Testability of a Swarm Robot
Using a System of Systems Approach
and Discrete Event Simulation

by

Matthew R. Hosking

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of
Master of Science
in Computer Engineering

Supervised by

Associate Professor Dr. Ferat Sahin
Department of Electrical Engineering
Kate Gleason College of Engineering
Rochester Institute of Technology
Rochester, New York
June 2009

Approved by:

Dr. Ferat Sahin, Associate Professor
Thesis Advisor, Department of Electrical Engineering

Dr. Shanchieh Jay Yang, Associate Professor
Committee Member, Department of Computer Engineering

Dr. Andreas Savakis, Professor and Department Head
Committee Member, Department of Computer Engineering
Title:

Testability of a Swarm Robot Using a System of Systems Approach and Discrete Event Simulation

I, Matthew R. Hosking, hereby grant permission to the Wallace Memorial Library to reproduce my thesis in whole or part.

Matthew R. Hosking

Date
Dedication

To my friends and family.

Of making many books there is no end,
and much study wearies the body.
Now all has been heard;
here is the conclusion of the matter:
Fear God and keep his commandments,
for this is the whole duty of man.
Acknowledgments

I am grateful for my advisor’s continual direction in completing this work as well as the friendship developed over the past year. I also appreciate my committee, Dr. Yang and Dr. Savakis, for their review and insights into this thesis. I acknowledge my colleagues in the lab, Ryan and Eyup, for helping me and providing the opportunities to enjoy engineering together. Finally, I owe many thanks to my brothers, sisters, parents, friends, and family for remaining patient with my schedule and for eagerly encouraging me to persevere.
Abstract

Testability of a Swarm Robot
Using a System of Systems Approach
and Discrete Event Simulation

Matthew R. Hosking

Supervising Professor: Dr. Ferat Sahin

A simulation framework using discrete event system specification (DEVS) and data encoded with Extensible Markup Language (XML) is presented to support agent-in-the-loop (AIL) simulations for large, complex, and distributed systems. A System of Systems (SoS) approach organizes the complex systems hierarchically. AIL simulations provide a necessary step in maintaining model continuity methods to achieve a greater degree of accuracy in systems analysis. The proposed SoS approach enables the simulation and analysis of these independent and cooperative systems by concentrating on the data transferred among systems to achieve interoperability instead of requiring the software modeling of global state spaces. The information exchanged is wrapped in XML to facilitate system integration and interoperability. A Groundscout is deployed as a real agent working cooperatively with virtual agents to form a robotic swarm in an example threat detection scenario. This scenario demonstrates the AIL framework’s ability to successfully test a swarm robot for individual performance and swarm behavior. Results of the testing process show an increase of robot team size increases the rate of successfully investigating a threat while critical violations of the algorithm remained low despite packet loss.
# Contents

Dedication ...................................................................................................................... iv

Acknowledgments ........................................................................................................... v

Abstract ........................................................................................................................... vi

Glossary ............................................................................................................................. xiv

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

2 Background Review ....................................................................................................... 6  
   2.1 Cooperative Robotics ................................................................................................. 6  
   2.2 Modeling and Simulation (M&S) ............................................................................... 10  
      2.2.1 Discrete Event System Specification (DEVS) ................................................... 12  
      2.2.2 DEVSJAVA ........................................................................................................ 15  
   2.3 System of Systems (SoS) ......................................................................................... 16  
      2.3.1 Integration Property ............................................................................................ 17  
      2.3.2 Interoperability Property .................................................................................... 18  
   2.4 Extensible Markup Language .................................................................................... 19

3 State of the Art Simulation Tools .................................................................................... 23  
   3.1 Simulators ................................................................................................................ 23  
      3.1.1 Robotics Simulators ............................................................................................. 24  
      3.1.2 HIL Support ........................................................................................................ 25  
   3.2 SoS and DEVS Applied to Net-Centric Design .......................................................... 27  
      3.2.1 DEVS/Service-oriented Architecture(SOA) ..................................................... 28  
      3.2.2 DEVS/HLA ........................................................................................................ 28  
   3.3 Testing Methodology .................................................................................................. 29  
      3.3.1 Formal Methods .................................................................................................. 30  
      3.3.2 Empirical Analysis ............................................................................................. 31
# Discrete Event Based SoS Simulation Framework

4 Initial Framework Support ........................................ 34
4.2 XML Architecture .................................................. 37
4.3 Interfacing Two Worlds .......................................... 40
4.3.1 Communication .................................................... 42
4.3.2 Synchronization .................................................. 43
4.4 Analysis of Systems ................................................ 43
4.4.1 Performance of A Single Robot .............................. 44
4.4.2 Evaluating Emergent Behavior in SoS ..................... 44
4.4.3 Measurement Methodology ...................................... 45

# Robust Threat Detection Example

5 Modeling the Other Systems ....................................... 47
5.1 Threat ................................................................... 47
5.2 Radar Stations (Sensors) .......................................... 48
5.3 Command Center .................................................... 49
5.4 Virtual Environment ................................................ 50
5.2 Mobile Agent Model ................................................. 52
5.2.1 Communication Layer ......................................... 53
5.2.2 Control Layer (Synchronization) .......................... 56
5.3 Simulations in Model Continuity Process .................... 60
5.3.1 Simulation Experiment ........................................... 61
5.3.2 Agent-in-the-Loop Simulation ............................... 63

# Testability Results & Discussion

6 Aggregating Data for Analysis .................................... 72
6.2 Measuring the Communication Statistics ..................... 75
6.3 Measuring the Response to a Threat ......................... 78
6.4 Measuring the Swarm Behavior ................................. 80
6.5 Scalability ........................................................... 84

# Conclusion

7 Summary .................................................................. 87
7.2 Future Work ........................................................ 89
7.2.1 AIL Framework .................................................. 89
7.2.2 Applications ...................................................... 90

Bibliography ................................................................ 91
List of Tables

4.1  XML Tag Translation .................................................. 40
4.2  Summary of XML Identifiers ................................. 41
6.1  Events and Statistics in Analysis Output file ................. 74
List of Figures

2.1 A photo of assembled Groundscout ........................................ 8
2.2 Levels of Conceptual Interoperability Model (LCIM) .................. 20
3.1 Abstraction Layers in DEVS Net-centric Approach .................... 27
4.1 AIL Simulation Framework Abstraction View ............................. 34
4.2 Correlation between an SoS and DEVS Models ......................... 36
4.3 Interaction between atomic model and activity ....................... 37
4.4 An XML based SoS architecture ............................................ 38
4.5 An XML based SoS architecture implementation ....................... 40
4.6 Supervisor Functionality .................................................... 45
5.1 DEVS Phase Diagram of Threat Model ................................... 48
5.2 Radar Station (Sensor) Model ............................................. 49
5.3 DEVS Block Diagram of Command Center Model ....................... 50
5.4 DEVS Block Diagram of Virtual Environment Model ................. 51
5.5 DEVS model of gsRobot (Groundscout) ................................ 53
5.6 Layers of AIL Communication Plug-in ................................... 54
5.7 DEVS model of communication layer ..................................... 55
5.8 Data flow in DEVS Model .................................................. 57
5.9 DEVS model of control layer .............................................. 58
5.10 Threat Detection Control Unit ............................................ 59
5.11 Synchronization Control Unit ............................................. 60
5.12 Graphics used in virtual and AIL simulations ......................... 61
5.13 Initial positions of agents in the SoS .................................. 62
5.14 Scout helicopters intercept enemy tank ................................. 62
5.15 Initial position of SoS (AIL Simulation) ................................ 64
5.16 Real agent investigates threat (AIL Simulation) ....................... 66
5.17 Virtual agent investigates threat (AIL Simulation) ................... 67
5.18 Groundscout in lab (AIL Simulation) .................................... 68
5.19 Larger Swarm Cooperating (AIL Simulation) ......................... 69
5.20 Formation Violations during AIL simulations ......................... 70
6.1 Analyzing the SoS: Input and Output Files ........................................ 73
6.2 Initial positions of robots in threat detection scenario ...................... 75
6.3 Transmitted Packets Dropped per Minute ........................................ 77
6.4 Percentage of Dropped Transmitted Packets vs. Swarm Size ............... 78
6.5 Percentage of Successful Threat Investigations vs. Swarm Size .......... 79
6.6 Average Response Time to Investigate a Threat ............................... 80
6.7 Formation Violations per Minute vs. Swarm Size ............................. 81
6.8 Formation Violations As Percentage of All Moves ............................. 81
6.9 Average Distance between Robots during a Formation Violation ......... 82
6.10 Critical Violations per Minute vs. Swarm Size ................................. 83
6.11 Critical Violations As Percentage of All Moves ............................... 84
6.12 Critical Violations As Percentage of All Violations .......................... 84
List of Listings

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>XmlStrEntity class declaration</td>
<td>39</td>
</tr>
<tr>
<td>5.1</td>
<td>XML Message from Radar to Command Center</td>
<td>65</td>
</tr>
<tr>
<td>5.2</td>
<td>XML Message from Command Center to Robots</td>
<td>65</td>
</tr>
<tr>
<td>5.3</td>
<td>XML Message from Robot to Robot</td>
<td>65</td>
</tr>
<tr>
<td>5.4</td>
<td>XML Message from Radar to Command Center</td>
<td>67</td>
</tr>
<tr>
<td>5.5</td>
<td>XML Message from Command Center to Robots</td>
<td>68</td>
</tr>
</tbody>
</table>
Glossary

**ACIMS** Arizona Center for Integrated Modeling and Simulation.

**AIL** agent-in-the-loop.

**AIS** Artificial Immune System.

**DDMS** Department of Defense Discovery Metadata Specification.

**DEV** Discrete Event System Specification.

**DEVSJAVA** A Java implementation of the DEVS formalism used as a simulator for models. Developed by ACIMS.

**DEVSML** DEVS Modeling Language.

**DoD** Department of Defense (United States).

**GIG** Global Information Grid.

**GPS** Global Positioning System.

**Groundscout** A modular micro robot for swarm intelligence and cooperative robotics research and applications. Developed by Dr. Ferat Sahin at RIT.

**HIL** hardware-in-the-loop.

**HLA** High Level Architecture.
IDE Integrated Development Environment.

IEEE Institute of Electrical and Electronics Engineers.

Integration process of combining multiple entities to form a larger component.

Interoperability a property referring to the ability of multiple entities to work together to achieve a goal.

LCIM Levels of Conceptual Interoperability Model as proposed by Dr. Andreas Tolk.

Liveness As presented by Dr. Winfield, a characteristic of swarm behavior in which the swarm is exhibiting desirable behaviors.

MAC Medium Access Control.

Model continuity a seamless migration from initial experimental simulations to deployed systems in the field.

Net-centric participating as a part of a continuously-evolving, complex community of devices, information and services interconnected by a communications network.

PIC Programmable Intelligent Computer - a family of micro-computer chips developed and manufactured by MicroChip, Inc.

RAM Random Access Memory.

RF Radio Frequency.

RIL robot-in-the-loop.
RIT Rochester Institute of Technology.

RS232 Recommended Standard 232 is an Electronic Industries Association standard for asynchronous serial binary data signals.

RX abbreviation for “receive” in the communications field.

safety As presented by Dr. Winfield, a characteristic of swarm behavior in which the swarm is not exhibiting undesirable behaviors.

SDK Software Development Kit.

SOA Service Oriented Architecture.

SoS Systems of Systems or System of Systems.


TDMA Time Division Multiple Access.

TX abbreviation for “transmit” in the communications field.

UAV Unmanned Aerial Vehicle.

XML Extensible Markup Language.

XPath XML Path Language.

XSLT Extensible Stylesheet Language Transformation.
Chapter 1

Introduction

Systems of systems (SoS) are comprised of systems which themselves are independent and complex systems that interact among each other to achieve a common goal. This goal requires a functionality greater than the functionality offered by any individual member of the SoS. For example, a Boeing 747 airplane is a system of an SoS, but an airport is an SoS. The goal of the airport is to check in passengers and luggage as well as provide security checks in addition to transporting them to another airport. The SoS concept is still in the developing stages and several formal definitions are available [1, 2]. For this work we considered the following definition: SoS are large-scale concurrent and distributed systems that are comprised of complex systems working towards a larger, common goal. This is an information systems view as it emphasizes the interoperability and integration properties of an SoS [3].

Considering a military operations example, it is obvious that many different ground, air, sea, and space units contribute data to an SoS. Information may be simple sensor data or it may be complex data from an aircraft carrier. Command centers attempt to aggregate data and inform their subsystems to accomplish many goals of the military. Each of the systems could be developed at different times and with different hardware, software, available bandwidth, and data format
requirements. These stovepipe systems were created by vertical integration techniques which connected them functionally but without the flexibility of reuse or concern of future growth. This creates a barrier to data aggregation and meeting goals of the SoS as the systems cannot interact and communicate with new systems that become available unless those systems are coupled directly to the present ones. The sources of different data and the detriments to the interoperability requirements introduced by stovepipe systems are discussed in [4].

One solution to achieve interoperability is to standardize the communication medium among the systems. Interoperability is discussed in more detail in Section 2.3 but conveys the ability of multiple systems to work seamlessly towards a goal. Two possible methods to standardize communication are [5]:

- Create a software model: each component in the SoS talks to the module embedded in itself

- Describe the data in a common language: each component in the SoS can understand and parse data from another system

The first approach creates a common interface among the systems through an additional software layer embedded in the systems. There is the potential for large overhead in generating software models. Each new member of the SoS would require the regeneration of software models and this approach also assumes, incorrectly, that the complete software model is available or practical to describe. In light of the difficulties of software models, this work promotes the use of a common language to describe the data. A common set of Extensible Markup Language (XML) tags are employed to standardize the information among the systems in the SoS. A
data-driven approach avoids the risk of potentially large overhead and more readily supports legacy components which may not have a software model available [6]. A common understanding of the data exchanged among the members of the SoS can also be captured and processed later to determine the degree of success which the SoS completes its goal.

In a multi-agent system, the agents operate autonomously but it is important they cooperate with other agents to take better actions for the overall goal of the SoS. Interoperability in this work is achieved when each system exchanges data according to an XML standard commonly understood in the SoS. A system correctly communicates within the SoS if other systems can receive, parse, and interpret the transmitted data as the sender intended.

The integration property implies that systems can connect and interact with the SoS components regardless of their hardware and software characteristics, operating systems, and internal data format. Integration permits a dynamic SoS in which systems may join or leave the SoS at any time. Other systems need to be aware of this change to effectively and efficiently use the available resources in the SoS to meet the larger goal.

Disparities can arise when transitioning from a simulated environment to the implementation of the control on actual hardware. In addition, unseen problems or over simplified assumptions in the initial modeling of the system can result in unexpected results during the implementation phase. It is the implementation phase which requires more time and money than the simulation phase. This phase also offers less flexibility to change systems or test new algorithms and sensors without
rebuilding a whole component. As the simulation model provides data increasingly close the observed implementation data, future test results can be expected with greater accuracy.

A seamless migration from initial experimental simulations to deployed systems in the field is referred to in general as *model continuity*. A simple four step process describing model continuity is found in [7]: conventional simulation, real-time simulation, hardware-in-the-loop (HIL) simulation, and implementation. A HIL simulation step allows part of the final system to be implemented and debugged before more resources are directed towards a flawed product.

The goal of this work is not to research new swarm intelligence algorithms but rather create a viable platform that is robust enough to aid in the development and testing of these algorithms and systems. AIL simulation also enables the examination of run time issues that may not be foreseeable in theoretical considerations.

This thesis has resulted in multiple conference publications: the initial XML communications framework [8], an extension to include hardware-in-the-loop simulation [9], and a more focused look on the framework along with initial results [10].

An outline of the thesis document follows. Chapter 2 gives a review of cooperative robotics, modeling and simulation concepts, the systems of systems approach, and information on XML. Current simulators, networked SoS abilities, and testing methodology of swarm robotics are presented in Chapter 3. The conceptual framework for agent-in-the-loop simulations this work contributes to the ongoing research is found in Chapter 4. The framework is exemplified through a robust threat detection example in Chapter 5. That chapter also includes detailed implementations
of the robot as a general application of the framework’s ability and walks through the simulation stages of a threat detection. The simulations provide the means to test a Groundscout. Chapter 6 discusses these results and demonstrates the ability of this work’s AIL framework to test a swarm robot. Chapter 7 contains final remarks and direction for continued research.
Chapter 2

Background Review

This chapter begins with an overview of cooperative robotics including the complexities that arise from their control algorithms. Modeling and simulation is often used to examine and verify algorithms and is also studied in this chapter. Some focus is given to discrete event systems simulation and DEVSJAVA before discussing the growing field of Systems of Systems. Systems of systems is an approach to address the issues arising in large interconnected systems. The abilities of XML used in this work to aide in the simulation process ends the chapter.

2.1 Cooperative Robotics

Robots have become smaller and more cost effective over the past few years and are now being deployed in cooperative teams. A team may be homogeneous with multiple instances of the same robot or the team may be heterogeneous and made up of robots with different characteristics. Research into cooperative robotics requires multiple functioning robots. The research and development time to design and test a robot for production usually comes at a high price.

With growing interest, however, comes many alternatives. In much the same manner as computer hardware is widely available to create custom systems at a
lower cost than proprietary ones, efforts have been documented which utilize lower
cost, off the shelf components to create a community of robots [11]. There are also
commercially available miniature robot platforms, such as the Khepera robot, used
in swarm robotics research [12].

Another important aspect for a robotic platform used in cooperative research is the
flexibility to be changed, updated, or expanded with new functionality. To this end,
the Groundscout modular micro robotic platform was developed at RIT [13]. These
robots place each functional resource on a different modular layer that connects to a
common bus. The addition or removal of sensors involves connecting or disconnecting
a layer from the robot. Ongoing research may require more or less functionality,
but a new robot design can be avoided with this modular approach. Groundscout
robots exhibit the following features: small footprint, differential drive mobility layer,
wireless communication, ultrasonic sensors, infrared emitters and receivers, proximity
sensors, GPS, and high level programming support.

A microprocessor and auxiliary PIC module for the wireless MAC protocol allow
freedom in deploying control and communication algorithms to the platform. With
cooperative robot teams, these algorithms must take into account the interaction and
functionality of the group in addition to local navigation and task procedures.

Robot teams are dynamic and complex systems that are difficult to fully define
due to changing environments and the distribution of each member of the team.
In addition to the typical constraints imposed by obstacle avoidance and path
planning/following approaches, a robot team may share global information about the
team’s navigation and task completion strategies. Formation control of robot team
formations can vary as much as the imagination; a thorough overview of general considerations for formation control and different situations is presented in [14]. Graph theory is applied as an analysis tool to create mathematical models and examine stability in groups [14, 15]. The goal of any approach arises from the need to know a vital set of information: the positions of the robots.

A leader-follower control approach appoints a robot to be the team leader (or perhaps two such robots) and the remaining robots attempt to follow the leader at specified distances and orientations. A single file line formation is such an example that may use this type of algorithm. Another approach to controlling a team is to use distributed control methods. That is, there is no distinct leader and local information is exchanged among the members to reach a consensus on the global state of the team. Ren has produced several papers researching the benefits and considerations of these techniques [16, 17, 18, 19]. A generalized coordinates approach using parameters to
specify the goal formation is discussed in [20] and uses a distributed algorithm.

A second type of distributed control where no leader exists is referred to in literature as behavioral methods or emergent control. There exists early work with behavioral methods and examples of different techniques and static formations applied to autonomous vehicles for the military [21]. In a behavioral-based formation control each robot follows a set of rules, or behavioral constraints, related to its current role in the team. Flocking, schooling, and herds are all examples of emergent behavior in nature. In these situations, each member operates in an almost independent manner. The goal is achieved only when the team is viewed as a whole. Robot swarms designed to mimic this natural phenomenon have been shown to be successful in different applications. Ant colony swarm intelligence [22] and an artificial immune system (AIS) approach [23] have been applied to mine detection.

Swarm intelligence is well fitted to threat detection schemes or surveillance of an area. The robot teams have loose requirements for communication and cooperation while scanning for land mines, chemical spills, or any other disaster that is being monitored. When the threat or spill is detected and the robot team is also designed to contain the threat, cooperation shifts to a formation based control algorithm. Thus, as the robot team becomes more autonomous and robust in its actions and dynamic goals, the complexity increases and different control approaches may be more well suited than others for a given state of the system.

Simulation can be used to evaluate the performance of a formation control or swarm algorithm. Multiple approaches can be combined in multiple systems if needed. Deploying robotic systems in a dynamic environment with unpredictable agents
and evolving team goals is well suited to be better understood through simulation. Some robots may achieve increased performance by changing the control algorithm deployed. Much work has been done on control algorithms and it is important to have a suitable modeling and simulation framework to test the various approaches.

2.2 Modeling and Simulation (M&S)

Modeling a system entails describing a real system in some way as to generate data similar to the real world. Simulation is the process of executing the descriptions of the system to generate the data given some specific set of inputs. An experimental frame defines and bounds the set of inputs, or circumstances, for which the model is a valid representation of the real system. A computational approach to evaluating systems and models is provided as well. *Theory of Modeling and Simulation*, authored by Dr. Ziegler, is a classic text in this field [24]. The DEVS formalism invented by Ziegler, as well as a Java based simulator, are reviewed later in this section.

Growing complexities in interactions of these robots working together turns research towards a software simulation based approach. It is difficult if not impossible to capture the many unknowns and couplings between different algorithms, scenarios, and platforms in a single analytical model. Simulations save time and costs associated with the resources that would otherwise be required to prototype, build, and perform tests on actual devices. Without simulation, data is only available for feedback and analysis after the system is implemented. Simulations allow insight into the operation of the system, even if it is not 100% accurate, before it is built and deployed. Proposed design revisions and different optimization attempts are also easily examined in a
The dynamic complexity present in cooperative robotics places additional value on using modeling and simulation practices to understand the system at hand and examine interactions that are not apparent before deployment.

Modeling and simulation can be used to understand the emergent behavior of an SoS if systems are modeled. Emergent behavior cannot be exactly predicted before run-time even if a single system operates with a known control algorithm. For example, one cannot predict the actions of an ant colony based on the understanding of individual ants. The behavior is affected by the environment and it is constantly evolving. Simulation can help explore the possible resulting behavior of an SoS or examine how that behavior may change if a new system or new capability is added to the existing SoS.

Much work has been done with hardware-in-the-loop (HIL) simulations in power analysis or other electronic design verification [25, 26, 27]. These advanced simulations allow a partial prototyping of the system under test while a software model replaces the rest of the system. This provides results much more reflective of real world performance than a purely computational model without the need for a complete and finished system. This concept of mixed simulation has also extended into the robotics field. RAVE implemented an environment in which virtual sensors could be added to real robots in a simulation [28]. This was extended to robot-in-the-loop (RIL) simulations in [29, 30, 31] and a test bed in [32]. HIL simulations allow designers insight into decision making early on in the development process. Incremental changes to a design can be tested and debugged before advancing to
the next step in the development. System testing is accelerated with the ability to dynamically change the test input based on the current state. This attempt to break the system with more precise tests is not always possible in the physical deployment.

The proposed work uses activities as part of a framework for flexible AIL simulation to support systems that do not have their own DEVS software. A data driven approach using XML reduces the need for detailed modeling of all components in a simulated SoS and enables easy integration and interoperability of heterogeneous systems.

2.2.1 Discrete Event System Specification (DEVS)

Discrete event system specification (DEVS) originated as a formalism for discrete event modeling and simulation but the methodology has grown to a wider use for systems theory within the research community [33]. It is a computational framework to support systems concepts and separates the model and the simulator. An abstract interface describes a model and a simulator and allows for different languages to implement them and provides for varying ways of simulating the model [34].

Event-driven models are an intuitive process humans use and understand. For example, a visitor arrives at your house and later leaves. A light bulb burns out and causes a replacement to be ordered and installed. These are all discrete events that cause a change in the current state of the system.

In a continuous time-based example, as in the filling or emptying a tank of water, the simulation must advance at very small increments of time to accurately reflect the current state. Even if the delta time change is small, the error accumulates and can adversely affect the results of the simulation. While modeling complex interactions
in a continuous domain can be simulated using event-driven methods, the complexity and overhead associated with this paradigm are generally more than a time-driven counterpart. However, if the simulation is time-driven and no state changes occur for extended periods of time, computing resources and execution time are wasted in evaluating each delta time period.

The event-driven paradigm and discrete event simulation method fit cooperative robotics system well, as much time could elapse between encountering obstacles, detecting a threat, or otherwise changing the behavior of the deployed team. The event driven architecture of DEVS enables the modeling of systems as naturally perceived by humans. This increases efficiency of simulating systems that may have extended time lapses between successive events as is sometimes the case in teams of robots working together to detect threats. A full explanation of modeling and simulation theory and the presentation of the DEVS formalism can be found in [24].

In DEVS, basic models, called atomic models, can be connected together to form larger, more complex models called coupled models. As stated formally using set theory notation, an atomic model in DEVS is a structure $M = \langle X, S, Y, \delta_{int}, \delta_{ext}, \delta_{con}, \lambda, ta \rangle$ [33] where
$X$: is the set of input values

$S$: is a set of states

$Y$: is the set of output values

$\delta_{\text{int}} : S \rightarrow S$ is the internal transition function

$\delta_{\text{ext}} : Q \times X^b \rightarrow S$ is the external transition function,

where $Q = \{(s, e) \mid s \in S, 0 \leq e \leq ta(s)\}$ is total state set

and $e$ is the time elapsed since last transition

$X^b$ denotes the collection of bags over $X$

(bags are sets in which some elements may occur more than once)

$\delta_{\text{con}} : Q \times X^b \rightarrow S$ is the confluent transition function

(to resolve simultaneous internal and external events)

$\lambda : S \rightarrow Y^b$ is the output function

$ta : S \rightarrow R_{0,\infty}^+$ is the time advance function

Couplings among models are the connections which specify the relationships between one DEVS model’s output port and another model’s input port. An output value from one model is transferred in a message object to be received as an input port-value pair for a second model if a coupling exists between them. A port can be the name of either an input or output of a model. Coupling is a closed operation as each coupled model can be represented as an atomic model [33].
2.2.2 DEVSJAVA

There are a few software packages that implement the DEVS framework. JDEVS [35], OMNeT++ [36], ADEVS [37] are general simulation environments for various systems. DEVSJAVA is a Java based implementation of the DEVS formalism from the Arizona Center of Integrative Modeling & Simulation (ACIMS) co-directed by Dr. Zeigler [38].

The DEVSJAVA software supports and provides a portable framework to simulate models conforming to the DEVS formalism. DEVSJAVA enjoys widespread use across the research community from its robust modeling and simulating capabilities: understanding how children and humans reason [33], monitoring industrial steel production to lower costs [39], researching wild fire containment methods [40], and the cooperative robotics arena [3]. There is little to limit the use of DEVSJAVA as a general purpose simulation environment because of its generic implementation and support for additional extensions as needed.

DEVSJAVA supports a message passing communication scheme among the models in the system. Message passing is an intuitive architecture to facilitate couplings between models as well as modeling the peer to peer communication links within an SoS. Thus, XML data encapsulation can be implemented more readily using this existing message passing scheme.

Through appropriate connections between models’ input and output ports, the complete system is specified. Messages are sent between entities to communicate relevant information. Messages are received on input ports, and messages are sent out on the output ports based on the current state. An incoming message can cause a state
change based on the external transition function, $\delta_{ext}$, or state changes can happen internally according to the internal transition function $\delta_{int}$ and the time advance function $ta$. It is possible that an external message arrives at the same time an internal transition is scheduled to occur. This conflict is resolved according to the confluent transition function $\delta_{con}$. Further implementation details of DEVS modeling and simulation with Java is found in [38].

### 2.3 System of Systems (SoS)

Systems engineering is moving towards finding ways to make the sum of the parts greater than the whole. Systems can work together to share information, or specialized abilities, instead of repeating and duplicating capabilities in less efficient ways. Tasks are complex and span across disciplines and geographic area and it is too cumbersome to design a single system with the task of accomplishing the goal. Existing systems, or new systems, are made to work together to accomplish the goal instead of designing a new independent system.

Definitions of key terms for this field as used within U.S. Department of Defense (DoD) specifications follow.

System definition: *a functionally, physically, and/or behaviorally related group of regularly interacting or interdependent elements; that group of elements forming a unified whole* [41].

SoS definition: *a set or arrangement of systems that results when independent and useful systems are integrated into a larger system that delivers unique capabilities* [41].

SoS engineering definition: *deals with planning, analyzing, organizing, and*
integrating the capabilities of a mix of existing and new systems into an SoS capability greater than the sum of the capabilities of the constituent parts [41].

The main goal of the SoS approach is to achieve greater performance from multiple systems and even find new capabilities not previously existing on any one system. There is a migration from end-to-end systems communication to global sharing of information among systems. Singular systems that have customized interfaces to other systems are no longer cost effective and flexible enough for the new net-centric infrastructure started by the growth of the Internet. The Global Information Grid (GIG) is the U.S. DoD’s concept for a unified communications framework among all systems across branches of the armed forces and intelligence communities [42, 43]. Net-centric design exposes the necessary interfaces for different systems to interact while maintaining a hidden view of the actual implementation technology. Network theories begin to play a greater role in understanding the interactions among the systems.

For this work we considered an SoS definition similar to the one presented: SoS are large-scale concurrent and distributed systems that are comprised of complex systems working towards a larger, common goal. This is an information systems view. An information system view focuses on the available data and it emphasizes the interoperability and integration properties of an SoS [3].

2.3.1 Integration Property

Integration has long had various levels and conceptual areas including the technology employed, the architecture chosen, semantic integration, and ability of the user to efficiently use a system [44].
“Integration is the process of combining different components to form a subsystem or the integration of subsystems to a chain of systems, also known as system integration [45].”

Just as subsystems are integrated to form a single system, such as an automobile, multiple independent systems need to be integrated into a single SoS which provides functionality beyond that of any individual system. The discrete systems should work together for this new functionality despite hardware or software structures, operating systems, and internal implementation. Integration implies a connection to the system is possible, but it does not imply that the connection method is flexible to include future systems.

2.3.2 Interoperability Property

The base levels of interoperability in the Levels of Conceptual Interoperability Model (LCIM) [46] require data to be shared among systems in an unambiguous manner. This technical interoperability is differentiated from the substantive interoperability [47] required to achieve automated re-use of simulations over a network architecture. Interoperability is a term which is applied in varying degrees and is often left as an ambiguous statement. The LCIM attempts to clarify this broad and widely used term.

This model is used in a discussion on composability in [48]. Composability is the ability to quickly construct new simulation systems from various components. These components could be models stored across different repositories accessible on a network. It is constructing simulations from modular and reusable components.
Composability is the highest abstraction of interoperability shown in Figure 2.2. This figure is adapted from [49]. An explanation follows.

Interoperability at Level 1 is supported by the underlying technology standard used as a communication link between the systems. This channel, such as TCP/IP, enables the possibility of sending data. Level 2 syntactic interoperability defines a common data structure for the systems and is directly enabled through the use of XML to wrap data. If the meaning of XML tags are well defined, such as through XSL, Level 3 semantic interoperability is also achieved. Level 4 pragmatic interoperability defines and presents an interface of the model to other systems and Level 5 dynamic interoperability exists when systems can adapt to other systems entering and leaving the scope so that the available functionality evolves over time.

This work, an implemented framework for AIL testing, addresses the first three levels of LCIM for the systems in the simulation. The framework itself needs also to be interoperable with other systems and simulations as part of a larger M&S testing and verification methodology. Generic languages, such as XML, are often used to achieve interoperability at various levels of the LCIM.

2.4 Extensible Markup Language

The DoD’s Discovery Metadata Specification (DDMS) [50] sets forth a common set of descriptive metadata elements (available as an XML schema) so that data from various repositories and systems is made visible to all other systems. XML is recommended for data exchange as part of an open standard supporting interoperability in [51].
DEVS Modeling Language (DEVSML) [52] is built on XML to provide transparent simulator implementation. That is, a model is not restricted to a simulator based on the same programming language, but the model is described in a generic manner using XML which can be automatically implemented in the programming language of the available simulator.

The XML data for an SoS is considered dynamic data. The data is updated asynchronously at non-regular intervals as each system interacts with the environment and the SoS. In contrast, static data is declared \textit{a priori} and streaming data is a continuous flow of data. One may tend towards a continuous flow of data from the
SoS in this case, but must remember that it is discrete event based and the systems may not transmit data on a regular basis.

Parsing techniques for Java, as described in [53], summarize the different options for implementing the XML parser in the simulator. The Document Object Model (DOM) parsing technique will be used to efficiently have support for updating, adding, and removing information nodes. The XML document for the SoS will be changing as new data is received from components and the DOM is well suited for updating this changing information.

Extensible Stylesheet Language Transformations (XSLT)[54] assists parsers in understanding the meaning of each XML tag. Tags in XML are user defined unlike a markup language such as HTML where all tags are standardized and well defined. XSLT is used to transform one XML document into another document. This document can be an XML file with different tags, or another type of document that is supported.

XML Path Language (XPath) is a query language used to find nodes in an XML document [55]. It allows easy navigation of the hierarchical tree structured nodes in the XML document. In simple terms, each level of the XML document defined by tags creates a node. XPath is a necessary component to successfully transform an XML document into another. An alternative method is to use code which traverses the DOM but this is a cumbersome procedure. Java 5 and later releases contain the javax.xml.xpath package to provide a standard API to an XPath engine. Advanced searches, or certain constructs still need to be handled in user code, but the complexity is reduced through initial use of XPath.
This work uses XML to describe the data of each component in the SoS. The details of how XML enables interoperability within the SoS is in the conceptual presentation of the framework in Chapter 4. Before presenting the new framework, current simulation tools are presented in Chapter 3.
Chapter 3

State of the Art Simulation Tools

Robots were historically expensive and it was considered dangerous to deploy a field unit without careful verification of the control algorithm. Simulated environments have become more detailed and robots themselves more powerful over the years. Lower cost robots, such as Lego Mindstorm™, are available, but robotics research can also use more expensive systems; either choice may rely on simulation to verify algorithms.

Networks have also continued to expand and are prevalent in most systems. The current state of simulations over a network is also discussed. Testing methodologies, including rigorous formal methods and empirical observations, have increased in comprehensiveness but still remain largely customized for each application.

3.1 Simulators

Simulators are the software applications providing a way to test and verify algorithms before implementation. Simulators can also provide an interface to connect to a real robot if designed for hardware-in-the-loop testing.
3.1.1 Robotics Simulators

There exist robotic simulation platforms but they generally do not easily integrate or interoperate with other general systems. Webots™[56] provide the ability to interact with external controllers via TCP/IP. However, there is no guarantee to maintain consistent simulation steps across a larger network. Webots™ also does not support AIL simulation but jumps from simulation experiment to deployment.

As the robotics simulation field continued to mature and reach a wider audience there became open source collaborative simulation environments available. The Player Project [57] provides free software for robots including Stage, a 2D multi-robot simulator, and Gazebo, a 3D multi-robot simulator. Virtual models can be simulated, but references to migrating this control code to “hardware” are misleading. Player uses a client-server approach, where the client resides on the robot and interfaces with the actuators and sensors and provides a communication link back to the host computer which runs the control algorithm. Gazebo provides realistic feedback based on a physics engine but no real hardware is deployed.

Microsoft Robotics Studio provides a proprietary framework for simulating robots, their environment, and creating 3D visuals as well as remote operation over the web [58]. Using what the company calls Decentralized Software Services (DSS) and Concurrency and Coordination Runtime (CCR) it is possible for a simulation to span across a network. Robot control algorithms are deployable in autonomous hardware only if the robot is running a MS Windows based operating system and the studio software.

These recent projects and advances have greatly increased the accessibility of
robotics simulators to researchers. However, to the systems designer, they fail to provide an effective way to implement AIL testing as part of the simulation to deployment transition.

Standalone, specialized simulation programs are no longer sufficient as we are now moving towards unified, distributed simulations of systems using web services and networks [59]. The systems we want to simulate now are increasingly complex, large scale, distributed systems. An SoS may be too large to practically simulate on a centralized computer system and need the computing power of a network. A distributed simulation inherently accounts for delays and transmission errors just as the deployed SoS may encounter. This locally distributed SoS simulation may then need to interoperate with other SoS simulations, or as the child of a parent who is part of a larger SoS. A simulation framework needs to be scalable and interoperate with systems across a network and share data among simulations in an SoS because an SoS is not limited to robotic systems.

3.1.2 HIL Support

Hardware-in-the-loop simulation is a missing link in most robotics simulators. HIL is not communicating commands to the system from a control module residing on a centralized host, but it is the control module residing on the system. This creates an autonomous system which then provides feedback to the simulation based on the provided stimulus from both the real and virtual worlds. A complete SoS can be analyzed and traced in real time even if only a portion, or single system, is available for deployment. Higher fidelity is achieved with a real system when analyzing the behavior and determining the interoperability of the system under test with the
existing systems in the SoS.

A thorough look at model continuity using discrete event simulation is found in [7]. The additions to the DEVSJAVA simulator infrastructure presented here are also fundamental to the AIL framework presented in this thesis. This work proposes the use of XML to encapsulate the data among systems and provides a method for including systems not running DEVSJAVA.

Early environments created to study a robot used overhead cameras and slow communication links to transfer virtual sensor data [60]. In [61], a special rig is built to perform HIL testing of an aerial vehicle’s sensors and flight algorithms without the need to fly the UAV. RAVE introduced a multi-robot simulator with support for real and virtual environments but no updated literature is found on the project [28].

A simulation based virtual environment to study cooperative robotics is discussed in [30] and robot-in-the-loop simulations are presented in [31]. An integrated test bed for robots utilizing an overhead camera system for monitoring the system is presented in [32]. In these frameworks, each robot is running a DEVS compliant simulator to create a distributed simulation. Synchronization is achieved at the DEVS simulator level and decisions can be made through knowledge of the global locations of each system.

This work’s initial XML based SoS simulation framework is demonstrated in conventional simulation steps [8, 62]. It is extended to support a mixed real and virtual simulation of systems. The new framework bridges simulation and implementation through the use of a discrete event based system of systems approach to agent-in-the-loop simulation.
3.2 SoS and DEVS Applied to Net-Centric Design

The U.S. military has been the biggest proponent for net-centric operations throughout its organization. The U.S. DoD believes sharing all information about a situation with its individual forces will give greater awareness, increased effectiveness, and even fewer friendly casualties [63]. This all depends on the Global Information Grid (GIG) to provide a reliable net-centric environment through which data is shared. Fast testing and evaluation methods are required to make timely deployment possible [63]. As the GIG continues to realize an interconnected physical network an efficient means to test systems’ functionality on this new infrastructure is required. Figure 3.1 shows an abstraction of the software layers in a DEVS net-centric framework to address this need.

![Figure 3.1: Abstraction Layers in DEVS Net-centric Approach](image)
3.2.1 DEVS/Service-oriented Architecture (SOA)

Service-oriented architecture (SOA) is a collection of self-describing modules which can communicate with one another. From a programmers’s perspective, an application can be built from these modules by defining calls to their functions and subsequent transfers of data to other services. This promotes re-usability of existing technology while continually updating functionality with the addition of new services. Its framework can be used to create new systems and simulations. DEVS-based simulation web services using SOA in distributed simulations is presented in [64]. This approach adds a layer on top of the actual simulator but allows the user to create systems using models described in DEVSML from various web-enabled repositories using open standards. The additional layer allows greater automation to achieve interoperability across a network of computers to address the DoD’s systems level net-centric testing and evaluation goals.

3.2.2 DEVS/HLA

High Level Architecture (HLA) is an IEEE standard (151623.2003) [65] for distributed computer simulations. It was developed by the DoD (HLA v.1.3) before adoption as a standard. This specification is implemented as middle-ware run-time interfaces (RTI); both commercial and non-commercial versions are available from various sources. The goal of HLA is to provide an interface between simulations on different systems to achieve interoperation among systems to provide large scale distributed simulations.

DEVS has been used in DEVS/HLA and is an HLA compliant framework endorsed by the Joint Interoperability Test Command (JITC) [66]. JITC is the
test organization responsible for certifying simulation systems as interoperable with existing and planned future net-centric operations of the DoD. RSync [67], a commercial spin-off of ACIMS, is a corporation focused on providing professional M&S products for military and industrial applications using a DEVS/HLA framework.

3.3 Testing Methodology

The goal of modeling and simulating in this work is to test proposed systems’ and/or algorithms’ ability to meet the goals of the SoS. Automated methods of comparing results to the desired results is necessary in large and/or complex simulations. When the expected result is straightforward, comparing to a golden algorithm is possible. With emergent systems with diverse and dynamic goals and multiple solutions possible, analyzing the “success” of a system becomes more difficult. It is still possible to determine final outcomes, such as “Did the agent reach the destination in the allotted time frame?”, but more difficult to answer questions as “Did the systems work together in an efficient manner to disable the threat?” or “Under what circumstances will the system fail to achieve any further goals?”

Testing is usually left out of the description of current robotics simulators. Human oversight of the simulation will provide the primary analysis and feedback of a successful outcome. It is desirable to have a framework which has testing and analysis as a consideration in normal work flow so that some subjectivity is removed from the evaluation.
3.3.1 Formal Methods

The field of cooperative systems and their formal analysis is still in infancy and no standard approaches have been developed. Ideally, systems’ interactions can be proven mathematically to give a desired outcome, or shed light on boundary conditions which should be considered when specifying a system’s ability. Some work has been done to provide formal verification methods of complex distributed systems; the subject has been approached by Winfield [68, 69, 70] for naive swarm algorithms but has not been applied to an SoS with heterogeneous and complex systems. There is a call for a more rigid design methodology for emergent systems and development of methodologies and practices for testing swarm systems.

Winfield’s approach discusses liveness, or exhibiting desirable behaviors, and safety, not exhibiting undesirable behaviors [68]. The swarm behavior will not cease when a goal has been met because the SoS will continue to evolve and meet other goals. The desired behavior should also not cease all operations if a deadline or goal is missed, but rather proceed to the next. An AIL simulation framework using DEVS presents a possible solution step to the testability problem. A hierarchical-web organization approach formally defines and analyzes a complex and dynamic evolution of the robot team in [71]. A lack of a formal computational mapping of abilities, rewards, and penalties for goals in the SoS prohibit the application of the approach at this time.
3.3.2 Empirical Analysis

Empirical methods rely on testing as many different situations as possible and aggregating the results into a metric of performance or expected reliability. Documenting the experimental results of simulations is the most plausible method for analyzing an SoS simulation. The results have greater impact if the models closely match the real world systems.

The use of an overhead camera, or any similar setup, to synchronize systems or provide global state data for analysis is not always practical for a geographically distributed SoS. It may not even be an option if we are considering, for example, UAVs or an entire military division. Satellite could be available for large scale operations by the military. Aerial surveillance can provide higher resolution updates but this would be costly and still may not provide the desired frequency for covering the entire SoS picture.

Other means of analysis, such as tracking and aggregating interactions and communications from the other systems in the SoS, must be employed. In perhaps the most subjective example, human ‘systems’ provide the feedback for localized systems. A human’s complex judgment and perception abilities can be employed to determine whether the desired behavior was exhibited or if the swarm failed to meet the requirements. In addition to this metric of pass/fail given a certain experimental setup, data from other systems could be used to more objectively verify the agent in the loop.

Capturing the data transferred from each system in the SoS and processing it against given conditions is the approach taken by the framework as presented in
Chapter 4. Prior to discussing the analysis of an SoS, the framework to support this testing is illustrated.
Chapter 4

A DEVS Based SoS Simulation Framework for Testing

The framework is presented in three conceptual areas: metadata and data requirements, discrete event system specification, and the interface between real and virtual worlds. All steps of the model continuity method (simulation experiments, real time simulations, and agent-in-the-loop simulations) are supported, although the focus of this work will be on connecting the simulated world to the real system for AIL setups.

The overview of this AIL simulation framework is contained in Figure 4.1 and shows the real world and virtual environment. They are connected through the agent model and a communications module. These modules each contain a driver to enable connections through the underlying operating system and hardware devices. The agent model resides in a virtual environment which contains information about the location of all agents and obstacles and resembles the surroundings of the real system.

The work presents a scalable framework for simulating SoS by utilizing XML to standardize the data created by each system. This XML enables interoperation among any set of systems which meets only one requirement: to provide data according to, and parse received data using the SoS’ XML standard. Scalability
is provided by the underlying DEVSJAVA implementation which executes any model conforming to the DEVS formalism. An approach to interfacing and synchronizing real and virtual systems eliminates the requirement that all systems execute a common simulator or operating system. The interface between non-DEVS compliant robotic systems is abstracted to a software layer using activities to link with the hardware communication protocol of the robot. A prototype virtual environment and XML engine provide the conceptual proof of the viability of this approach.

4.1 Initial Framework Support

The interactions between the independent systems within an SoS are asynchronous in nature and can be effectively represented as discrete event models [3]. Choosing the DEVS formalism and the DEVSJAVA object oriented implementation provides some inherited background functionality. Communication overhead for distributed
simulations is reduced, models and their relationships can be added and removed during runtime, and a generic interface declaration supports external Java Thread execution.

The DEVS simulator interface allows the models to be executed in different ways. A fast mode simulator executes the models for initial simulation experiments. The simulation time advances as fast as the host can compute it. In contrast, a real time simulator synchronizes with wall time. Simulators are also categorized as centralized or distributed. A distributed simulator coordinates time and messages among simulators running on nodes connected by TCP/IP whereas a centralized simulator runs on a single workstation. These various simulators are used for the different stages of model continuity [7].

The system-subsystem hierarchical organizational view in a system of systems approach is intuitively supported by coupling DEVS atomic models. Figure 4.2 maps an SoS hierarchy to DEVS models and demonstrates the organization of simulated models. At the topmost hierarchical level, the SoS is represented by a DEVS coupled model containing one or more instances of each of the models describing the independent systems. Each dot represents an input or output port where data enters or leaves the system. Figure 4.2b lists the structure of the XML corresponding to this SoS. Communication links are implied by coupling the models together. The method

```
addCoupling(src,"portOut",dst,"portIn")
```

creates a directional communication channel from src’s portOut port to dst’s portIn port. src and dst are models of systems in the SoS. Models can be
added to a DEVSJAVA simulation and also removed by calling `addModel(m)` and `removeModel(m)`, respectively. These dynamic methods handle creating a new simulator to bind to the model and updating all affected simulators when couplings are affected by changing models. Adding or removing models may also change the couplings. Couplings can be added or removed directly, as well. This may be the case when communication links among the models (systems) changes during the simulation as the result of some interaction or distance. A DEVSJAVA message, in basic structure, is one or more port-value pairs. A `message` which appears on a model’s outport is transferred to all inports connected by a coupling. A more generic class, `entity`, is used to provide a basic interface to most objects within DEVSJAVA. These objects, whether included in the DEVSJAVA library or user defined, are then compatible with DEVSJAVA storage containers and interfaces.

DEVS `ActivityInterface` provides an interface for external computations to asynchronously enter into the simulation loop. Activities are bound to a single atomic model, but each model may have multiple activities running concurrently. As a Java
Thread an activity can complete any task independently of the simulator. “Results” of the task are returned to the model in a message via the interface’s methods. Figure 4.3 illustrates the sequence of interaction.

![Diagram](image)

Figure 4.3: Interaction between atomic model and activity

An activity can be used to drive a system actuator or receive signals from a sensor and create a message for the atomic control model if the system supports a DEVS simulator. A virtual model of this same sensor or actuator should have an interface similar to the real device. Adhering to this principle of model continuity, a common interface to a component in both real and virtual worlds, enables seamless migration from experiments to testing the new component in the field. abstractActivity implements the same interface but runs as an atomic model, not a Thread, in the simulation. The ActivityInterface is used in this work to create communication drivers for any system to enter into the loop as well as interacting with the virtual world just as a system would in the physical realm.

### 4.2 XML Architecture

In [3], an SoS simulation framework utilizing XML is proposed in order to wrap data originating from different entities in a common way to address the interoperability and
integration requirements of an SoS. XML’s hierachical structure inherently reflects the hierarchical structure of the systems composing an SoS. As seen in Figure 4.4, each system definition contains at minimum some identification and a description of the output as well as the actual data. A system can contain another system as an element in a recursive manner to construct the SoS hierarchy as it exists. The metadata tags can be used to describe the data’s source, type, security level, importance level, or age. This data view of the systems allows each system to take actions to meet individual goals and the goals of the SoS. Even if a new system enters the SoS which has
Listing 4.1: `XmlStrEntity` class declaration

```java
public class XmlStrEntity extends GenCol.entity {
    protected String value;
    ...
}
```

defined its own set of tags, data exchange can occur, and the system can interoperate with the SoS. Extensible stylesheet language transformations (XSLT) would define a conversion method to adapt the data into the standard used by the larger SoS.

Within the DEVSJAVA software, XML string data is wrapped as `XmlStrEntity` objects to inherit the properties of an `entity`. This allows the data to be sent as `messages` among models in the simulation. `XmlStrEntity` also provides basic parsing and encoding methods for the simple XML architecture in use. The architecture shown in Figure 4.5 is an implementation of the general SoS XML architecture shown in Figure 4.4. The new syntax arises from the necessity to fit data within the storage available in the communication layer of the robot deployed as the real agent. The robot, a Groundscout [13], and its model is further explained in Chapter 5. The IDs of the system and the sensors are single bytes but these are transformed into more readable `String` types in Java by the communication layer. This aides in presenting the information to the user and is not a general constraint when working with the XML framework. This simple transformation between two ways of representing the data demonstrates XML’s flexibility and can be further developed using an XSLT definition. Table 4.1 summarizes the transformation into the smaller XML structure embedded in the deployed system. Any imaginable sensor or data type can be implemented but the basic sensor/data types implemented are in Table 4.2.
These include position and destination integer coordinates, threat detection status, ultrasonic distance sensors and corresponding “virtual distance” sensors.

4.3 Interfacing Two Worlds

A virtual representation of the environment is implemented as an underlying database object to provide feedback to the models. This virtual environment not only contains virtual objects, but representations of objects in the physical world, too. Stationary obstacles and mobile systems are both included. When a virtual representation of the
Table 4.2: Summary of XML Identifiers

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Hex</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>0x24</td>
<td>Current (x,y) location of system</td>
</tr>
<tr>
<td>+</td>
<td>0x2B</td>
<td>Destination/Target (x,y) location</td>
</tr>
<tr>
<td>‘‘</td>
<td>0x22</td>
<td>Threat Sensor ID</td>
</tr>
<tr>
<td>!</td>
<td>0x21</td>
<td>Robot status</td>
</tr>
<tr>
<td>#</td>
<td>0x23</td>
<td>Sonar Distance sensors</td>
</tr>
<tr>
<td>‘</td>
<td>0x2C</td>
<td>Virtual Distance sensors</td>
</tr>
</tbody>
</table>

real agent is included in the simulation environment the real agent becomes another model from the simulated SoS’s point of view. The environment database and model representations of real systems works in conjunction with plug-in communication drivers to establish a connection and synchronize the state of the physical systems with the virtual SoS.

An external system’s software/operating system falls into two classes: DEVS compliant and non-DEVS compatible. If the system is executing a DEVS simulator, or can support an additional software layer to enable DEVS compliance, the DEVSJAVA framework enables a distributed simulation if the components are connected to a TCP/IP network. Interoperability, data exchange, and synchronizing the simulation clock are inherited from DEVSJAVA just as though all systems were on a single host. The SoS based XML AIL framework in this work focuses on enabling the systems, specifically autonomous robotic platforms, which do not run DEVS compatible software, to be tested as an agent in an SoS. This example SoS is thus composed of both real and virtual systems. Consider a legacy system or micro-system which does not have the ability to run high level code, or cannot be easily reconfigured/reprogrammed: using plug-in communication drivers in the simulation framework allows the legacy system to execute as an agent-in-the-loop. The data the
system sends is used to update its virtual representative model in the simulation.

It is assumed the system has some type of communication, otherwise it would be very limited in its usefulness as part of an SoS. This framework for an AIL simulation supports various communication medium and standards using a plug-in approach.

Synchronization between the real system’s operation and the virtual agents in the SoS must also be considered. The virtual representation of the system in the simulation interacts with the virtual environment on behalf of the external system. This model representation, however, does not need to provide any control or implementation details of the system. The model, along with the communication plug-in, creates a means to transfer data back and forth between the simulation environment and the real world to both synchronize states and cooperate as an SoS.

4.3.1 Communication

A communications driver is written in Java for compatibility with the simulation framework and to access the underlying hardware communications device. This driver is plugged into the AIL simulator framework as a back-end service which enables the virtual representation of the real agent to communicate from the simulation. Optionally this module can be expanded into a software layer which packs and unpacks data into XML data if a system is unable to be updated to transmit its data according to a common language standard. In addition to the back-end driver, a communications layer is included in the model of each mobile system in the SoS. These two elements work together to facilitate all communication cases.
4.3.2 Synchronization

The data communicated among the systems is used to synchronize the external agent with the SoS simulation. Synchronization is achieved when the virtual representation of the agent in the virtual environment accurately reflects the state of the agent in the physical world. It is the virtual representation which is used for generating sensor readings and location data related to the other simulated systems. For this reason, the virtual representation of an agent-in-the-loop also includes a “control” layer just as a virtual model does. This control layer, however, does not affect the behavior of the real agent. It interacts with the virtual environment based on the data made available to it by the communication layer. As coordinate data is received, for example, the synchronization control layer requests the virtual representative model be relocated within the virtual environment. This happens in much the same manner as a virtual system would initiate a move. The updated sensor data based on the inclusion of virtual objects can then be forwarded back to the external agent in a reverse process.

4.4 Analysis of Systems

Simulation of an SoS attempts to better understand and analyze the behavior of the systems. It may be difficult to have a set of concrete metrics to analyze the general swarm behavior. The emergent swarm behavior is achieved through the synergism of many systems’ autonomous algorithms. An autonomous algorithm is one in which the robot can operate and achieve goals on an individual level as well as part of a swarm team. This work distinguishes between the performance of an individual robot and the emergence of swarm behavior by the robot when cooperating with other systems.
4.4.1 Performance of A Single Robot

The performance of a robot is closely related to the technical specifications claimed by the unit. Position, heading, speed of travel, rate of communication, and accuracy of sensor readings can be measured very objectively. These factors may or may not affect the emergence of swarm behavior depending on the desired goal of the swarm and the algorithm deployed on the agents.

Convoy speed, number of adjustments, formation coherence, scalability, and sensitivity as performance metrics for the robot following convoy example in [29]. In the threat detection example of Chapter 5, there is no predefined formation to expect. Consistent communication, both a valid link and correct interpretation of data, is important for the system.

4.4.2 Evaluating Emergent Behavior in SoS

The desired swarm behavior of the mobile systems in the threat detection scenario is to converge on a given target location, yield to and hold the current position when encountering a teammate who is closer to the threat target, and return toward a home position when a threat no longer exists.

The SoS based approach to AIL simulation allows for the collection of data transferred among the systems. This data can be interpreted and checked against known conditions of failure or known conditions of successfully completing a mission objective.
4.4.3 Measurement Methodology

Data is captured from each system as it becomes available to the SoS. This data is aggregated and recorded by a specialized command center, referred to in this work as a *supervisor*, as a record of the data interactions in the SoS simulation. This supervisor, depicted in Figure 4.6, compiles the data for comparison against a set of specified metrics. These metrics can be adapted as the SoS or experimental test requires. The results of this analysis, a summary of both desirable and undesirable events, are then presented to the user.

The scalability of the swarm SoS is viewed by comparing the supervisor’s summary across multiple runs with increasing numbers of agents. It is also possible, by comparing the same metrics, to evaluate different configurations of the SoS. Configurations involve the deployed locations of systems in the SoS but could also examine different systems’ ability to cooperate.

![Figure 4.6: Supervisor aggregates data and summarizes critical events](image)

Figure 4.6: Supervisor aggregates data and summarizes critical events
Chapter 5

Robust Threat Detection Example

A robust threat detection scenario serves as the case study for the framework presented in this work. The scenario follows that of [8, 9, 62]. In this example there is a team of helicopters which can detect enemy tanks. The helicopters are dispatched by a command center to a location of the threat.

The helicopters are each stationed at a unique location in the map to provide a faster response time for detecting the enemy tank. It is expensive to continuously operate the team of scout helicopters. Sensors have been deployed in the area to create a sensor network to alert the presence of the tank.

The threat is initially detected by one of the seven stationary sensors. These sensors, the radar stations, are cost effective for continuous operation but cannot pinpoint the exact location of the enemy tank. The command center processes the sensors’ data and dispatches the scout helicopters. The scout helicopters cooperate according to a basic swarm behavior control: the helicopter closest to the supposed tank location takes the lead in investigating that area while the other team members hold their positions at a distance from the lead helicopter and one another. A Groundscout robot is deployed in the lab as a “helicopter” to cooperate with the virtual team members. In the example, the models only move in two dimensions so
a mobile robot on the ground is acceptable for testing.

Through this example application of the DEVS based AIL framework the work is shown to be a valid approach to simulation and analysis of SoS. The results of the threat detection simulation illustrate the behaviors of the components in the SoS and the successful communication using XML data messages. Next, we explore the individual components of the robust threat detection scenario. Then the simulations will be presented.

5.1 Modeling the Other Systems

The tank threat, radar station sensors, and command center are modeled in DEVS and together with the team of helicopters form the example threat detection SoS. The virtual environment model is also presented before focusing on the DEVS model for the robot. The control and names are specific to this example, but the general approach to modeling the Groundscout can be extended to any mobile system in order to be tested as an agent-in-the-loop using the DEVS based AIL simulation framework.

5.1.1 Threat

A threat is any undesirable agent within the operational area of the SoS: a chemical spill in a factory, a fire in a national forest, or in this example, an enemy tank within a friendly zone. The threat moves around randomly and is never disabled. This enables a more thorough investigation of the swarm robot operating over time as the threat continues to move in and out of the sensors’ range and the SoS continues to react to the evolving situation. After initialization, the threat remains in the terrorizing
phase to move randomly as seen in Figure 5.1.

An arrow pointing to a rectangle signifies a message is created in the output function while in the given state. Inside the message rectangle the outport and specific object type of the message is listed in the form of \textit{outport} : \textit{object}\_type. A short description of the XML data contained in the message follows the arrow symbol (→). A number next to a dashed arrow is sigma, the time until the next internal event. Figure 5.1 shows the threat relocates every five time units using its internal transition function. A dark arrow from one state to another represents an external event transition. In the threat’s phase diagram, this arrows shows an \texttt{XmlStrEntity} message object type on the \texttt{startIn} inport causing the event (\textit{inport} : \textit{object}\_type). Again, a description of the XML data in the message follows the arrow.

![Figure 5.1: DEVS Phase Diagram of Threat Model](image)

5.1.2 Radar Stations (Sensors)

Each radar station can detect a threat within its range. When a threat is detected, or when a threat leaves its detectable area, a message is sent to the command center. To reduce communication costs but also keep an updated status of threats to the command center, a radar station will continue to transmit at regular intervals if a
threat remains within its range. The exact threat location cannot be determined by the sensor itself, so only the location of the station is sent to the command center. A block view and a phase diagram are shown in Figure 5.2.

![DEVS Atomic Block](image1)

(a) DEVS Atomic Block

![DEVS Phase Diagram](image2)

(b) DEVS Phase Diagram

Figure 5.2: Radar Station (Sensor) Model

### 5.1.3 Command Center

The command center serves as an interface between the multiple stationary sensors and the robot team. In this example, the command center searches for the presence of a threat in the incoming data. When a threat is detected, the command center broadcasts the target location to mobilize the robotic swarm. If the command center is tracking a threat which is no longer detected by any radar station, the robotic swarm is also notified. The command center makes this information available to all the mobile
systems in the SoS, but each system decides what it will do with this information. The block diagram view of the model, in Figure 5.3, also shows a `serialPortThread` contained in the command center. This is the thread responsible for handling operating system level calls to the serial port. Incoming and outgoing packets are each stored in a buffer monitored by the virtual robot model’s communication activity. The flow of data operates independently of the command center control logic; this is just an intuitive place to locate this part of the communications framework.

![Figure 5.3: DEVS Block Diagram of Command Center Model](image)

**5.1.4 Virtual Environment**

The virtual environment model contains two components: a location coordinator for spatial logistics and time coordinators for temporal logistics. This is the approach found in the previous work of Hu [7]. Each mobile robot has a time coordinator to manage the elapsed time in the simulation. A single spatial coordinator wraps an internal database and environment engine to maintain the location data and sensory data of the mobile robots. In Figure 5.4, messages are sent to the environment model from the abstract activity in the control layer of the robot model. The robot model is discussed in detail in Section 5.2. The request appears to both the time coordinator
associated with the model and the spatial coordinator. The spatial coordinator does the computations necessary to update the internal database of locations and sensor readings of the system. After a time delay, emulating the time it would take the robot to move in the real world, the time coordinator notifies the spatial coordinator. At this point a message containing the new location and sensory data is sent back to the robot model.

![DEVS Block Diagram of Virtual Environment Model](image)
5.2 Mobile Agent Model

Swarm agents receive target coordinates from the command center and navigate autonomously from their current location to the destination. They communicate with one another to more efficiently use the resources in the SoS when going to a threat’s purported location. When no threat is present in the area, the swarm agents return to their respective home positions.

The AIL simulation framework’s interface to real agents is also presented here using the Groundscout robot as an example. This will highlight the structures and methodology to provide support for AIL simulation and testing of systems not executing a distributed DEVJSJAVA simulator. The similar model design between a real and virtual agent reflects model continuity principles as well as the goal of testing swarm behavior of the robot. It is possible to add control to the virtual representation of a real agent to test the affects of deploying new functionality on the model.

This work focuses on modeling the behavior of the robot. Specifically, focus is given to the data flow and responses of the robot to incoming data. Dynamic issues such as motor start up, decreasing battery levels, and wheel slippage are not detailed by the model. The speed of the robot is adjusted by changing a time delay in the virtual environment to reflect varying conditions such as battery level and to calibrate the virtual models’ speed to the speed of the Groundscout robot. The data-driven behavioral model shows that detailed dynamics of complex systems are not required to test their effectiveness in an SoS environment. It also enables heterogeneous systems to integrate and interoperate using only a commonly shared data format which overcomes implementation differences.
The DEVS model of a swarm agent is a coupled model as shown in Figure 5.5. A coupled model allows the swarm agent model to more closely represent the hardware architecture of the Groundscout robot as it processes data. The Groundscout robot has separate control and communication modules which are modeled in the communication and control layers. The layers work together to communicate and synchronize the real robot with the virtual systems.

![Figure 5.5: DEVS model of gsRobot (Groundscout)](image)

5.2.1 Communication Layer

The communication layer is responsible for sending and receiving the XML data to the other systems. As presented in Section 4.3 a communications plug-in driver is created to support any real system as an agent-in-the-loop. Figure 5.6 maps the layers of the plug-in to the objects implementing the driver for the example Groundscout.
A hardware driver library containing a `serialPortListener` interface enables the operating system to interact with the underlying RS232 hardware. A `serialPortThread` implements this interface and runs in the background to manage the RX and TX buffers. An activity associated with the communications layer model provides the backend DEVS support to the hardware driver’s buffers. The communications layer, an atomic DEVS model, implements the behavioral functions of the layer. The communications layer is explained from both a virtual system’s point of view and from the view of a model representing a real system. This highlights the similarities and differences between the models.

Figure 5.6: Layers of AIL Communication Plug-in
Virtual System Model

A closer view of the communications layer is illustrated in Figure 5.7. There is an inport, \texttt{rxMsg}, and an outport, \texttt{txMsg} to facilitate communication with other systems in the simulation. The communication layer also must connect to the control layer in order to forward data to it and transmit any requested information to other systems. \texttt{fromCtrlLayer} and \texttt{toCtrlLayer} provide this path. The virtual systems’ communication layers contain an abstract activity (\texttt{gsCommAbsActivity}) associated with them to maintain the same control. A virtual system does not need access to any hardware devices; the abstract activity functions only as a place holder where the model representing a real system would use an activity (\texttt{gsCommActivity}).

![Figure 5.7: DEVS model of communication layer](image)

Real System Model

To facilitate the connection with the Groundscout robot the DEVS model includes \texttt{gsCommActivity}. The activity previously illustrated in Figure 5.7 is now explained. The activity runs as a Java \texttt{Thread} and has access to the serial port’s transmit and receive buffers for data packets. The activity retrieves packets from the receive buffer...
if the packet destination matches the model’s ID. It also places outbound data packets in the transmit buffer. The understanding of the data transfer enabled by the activity further emphasizes the data driven approach of describing the behavioral model of the robot.

The communication layer’s activity listens for updates to the incoming serial packet queue using the Java Observer-Observable API. The activity is an Observer to the Observable serial port packet buffer accessed by the communication layer’s activity. The data packets arrive as well-formed packets of bytes containing header information and a data envelope with XML encapsulated data. Then the activity extracts the XML messages from the packets and makes them available to the communication layer via the DEVS external transition function. The communication layer forwards the message to the control layer and/or broadcasts the message to other virtual systems. The control synchronization layer then makes a request to the virtual environment when a new location or sensor data from the real robot is contained in the message. The response from the virtual environment is given to the control layer and it then broadcasts the feedback to the virtual systems via the communication layer’s output port. These steps are enumerated in time order and summarized in Figure 5.8.

5.2.2 Control Layer (Synchronization)

The control layer implements basic swarm behavior if it is part of a virtual agent model, but implements logic to update the virtual model if the control layer is part of the model representing the real Groundscout in the simulation. The control layer defines the behavior of the robot based on the data received. An abstract activity is associated with the control layer in Figure 5.9. The activity interacts with the virtual
environment; this internal communication does not interfere with the control loop logic of the model.

**Virtual System Model**

The virtual control begins in the **idle** phase and transitions to the **move** phase when the current location of the robot is not near the destination location. This is initially triggered by receiving a mobilization message from the communication layer which handled the incoming message from the command center. This external
event corresponds to the solid arrow transition between *idle* and *move*. The control remains in the *move* phase until the abstract activity confirms the move with the virtual environment. A sigma of zero on this transition immediately triggers the output message in the rectangle. This message is sent to the model’s communication layer where it is then broadcast to the other systems in the simulated SoS. Receiving a new destination while in *move* changes the internal destination coordinates but does not affect the phase of the model. An external event with sigma set to *continue* does not change the time until the next internal event.

When an update message from the environment initiates a return back to *idle*, and the subsequent internal transition function is executed, the updated current position is compared with the destination. If the robot is not close enough ($cur(x,y)!=$dest($x,y)$) the model initiates a transition back to *move*. The internal transition function continues this cycle until the destination is reached and sigma is set to infinity. Figure 5.10 depicts the algorithm for the virtual system’s control layer.

A simulated robot’s control layer contains the algorithm or behavioral definition to define its actions in the SoS. It also uses an abstract activity to interface with the virtual environment. The control layer in a real robot’s virtual representation contains
behavioral logic to update the virtual environment model when updates from the real agent are received. This keeps the virtual environment in sync with the real world.

**Real System Model**

The synchronization in the control layer operates in a similar fashion using **idle** and **move** phases. The synchronization is between the real and virtual environments. The virtual environment is updated when data is received by the model which indicates some change in behavior such as moving. This permits other virtual systems to interact with the model representation and make decisions assuming the virtual environment correctly reflects the state of the world. Figure 5.11 is the real system’s synchronization algorithm described in DEVS. A move in the virtual environment, however, is *only* initiated if the control layer receives from the communication layer a message from the Groundscout robot with a position that is not equal to the current virtual copy of the robot’s position. The virtual environment moves the robot in zero simulation time as there is no need to simulate the time it takes to move when synchronizing with the real system.
The behavioral models presented for the robust threat detection example are then simulated.

5.3 Simulations in Model Continuity Process

The focus of this work is agent-in-the-loop simulation although initial simulation experiments will also be discussed. Real time simulation experiments would follow a similar approach. Existing literature shows model continuity practices are supported if the real agent is Java enabled. Even if the deployed system has a different software configuration than the DEVS simulation, as does the Groundscout, the framework in this thesis enables interoperability. Focus and emphasis is given to the data generated and exchanged by the system to achieve a seamless simulation of mixed real and virtual agents. Viewing the conceptual aspect of the systems’ interactions in an SoS will create continuity in the behavioral model through all stages of simulation, despite different implementation technologies used in the real world and in the DEVS models.

Figure 5.12 depicts the graphics used in all simulation stages. A circle surrounding a radar station gives notice of the range at which it can detect the enemy tank. Seven
radar stations form a sensor network area. The range of each sensor is illustrated by a circle. The radar stations are labeled left to right and from top to bottom so that the first station, \textit{sr01}, is on the top left, \textit{sr04} is in the center of the network, and \textit{sr07} is the lower right station.

![Diagram showing radar stations and their labels](image)

\begin{figure}[h]
\centering
\includegraphics[width=0.8\textwidth]{radar_stations}
\caption{Graphics used in virtual and AIL simulations}
\end{figure}

The simulations are executed on a single core, Pentium 4 3.2GHz x86 workstation with 1GB of RAM installed. The software is developed using Java SDK 1.6, Windows XP Professional and NetBeans IDE 6.1 with a beta version of DEVSJAVA after v3.1.

### 5.3.1 Simulation Experiment

The initial \textit{fast mode} simulation experiment involves five virtual agents which investigate a threat. In fast mode simulation, the behavioral algorithms are verified using more agents and longer simulation times. Fast mode does not correlate model delays to wall time and allows many more situations to be tested in a short time period because no real agent is included. This stage verifies the foundational XML data communication in the framework as well as the control algorithms used by the systems. If the SoS goal is not met, changes need to be implemented.

In Figure 5.13 the systems are in their initial positions and Figure 5.14 shows a swarm agent emerging as the leader to investigate the threat while the other remain
in support positions. The figures are captured from the graphical display of the system. This GUI monitors the data exchanged among the systems and displays an icon for each system. For the sake of brevity, the virtual simulation was successful and allowed continuation to the AIL step. The same threat detection scenario and control algorithms were used in all stages of simulation. More simulation experiment stage details can be found in [8]. A more thorough explanation of the interactions among
the components in the example SoS is the agent-in-the-loop simulation presented in the next section.

5.3.2 Agent-in-the-Loop Simulation

Introducing a real agent into the simulation loop requires a real time simulator in DEVSJAVA. A real time simulator directly correlates a time delay in the model with wall time; both real and virtual worlds advance in time at the same rate. This simulation stage makes use of the RS232 communication module to interact with the Groundscout. The agent is monitored through the continued use of a graphical display of the SoS environment to verify its operation according to the emergent behavior desired in the swarm. The completion of this step demonstrates the ability of the framework to integrate and interoperate with a non-DEVS system and uses the data exchanged among systems to analyze the effectiveness and success of the SoS to meet the desired goals. [9]

The virtual robot models’ relocation speed is adjusted using both time delays and step distances. A change in time delay affects the virtual environment’s response time for updating to a new location or heading. The time delays are directly related to wall time while using a DEVSJAVA real-time simulator in the AIL simulation stage. A change in the step distance affects the robot’s control layer requests on how far to move or change heading each iteration. Simplifying the robot’s continuous motion in the real world to small, discrete steps is a common approach to reduce the computational requirements significantly while maintaining fidelity with the robot’s operation.

Figure 5.15 shows three scout helicopters stationed around the radar network.
Figure 5.15: Initial positions of agents in the SoS for AIL simulation

The darker green helicopter gs02, stationed in the bottom left corner of the plot, is deployed as a Groundscout robot in the lab environment. The other two lighter, yellow helicopters are virtual systems in the simulated SoS.

As the simulation evolves from the initial state shown in Figure 5.15, a radar station detects a threat and notifies the command center with an XML message. The message contains the location of the station and the status of its sensor. The message in is Listing 5.1. A 0 or 1 is used for no threat detected or threat present, respectively.
The command center interprets the messages from the radar stations and then transmits the XML message found in Listing 5.2 to the scout helicopters.

Each scout helicopter receives this message and interprets the parsed coordinate data as the destination location to investigate. According to the swarm logic of the robots, each agent converges on the destination until a member of the swarm emerges as the leader. The swarm agents communicate with one another using XML messages, as shown in Listing 5.3 containing their current locations.

Figure 5.16 shows a helicopter, the virtual representation within the simulation of the Groundscout robot deployed in the lab, over sr85 with the remaining two virtual
Figure 5.16: Scout helicopters investigate enemy tank during AIL simulation: Groundscout leads agents holding in close proximity. Figure 5.18 shows the Groundscout in the lab environment and corresponds to the same simulation time as Figure 5.16. The command center’s logic could be extended to triangulate a more precise location of the threat using data from multiple swarm agents and the radar stations. A more precise location would be useful for more efficient deployment of expensive attack units as they enter the SoS to disable the threat. At another time during the simulation, a virtual agent may take the lead and require the Groundscout to yield. Figure 5.17 demonstrates this direction of communication and resulting cooperation with a virtual agent as the lead investigator over sr81.

Eventually the enemy tank will leave the area. The sensor notices the change and sends an XML message to the command center. Listing 5.4 contains an example of
Figure 5.17: Scout helicopters investigate enemy tank during AIL simulation: Virtual robot leads this message.

Listing 5.4: XML Message from Radar to Command Center

```xml
<y><i>sr05</i>
  <s>
    <i>$</i><d>509,384</d>
  </s>
  <i>"</i><d>0</d>
</s>
</y>
```

If the command center determines from the sensors’ data that a threat is not present anywhere in the area, it proceeds to send an XML message with cleared destination coordinates to the swarm. The command center is aware of the change by parsing the 0 reading of the threat sensor in Listing 5.4. Listing 5.5 is an XML message sent to notify the robot team to stop the investigation.
Figure 5.18: Groundscout robot in lab investigates sr05 during AIL simulation

Listing 5.5: XML Message from Command Center to Robots

```xml
<y>
  <i>bs09</i>
  <s>
    <i>+</i><d>0,0</d>
  </s>
</y>
```

and once the lead helicopter has investigated the last reported location of the threat, all swarm agents return to their initial locations and wait for an XML message from the base station.

The threat detection example was also simulated with varying numbers of agents cooperating. Figure 5.19 shows a scaled simulation involving a Groundscout robot and four virtual team members. At certain times, the team failed to cooperate in an ideal manner and came in closer proximity than desired. The distance desired to be maintained among the systems is larger than the size of the systems, so violating the formation does not mean the systems have crashed into one another. It can be
Figure 5.19: Groundscout robot and four virtual agents during AIL simulation

assumed, however, that the closer the systems operate the greater the probability of a mishap in the field. In Figure 5.20 the helicopters are shown to be close but not occupying the exact same area. This discussion and analysis of the behavior of the robot team continues in Chapter 6.
Figure 5.20: Formation Violations during AIL simulations
Chapter 6

Testability Results & Discussion

This chapter presents the testability of a swarm robot and its control algorithm using the DEVS based SoS AIL simulation framework. This work does not intend to present any new or original swarm behavior algorithms. The algorithm implemented on the mobile agents in the SoS is not optimized, nor is any attempt made to optimize this control based on the analysis of the example threat detection scenario. The results presented in this chapter demonstrate the ability to examine and test a robot’s swarm behavior to compare against the desired outcome of the control algorithm.

It is observed through the AIL simulation that the Groundscout robot does exhibit the desired swarm behavior. The robot travels toward the threat location and yields to other team members or emerges as the leader. Motion towards a home location is executed while the threat is not visible to the sensor network.

The robust threat detection example is examined with varying configurations of the SoS. The swarm robot used in the SoS is tested for both individual performance and swarm behavior. The swarm behavior is examined by two metrics in this work:

- Time for a member of the robot team to investigate a threat
- Adherence of agents to formation boundaries (swarm behavior)
The former is a metric for the liveness of the swarm while the latter relates to the safety of the swarm as discussed previously in Section 3.3. A threat is investigated when a robot arrives at the threat’s location. The members of the robot team should stay a minimum distance apart from one another. If they are closer than this threshold distance, it is considered a formation violation. If the formation violation is severe enough it would result in a potential collision in the real world. Potential collisions are classified as critical violations in the analysis.

The individual performance of the robot is also tracked using two metrics:

- Communication statistics for the real agent
- Scalability of the deployed agents to cooperate in the SoS

The metrics are displayed in graphical format for easier interpretation after the data has been collected during the simulation stages of the SoS threat detection scenario.

### 6.1 Aggregating Data for Analysis

The measurements are calculated based on the data captured in the SoS. The data is captured by the supervisor. The supervisor concept in this work is first introduced in Section 4.4. The supervisor is a DEVS atomic model; each component’s output in the example SoS is connected to its input port. When messages arrive and cause the external transition function to execute, a Java `BufferedWriter` writes each XML message to a data capture output file. A graphical summary of these input and output files is found in Figure 6.1.

A `BufferedWriter` is used to increase the efficiency of writing the hundreds or thousands of XML data messages to disk. In a similar manner, a `BufferedReader`
is added to the communications driver. This enables the `serialPortThread` to capture statistics for the RS232 communication between the simulated systems and the Groundscout deployed in the lab. Another file, the requirements file, is created by the user and defines the variable conditions to flag when parsing through the captured data. The variables set by the input file are formation violation distance, critical formation violation distance, and a threshold for determining arrival at a desired location. These variables assist in determining the metrics for the swarm behavior in this work: time to investigate a threat and adherence to formation distances among the robot team members.

![Diagram](image)

**Figure 6.1: Analyzing the SoS: Input and Output Files**

Post processing of the captured data file is completed by the `analyzer`. This
module creates an output file summarizing events and statistics as determined by the requirements input file and the desired swarm behavior. This algorithm in the analyzer is customized to check for the desired swarm behavior. The analyzer’s output file for the threat detection scenario contains the events and statistics listed in Table 6.1.

<table>
<thead>
<tr>
<th>Events (each includes time occurred)</th>
<th>Additional Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>New system enters SoS</td>
<td>System name, location</td>
</tr>
<tr>
<td>Threat is detected</td>
<td>Suspected threat location</td>
</tr>
<tr>
<td>Threat relocates</td>
<td>New threat location</td>
</tr>
<tr>
<td>Threat is investigated</td>
<td>Investigating system, location, time elapsed</td>
</tr>
<tr>
<td>Formation Violation</td>
<td>Violating systems, distance</td>
</tr>
<tr>
<td>Threat leaves area</td>
<td>Time threat was in area but not investigated</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Statistics (based on events)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total_time</td>
<td>Run time of simulation</td>
</tr>
<tr>
<td>Num_Violation</td>
<td>Total formation violations</td>
</tr>
<tr>
<td>Critical_Violations</td>
<td>Formations violating swarm behavior</td>
</tr>
<tr>
<td>Avg_Violation_dist</td>
<td>Mean distance of all formation violations</td>
</tr>
<tr>
<td>Num_Success</td>
<td>Instances of a threat being investigated</td>
</tr>
<tr>
<td>Avg_Success_time</td>
<td>Mean time to investigate a threat</td>
</tr>
<tr>
<td>Num_Failure</td>
<td>Instances of a threat leaving area before investigated</td>
</tr>
<tr>
<td>Avg_Failure_Time</td>
<td>Mean time threat was left un-investigated in a failure</td>
</tr>
</tbody>
</table>

Data is collected and analyzed for multiple simulation runs. For each team size, data was collected over at least three different simulation runs totaling sixty minutes or more. Measurements were converted to a “per minute” base for comparison to eliminate the effects of varying simulation time.

The Groundscout robot is tested in a swarm of one to seven other virtual robotic agents. The initial positions within the example SoS are depicted in Figure 6.2. The robots (shown in the figure as filled black circles) were evenly spaced around
the perimeter of the area in the threat detection scenario. As the speed of the Groundscout changes with its battery charge level, the virtual agents are assigned a random speed that ranges from full charge to minimal charge. With a full charge, the Groundscout covers fourteen feet per minute (14 ft/min); minimum simulated speed was half this rate. The speed does not change with time, but remains constant for the duration of the simulation run.

![Figure 6.2: Initial positions of robots in threat detection scenario](image)

The results for each robot team are plotted to examine the trends of the statistics in Table 6.1 and to view the swarm robot’s testability using the AIL framework presented in Chapters 4 & 5.

### 6.2 Measuring the Communication Statistics

The agent in the loop undergoing testing is connected to the simulation environment through a communication link. The Groundscout robot connects via an RS232 connection and transmits from the base to the robot on a 916 MHz radio frequency (RF) signal. The robot will integrate and interoperate with the SoS only if the data it attempts to communicate is received consistently by the SoS.

The Groundscout broadcasts updated state information as it moves to investigate
a threat. The new state information contains the current location of the robot and is sent out approximately once every 1.5 seconds. Traveling at a full speed of 14 ft/min this results in an update available for every four inches the Groundscout traverses. A Groundscout in constant motion sends out forty-two packets per minute; the robot, however, is stationary as much as it is in motion during the simulation.

Messages broadcast from the Groundscout need to be captured and broadcast to the virtual systems in the example SoS simulation. This effectively synchronizes the state of the real world with the state of the virtual environment. It is important that this synchronization happens in a timely manner to maintain valid results. Timing measurements were captured from the instance a packet is received by the serial port thread on the workstation to the instance when the virtual representation of the Groundscout broadcast the packet to the virtual systems. This time to synchronize the state of the two worlds averaged 741 mS for a single Groundscout operating as an agent-in-the-loop. While in motion at full speed, this delay corresponds to a two inch discrepancy in the location of the robot.

No method of communication can guarantee the perfect transmission. The TDMA scheme employed by the Groundscout wireless link does not have error correction so a single corrupted bit can cause the packet to be discarded. A checksum byte is used for data integrity. Figure 6.3 shows the average rate at which XML data packets are dropped while attempting to send messages to the Groundscout robot during the course of an entire simulation run. As more virtual agents are added to the SoS, more data needs to be communicated to the agent, and it is expected that the drop rate of packets will increase. To determine if the performance is getting relatively higher
or lower we examine the overall percentage of packets that are dropped. Packet loss is inherent in non-ideal communication protocols and is not a reflection of the swarm algorithm. The number of dropped packets relative to the total number of attempted transmissions is presented in Figure 6.4. This graph shows a constant packet loss of between 4% and 10%. The rate of packet loss does not appear to correlate with the size of the robot team. It is considered to be the expected rate of packet loss using the RF TDMA wireless communication of the Groundscout. Packet loss may not be caused by the swarm algorithm, but it will affect the performance of the robot team. If systems do not receive updated data they will make decisions based on outdated data. This results in decreased performance of the swarm behavior to investigate a threat and maintain the required spacing among team members.
6.3 Measuring the Response to a Threat

The emergent behavior of the robot team achieves a goal bigger than a single robot can handle - a single robot has an observed success rate of zero percent! An individual swarm robot with minimal battery charge moves too slow and cannot investigate the threat before the threat relocates or leaves the area. The emergent behavior of the robot team within the SoS threat detection scenario is to investigate a threat when it is detected by a radar station. It is expected that as more robots are deployed to the area, it is more likely the threat will be investigated. A robot arriving at the threat’s location is counted as a success. The threat is a moving agent and if the robot team’s response is too slow, the threat will move out of the area without being verified. If no member of the robot team arrives at the threat’s location before it leaves the area this is not counted as a success. The success rate increases non-linearly but begins to saturate as it nears 100% in Figure 6.5. Deploying more of the same robot to the
area may not be an efficient use of resources as a larger team will only marginally increase the success rate of investigating a threat. As the robot team successfully investigates the threat the time to do so is tracked. Perhaps the success rate of investigation saturates but the time required to investigate will decrease, and a larger swarm will provide continued benefit for response time. The time elapsed between the SoS detecting a threat at a given location and a robot arriving at that location is given in Figure 6.6. The mean time from threat detection to threat investigation is neither increasing nor decreasing but remains between twenty and thirty-five seconds. Thus, it does not seem that further increasing the size of the robot team would yield a faster investigation of the threat. A high success rate of verifying a threat’s location is the end goal of the SoS example, but equally important in this testing is the robot’s behavior. The robot should exhibit swarming behavior with the virtual agents at all times regardless of the success or failure of investigating the threat before it leaves the area.

Figure 6.5: Percentage of Successful Threat Investigations vs. Swarm Size
6.4 Measuring the Swarm Behavior

The individual robots in the SoS should not emerge at a single location but rather hold at a distance from one another when in close proximity. This creates a swarm covering a larger geographic area than a single robot and enables the team to more efficiently use its resources. A single robot can verify a threat, and the remaining robots will be monitoring the surrounding area to check for the threat. When the robot team does not adhere to this constraint, the swarming behavior is not emerging as expected. Thus, the occurrence of the violations of the formation constraint on the team is used as an indicator of the safeness of the robots and their algorithm. Figure 6.7 presents the results of this metric. As more robots join the team, they are in closer proximity to other robots and we would expect the number of interactions to increase. As the robot team size increases the formation violations increase linearly. Viewing the formation violations as a percentage of all movements by the robot team is less
intuitive. An increase in the rate of formation violations is observed in Figure 6.8 as more robots join the team. However, this sharp increase levels off between 12-13% for team sizes greater than five. It is unknown whether this is a steady state condition or the beginning of a step-like function. In addition to the number of occurrences of
formation violation, the magnitude of the violations are captured. Figure 6.9 exhibits this trend as the robot team increases in size. A subtle linear increase in the average magnitude of a given violation is noted. With more robots there is a greater chance of crossing paths with multiple team members; even if communication between two robots was temporarily lost, the robot may receive communication from another team member within a threshold distance. Receiving both packets or either one results in the robot holding its current position and potentially avoiding the other two team members.

Positioning closer than desired to a neighboring robot does not necessarily mean a failure of the swarming algorithm in real world situations. It is certainly less desirable than intended but may not disable the agents. The formation violations are further categorized as critical formation violations. These are instances of two robots in the team closer than safely allowed in the real world; this is a distance of fourteen units
in the simulated environment. Figure 6.10 shows these critical formation violations as instances per minute for the different size robot teams. The occurrences of critical violations appears to increase with the team size but this remains a very small percentage of all moves recorded by the robots in the team. Figure 6.11 shows a maximum rate of just above one percent for all critical swarm behavior violations in the simulations. Figure 6.12 shows the relative decrease in number of critical violations compared to the occurrence of all formation violations within the robot team. This is the direct result of comparing Figure 6.7’s slope with the flatter slope of Figure 6.10. The robot team appears to become more stable, exhibiting more safety, as the team size increases.
6.5 Scalability

A scalable algorithm should perform as good or better than a smaller implementation of the algorithm. That is to say more robots on the team should not reduce the effectiveness of the swarm, but ideally increase the success rate. It is expected that
non-desirable behavior, safety issues, will increase in magnitude as the number of systems interacting increases. This increase should not accelerate if the cooperation algorithm is scalable.

The threat detection algorithm is not optimized but still appears to be scalable at least for the robot team sizes tested. Figure 6.8 shows stability in formation violations around 13% of all movements in the robot team. Critical violations, ones that may disable the robots in the team, remain in check in Figure 6.11. The relative occurrence of encountering a critical violation as a formation violation is decreasing. While more robots would not cause swarm behavior to break down, it may not be efficient to keep adding resources to the example SoS. Figure 6.5 shows a continuing increase in the rate of successful threat investigation that levels off as we approach a 100% investigation rate. The TDMA scheme used on the Groundscout robots has a limit of sixteen connected devices. A team size of sixteen robots would simulation this situation.

The focus of this analysis is the testability of a swarm robot in order to examine the example cooperative algorithm. Larger robot teams were not tested due to the lack of computing power available on the single core workstation available. A DEVSJAVA real time simulator, such as the one used in the AIL framework, assigns each atomic model as an independent Thread. The hierarchical DEVS model and associated activities for multiple robots fill the available CPU time. Each behavioral robot model added to the simulation contributes four new threads: the communication layer and associated activity as well as control layer and associated activity. Another thread is spawned as a time coordinator in the environment. The spatial coordinator
in the environment is not optimized and contains double and triple nested loops for calculating sensor readings among all the components.

The performance of the robot model is not compared against more or less complex models. The model’s complexity is dependent on the desired focus of the simulation experiment and the behavioral model used in this work provided an appropriate level of detail from a data oriented viewpoint. DEVSJAVA does not inhibit the models from increasing in complexity (i.e. including mechanical dynamics of the motors), but the complexity would be limited by the timely computational response necessary to facilitate an accurate agent-in-the-loop simulation.

All robots and other models are being executed at once, resulting in multiple functions, mostly sending and receiving messages among the components, occurring at the same time in the simulation. The $n^{th}$ robot model in the simulation creates an additional $2n(n-1)$ couplings to the other robots. Each pair of couplings between models may create a potential thread lock situation as each waits for the other to send a message. A very small random delay, on the order of 200 ms, is added to each model in an attempt to reduce a simultaneous send/receive deadlock. As it is, a simulation run sometimes became unstable and one or more agents in the environment would no longer communicate or respond to communication. In most cases of failing thread communication the simulation run was discarded and replaced with a run which more accurately reflected the interactive behavior of the components in the SoS.

Chapter 7 follows with a summary of the AIL simulation framework using SoS/DEVS as well as future directions for this research.
Chapter 7

Conclusion

Research is an ongoing effort in cooperative robotics, simulations, and the analysis of complex systems working together to achieve a goal. In this chapter the contributions of this work are summarized and then ideas are given to guide future researchers after this thesis.

7.1 Summary

It is important not only to be able to efficiently model and simulate system of systems, but also to integrate these simulations with the systems deployed in the field. It is desirable to maintain the same modeling from simulation to implementation and verify the interoperability of the system within an SoS before a full scale implementation. An XML based systems of systems agent-in-the-loop simulation framework accomplishes this task and provides an important step in the model continuity process. Interoperability is achieved by wrapping the data exchanged among the systems with commonly understood XML tags. This approach avoids the potentially extensive work involved in creating detailed software models of each system in an SoS. DEVS activities provide a flexible link between the simulation world and the physical world for systems that do not support a local DEVS simulator. This
thesis uses the DEVSJAVA simulator which has been shown in other publications to provide the scalability and interoperability towards composability for use in a net-centric environment.

The framework uses the separation of communication and control in the mobile systems’ model as well as the implementation of plug-in communication drivers to readily integrate new and diverse systems. This enables systems to be tested in a cooperative environment with other real and virtual components to verify the proposed abilities of an SoS. The Groundscout robot was successfully tested using the framework and exhibited the ability to work with other virtual robots in a simulation of a robust threat detection scenario. Data for analysis is collected from the broadcasts of each systems in the SoS and then analyzed to determine how often desirable and undesirable conditions were exhibited by the SoS components. The robot team exhibited increased success rates of threat investigation as the team size was increased to include eight members. Test results also show the robot team maintains a low percentage of critical violations in the swarm algorithm.

The framework is able to effectively synchronize the real world events into the virtual simulation world in a timely manner. Scaling the size of the simulated SoS quickly saturates a single workstation as the additional threads for each DEVS model compete for time on the central processor. In addition to contention for CPU time the Java threads also begin to encounter deadlock as they attempt to transmit and receive messages simultaneously among the system models.
7.2 Future Work

The future work of this research falls into two categories: the AIL framework, and applications of the framework. Continuing work on the framework will provide further increases in the efficiency and accessibility of agent-in-the-loopsimulations to test systems. Applications of this work further demonstrates the benefits of an SoS approach and the ability of DEVS and DEVSVJAVA to meet the simulation demands of cooperative robotics.

7.2.1 AIL Framework

The XML structure will continue to be developed into a proposed standard for systems of systems based approaches to analyze and implement large, complex, and distributed systems. XPath has been implemented concurrently in an other research project in the Multi-Agent Biorobotics Lab (MABL) at RIT [72]. This integrates the search queries for more efficient and standard implementation of the XML support architecture. A formal XML schema is needed to expand the reach of this work; consideration and examination of the DoD’s metadata structure is important so the framework will be compatible with this desired standard. Along with the schema, an XSLT will enable multiple SoS configurations to parse data and provide flexible integration options. Additionally, a standard environment database will be deployed as part of the XML SoS DEVS simulation framework. PlayerStage [57] is one such possibility for providing a proven, expandable, open-source environment simulation package.

As these future improvements are implemented, it is desirable to make a package
available that others can use for agent-in-the-loop simulations of robotic systems, and SoS in general. The package would require the user to obtain a license for DEVSJAVA and an optional DEVS simulator that can be embedded on the deployed system. An environment database with a DEVSJAVA wrapper, a flexible XML library for SoS, and communication plug-in drivers for common protocols for interfacing non-DEVS systems would provide easy access to AIL testing abilities for a wide range of research opportunities.

7.2.2 Applications

Work is being planned to use this DEVS framework for AIL simulation of SoS with heterogeneous systems including mobile robots, video cameras, and possibly an unmanned aerial vehicle (UAV) operating outside the lab environment. Addition of AmigoBot robots [73] to the SoS example is in progress. These robots are used in conjunction with a stationary security camera to detect people walking in a monitored area. The robot will then be sent to investigate the location. An updated version of the Groundscout robots is a current senior design project (http://edge.rit.edu/content/P09207/public/Home). The next generation features a fully programmable processor capable of running higher level operation systems, TCP/IP communication over 802.11b/g wireless link, GPS and camera modules for more advanced algorithms.
Bibliography


