

1. Representing Movement in DES

Time-step models, in general, use grids for representing terrain. (Hexagonal and rectangular grids have both been used — each having some advantages and some disadvantages.) An agent is allowed to occupy exactly one cell on the grid. After each time-step a new location is calculated for every agent on the grid, depending on the agent's speed and direction of movement. The new location is likely to be an adjacent or

nearby cell (it may be the same cell or even completely out of the grid, though). Note that the grid's scale must be appropriately chosen, so that no large rounding errors occur in the agents' movements. This can be especially challenging when modeling agents with greatly varying speeds (foot soldiers and aircraft, for instance). Agents moving at relatively high speeds may travel across more than one cell in a given time-step. The greater the speed, the more cells will be traversed. The simulation has to check for occurring events (such as detection) at every cell the agent crosses. Selecting too fine a grid increases the number of traversed cells even more, thus slowing down the simulation run.

A DES represents movement without relying on the use of an explicit location state for the agent. Instead, by storing a set of initial conditions it computes the agent's location "on demand." For an agent that moves in a uniform, linear fashion, it suffices to store the initial position, the time at which movement starts, and the agent's velocity vector. Simple uniform and linear motion equations can calculate the agent's position as a function of simulated time. Subsequent events involving the moving agent (detection by a sensor, for instance) are scheduled with respect to the initial conditions, and can be calculated at the start of movement. Those calculations are valid for the duration of the movement process, and need only be reassessed when and if one of the parties involved changes its motion status, i.e., when the corresponding change movement event occurs.

Movement representation in DES need not be limited to uniform linear motion though. Almost any closed-form equation could replace the uniform and linear motion equations of the previous example. A. H. Buss and P. J. Sánchez claim, however, that according to their experience, "linear motion can provide a wide range of possibilities. Acceleration and turning, for example, can be modeled using a piecewise linear approximation to smooth curved trajectories." This is as high a level of detail as most medium to large-scaled combat simulations really need. In practice, this should be no worse than the piecewise approximations represented by grid-based movement, and may be substantially better.

2. Representing Detection in DES

Consider a stationary detecting device, and a target positioned in the vicinity of the detector, that starts moving in a random direction. Let P be the probability that the detector detects the target. One of the simplest representations of a detecting device is where the target, once within a fixed distance from the device, is detected with probability p_d=1. The detection range of such a device can be represented as a circle centered at the device. Outside the circle, the probability of detection is p_d=0. We call such a detecting device a "Cookie-Cutter sensor" (Figure 11).

Consider the target's line of motion and the ring defining the Cookie-Cutter sensor's range. The intersections between the target's path and the sensor ring, if any, represent the points where the target enters or exits the sensor's range. The target's arrival at any of these points triggers an "Enter Range" or an "Exit Range" event, accordingly. Cookie-Cutter sensing requires that a "Detection" event be scheduled immediately after an "Enter Range" event and an "Undetection" event immediately after the "Exit Range" event. The times at which targets enter or exit a sensor's range can be calculated by solving a simple quadratic equation. Depending on the target's initial location and velocity, there may be two, one, or no solutions. If a target stops moving, changes speed or changes direction, all detection events already scheduled have to be cancelled and detect/undetect times have to be recomputed. This example can be generalized to include the case where both the sensor and the target move, by considering one of them to be stationary and using their relative velocity to calculate detection time.

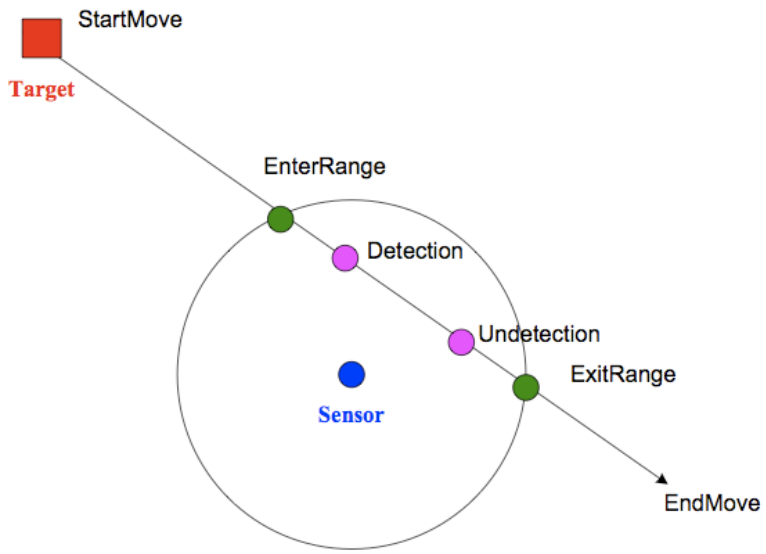


Figure 11. A Cookie-Cutter sensor. (From: Buss and Sánchez, 2005.)

Cookie-Cutter detection is the simplest detection type possible and may be unsuitable for many modeling needs. It can be used, however, as a basis for other, more complex sensing models and sensors. Scheduling of a "Detection" event can be accomplished by a computational algorithm, triggered by an "Enter Range" event. Consider, for instance, a ground radar station and an aerial target. Suppose that a target enters the radar's maximum range but detection does not immediately occur (because of unfavorable weather conditions, or due to the radar operator's cognitive processes). We can represent the radar station by a Cookie-Cutter sensor and the detection hindrance by a suitably chosen random distribution. The "Enter Range" event will trigger a "Detection" event with a delay determined by the random distribution. The radar's operator may even miss the target entirely if the target exits the radar's maximum range before detection occurs. The "Exit Range" event will cancel any subsequent "Detection" events for the particular target.

It is the DES's ability to effortlessly reproduce time-step rules that most clearly exhibits its power as a modeling tool. Consider the following rule: once a target enters a sensor's maximum range, there is a probability of detection p_d at every time increment Δt. This simple rule, that typifies the representation of sensing and detection in time-step

models, can be easily reproduced in DES. Note that, according to the rule, the process of detection is equivalent to a sequence of Bernoulli trials, with probability of success p_d. Let N be the number of trials until detection; N is a geometric random variable with parameter p_d. In a DES, the "Detection" event could be scheduled after the "Enter Range" event, with delay corresponding to N = n failures. The time to detection would be equal to the product of failed attempts by the time increment length, $n \cdot \Delta t$. It is well known that the continuous analog of this process is the exponential distribution. One could model the time to detection using an exponential distribution if the sensor was omnidirectional, or using the geometric distribution if the sensor periodically inspected a given sector, such as a sweep radar.

3. DES and Agent-Based Models

Buss and Sánchez showed that it is possible to represent movement and sensing in a DES model. An implementation of their ideas can be found in Simkit, a free, Java-based DES package. Simkit has been used by NPS students to simulate various battlefield situations incorporating entities that move and sense. Very few of the existing implementations, though, approach the modeled scenarios from an ABS standpoint. Agents in ABS are considered to be autonomous and self-governing. In most of the existing implementations, the moving entities are governed by single decision-making entities lying "outside" of the moving entities themselves.

Furthermore, most of the existing combat simulations in Simkit are "single-use" models, written for a specific scenario. Object-oriented programming languages, like Java, offer the potential for model and component reuse. The creation of a library of Simkit-based reusable components would allow the development of complicated models, by the use of simple and tested building blocks. These components should be designed in a way that would not limit a user's prerogative to implement his/her own ideas. On the contrary, modelers should be able to tailor existing components to suit their particular modeling needs.

In the following chapter, we will take some first steps toward the creation of a Java library to expand Simkit into the realm of ABS. We will design a framework upon which an agent-based model could be built. We will make extensive use of Java interfaces to make our code extendible and reusable. And finally, we will implement our designs in a small simulation experiment.

THIS PAGE INTENTIONALLY LEFT BLANK

IV. DISCRETE EVENT METHODOLOGY AND AGENT-BASED MODELS: FROM THEORY TO IMPLEMENTATION

Little by little does the trick

- Aesop

A. INTRODUCTION

In this chapter, we describe our efforts to build Agent Based (ABS) models using the Discrete Event paradigm. Our implementation is written in the Java programming language and uses the Simkit DES package. We make the assumption that the reader is familiar with Java basics, Simkit, and the Event Graph methodology. If a review of these topics is needed, any introductory Java text, combined with Buss, 1996 and 2001, and Buss and Sánchez, 2002 should provide the necessary background.

We will describe the framework we created to represent our agents, our extension of Simkit's Cookie Cutter sensing model, and a simple Agent-Based model we built to implement our design. By convention, Java file names consist of a single word, written in a "CamelCase" fashion. CamelCase is the practice of connecting multiple words by capitalizing each word's first letter, as in "UniformLinearMover." Throughout this chapter, we will make extensive use of the CamelCase convention.

B. THE AGENT

1. Agent

The agent is to the Agent Based Simulation what the actor is to the play. The agent, whether walk-on or protagonist, influences, more than any other element, the shape and course of all action in our representation of the world. It would not be unreasonable to claim that designing an agent-based model is more about designing the agents themselves.

Stuart Russell and Peter Norvig give a very simple definition of the agent: "An agent is anything that can be viewed as perceiving its environment through sensors, and acting upon the environment through actuators." This definition implies the existence, within the agent, of an internal mechanism translating inputs to outputs, percepts to actions. Mapping any sequence of an agent's perceptual inputs to specific actions is tantamount to fully describing the agent's behavior [Russell & Norvig, 2003].

Agent Based Simulations involve models whose functions and outcomes mainly depend on the interactions between autonomous, self-sufficient entities. The function of an Object-Oriented Program also derives from the interactions between autonomous components called Objects. It is, therefore, easy to see why an Object-Oriented Programming language, like Java, provides an excellent platform for representing agents in an ABS.

Our proposed design is fulfilled in the Agent interface. We consider the Agent to be our model's most important design element. Our framework follows a blueprint outlined by Russell and Norvig's simple definition. We equipped our agents with sensors, allowing them to perceive their environment. We infused them with special behaviors, delineating each agent's distinctive characteristics. Finally, to some behaviors, we attached appropriate actuators, so that decisions could be turned into actions (Figure 12).

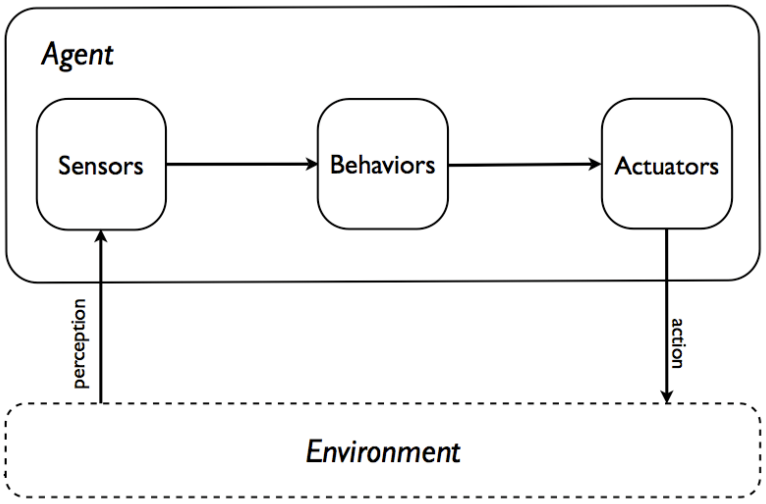


Figure 12. Russell and Norvig's agent.

2. Mover

In the Combat-oriented models that we plan to create, agents will be moving entities more often than they will be stationary ones. To depict a moving entity's functions, Simkit incorporates a dedicated interface named Mover. The UniformLinearMover class that Simkit uses to represent moving entities implements the Mover interface. UniformLinearMover (ULM) objects model entities that move at constant speeds in straight lines. This is as high a level of detail as most models really need since accelerations can be well-approximated with a sequence of velocities, and non-linear trajectories can be well-approximated with a series of piecewise-linear segments. Our Agent objects, though, do not extend the ULM class. They are designed, instead, to implement the Mover interface themselves. Current Agent implementations use ULM objects as instance variables, and reflect a ULM's functions through methods enforced by the Mover interface. This scheme allows the currently used ULMs to be replaced, in existing Agent objects, by any future design that implements the Mover interface.

3. Listener Patterns [Buss & Sanchez, 2005]

To allow communication between different components, Simkit implements two listener mechanisms, or patterns: the "Event Listener Pattern," and the "Property Change Listener Pattern." Both are extensively used in our models.

The "Event Listener Pattern" allows Simkit components to "listen" to each other's events. When an event occurs at a component that is being monitored, its registered Event Listeners are notified. Events that belong to the Event Listeners, and that share the same name and signature with the heard event, are executed as if they were scheduled by the Listeners themselves. A graphic representation of the Event Listener Pattern is provided in Figure 13. The stethoscope-looking edge between Component A and Component B indicates that Component B listens to events occurring in Component A.

Simkit components may also use the "Property Change Listener Pattern" to hear Property Change events occurring in other Simkit components. A Property Change event is usually triggered when a state variable, within a component, changes its value. The Property Change event broadcasts the property name, and either 1) both the old and the new value of the state variable, or 2) the new value only. A graphic representation of the Property Change Listener Pattern is provided in Figure 14. The edge between Component A and Component B indicates that Component B listens to Property Changes occurring in Component A.

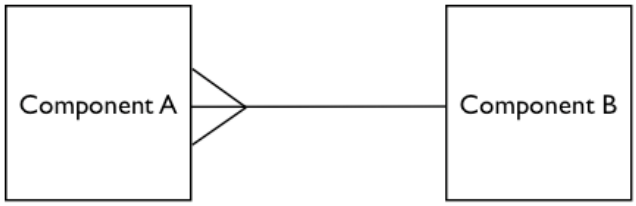


Figure 13. The "Event Listener Pattern"

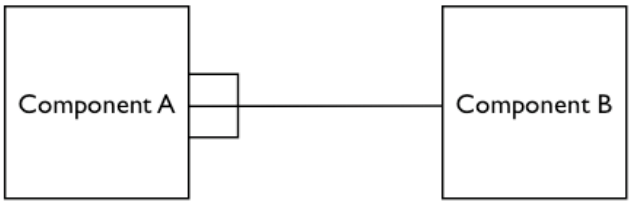


Figure 14. The "Property Change Listener Pattern"

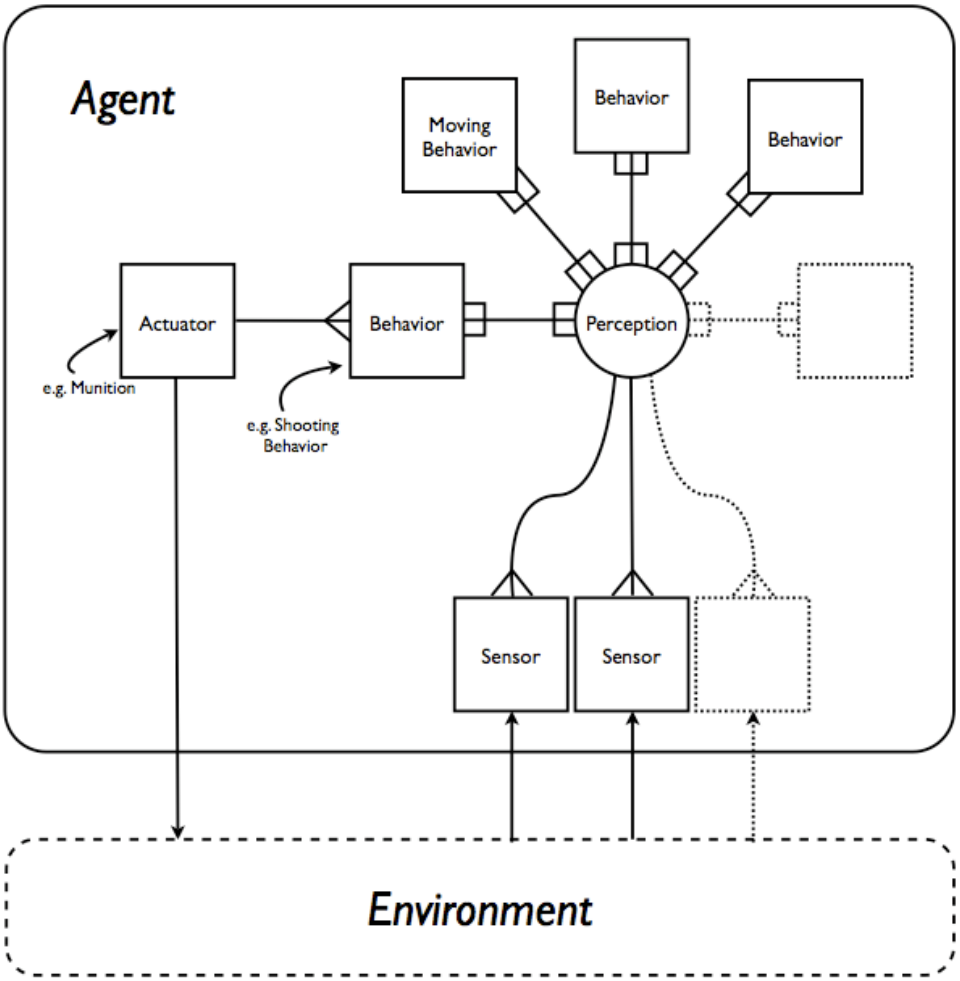


Figure 15. Implementation of the Agent

4. Sensor

Agent objects perceive their environment through a set of Sensor objects, which are Simkit components implementing the Sensor interface. There is no theoretical limit to the number of Sensor objects that an agent can "carry." Sensor objects communicate information to a Perception component using the "Event Listener" pattern. The Perception component is responsible for the distribution of information, collected by the Sensor objects, to the agent's Behavior objects.

5. Behavior

The Behavior components are responsible for translating information, which the Agent's Sensor objects provide, into actions. Behavior objects are built as essentially autonomous components, performing simple tasks. They receive their input from, and deliver their output to, the agent's Perception via the "Property Change Listener Pattern." A String variable, named "beholder," tags the Behavior components. The beholder variable denotes the agent to which a Behavior belongs; thus, a Behavior can distinguish Property Change events "fired" by Behavior objects belonging to the same owner.

Existing Behavior components consist of two parts: a basic one and a derived one. The basic part holds the state variables and is responsible for the scheduling of events. The derived part permits communication and interaction with the Behavior object's immediate environment. The basic part is declared an "abstract" class and cannot be instantiated. To use an existing Behavior module, one must derive a new class from the corresponding basic part. As soon as the derived part is created, the user must determine the Behavior's way of responding (in terms of event scheduling) to Property Change events being rebroadcast, or triggered, by the agent's Perception. Depending on the situation being modeled, a Property Change event may schedule different simulation events in the Behavior module that hears it, and vice-versa; different Property Changes may schedule further occurrences of the same simulation event. This proposed design

allows the synthesis of Behavior modules within the agent framework in multiple ways to form different, model-specific, behavioral patterns. Furthermore, the reusability of existing Behavior modules is enhanced.

6. Perception

The Perception component has been designed to mainly function as a hub to which an agent's Behavior and Sensor modules are connected. Perception is connected to the agent's Sensor objects as an "Event Listener." It "listens" to all events triggered by the Sensor objects, like "Detection," "Undetection" and "Classification." These events carry, as arguments, relevant information collected by the Sensor. As soon as the Perception component "hears" one of these events, it redistributes the information to the agent's Behavior objects, through an appropriate Property Change Event. Perception also allows communication between the agent's Behavior objects. Behavior objects have a reciprocal "Property Change Listener" relationship with the Perception component to which they are connected. The Perception component listens to, and rebroadcasts, all Property Change events fired by the Behavior objects to which it is attached (Figure 15).

Every agent may have only one Perception component. Many agents, however, may share the same Perception component. This scheme may be utilized to represent the sharing of information within a group of agents. Such a group can be defined by the sharing of a common Perception object. Because a shared Perception object will be an Event Listener to the Sensor objects of all agents belonging to the same group, any events scheduled by an agent's Sensor will be communicated to (the Behavior objects of) all the agents belonging to the group.

7. Actuator

Agent objects use instances of Actuator to interact with their environment. Each Actuator object must be controlled by a specific Behavior. For the time being, only Munition objects have been used as Actuator objects.

C. THE SENSING MODEL

The Cookie Cutter sensing model used by Simkit, albeit simple, is flexible and powerful. However inadequate it may seem when it comes to the representation of sensing in modern battlefield environments, it actually allows the modeling of complicated concepts such as the scanning of an area by a sweep radar, or the delay in a target's detection due to the operator's cognitive processes. We created a number of new, Simkit based, components to demonstrate this capability.

1. RandomDelaySensor

The RandomDelaySensor (RDS) extends the Sensor interface by adding setter and getter methods for a RandomVariate object. (An instance of RandomVariate generates pseudo-random numbers from a specified random distribution.) The RandomVariate object may represent characteristics belonging to the modeled sensor or its operator, and is meant to be used by a Mediator in determining the delay between the "Enter Range" and "Detection" events. When a target enters a Sensor object's maximum range, an "Enter Range" event immediately occurs and a "Detection" event enters the Pending Events List (PEL). In an RDS, the delay between the "Enter Range" event and actual "Detection" may be represented by generation of a random value from the RandomVariate object. This design can model a number of different concepts, depending on the choice of the random distribution used by the RandomVariate. For instance, a delay generated by an Exponential distribution may represent the scanning of an area by a sweep radar, with the probability of detection being equal to the probability of success in a series of Bernoulli trials.

2. ClassifyingSensor

The ClassifyingSensor (CS) interface extends the RandomDelaySensor by incorporating: a "Classification" event; a getter method, returning an instance of Map; and setter/getter methods for a RandomVariate instance variable.

In many cases, an agent needs to "know" more about a target than merely its location and velocity. The "Classification" event has been introduced to allow the identification of a target's qualities and characteristics. The CS interface enforces a getter method for an object that maps instances of Moveable to instances of Object. Such a mapping could be used to assign properties to targets, and thus, store collected information for future retrieval. The RandomVariate object, for which setter/getter methods are enforced, may be utilized to represent the delay between a "Detection" and a "Classification" event, and possibly, between subsequent "Classification" events.

3. ClassifyingCookieCutterSensor

An implementation of the CS interface can be found in the ClassifyingCookie CutterSensor (CCCS) class. The CCCS extends the CookieCutterSensor class, thus inheriting all its methods and instance variables.

The "Classification" event, enforced by the CS interface, models the identification of a target's characteristics by a Sensor. "Classification" events are never scheduled by the Sensor itself, but rather by an appropriately designed Mediator. The same Mediator also provides, as arguments to the "Classification" event, the object that represents the identified characteristic and the target to which the characteristic belongs.

It is assumed that the Sensor collects a target's properties one at a time. Collected information is stored in an instance of Map, which associates each target to a set of collected properties. Any type of object can be added to the set of properties, but only one object of a particular class can be found in the set. For example, if a target is classified as belonging to a specific affiliation, the "Classification" event will attempt to add a Side object to the set of the target's properties. If a Side object already exists there (added by an older "Classification" event), the old Side object will be removed and the new object will take its place.

Because CS extends the RDS interface, a CCCS will have two instances of RandomVariate. These generate detection and classification delays, and may be used to model aspects such as inherent sensor characteristics, varying weather conditions or the operator's personality traits.

4. MoverAlias

Simkit uses instances of the Mediator interface to schedule "Detection" events on behalf of the Sensor objects. The "Detection" event takes, as arguments, objects of type Contact generated by the Mediator. Contact objects serve as aliases for detected Mover objects, reflecting only their velocity and location. The Sensor objects that Simkit implements are designed to only discern a target's location and velocity, as most actual sensors do. If "Detection" events provided access to the Mover object itself, this principle would be violated.

Contact objects, however useful they may be, are really rudimentary and lack some basic functions. For instance, Simkit's Contact objects are not able to reflect the Mover object's Movement State. Therefore, if a Mover changes course or speed while being tracked by a Sensor, there is no way for the Sensor to "know." For that reason, we created and implemented a new class, named MoverAlias, whose instances can be registered as a PropertyChangeListener to the Mover they represent. As long as the Mover remains detected by the Sensor, the corresponding MoverAlias object will rebroadcast any changes in the Mover object's movement state.

5. RandomDelayCookieCutterMediator

The RandomDelayCookieCutterMediator (RDCCM) extends Simkit's CookieCutterMediator by overriding the "Enter Range" and "Exit Range" events, and by replacing Simkit's Contact with our MoverAlias class.

When an "Enter Range" event is heard by the RDCCM, a check is made as to whether the Sensor is an instance of RandomDelaySensor. If this is the case, the Sensor is cast to RDS and a "Detection" event is scheduled, with a delay generated by the RDS's RandomVariate object. If the target exits the Sensor object's maximum range before "Detection" takes place, the "Detection" event is cancelled and "Undetection" immediately occurs. Thus, a target crossing the Sensor object's maximum range and quickly exiting, or a slower but stealthier target, may not be detected at all.

6. ClassificationMediator

The ClassificationMediator (CM) extends Simkit's Mediator interface by adding a "Categorization" event. A CM's main function is to retrieve properties from detected targets and to schedule corresponding "Classification" events for ClassifyingSensor objects.

The CM uses the "Categorization" event to initiate the process of classifying a target. It is similar to the "Enter Range" event, in that it retrieves relevant information from the target and schedules a corresponding "Classification" event for the CS. (Analogously, the "Enter Range" event generates a MoverAlias object and schedules a "Detection" event for the Sensor.)

7. StackedPropertiesMediator

An example of a CM implementation is the StackedPropertiesMediator (SPM). The SPM is an RDCCM that implements the CM interface. An SPM holds an ordered list of properties that are to be retrieved from a target. When an "Enter Range" event occurs, and after scheduling a "Detection" event, an SPM iterates over the list of target characteristics. It then schedules appropriate "Categorization" and "Classification" events to pass the detected characteristics to the corresponding Sensor objects.

D. CREATING AN AGENT

1. Registrar

When an agent is created, a series of declarations have to be made in order for the agent's components to integrate with the model. Similarly, when an agent needs to be removed from the model (e.g., it "dies") its components have to be removed, and any associated scheduled events have to be cancelled. In our models, these tasks are typically performed by an instance of the Registrar interface.

2. BasicClassifyingPerception

Agent objects in our models are built "around" an instance of Perception. The Perception components that we use are objects of the BasicClassifyingPerception (BCP) class. The set of Agent objects, which own an object of the BCP class, are stored in an appropriate list within the Perception object. A BCP listens to all Sensor objects that are attached to it for "Detection," "Undetection" and "Classification" events. When a "Detection" or "Classification" event occurs, corresponding Property Change events are fired for each Agent in the list. When an "Undetection" event is heard, it means that one Sensor has lost a target. The BCP checks the contacts of the remaining Sensor objects to which it is connected as an Event Listener, and will not fire an "Undetection" Property Change unless all of them have lost the target.

3. Agent Creation

Every created agent, in our models, extends the AgentBase (AB) class. The AB class implements the Agent interface and Simkit's PropertyChangeListener (PCL) and Target interfaces.

The PCL interface ensures that the agent will be able to hear and rebroadcast the underlying Mover object's movement state. Depending on the model, this information may or may not be used by the agent's Behavior objects. The Target interface, on the other hand, enforcing methods with names such as kill(), hit() and isAlive(), is a reference to the agent's mortality.

There is series of actions and declarations one should carry out in order to create an agent. The existing design demands that the user: (1) create an AgentBase object; (2) add an instance of Registrar as an Event Listener to the AgentBase object to undertake any necessary declarations regarding the agent's Mover, Sensor and Actuator objects; (3) create a Mover object and add it to the agent; (4) generate the agent's Sensor, Behavior and Actuator objects, in that order, and add them to the agent; (5) activate a MoverManager by use of the corresponding start() method.

To demonstrate our library's functionality, we created two simple models involving agents that move, sense and perceive their environment in a partial way.

E. THE RANDOM SEARCH MODEL

In the "Random Search" model, an agent of Red affiliation (villain) and a team of Blue agents (searchers) enter a rectangular area. In the area, there are numerous agents of Green affiliation (civilians). All agents in the model are moving independently of each other and randomly, in terms of speed and direction. The simulation ends when the villain gets located by one of the searchers. The villain does not know that the searchers are looking for him, and does nothing to avoid them.

With the rather simple "Random Search" model, we want to demonstrate how much more scalable a DES simulation can be when compared to a time-step simulation. In "Random Search," hundreds of agents move, detect and classify each other, with the running times remaining manageable in size.

Our model uses three types of Agent classes, namely Civilian, Searcher and Villain. All Agent objects have one Sensor object extending the CCCS class, called Eyesight, and two Behavior objects: a moving Behavior (an object implementing both the Behavior and Moving interfaces) holding each Agent's MoverManager, and a classifying Behavior (an object implementing both the Behavior and Classifying interfaces) to store the classifications of detected contacts.

1. Searcher Objects

Agent objects belonging to the Searcher class are moving in a random fashion with their sole intent being the detection of at least one agent of Red affiliation. They conduct what is called a "Random Search." The Searcher objects move independently of each other. They are considered, however, to belong to a group, and therefore share a common Perception module. "Detection" events generated by any Searcher object's Eyesight are communicated to every classifying Behavior component in the group. The classifying Behavior allows the Searcher objects to classify other agents they detect by gender and by affiliation. The Searcher objects' ability to classify by gender plays no

significant role in this model. It helps us, however, to evaluate simulation components such as the StackedPropertiesMediator and the BasicClassifyingBehavior by adding one additional level of classification. It can also be useful in future implementations, involving more detailed cognitive processes in the identification of detected contacts by the Searcher objects.

2. Civilian and Villain Objects

In this implementation, the Civilian objects are used in large numbers to demonstrate our model's scalability. They move around, detect and classify other Agent objects, albeit with no particular objective. The Villain is behaving in the exact same way. He, however, has a role: to be found by the Searcher objects.

3. The Simulation

Action, in our model, takes place in a rectangular area, similar perhaps to a market area. The Civilians are generated at random locations, uniformly dispersed across the action area. Both the Civilian and the Villain objects move in random directions, with speeds uniformly distributed between 1.0 and 5.0 units. Every time they change their direction of movement, they also change their speed. Each Searcher conducts a random search, independently from the other Searchers, at a predefined speed. As soon as any Searcher detects the Villain, the simulation ends.

A Random Search for a stationary target within a specified area has an expected time of detection $E(T) = A/(V \cdot W)$, where A is the size of the searched area, W is the sensor's sweep width (twice the detection range) and S is the searcher's speed.

Now, assume that the target moves at speed U. It can be shown that in an equivalent system, where the target is stationary and only the searcher is moving, the searcher's speed will be greater than the largest of V and U. In such a system we would expect the time to detection to be somewhat smaller than $A/(V \cdot W)$ [Washburn, 1996]. The question is, how much smaller?

To answer this question, and to verify our model, we ran a small experiment, with one Searcher and one Villain, randomly located within a rectangular area. The Villain is moving randomly, at constant speed, whereas the Searcher performs a random search. We set the size of the area to 500,000 square units, the sweep width to 50, and the speeds to 5 and 100, for the Villain and the Searcher respectively. If the Villain were stationary, the expected time to detection would be 100 time units (500,000/(100·50)). We executed 100 runs. The mean time to detection was 87.67, with a standard deviation of 60.67. The 95% CI for the true mean was [75.78, 99.56], which is less than 100, as we had initially presumed.

With this at hand, we tried to model the following scenario: a Searcher object enters a rectangular area from the lower left corner, and a Villain object from the upper right. The simulation ends when the Searcher detects and classifies the Villain. Our objective is to model the simulation running time by the number of agents involved. We packed one thousand Civilian objects in a rectangular area along with the searcher and the Villain. We set the diagonal of the area at 1000.0. All agents moved at random: the Searcher at a speed of 100.0; the Villain at a speed of 5.0; and the Civilian objects at a speed uniformly distributed between 1.0 and 5.0. All agents were equipped with one sensor, with a maximum range of: 50.0 for the Searcher, and 10.0 for the Civilian objects and the Villain. All agents had the ability to detect and classify each other by gender and by affiliation. We initially ran the model 10 times to get some initial idea of the run-time requirements for a larger experiment. The average time required for a single run in our 2.0GHz 2GB RAM MacBook, was 32.03 seconds, with a standard deviation of 10.35 seconds. We believe that this is quite fast, given the size of the simulation.

We attempted to model a similar scenario in MANA. We created two squads, one of blue and one of red affiliation, with one agent each, to represent the searchers and the villain respectively. We created one hundred squads of neutral affiliation, with ten agents each, to represent the civilians in the model. The rest of the model's characteristics regarding the size of the search area, agent speeds and sensor ranges, were similar to the discrete event implementation.

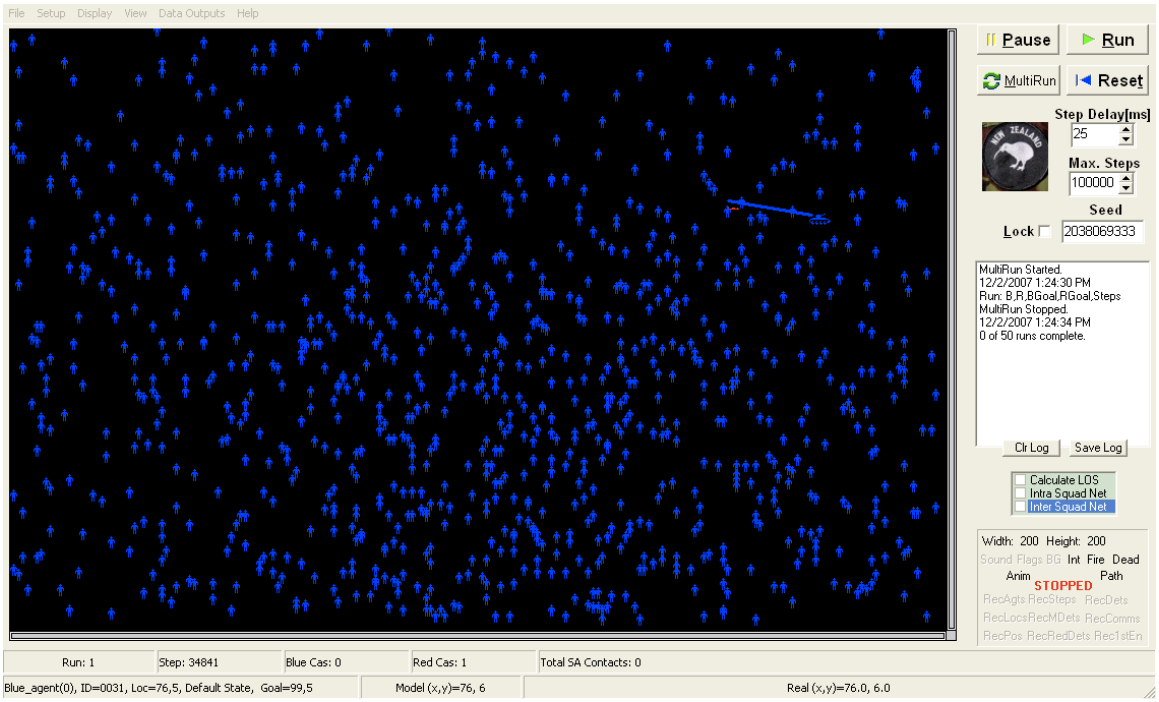


Figure 16. The Market Search Simulation in MANA

Note that MANA does not provide, in its GUI modeling tool, the possibility for the user to simulate random movement by generating random waypoints across the search area. Only user-defined waypoints can be entered through MANA's GUI. Agents in MANA move following their specific impulses and desires. We simulated random movement by removing from the agents almost all movement impulses. Thus, at each time step, agents would choose their position at the next time step in a random fashion. This resulted in a lot of jitter in their movement because it was not uncommon for an agent to return to its initial position immediately after moving away from it.

Using this scheme, we generated ten runs. The average time required for a single run ranged from 15 to 17 minutes, with an average time of sixteen minutes, approximately. The difference is impressive; one cannot be sure, however, what proportion of it should be attributed to the inefficient representation of search by random movement in our MANA model.

To see how the number of agents affects the running time, we ran the DES implementation several times, increasing the number of Civilian objects from 100 to

1000, in steps of one hundred. We generated ten runs at each case, and calculated the mean run time in milliseconds. We repeated the same procedure for the MANA model. The data were analyzed in JMP 7.0. We discovered that in the DES model the run time increased proportionally to the square of Civilian objects in the model. In MANA, however, the increase in running time was linear with the number of Civilian objects. Furthermore, the DES implementation proved to run significantly faster than MANA, throughout the range of the number of agents in the model. More specifically, the equation modeling the time required for a single run in the DES model was:

$$t_{DES} = 2380.73 - 11.41 \cdot N + 0.0392 \cdot N^2$$

where:

- t_{DES} is the time required for a single run in milliseconds, and
- N is the number of Civilian objects in the model.

The respective equation for the MANA model was:

$$t_{MANA} = 326859.62 + 616.78 \cdot N$$

where:

- t_{MANA} is the time required for a single run in milliseconds, and
- N is the number of Civilian objects in the model.

Some of the analysis details can be seen in Figures 17, 18, 19, 20.

Note that, in both cases, the intercepts represent the run time when only one searcher and the target are included in the model.

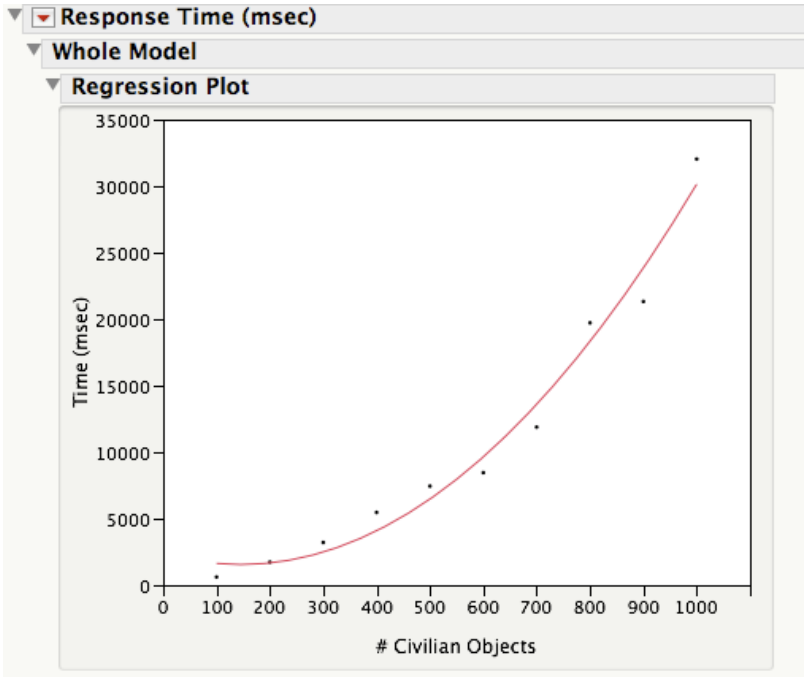


Figure 17. Running time vs. number of Civilian Objects in the DES implementation

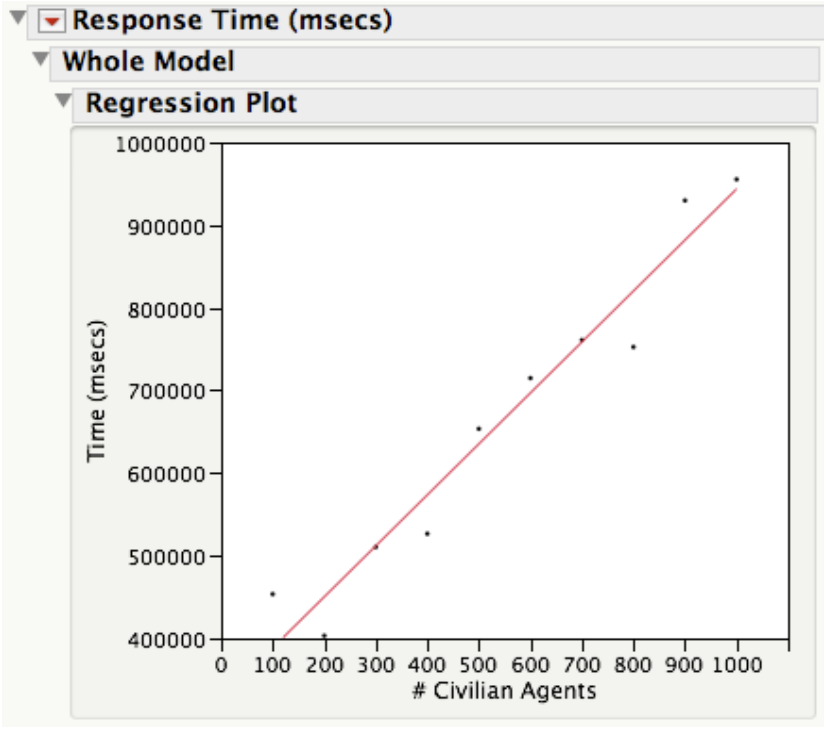


Figure 18. Running time vs. number of Civilian Objects in MANA

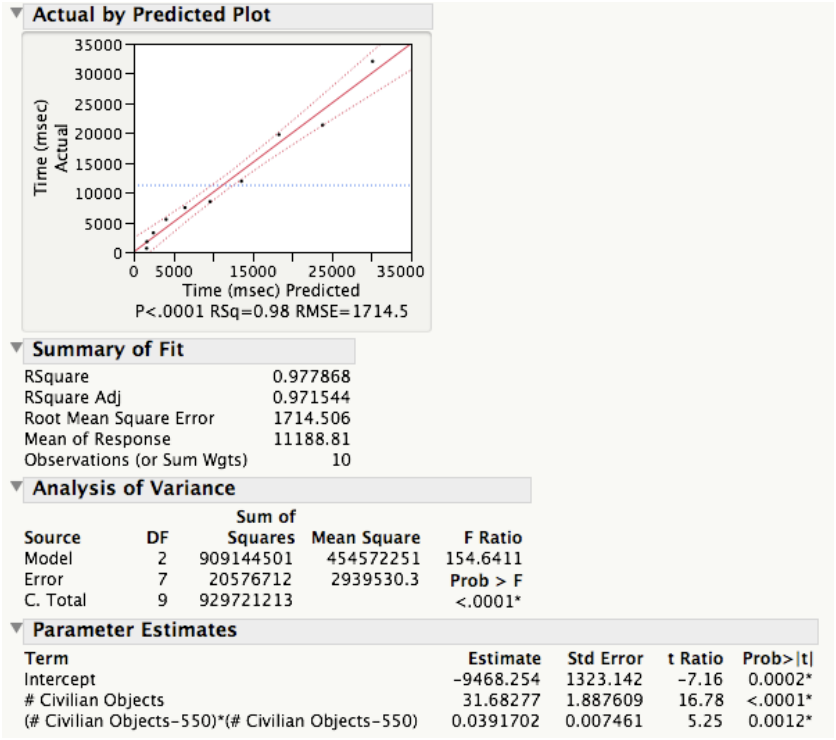


Figure 19. Analysis of running times in the DES model

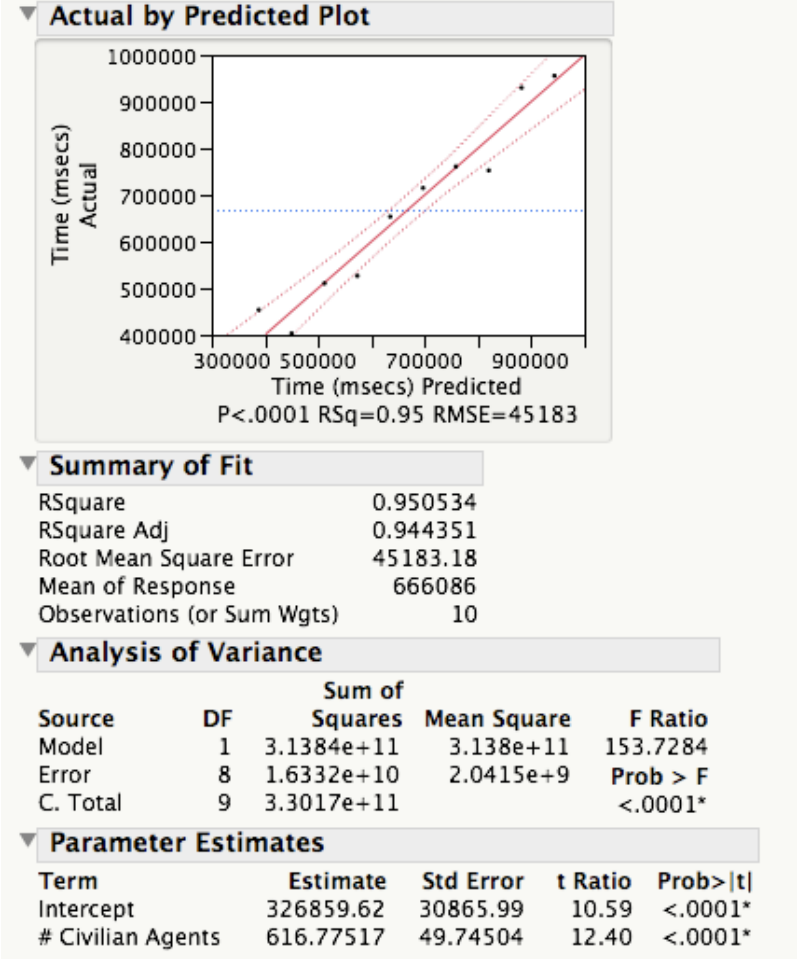


Figure 20. Analysis of running times in the MANA model

The "Random Search" model is not meant to be used as a realistic scenario, but rather as an illustration of the differences between the two modeling approaches. We created the "Random Search" model to prove two things regarding our DES implementation: the first is that it can be used for the representation and analysis of a simple Agent-Based model; the second is that it provides a faster and more scalable alternative to traditionally used time-step models, such as the excellent, in all other respects, MANA and Pythagoras. Our model is intended as a proof-of-concept prototype. It is still far from representing the rich features of the impulse-driven entities that MANA and Pythagoras introduced. Because our design ensures, however, the reusability and the extensibility of its components, we aspire to its further enhancement and development.

V. CONCLUSIONS

Come and get them

- Leonidas I, King of Sparta

A. SIMKIT AND EVENT GRAPHS

The Event Graph methodology is an efficient and intuitive way of graphically representing Discrete Event Simulations (DES). The Event Graph, the only graphical paradigm that directly models the event list logic, has no limitations in creating even the most complicated of DESs – it has been shown to be "Turing complete." More simple and extensible than other constructs used in DES, such as Petri Nets, it makes an ideal tool for creating and representing any kind of simulation model. Furthermore, its straightforward and intuitive nature allows the modeler to spend more time on model formulation, than on learning the intricacies of the paradigm [Buss, 1996].

Simkit was developed as a means of translating Event Graphs into sets of instructions for a computer to execute. Simkit is written in Java, an object-oriented language, which is popular for numerous reasons, including its portability; programs written in Java can run on almost any platform and almost any operating system. To write a DES in Simkit, however, one need only master the Java basics. Many NPS students who use Simkit for their theses, including the author, are not experienced programmers. A simulation model should not be an opaque box to the model builder or analyst. We believe, on the contrary, that it is a good practice for the analyst to be knowledgeable about the inner mechanisms and assumptions of the tool upon which his/her research is based. We think this is a strong argument in favor of using a tool such as Simkit rather than one of the plethora of off-the-shelf simulation packages available.

B. SIMKIT AND AGENT-BASED SIMULATION

Agent based simulations, such as those that are used to represent combat, have traditionally been modeled by use of the time-step approach. The time-step approach,

however convenient it may seem in the modeling of aspects like movement and sensing, is associated with many problems and limitations. DES is free of these problems and limitations. Modelers, however, have overlooked the potential to implement the DES paradigm in the creation of ABSs. Contrary to the popular belief that DES cannot easily model movement and sensing, work by Buss and Sánchez [2005] has demonstrated that such a representation is not only feasible, but also desirable from several standpoints.

The most important element of a combat-oriented ABS is the agent itself, and movement and sensing are the agent's most fundamental modeling aspects. Simkit provides the tools for designing entities that move and sense. We tried to complement these aspects of Simkit by creating a versatile and extensible framework upon which new agent behaviors and capabilities could be built. We also built a small-scale model, which we used to conduct a simple experiment to demonstrate the functionality of our components.

Our simple search model demonstrates that DES can be used to create models that include thousands of agents. Furthermore, the use of DES yields much shorter run times. A typical single simulation run, including approximately 1000 agents packed in a relatively small area, moving randomly, detecting and classifying each other, would take less than one minute to complete on a 2.0 GHz, 2 GB RAM MacBook.

C. PROBLEMS

Several iterations of Agent design were tried and discarded before the current design was settled on. Along the way, we encountered difficulties in integrating our model with existing Simkit components.

One of the difficulties had to do with the integration of Agent objects with older Simkit classes. More specifically, classes like the PathMoverManager or the SensorTargetReferee did not work correctly with an instance of Agent (instead of a Mover object, such as the UniformLinearMover), despite the fact that the Agent interface extends the Mover interface, i.e., an Agent is always a Mover as well. As a workaround, we used the Agent's Mover instance variable where appropriate. Some of these classes, such as certain instances of Mediator, may need a reference to the Agent object to which

the UniformLinearMover belongs, in order to perform some of their basic functions. For these classes, we store the Agent object as an Added Property to the UniformLinearMover, under the name "myAgent." This operation takes place whenever an AgentBase object is created and is "invisible" to the user.

Another problem that we encountered while experimenting with the Sensors' maximum ranges was that, sometimes, "Detection" events would not occur at all. This was happening when the maximum Range of the Searcher's Sensor was chosen to be too large. We found the source of the problem to be in the design of the Sensor Target Referee. For targets (instances of Mover) that are instantiated within a Sensor's footprint, the Sensor Target Referee will schedule an "Exit Range" event but not an "Enter Range." As a result, no "Detection" or "Classification" events will be scheduled either. This can create missed detections when agents are initially generated within each other's detecting range.

D. LIMITATIONS/FURTHER RESEARCH

Our model lacks many characteristics that would enable it to even remotely represent a modern battlefield environment. Each one of these characteristics could be an object of further research.

Our agents, for instance, are dimensionless entities that move in a 2D environment. As a result, an arbitrarily large number of agents can be stacked in one location. Another serious limitation is that they can only detect and interact with entities of similar nature. Our agents cannot currently detect geometrical shapes, cannot hide behind opaque walls, and are not blocked by impenetrable obstacles while moving.

The simple entities we created obey simple rules: "if you classify the target as enemy, shoot immediately." Producing the multitude of outcomes that models like MANA and Pythagoras generate will require substantial development of state-dependent behavior rules. The addition of Behavior components representing the agents' impulses and desires would bring them one step closer to behaving like intelligent, autonomous entities.

E. EPILOGUE

Our model is a first attempt to use Java programming and existing Simkit components to create Agent-Based Simulations within the Discrete Event paradigm. Despite the numerous off-the-shelf simulation packages that are available, we strongly advocate the creation of a library consisting of ABS components, developed and maintained by users. This thesis prototypes an architectural design which is generalizable, reusable, and extensible. We have created an initial set of model elements that demonstrate the approach we are advocating, and anticipate future growth and enrichment of these components. Although there is a long distance to be covered before our agents can be a part of large-scale, combat-oriented simulations, incremental improvements should be able to eventually bridge this distance. Little by little does the trick.

APPENDIX

A. THE AGENT INTERFACE

```
package myEntities;

import java.util.Map;
import java.util.Set;
import myBehaviors.Behavior;
import myBehaviors.Perception;
import simkit.SimEntity;
import simkit.smdx.Mover;
import simkit.smdx.Sensor;

/**
 * The Agent interface implements Russell and Norvig's definition of an
 * intelligent agent: "An agent is anything that can be viewed as perceiving its
 * environment through sensors, and acting upon the environment through
 * actuators." The Agent interface enforces setters and getters for instances of
 * the Sensor, Behavior and Actuator interfaces. The instance of Perception,
 * whose getters and setters are enforced, can be used as a hub, to which all
 * Sensors and Behaviors are attached. One common Perception may be shared by
 * many agents, thus establishing a "collective" perception among them. The
 * Agent instance is designed to "reflect" all the Mover's public methods and
 * properties.
 *
 * @author Alexandros Matspopoulos
 */
public interface Agent extends SimEntity, Mover {

        /**
         *
         * @return the Agent's Mover
         */
        public Mover getMover();

        /**
         * Sets the Agent's Mover.
         *
         * @param mover
         */
        public void setMover(Mover mover);

        /**
         *
         * @return the Agent's Perception.
         */
        public Perception getPerception();

        /**
         * Sets the Agent's Perception.
         *
         * @param perception
         */
        public void setPerception(Perception perception);

        /**
         *
         * @return the Agent's set of Sensor instances.
         */
        public Set<Sensor> getSensors();

        /**
         * Sets the Agent's Set of Sensor instances.
         *
         * @param sensors
         */
        public void setSensors(Set<Sensor> sensors);

        /**
```

```
     * Adds a Sensor instance to the Agent.
     *
     * @param sensor
     */
    public void addSensor(Sensor sensor);

    /**
     * Removes the corresponding Sensor instance from the Agent.
     *
     * @param sensor
     */
    public void removeSensor(Sensor sensor);

    /**
     *
     * @return the Agent's set of Behavior instances.
     */
    public Set<Behavior> getBehaviors();

    /**
     * Sets the Agent's Set of Behavior instances.
     *
     * @param behaviors
     */
    public void setBehaviors(Set<Behavior> behaviors);

    /**
     * Adds a Behavior instance to the Agent.
     *
     * @param behavior
     */
    public void addBehavior(Behavior behavior);

    /**
     * Removes the corresponding Behavior instance from the Agent.
     *
     * @param behavior
     */
    public void removeBehavior(Behavior behavior);

    /**
     *
     * @return a mapping of the Agent's Actuator instances to the Behavior
     *         instances that control them.
     */
    public Map<Actuator, Behavior> getActuators();

    /**
     * Sets the Agent's Set of Actuator instances.
     *
     * @param actuators
     */
    public void setActuators(Map<Actuator, Behavior> actuators);

    /**
     * Maps a new Actuator instance to the Agent's Behavior instance that will
     * control the Actuator.
     *
     * @param actuator
     * @param behavior
     */
    public void addActuator(Actuator actuator, Behavior behavior);

    /**
     * Removes the corresponding Actuator instance from the Agent.
     *
     * @param actuator
     */
    public void removeActuator(Actuator actuator);
}
```

B. THE BEHAVIOR INTERFACE

```
package myBehaviors;

import java.beans.PropertyChangeListener;

import simkit.SimEntity;

/**
 * Instances of Behavior are designed to be attached to an Agent via a
 * Perception instance. The setter and getter methods that this interface
 * enforces are for a String that serves as a unique identifier for the Agent
 * instance to which the Behavior belongs.
 *
 * @author Alexandros Matsopoulos
 *
 */
public interface Behavior extends SimEntity, PropertyChangeListener {

        /**
         *
         * @return the String serving as a unique identifier of the Agent instance to
         *         which the Behavior belongs.
         */
        public String getBeholder();

        /**
         * Sets a String serving as a unique identifier of the Agent instance to
         * which the Behavior belongs.
         *
         * @param beholder
         */
        public void setBeholder(String beholder);

}
```

C. THE PERCEPTION INTERFACE

```
package myBehaviors;

import java.beans.PropertyChangeListener;
import java.util.Set;

import simkit.SimEntity;
import simkit.SimEventSource;
import simkit.smdx.Moveable;

/**
 * The Perception component has been designed to mainly function as a hub, to
 * which an agent's Behavior and Sensor modules are connected. Perception is
 * connected to the agent's Sensor objects as an "Event Listener". It "listens"
 * to all events triggered by the Sensor objects, like "Detection",
 * "Undetection" and "Classification". These events carry, as arguments,
 * relevant information collected by the Sensor. As soon as the Perception
 * component "hears" one of these events, it redistributes the information to
 * the agent's Behavior objects, through an appropriate Property Change Event.
 * Perception also allows the communication between the agent's Behavior
 * objects. Behavior objects have a reciprocal "Property Change Listener"
 * relationship with the Perception component to which they are connected. The
 * Perception component listens to, and rebroadcasts, all Property Change events
 * fired by the Behavior objects to which it is attached.
 *
 * Every agent may have only one Perception component. Many agents, however, may
 * share the same Perception component. This scheme may be utilized to represent
 * the sharing of information within a group of agents. Such a group can be
 * defined by the sharing of a common Perception object. Because a shared
 * Perception object will be an Event Listener to the Sensor objects of all
 * agents belonging to the same group, any events scheduled by an agent's Sensor
 * will be communicated to (the Behavior objects of) all the agents belonging to
 * the group.
 *
 * @author Alexandros Matsopoulos
 *
 */
public interface Perception extends SimEntity, PropertyChangeListener {

        /**
         *
         * @return the objects to which the Perception instance is registered as an
         *         Event Listener.
         */
        public Set<SimEventSource> getSimEventSources();

        /**
         * Adds a SimEventSource to the Perception instance.
         *
         * @param source
         */
        public void addSimEventSource(SimEventSource source);

        /**
         * Removes the corresponding SimEventSource from the Percpetion instance.
         *
         * @param source
         */
        public void removeSimEventSource(SimEventSource source);

        /**
         * Method to be "heard" by a registered Sensor instance.
         *
         * @param contact
         */
        public void doDetection(Moveable contact);

        /**
         * Method to be "heard" by a registered Sensor instance.
         *
         * @param contact
         */
        public void doUndetection(Moveable contact);
```

```
        /**
         * Each String in the Set that this method returns is designed to denote an
         * Agent instance to which the Perception belongs.
         *
         * @return
         */
        public Set<String> getBeholders();

        /**
         * Sets a Set of Strings, designed to denote an Agent instance to which the
         * Perception belongs.
         *
         * @param beholders
         */
        public void setBeholders(Set<String> beholders);

        /**
         * Adds a String, designed to denote an Agent instance to which the
         * Perception belongs.
         *
         * @param beholder
         */
        public void addBeholder(String beholder);

}
```

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

Banks, J., J.S. Carson II and B.L. Nelson. 1996. *Discrete-Event System Simulation*. Upper Saddle River, NJ: Prentice-Hall, Inc.

Bitinas, E.J., Z. Henscheid and L.V.Truong. 2003. Pythagoras: A New Agent-based Simulation System. Technology Review Journal, http://www.ms.northropgrumman.com/PDFs/TRJ/TRJ-2003/SS/03SS_Bitinas.pdf (accessed November 2007).

Brutzman, D., A. Buss, C. Blais and CAPT S. Starr King USN. 2004. Connecting Simkit Discrete Event Simulation (DES) and the Naval Simulation System (NSS) via Web Services for Extensible Modeling & Simulation (XMSF)-Capable Analysis. Modeling Virtual Environments & Simulation (MOVES) Institute, Naval Postgraduate School, Monterey California, http://www.movesinstitute.org/xmsf/projects/WCM/WorldClassModelingProject OverviewNpsMOVES.2004April19.pdf (accessed November 2007).

Buss, A. 1996. Modeling With Event Graphs. Proceedings of the 1996 Winter Simulation Conference. ieeexplore.ieee.org/iel5/6969/18768/00873273.pdf (accessed November 2007).

Buss, A. 2001. Basic Event Graph Modeling. Simulation News Europe 31:1- 6.

Buss, H. and P. Sánchez. 2002. Building Complex Models With LEGOS. Proceedings of the 2002 Winter Simulation Conference, http://www.informs-sim.org/wsc02papers/094.pdf (accessed November 2007).

Dewar, J.A., J.J. Gillogly and M.L. Juncosa. 1996. Non-Monotonicity, Chaos and Combat Models. *Military Operations Research* 2 (2): 37-49.

Ferber, J. 1999. Multi-Agent Systems: An Introduction to Distributed Artificial Intelligence. London: Addison-Wesley.

Henscheid Z., D. Middleton and E. Bitinas. 2006. Pythagoras: An Agent-Based Simulation Environment. Proceedings and Bulletin of the International Data Farming Community, http://harvest.nps.edu/scythe/Issue1/scythe_1_1.pdf (accessed November 2007).

Law, A. 2007. *Simulation Modeling and Analysis*. New York: McGraw-Hill.

Law, A. and D.W. Kelton. 1982. Simulation Modeling and Analysis, New York: McGraw-Hill.

Leemis L.M. and S.K. Park. 2006. *Discrete-Event Simulation – A first Course*. Upper Saddle River, NJ: Pearson Education, Inc.

McIntosh, G., D.P. Galligan, M.A. Anderson and M.K. Lauren. 2006. *MANA: Map Aware Non-uniform Automata Version 4.0 Users Manual.* Devonport, Auckland: Defence Technology Agency. (Draft version).

Rocha, Luis. 1999. From Artificial Life to Semiotic Agent Models. Review and Research Directions. Los Alamos National Laboratory, http://lib-www.lanl.gov/la-pubs/00460075.pdf (accessed November 2007).

Washburn, A.R. 2002. *Search and Detection, 4th ed*. Linthicum, MD: MAS of INFORMS

Watson, H.J. and J.H. Blackstone, Jr. 1989. *Computer Simulation*. New York: Wiley.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Dr. Paul J. Sánchez
   Naval Postgraduate School
   Monterey, California

4. Dr. Arnold H. Buss
   Naval Postgraduate School
   Monterey, California