Combat System Modeling:
Modeling Large-Scale Software and Hardware Application Using UML

By

MOHAMMAD S. AL-AQRABAWI

Thesis submitted to the faculty of the
Virginia Polytechnic Institute and State University
In partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Engineering

Committee

Dr. Ali H. Nayfeh, Co-Chairman

Dr. Pushkin Kachroo, Co-Chairman

Dr. Abdul-Razzaq R. Habayeb

Dr. Scott F. Midkiff

May 2001
Blacksburg, Virginia

Keywords: Software Engineering, Unified Modeling Language, Combat System, Command and Control

Dedication

For the roots and the fruits, for the leaves that shaded them, and for the soil from which they all came…

For my parents whose roots stabilized me when the land was shaking under… whose love can never fade nor die… and will always stay warm and tender…

For my unborn children… that I already named and loved before seeing… for whom I work to build a better future… better life… and an eternal home…

For my wife… whose shade has always covered me… hand in hand, we walked through the hard times… and hand in hand, we will make it through the harder…

For the soil that we all call home… from which we grew… and to which – someday - we will all be back…

For all of them, I dedicate this effort…

Combat System Modeling:
Modeling-Large Scale Software and Hardware Application Using UML

Mohammad S. AL-Aqrabawi

ABSTRACT

Maintaining large-scale legacy applications has been a major challenge for software producers. As the application evolves and gets more complicated, it becomes harder to understand, debug, or modify the code. Moreover, as new members are joining the development team, and others are leaving, the need for a well-documented code arises. Good documentation necessitates the visualization of the code in an easy to understand manner. The Unified Modeling Language (UML), an Object Management Group's (OMG) standard, is a graphical modeling language used for specifying, visualizing, constructing, and documenting software intensive artifacts. UML, which has been accepted as an industry standard in November 1997, has aided the design and maintenance of object-oriented legacy applications. While the software developers were building UML models for their existing applications as part of a reverse-engineering process, development of next generation software applications started from the models (forward-engineering process). In the forward engineering process, the system's code is specified and constructed from the UML models, which evolve as the system evolves in order to maintain consistent documentation and visualization of the system. Moreover, UML has the power of hiding unnecessary details of the system by the ability to model its different views. This enables visualizing the system at different levels of hierarchy.

This thesis documents how to use UML to model a software-intensive simulation for the combat systems of a fully automated naval "digital ship". This process started with building the use case diagrams based on the system requirements given by the domain experts. Then activity diagrams were used to describe the exact performance of the use cases. The logical

view of the system was built using class, interaction, and activity diagrams. Then, the physical view of the system was built using component diagrams. Finally, an example of the code generation process from the UML models was carried out for one of the system components.

These models are to be maintained as the application evolves. Using UML has aided in building a well-structured object-oriented application, validating the use cases of the application with the domain experts, visualizing and validating the structure of the source code before writing it, communicating between different members of the development team, and providing an easily understandable documentation of the system.

ACKNOWLEDGEMENTS:

I would like to thank all the people that helped me carryout my part of this research and my thesis work.

I greatly appreciate the efforts of my advisor Dr. Nayfeh for his endless support and encouragement during the hardest times of my work. I strongly appreciate the chance he gave me in joining the research that matched my interest. I do strongly appreciate the patience and tolerance he showed all over the time when I worked under him. The freedom he gave me on using the time and resources to achieve the specified goals were a great aid in my success. Finally, the time and effort he spent reviewing my thesis and giving me the feedback are most appreciated.

Special thanks are due to Dr. Habayeb who acted as the domain expert on the material presented in this thesis. I thank him very much for the time he devoted to explain to me the requirements of the Naval clients and validate the diagrams of the use case view of the combat system. His support and encouragement had a great effect on my devotion to the work. Moreover, special thanks are due to him for the time he spent to review my thesis before it was submitted to the advisor committee.

I would like to thank Dr. Kachroo whose efforts brought me to the world of UML and modeling. His stimulation to start and finish my work with the endless race with time had a great effect on finishing in time.

I am thankful to Surendranath Ramasubbu for his suggestion to explain the UML use cases using activity diagrams rather than text, which aided me in showing the use cases in a clear and easy to understand manner. I appreciate very much his building the flow chart of a typical software development process.

I could never forget the endless support and tolerance of my beloved wife Fangfang. She stood by my side all through the hard lonely nights that we had to spend until I finished my work. She has always been such an understanding person.

Finally, never can I find the right words to thank Almighty Allah for giving me life, opportunities, ability, and all of the countless bounties I have ever had, which enabled me to go through this life. Praise be to Allah, the most Beneficent, and the most Merciful.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1:

INTRODUCTION

This chapter provides an overview of the research conducted for a large-scale combat system that involves software and hardware. The work presented in this thesis was conducted as a part of a U.S. Navy sponsored project, called NAVCIITI[1]. The objective and motivation behind the NAVCIITI project are introduced. Then, an introduction to the ship command and control grid is presented. The relevant literature in the field of large-scale modeling and simulation is reviewed and referenced. Finally, an outline of the thesis and the contents of each chapter are discussed.

1.1 Motivation for Current Research

Command and control plays a key role in all battles, especially the new ones because of the use of newer technology. Even though command and control does not take part in the fighting, it is a major force multiplier in the battlefield. An army that is fighting aimlessly without a commander is not different from a mob. Throughout history, the greatest victories were always credited to the success of army commanders. The situation is not much different nowadays. However, the armies are getting larger in size and power. Decisions in the battlefield need to be faster and more efficient. So far, most of the military systems have been automated; electronic sensors and radars for data gathering and launching interceptors are highly automated. The command and control operation is performed by humans. Human efforts are characterized by the fact that they are more error prone, have slower response times, and overall have a higher training and running cost than automated systems. Distributed battlespace makes automated command and control a necessity. This thesis is a part of the

[1] NAVCIITI stands for Navy Collaborative Integrated Information Technology Initiative. The NAVCIITI project is aiming towards building a real-life test environment for a digital ship. More about NAVCIITI and the digital ship follows later in this chapter. < http://www.rgs.vt.edu/navciiti/ >

research efforts supported by the US Navy and performed by the researchers at Virginia Tech. This research effort aims at building a simulated environment for a digital test platform. The test platform will aid efforts towards the automation of command and control systems for the operations performed within the ship, and operation performed with other ships in the fleet. The next section takes a closer look at command and control systems in general.

1.1.1 Overview of Command and Control

Command and control are two different functions joined to provide a higher-level functionality; they are used jointly to imply the process of identifying targets and making decisions in the battlefield. Command is defined by the " DOD Dictionary for Military Terms" [4] as:

> *"The authority that a commander in the Armed Forces lawfully exercises over subordinates by virtue of rank or assignment. Command includes the authority and responsibility for effectively using available resources and for planning the employment of, organizing, directing, coordinating, and controlling military forces for the accomplishment of assigned missions. It also includes responsibility for health, welfare, morale, and discipline of assigned personnel"[4]*

Therefore, command is the force directing and coordinating the fleet towards achieving the mission goals. Command implies a high level of knowledge and situation awareness. Command is the brain that directs the resources in the battlefield.

Control, however, is exercised over a part or an organization of the ship to assure proper execution of all tasks. Control is defined by [4] as:

> *"Authority which may be less than full command exercised by a commander over part of the activities of subordinates or other organizations."[4]*

Both the command and control tasks are carried out by the commanding staff of the ship; therefore, command and control terms are used jointly as one term. A joint definition of command and control is listed in [4] as:

> *"The exercise of authority and direction by a properly designated commander over assigned and attached forces in the accomplishment of the mission. Command and control functions are performed through an arrangement of personnel, equipment, communications, facilities, and procedures employed by a commander in planning, directing, coordinating, and controlling forces and operations in the accomplishment of the mission."[4]*

Command and control is implemented in all systems of the ship organization: combat, operations, supply, weapons, and engineering. The commander should be aware of all the organizations of his ship and partner ships, aircraft, submarines, and ground forces involved in the same mission. This knowledge is referred to as *situation awareness*. This thesis is aimed at modeling combat systems for an emulated ship environment called a "Digital Ship".

The concept of the digital ship implies a complete digitally simulated environment for all components of a Naval battleship. Some parts of the ship can be simulated completely in software, some in hardware using scaled models, and some by interfacing with the actual data from a real ship.

The automation of the commander role in a ship is a rather challenging task. It requires an examination of the different doctrines and strategies carried out by commanders. Replacing the human role in such a vital position would be a big mistake if the automated system were not sufficiently tested. Testing such a complex system needs a good simulation platform. Building such a simulation platform is a software intensive task. Software intensive tasks require good modeling and engineering of the system.

Since the digital ship system contains software simulation modules, as well as scaled hardware and real-time data interfaces, there is a need to develop the simulation model. This thesis

proposes the use of the Unified Modeling Language (UML) as a tool for modeling and designing this large system.

1.2 Research Objective

The objective of this research project is to build a test software for the integration, development, demonstration, evaluation, and testing of automated command and control technologies in the framework of the digital ship. This test bed serves as an integration mechanism for the on-going efforts of the research initiatives of the NAVCIITI project at Virginia Tech.

1.2.1 The Digital Ship

The digital ship should be able to emulate a real-world ship in order to evaluate its performance and behavior under different states, such as, underway and under fire. The digital ship should be able to simulate the operation of a ship under degraded modes of operation.

The test bed of the digital ship will provide the visualization needed to evaluate the operation and behavior of the ship in various modes. Since these states are hard to test in real life, simulation of the ship would be needed.

Simulation as a tool, especially for a large and expensive system such as a ship, has many advantages. In simulations, one can very inexpensively simulate disasters and catastrophic failures without having to destroy any actual hardware. Another advantage of simulation over real experimentation is that one can very easily make modification, which could be potentially expensive and time consuming in real systems.

Effectiveness and validation of simulation environments can be greatly enhanced by providing an immersive virtual environment for visualization. Computer Automated Virtual Environment (CAVE), a three-dimensional virtual reality environment used as a tool for scientific visualization, provides an ideal environment for visualization. This will enable the users to have a closer look at the whole warfare environment surrounding the digital ship and understand the performance of the automated command and control system under testing. This

involves the simulation of the ship's surrounding environment (other ships, aircraft and satellites) along with the ship itself.

1.2.2 The Digital Ship in a Network Centric Environment

The digital ship should be capable of communicating with other partner ships, aircraft, and satellites. This can improve the performance of the whole fleet, even under graceful degradation of the ship. The whole fleet can achieve much more as a team than each element can operating by itself. The digital ship can be represented as a single node in a network centric environment (NCE).

This NCE gives special importance to the automated command and control technology as a force multiplier. The NCE is based on three main grids: sensor grid, C4ISR information[1] grid, and shooter grid. C4ISR stands for command, control, communication, computation, intelligence, surveillance, and reconnaissance. The sensor grid consists of the sensors that provide the data input to the NCE. The shooter grid is composed of the execution units that take the actions and counter actions based on the decision made at the C4ISR grid using the data from the sensor grid. The shooter grid represents the offensive and defensive capabilities of the ship.

1.2.3 The Digital Ship's Complexity

The digital ship is an integration platform that combines the efforts of several research teams. It is an emulation of software and small-scale models of hardware. The whole system will be too complex and software intensive. The project lifetime is expected to be longer than the time required by typical research assistants to finish their academic work. All of these factors add to the complexity of the system, which makes the traditional methods of software development inefficient. It takes a huge effort for a developer to understand code artifacts developed by somebody else.

The Unified Modeling Language (UML) is an effective tool to maintain the continuity in the software/ hardware development process. UML diagrams provide an efficient means of

[1] Also referred to as Command and Control grid

understanding system modules without having to read the actual code. This saves tremendous effort for new developers who join the development team at various times.

1.3 Literature Review

An exhaustive search for related publication was performed at the beginning of this project. This thesis covers two aspects, the military and engineering aspects of the problem. Therefore, the literature search took two parallel directions covering these aspects. The IEEE[1] library was searched mainly for technical engineering papers about modeling and simulation. The INSPEC[2] library, on the other hand, was used to search for both technical and military literature and for statistical information covering each of these aspects. The technical literature search investigated both the traditional way of system simulation and the use of UML for modeling simulation.

The INSPEC library gave a large set of references for military and technical papers. UML papers were searched using keywords like: modeling, simulation UML, and combinations of both. Military papers were searched using terms like: military, communication, equipment, command and control, ships, and combat systems. The literature search tree hits for military and engineering papers are listed on Figures 1.1 and 1.2, respectively.

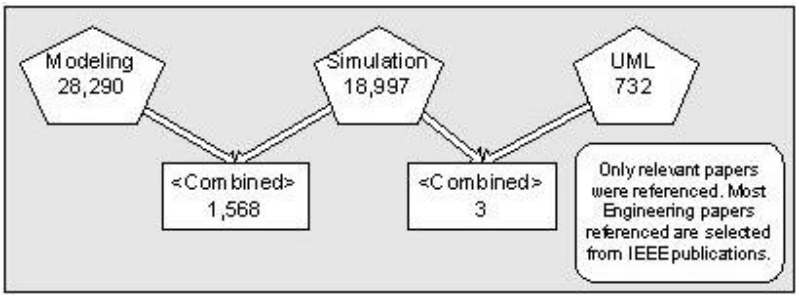

Fig 1.1: Keyword search results for UML literature using INSPEC.

[1] IEEE explore on-line library. < http://ieeexplore.ieee.org/ >

[2] INSPEC: is a database for physics, electronics, and computing literature papers.
< http://gateway1.ovid.com/ovidweb.cgi?T=JS&PAGE=main&D=insz >

Figure 1.1 illustrates the fact that there is not much literature written on using UML for modeling simulation applications (3 hits). On the other hand, the hits of work in modeling simulation applications without referring to UML are much more (1568 hits). This gives an idea about the importance of modeling for simulation applications. At the same time, it emphasizes the fact that the use of UML as a modeling language is a new field of study. The relevant papers will be reviewed in more detail later in this section.

The military literature was surveyed as well. Figure 1.2 gives an indication about the amount of research performed in the field of military applications. Relatively, a large amount of research has been performed in the field of military computation. The command and control is the field of research most relevant to this project and thesis. Several papers were surveyed that cover both combat systems and command and control of military units.

In the following section, the literature related to the military aspects of the project are reviewed first, and then the technical literature about the modeling and simulation is discussed.

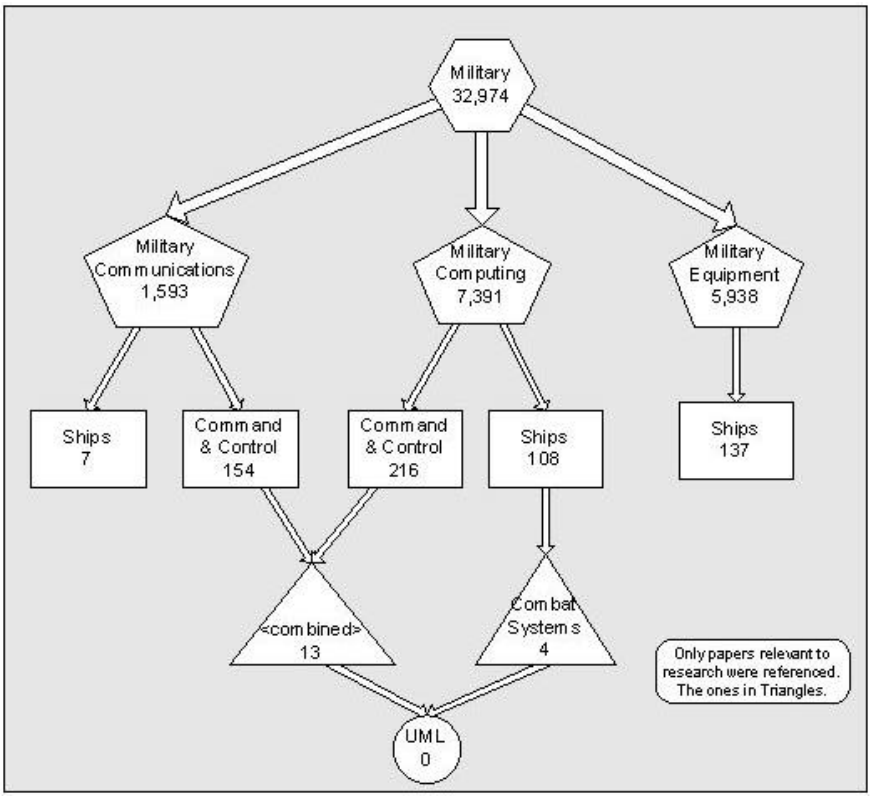

Fig 1.2: Literature results for military keywords using INSPEC

1.3.1 Command and Control in Literature

Hollenbach, W. and Misch, L. in [1] discussed the importance of simulation and modeling in the interoperability of C4ISR. The traditional C4ISR systems, according to [1] were more mission focused, and designed for operational requirements. This paper indicated that the major importance of distributed C4ISR system was in the training of personnel and mission planning. Better training can be achieved by the integration of simulation systems with actual C4ISR systems to provide dynamic environmental data. Mission planning is facing two challenges; one challenge is the variation of events that might occur during the plan execution, the other challenge is the different actions that can be taken to execute the plan. Mission planning has not been totally automated yet. The authors suggested simulation would assist the planner and the decision maker in testing the plan and rapidly re-plan and re-test as the situation changes. This required faster simulation systems for training, hence distributed simulation systems would be needed to achieve the required speed. The paper then discussed the importance of visualization in plan simulation. Visualization would provide a strong tool for plan validation after simulation.

Janowiak, T.J. [2] described a simulation of the Combat Direction System (CDS). CDS is the layer that performs the command and control for combat systems. The CDS obtains data from different sensor systems like radar, sonar and communication systems. The CDS goes through a decision-making process to decide on action to be taken against each threat. Finally, it interfaces with the weapon systems to achieve engagement with the target. This paper introduced the problem of simulating a system that interfaced with other subsystems. The initial step according to [2] was to design Interface Design Specification (IDS). IDS described the communication protocols and messages between the different subsystems. Fidelity for IDS is a major challenge for the subsystem interface. The U.S. Navy has established the Integrated Combat System Test Facility (ICSTF) to validate and test the simulation programs of combat systems. Combat System Simulation (CSS) is the tool used for simulation and testing of CDS. The CSS supports several modules to enable user interface with the system, simulate ship motion and provide a means for setting target tracks and calculating bearing, range and elevation values with respect to the ship local coordinate system.

In [3], Hyer, S.A. et al. worked on a simulation of combat systems for ships to be used as a test bed for different tactics suggested by the US Navy. They simulated the three grids of the combat system: detect (sensor grid), control (command and control grid) and engage (shooter grid). In [3], the authors focused on evaluating the effectiveness of the three grids of Anti-Air combat systems. Airborne threats are faster and harder to detect by defensive systems. Another challenge the ship faces in the battlefield is the unpredictable environmental conditions that might affect shipboard sensors. Radar jamming and electromagnetic interference from other ships add more unpredictability. The authors used random statistical models of dynamic parameters to model unpredictable factors. Then they evaluated each of them based on specific characterizations and requirements.

Detection sensors were evaluated by [3] based on target irradiance, radar cross section and emission characteristics. On the other hand, the command and control subsystem was evaluated based on its reaction time and delay for making decision and communication. Engagement subsystems were evaluated according to their type: hard-kill weapons, such as surface-to-air missiles (SAM) and soft-kill weapons, such as electronic countermeasures (ECM). Countermeasure is defined by the DOD Dictionary of Military Terms [4] as:

> *"That form of military science that, by the employment of devices and/or techniques, has as its objective the impairment of the operational effectiveness of enemy activity "* [4]

Hard-kill weapons were evaluated based on their range, kinematics and probability to kill target. Soft-kill weapons on the other hand, were evaluated in terms of their effectiveness to impair the target and their deployment time.

The authors of [3] then discussed the use of modeling and simulation as analytical tools. They built an anti-air warfare combat system. After a model is drawn and simulated for that system, the simulation results were compared with that of real systems. The overall system was divided into several subsystems. Each subsystem was modeled separately, taking care only of having successful interface between different modules. The authors had some software modules for sensors, environmental effects, scenario generation for targets and several models

for engagement systems. Threat vulnerability models were used to determine the effect of missile warheads in destroying the target.

Jo, K.Y. et al in [5] discussed the modeling of the architecture of command and control systems. The authors addressed several types of architectural elements: operational concept, functional architecture, physical architecture and operational architecture. Functional architecture was modeled by functional decomposition of the system at different levels. This paper covered the interoperability of heterogeneous command and control systems. This paper focused on the communication level modeling of command and control systems. It considered network traffic, distributed processing and centralized control.

Allen, R.T. et al in [6] discussed the need for integrated C4I systems. The authors emphasized the point that weapons need faster response time and minimum human interference. At the same time, the automated response should be as reliable and accurate as possible. The paper discusses the information attributes necessary for delivering trusted information to the C4I system. Information attributes include accuracy, timeliness and completeness. System attributes were then discussed. System attributes are security, flexibility, availability, interoperability and affordability. The authors then discussed the architecture of the C4I system. The system was composed of control and management functions unit, information management functions unit and communication support system (CSS). The authors then discussed the levels of control in the network management: local on-platform control, intra-platform local system coordination, inter-platform network coordination and mission coordination.

Rebbapragada,V.L. in [7] explored the challenges faced in distributed battle management (DBM). According to [7], DBM was more challenging than normal distributed information management because of battlefield dynamics and the need for the availability of timely information to the commander. The author addressed the battlefield dynamics associated with light fighting force, with special emphasis on support for battle commander by providing needed information in combat situations. The paper discussed the DBM concept from the point of view of three areas: Command and control, advanced information management, and distributed environments. The author first discussed Command and Control by defining the

distinction between the two terms: Command, according to [7], indicates the ability to process information with analytical skills to guide to mission accomplishment. Control, on the other hand, is monitoring and measuring performance and correcting deviations from requirements. The paper then discussed the importance of not only the data-information modeling, but also modeling the knowledge base of the battlefield. Object-oriented modeling of the battlefield has been proven to capture the dynamics of the battlefield better than any other data and information models. As the battlefield gets more automated, the amount of information provided to the commander becomes hard to manage. An automated information management system would alleviate this problem. Distributed information management systems perform better than centralized information management in both control and strategic environments. Distributed information management implies that information is coming from different sources at different speeds in different formats, and is a challenging goal that has not been reached yet. The paper examined two approaches for representing distributed environments: distributed simulation and rapid prototyping. The author explains the importance of simulation as a low-cost low-risk method for testing the DBM. Rapid prototyping on the other hand, is needed for focusing the users on relevant information. This leads to a faster decision-making process. The author discusses the Course of Action (COA) analysis example. COA is a time consuming process. Most of the time in COA is spent in information gathering and intelligence, and consequently less time is left for analysis, choosing and executing the selected plans. Huge amount of information is provided to the commander by the automated distributed information management systems, which makes it harder for the commander to come to a timely decision. Rapid prototyping helps in giving only the information needed by the commander to make relevant decisions.

Some of the reviewed papers concentrate on the decision-making process. Holt, J. et al in [8] describe a simulation of human decision-making behavior in naval Anti-Air Warfare (AAW) situations. The authors present a computer model for an AAW officer that was integrated with a combat system. The decision making model is to be used for testing tactical procedures and command and control of Royal Navy destroyer. In this paper, the human model has both formal and informal knowledge about how the task should be accomplished. The model is intended to be error prone to simulate human errors. Task prioritization and scheduling

algorithms are used to model human decision-making process. Cognitive mapping technique is used to elicit knowledge based on information about the decision-making tasks. The time-based aspect of human decision-making is modeled based on Rasmussen's Model of Decision Making[1]. Rasmussen's model is activated whenever new information is introduced. The model goes through knowledge analysis steps through knowledge-based planning steps to execution phase (I am not sure what you mean in this sentence). The Rasmussen's model accommodates faster processing when a new situation is rehearsed or when the time is insufficient for detailed processing. The authors of [8] describe the Structured Knowledge Elicitation (SKE) approach for building the task's knowledge base. SKE involves running test scenarios on experienced AAW officers. The experts were asked about their responses for each step as some sample scenario was running. After the scenario ended, the whole experiment would be performed again. The AAW officers were asked if their response would have changed if they had more information about the situation. The discussion of the paper indicates that the method used was resource intensive. Moreover, modeling the human errors is unwanted in mission-critical decision-making applications.

Qi Yaohu in [9], on the other hand, studied the main areas of Countermeasure Decision-making Supporting Systems (CM DSS). CM DSS are systems used for using information from the enemy's C3I systems to destroy or weaken them. CM DSS include multi-sensor information processing, schemes selection, resources distribution and systems management. The CM DSS use techniques like intelligence, jamming, destruction and secrecy to disable the enemies C3I systems.

In [10], Siliato J.M. et al emphasize the importance of modeling and simulation in developing field-able C3 systems. Simulation models allow for requirement analysis, design evaluation and test planning.

Schwalm N.D. et al in [11] show integrated prototypes for command and control systems based on object-oriented software. The authors include analysis of collaborative decision making of distributed C2 systems. This paper covers information about scenario generation features and it's the corresponding operating characteristics.

[1] A state machine model the represents the process that the human goes through when making a decision.

In [12], Perse R.M. et al discuss the development of an automated tool to enable the ship to fight by continuing to defend itself while under fire[1]. This is achieved by integrating functions of Damage Control (DC) and combat systems. The authors present models of the human information and decision-making processes. The activities during damage scenarios include repairing vital systems and defending against attacks. The system was tested under fire and flood scenarios.

1.3.2 The UML in Literature

In [13], Savino-Vazquez, N.N. and Ruigjaner, R. describ simulation as a three-stage process of model design, model execution, and execution analysis. The paper discusses the mapping of UML to Hierarchical Object Oriented Modeling Approach (HOOMA) (explain what this is) as a way to model the dynamic and static behavior proposed system (Network Queue (what is this and how is it relevant?). The authors specify the decomposition of the system specifications into three domains: Structural domain, interaction domain and computing domain. In structural domain, the static components of the system are modeled and integrated. In the interaction domain, the dynamic behavior of the system is captured. This includes the interaction between different components. Finally, in the computing domain, the methods that carry out the system's behavior are specified.

In [14], Shushen Y. et al present a model for parallel software. In their model, the authors use UML to represent the structure of the system and internal behavior of the objects and interaction between the objects in the system.

UML was adopted by Huang, H.P. and Yeh, C.F. in [15] in the development of a virtual factory (VF) emulator. This emulator is suitable for modeling systems with concurrent and asynchronous behavior by running several instances of the VF emulator.

Kortright, E.V. in [16] describe a method for using Java for simulation. The author used UML as a modeling and simulation language and Java as the simulation platform. The author added controllers to the model in order to generate and handle events of the system. Views are added

[1] The author refer to this situation as "fight while hurt"

to the system for animation and checkpoint recording. The author shows mapping from UML models to Java codes for the proposed simulation system.

1.4 Thesis Outline

This thesis starts by a brief introduction to UML syntax and semantics, followed by the specifications of the digital ship in Chapter 2. The use case view of the combat system is discussed in Chapter 3. Chapters 4 through 6 discuss the logical view of the combat system, which covers the different UML generated diagrams. Chapter 4 introduces the structural domain of the combat system; it includes the package and class diagrams. Chapter 5 explains the interaction domain for the combat system; it includes the sequence and collaboration diagrams. Chapter 6 covers the operational domain for the system, including its activity and state chart diagrams. The physical view (component diagrams) of the system is covered in Chapter 7 along with an introduction to the process of code generation from UML models. Chapter 8 contains the conclusions and recommendations for future work. Appendix A covers the source code listing for selected components of the combat system.

Chapter 2:

BRIEF OVERVIEW OF UML

This chapter discusses the Unified Modeling Language (UML). The syntax and semantics of the UML are discussed. First, the definition of the language, its importance, and use are presented. Then the structure and diagrams are presented. Using UML for system modeling is introduced. Mapping of the digital ship's combat system requirements into UML models is explained.

2.1 Definition and Use of UML

The UML is defined by [17] as a graphical language used for specifying, visualizing, constructing, and documenting the artifacts of software intensive systems. The UML is considered a *language* in the sense that it is a means of communicating ideas. These ideas are mainly about the structure and behavior of a certain system.

UML helps in specifying the system by helping the developer build models of the system that are clear, easy to understand, and fully represent what the designer wants. During the problem specification, the problem needs to be modeled in a way that will be easy to communicate to other members of the team. Every party participating in the project should be familiar with the models written by the others. If the models were graphical, it would be easier for a person to have a wider vision of the system at different levels of hierarchy. This will help one understand, comprehend, and debug the system.

When the models of a system are built successfully, they can be mapped to a source code of some object-oriented programming languages, such as C++, Java, Visual Basic, or tables in a relational database. Mapping the models into a code is known as "forward engineering". Forward engineering is not applicable to all UML graphs. Rather, it is highly dependent on the

level of abstraction and details of the model. For instance, *class diagrams*[1] can be mapped to C++ or Java classes. Class diagrams are UML diagrams, which specify and visualize the classes used in a system and the relationships between classes. It is a good way to show class hierarchy and dependency. UML models the classes in a language independent format, that is, the same model can be mapped to a C++, Java, or VB class. Other diagrams might not be built to construct a code; they might be needed to help the developer understand the system before writing his own code or diagrams. *Use case diagrams* are examples of these diagrams; they are used to capture the behavior of the system from a high level of abstraction.

Finally, the code is generated and tested, and the executables of the software are released. Still this is not the end of the system development lifecycle. Updates are issued all the time; a well-known example of these updates is the service-packs, which the software vendors provide to their customers, even after purchasing the software. These service-packs contain bug fixes of the sold software. Sometimes the requirements of the project might change, even after the release of the product. An example of this situation is more common in a contract-engineering project, where the products are built for a certain customer, the customer needs might evolve, and hence the project requirements would evolve too. This points to the need for a well-written documentation of the system, not only to make it easier for developers to migrate the system to meet the new requirements, but also to enable new team members understand the work that is inherited from other members. The longer the product lifetime is, the more important it is to have understandable and unambiguous documentation. Long-term projects have members leaving and joining the teams more often. New members need to learn and understand the "legacy application" before they can start working.

Even though UML is mainly designed for modeling software intensive systems, it has also been used in a broad range of other non-software systems, such as e-business, financial applications, information technology, and supply-chain applications [17].

[1] More about classes and class diagrams later in this chapter

2.2 Structure of UML

In this section, the building blocks of the UML diagrams are discussed and their use and syntax are introduced. *"The Unified Modeling Language: User Guide"*[17] is the main reference for this section.

2.2.1 UML Building Blocks

The building blocks of UML can be classified into three types: things, relationships, and diagrams [17]. Diagrams of UML are built of a group of things that are tied together using relationships. Diagrams visualize different aspects of the system. They vary in the level of abstraction, content, and structure. No one diagram will be enough to model the whole system. There is always a need to have several diagrams, each of which concentrates on a single view of the system.

2.2.1.1 Things in UML

Things in UML are classified according to their use. Things can be *structural*, *behavioral*, *grouping*, or *annotational* [17].

Structural things represent the static parts of the system. In other words, they represent the building blocks of the system. Structural things include *classes*, *collaborations*, and *use cases.*

Classes in UML are similar to the classes in all object-oriented programming languages. A class is a description of objects that have some attributes and characteristics in common. Classes have hierarchical relations, discussed later. In general, child classes inherit attributes from their parents. Class notation in UML includes the class name on the top rectangle, its attributes in the middle rectangle, and its operations in the lower rectangle. An example of a class is a ship. A ship is a class of vehicles that travel on the surface of water. All ships in the world have some attributes and functions in common. All ships have specific capacity, size, and speed. All ships sail and dock. The class "ship" does not refer to any specific ship (instance); it is rather a general abstract term that gathers all objects (instances of a class). The digital ship under construction is an object (instance) of

the class “ship”. Figure 2.1 shows the UML notation of an object (right) and a class (left) and its attributes and operations.

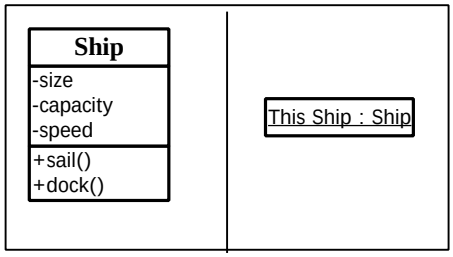

Fig 2.1: UML notation for class (left) and Object (right)

Association classes are a special kind of class; these classes are linked to association relationship[1] between normal classes to specify the nature of this relation between these clases. The association classes are rendered as a normal class with a link to the association this class specifies.

Collaborations in UML have both structural and behavioral extents. Collaboration is made up of several objects and relations that interact together to achieve one behavioral task of the system. A decision making process of the shooter grid of the ship is modeled by a collaboration; this collaboration contains all of the objects that represent the decision making process. Figure 2.2 shows an example of the UML notation of collaboration things.

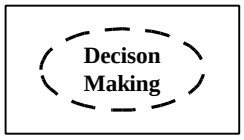

Fig 2.2: UML notation for a collaboration

Components in UML are used to model the physical aspects of the system. By modeling the source code, files, libraries, and executable versions of the code, the components can guide the code generation process from the UML models. Figure 2.3 shows the UML notation of a component.

[1] More about associations later in this chapter

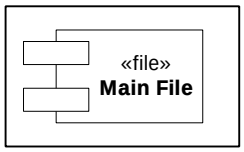

Fig 2.3: UML notation for a component

Use cases are used in UML to capture the behavior of the system at a high level of abstraction. A use case represents a sequence of actions taken in response to an external actor. A sequence of use cases introduces what that actor expects from the system. Collaborations realize use cases since they are more abstract than the collaboration. The collaboration in turn is the link between the behavior of a use case and the structure of the class objects. An example of a use case is "make decision", which is realized by the decision making collaboration. Here the name of the use case implies an action more than a structure. Use cases are classified as structural things because they are used as building elements for use case diagrams that capture the structural behavior of the system. Figure 2.4 shows UML notation for a use case.

Fig 2.4: Use case's notation in UML

Behavioral things in UML represent the dynamics of the system. They are used in visualizing the system's operations and interactions. Behavioral things build the semantics of the system's behavior; they include *interactions* and *state machines*.

Interactions represent the exchange of messages between different objects of the system. This exchange of messages is carried out to achieve a specific task. Interactions are the building blocks for collaborations.

State machines represent the behavior of an interaction or an object. State machines capture the sequence of event-triggered states; the state of an object or an interaction does

not change on its own. An example of a state of the digital ship is "Under Fire." The ship enters this state when it is under attack, and it might change this state in case the attackers are destroyed. Figure 2.5 shows the UML notation for a state.

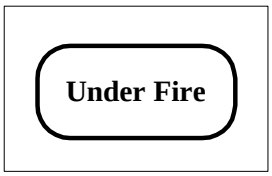

Fig 2.5: State's notation in UML

Grouping things in UML are mainly packages. A package is used for organizing elements of the model into related groups. Packages in the UML context are similar to folders in the operating system; folders are used by the operating systems to group and organize files. An example of packages is "Combat Systems." This package contains all of the classes and diagrams that build the structure of the combat system of the ship. Figure 2.6 shows the UML notation for a package.

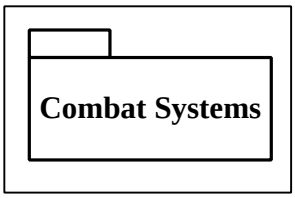

Fig 2.6: Package's notation in UML

Annotation things in UML are mainly notes; notes are used to explain diagrams or to add comments and constraints to the diagrams. The notes are expressed in textual language, which is best understood by a human reader who tries to understand the model. Figure 2.7 shows the UML notation for a note.

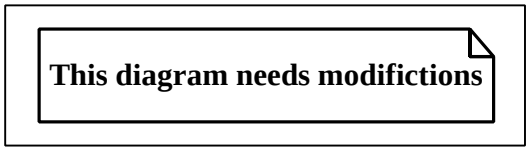

Fig 2.7: Note's notation in UML

2.2.1.2 Relationships in UML

Relationships in UML are the mechanisms that connect related things to form a diagram. There are four relationships in the UML notation: dependency, association, generalization, and realization.

A **dependency** relationship indicates that any change in the destination object will affect the behavior of the source object. In UML, dependency is rendered as a dashed arrow from the source thing (dependant) to the destination thing. Figure 2.8 shows the notation for a dependency relationship in UML.

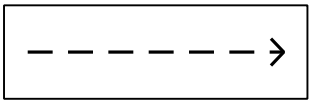

Fig 2.8: Dependency notation in UML

Associations in UML are structural relationships that represent the links between pairs of things. Unlike the dependency relationships, the association relationships might occur between peers. *Aggregation* relationship is one type of association; it represents decomposition of a whole at one end into its parts at the other end. The ends of the association relationships can have text, which indicates the role of each end with respect to the other. Moreover, these ends can have numbers, which indicate the *multiplicity* of each end with respect to the other; these multiplicity numbers can be indicated as a range (3…7) or even as wild cards[1] (*). Figure 2.9 shows an example of the notation for association (left) and aggregation (right) in UML. From the Figure, one can note that the ship could have only one commander (left), but it can have one or more combat systems (right). The combat system is part of the ship (right), but the commander is not (left).

[1] Wild cards are characters used to represent the "ANY" logic in a string of characters. In most programming languages, wild cards are rendered as (*). An example of a wile card use is (*ed): for any string ending with (ed).

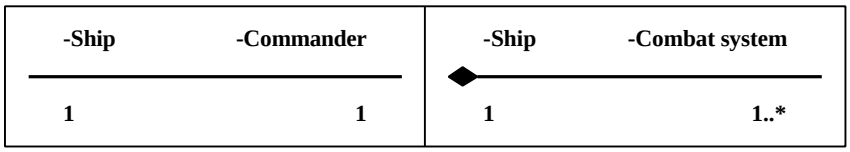

Fig 2.9: UML notation for Association [left] and aggregation [right]

Generalization in UML indicates the relation between a parent and a child. This relationship is used to represent the *inheritance* in object-oriented programming. A child inherits some characteristics of its parent in addition to its own characteristics. For example, the class "*vehicle*" is defined as a machine used for transportation. All transportation machines are called "vehicles" and are instances of the class "vehicle." Ships are a special kind of vehicle that moves on the surface of water. The "ship" is another class that joins the attributes and operations of all objects that move on the surface of water. Meanwhile, every ship is a vehicle, hence a "ship" is a specialization of "vehicles" and a "vehicle" is a generalization of "ships." The "ship" class contains all of the attributes and operations of general vehicles, like speed. On the other hand, ships have specialized attributes and operations, like dock and sail. In object-oriented programming, the class *vehicle* is considered the parent (generalization) of the class *ship*. In the UML notation, the "ship" class is the source of a generalization relationship and the "vehicle" class is the destination. Figure 2.10 shows an example of the use of generalization to link a class ship to its parent vehicle.

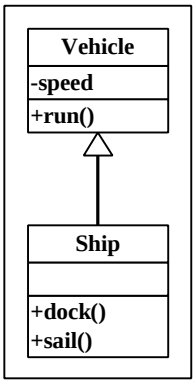

Fig 2.10: Generalization relationship in UML

Realization in UML is the relationship between two classifiers where one of them carries out (realizes) the behavior specified by the other. The best example is the relation between a use case and collaboration. The "*make decision*" use case is realized by the "*decision making*" collaboration. The use case specifies what is required at a higher level, and the collaboration carries out the decision making process by specifying the interaction of objects to achieve that task. Figure 2.11 shows the notation for a realization relation in UML. The "Decision Making" collaboration realizes the "Make Decision" use case.

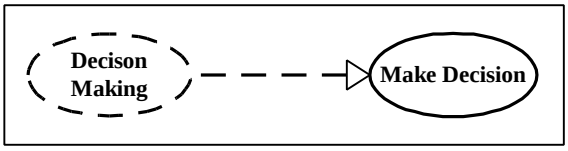

Fig 2.11: Realization notation in UML

2.2.1.3 Diagrams in UML

Diagrams are used in UML to show the structure and behavior of the system. Diagrams are usually drawn as graphs with things as vertices and relationships as edges. Some diagrams show the structure of the system by showing how different things are related and connected by relationships. Other diagrams emphasize the behavior of the system by showing how different things interact with related things. As indicated at the beginning of this chapter, the system cannot be totally specified by one graph. Each graph captures one view of the system. Different views of software systems are discussed later in this section. Next, we describe the different UML diagrams.

Class diagrams: Class diagrams capture the static view of the system and include classes, collaborations, and relationships. Class diagrams might show class hierarchy, inheritance, and realization.

Use case diagrams: Use case diagrams show the static use case view[1] of the system. They are made of use cases and actor classes. Actors are a special class kind, which are external to the system and are rendered in UML as stick men. They usually model the action of the system's

[1] More on use case view later in this section.

users. Sequences of operations are usually triggered by actors, which in turn triggers other use cases. The use cases work together to model the overall system task.

Interaction diagrams: Interaction diagrams are made of objects (instance of classes), their relationships, and messages. They describe the collaboration things, which in turn realize the use cases. The dynamic aspects of the use case view are captured by interaction diagrams. Interaction diagrams include both sequence diagrams and collaboration diagrams. Sequence and collaboration diagrams are isomorphic; that is, they both represent the same information, but each of them emphasizes some aspect of the system. Next, we describe the different interaction diagrams.

Sequence diagrams: Sequence diagrams emphasize the behavioral aspects of collaboration. They model the flow of control, emphasize the time ordering of messages and show the lifetime of transient objects. Figure 2.12 shows an example of a sequence diagram.

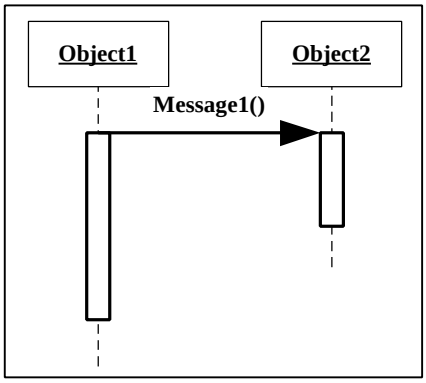

Fig 2.12: Sequence diagram

Collaboration diagrams: Collaboration diagrams show the structural aspects of collaboration. They emphasize the structure of objects that send the message more than the time line of sending the messages. Collaboration diagrams can model iterations, branching, and concurrency better than sequence diagrams. Figure 2.13 shows an example of a collaboration diagram.

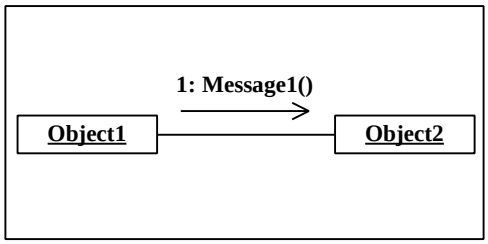

Fig 2.13: Collaboration diagram

State chart diagrams: State chart diagrams are equivalent to the well-known state machines. They are made up of states, the activities performed at each state, and the transition from one state to another. Transitions occur due to an event in the system. States are all stable; the system stays in the current state unless an event occurs. State chart diagrams capture the dynamic aspects of classes or collaborations.

Activity diagrams: Activity diagrams are similar to state chart diagram. However, the transition from one activity to another occurs due to the finishing of the first activity. Flow of control in different functions of the system can be better viewed in the activity diagrams.

Component diagrams: A component diagram guides the process of code generation from the UML models by linking the logical[1] elements into physical source code files. Component diagrams include packages, components, and dependency relationships between them. The dependency relationships are needed to provide the visibility between the different components; a dependency relationship would result in a "#include "" line in C++ files.

2.3 Modeling and the Software Development Process

A successful software design process has three major characteristics: it should be use case driven, incremental, and iterative. In use case driven systems, the use cases are the major

[1] Logical view of the system is the term used to describe its abstract classes. These classes would otherwise have no link to the source code, which realizes the class diagrams. The source code is modeled by component diagrams.

artifacts that guide the development, testing, and validation of the system's behavior towards the desired behavior. An incremental process starts with a low level of details and complexity. The complexity and amount of detail increase as the system evolves, which allows it to grow as the requirements grow. The iterative process is the one that results in several executables. As the new code is written, it is run, tested, and validated. Then another version of the code is developed to fix the problems (if any) in the previous code and improve the performance of the system. These three characteristics apply to UML models too.

Before modeling of a software system using UML, one needs to know the requirements of the end user or the client. The process of building a well-engineered software system consists of two stages: the use case stage[1] and the architecture stage. The use case stage is the stage where the system's behavior is specified by the end users. This stage is not concerned with the internal structure of the system. Rather, it is only concerned with the interface and behavior of the system. This stage takes iterative steps between the end user and the system architect until the user agrees on what he expects the system to achieve in response to his/her requests. UML supports modeling of the static aspects of this stage using the use case diagrams and its dynamic aspects using activity diagrams. Activity diagrams can be used to describe the behavior of some use cases as a sequence of activities.

When the users agree on the architect's suggestions, the system enters the second stage: the architecture stage. The process of specifying the system in the architecture stage can be broken down into three domains [13]: the structural domain, the interaction domain, and the computational domain. In the structural domain, the static aspects of the system are specified. Class diagrams are used to capture the aspects of this domain. The interaction domain specifies the interaction between different instances of classes (objects) in the system and shows its behavioral scenarios. Interaction between objects occurs when messages are sent between them or when their states change in response to certain messages. Message sending is modeled by interaction diagrams: sequence and collaboration diagrams. The computational domain, on the other hand, specifies the methods and functions that classes use to achieve their required functionality. State chart and activity diagrams are the best way to model the

[1] Also called use case view

computational domain. In this thesis, UML models are drawn for the system starting from the use case driven stage and ending with the computational domain. In Chapter 6, code generation is shown for sample parts of the system.

The UML modeling process is incremental; that is, the system is first captured at a high level of abstraction. As the design evolves, more and more details are introduced. These incremental steps continue even after the design is fully specified, because maintenance would always be needed. This method enables the system architects and designers to stay focused at the level of information they need without getting overwhelmed by details from the very beginning.

The code generation process follows the modeling process in UML. As the models evolve, the code should evolve, and vice versa. Any changes in the application's code should be reflected in the models, and vice versa. This would guarantee that both the models and the code stay up to date. Whenever someone joins in, he/she can easily understand the code by reading the models. Figure 2.14[1] shows the flow chart for a typical software development process.

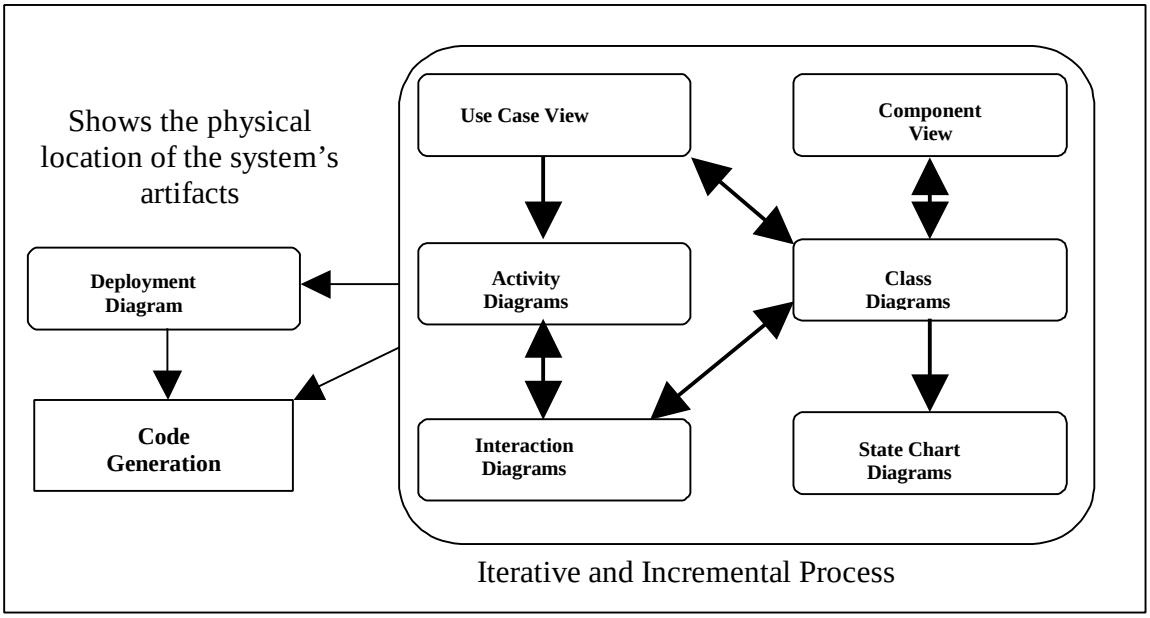

Fig 2.14: Flow chart of a typical software development process

[1] Courtesy Surendranath Ramasubbu GRA at VT

2.4 A Systematic Approach to Model the Digital Ship Using UML

This section discusses the approach taken in modeling the digital ship using UML. This discussion starts with a description of the project specifications that need to be modeled. Then mapping of the digital ship's requirements to the UML is discussed.

2.4.1 Digital Ship's Specifications

In this thesis, the combat systems of the digital ship are modeled. The behavior of the combat systems - from the detection of the target to the firing of the interceptor - goes through three interconnected grids: the sensor grid, the command and control grid, and the shooter grid. Figure 2.15[1] shows these three interconnected grids.

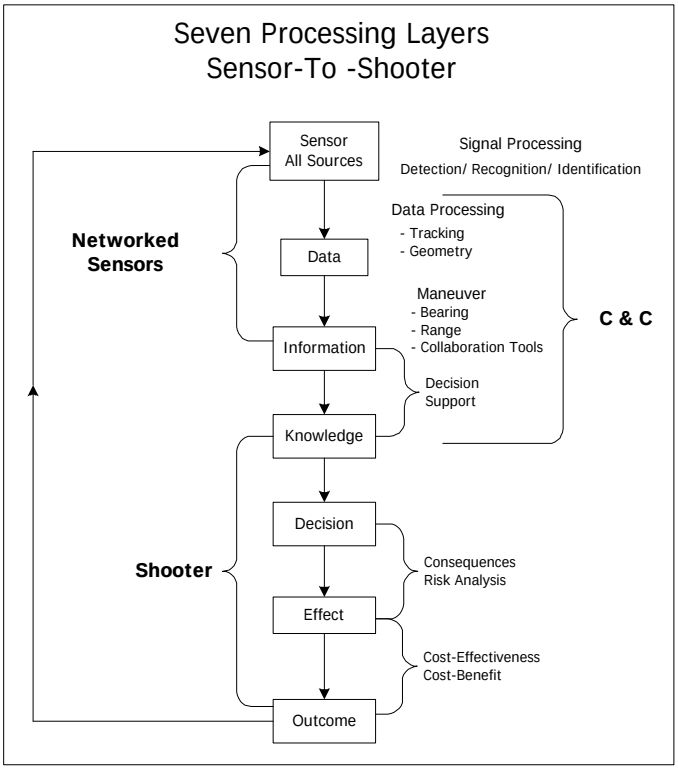

Fig 2.15: Seven processing layers

[1] Courtesy Dr. A. Habayeb

In the sensor grid, the surveillance and reconnaissance data are collected by RF sensors and intelligence scanning the sky, sea, and land for surrounding potential threats. The data from these sensors are processed in order to obtain information about the target. This information is used for detecting the target and computing its range and bearing. This piece of information is sent to the command and control grid.

The command and control grid initially processes the information from the sensor grid along with data from the acquisition sensor[1]. The acquisition sensor is a higher resolution sensor than the surveillance sensor; it can use bearing and range information from the sensor grid to obtain higher resolution data about the target. After the information processing stage, the command and control grid is able to recognize the target by knowing its intent and geometry. This knowledge is processed to identify the target and create the situation awareness of the environment. The command and control grid triggers the process of the shooter grid.

The shooter grid goes through the loop of making the decision about the target; this decision is based on the evaluation of the expected outcome and the processing of the consequences and effects in case an action is taken. If a decision is made for firing an interceptor, damage assessment will be needed to estimate the level of damage that occurred to the target. Firing of the interceptor triggers the effect processing stage, where the damage assessment occurs and the draw down[1] of the target's capabilities is evaluated. The effect processing stage triggers the outcome evaluation stage, which determines the success or failure of the engagement and the need for further actions.

Damage levels of targets are categorized as

- 0-kill: (Zero kill) if the target is not affected at all by the interceptor.
- B-kill: (Base kill) if the target suffers enough damage that it cannot return to base.
- F-kill: (Fighting kill) if the target cannot fight anymore, which implies that it does not pose any more danger to the ship.

[1] Also know as the "target sensor"

- M-kill: (Mobility kill) if the target cannot move.
- K-kill: (Killed) if the target is destroyed.

Again, decision-making is revisited, but this time to decide whether or not to take any further action against the same target. Choosing when to stop firing at the target is highly heuristic and is highly dependent on the overall situation awareness of the current ship's status and its weapons and supplies. Knowledge about nearby partners and their capabilities will also affect the engagement decision. So far, commanders make these decisions based on their own experience; many times these decisions are not optimal. There are no clear-cut methods for determining when engagement is preferable over non-engagement; the alternatives for different situations are unlimited. This raises the importance of simulation; the digital ship will be used as a test bed to evaluate how these decisions affect the performance of the fleet in different situations and missions. This of simulation can assist automated commanders, and even human commanders, of ships in making their engagement decisions.

2.4.2 Mapping the Digital Ship's Requirements to the UML

In the process of modeling the command and control systems of the digital ship, communications with the domain expert, Dr. A. Habayeb, set the requirements of the naval clients for this project. These requirements were modeled as use case view; this view is composed of use case diagrams and the activity diagrams describing them. Three use case diagrams were built for the ship's combat system: one for the sensor grid, one for the command and control grid, and one for the shooter grid. This division aided in keeping the sizes of the diagrams small and manageable and in focusing the attention of the person who reads these diagrams on one subsystem at a time and its interactions with the other subsystems. The behavior and actions carried out by each of the use cases were described by activity diagrams. The structural domain for the organizations of the digital ship was built using package diagrams. Five packages were used to represent each of the ship's organization: combat system, operations, weapons, engineering, and supplies. Then package diagrams were built for the combat system; one package was used to group the components of each grid of the combat

[1] Draw down is an indication of the degradation in the abilities of the target after being hit by an interceptor.

system. Class diagrams for each grid were built. The choice of classes in an object-oriented programming language is an iterative process. The initial step included considering the objects necessary to carry out the processes specified by the use cases. Similar objects were grouped together into classes, as the system evolved, more classes were added and removed. On the other hand, interaction diagrams were drawn for each use case to model the interaction domain of the system. Activity and state chart diagrams were built to model the computational domain of the system. Finally, component diagrams were built to model the physical implementation of the system. These component diagrams were used for generating the code for the combat systems.

Chapter 3:

USE CASE VIEW FOR THE COMBAT SYSTEMS

This chapter discusses the development of the use case views for modeling the combat system of the digital ship. The use case view consists of use case diagrams that describe the behavior of the ship and the activity diagrams that describe their behavior. They are driven by the sequence of events' (SOE)[1] requirements. Use case diagrams of the three grids of the combat system are discussed and explained in this chapter.

3.1 Project Specifications

The initial stage of this project was to learn from the domain experts how the system works. This provided the insight for generating the SOEs. Understanding how the system responds to certain stimuli created by the SOE led to structuring the system's use cases. They were grouped in use case diagrams. According to the domain expert description discussed in Section 2.4.1, the combat system is partitioned into three overlapping grids. Next, we discuss the requirements for each grid and their UML use case models.

3.2 Sensor Grid

The sensor grid collects intelligence, surveillance, and reconnaissance data about the targets surrounding the ship. The data is processed into information about the target range and bearing, and sent to the command and control grid.

Figure 3.1 shows the use case diagram for the sensor grid in the combat system. Here, the system boundaries are specified. The sensor grid is responsible for converting the signals for

[1] Sequence of events refers to the events followed in building the knowledge base from sensor to shooter grids.

each of the targets into data and processing it to obtain information about the target's range, bearing, and kinematics. Therefore, the nodes: "Use Surv Recon Sensor", "Use Target Sensor", "Use Fire Control Sensor", and "Process Data" establish the use cases of the sensor grid. On the other hand, the "DSA[1] Command and Control" and "DSA Target" actors are external to the sensor grid; that is why they are modeled as UML actors[2].

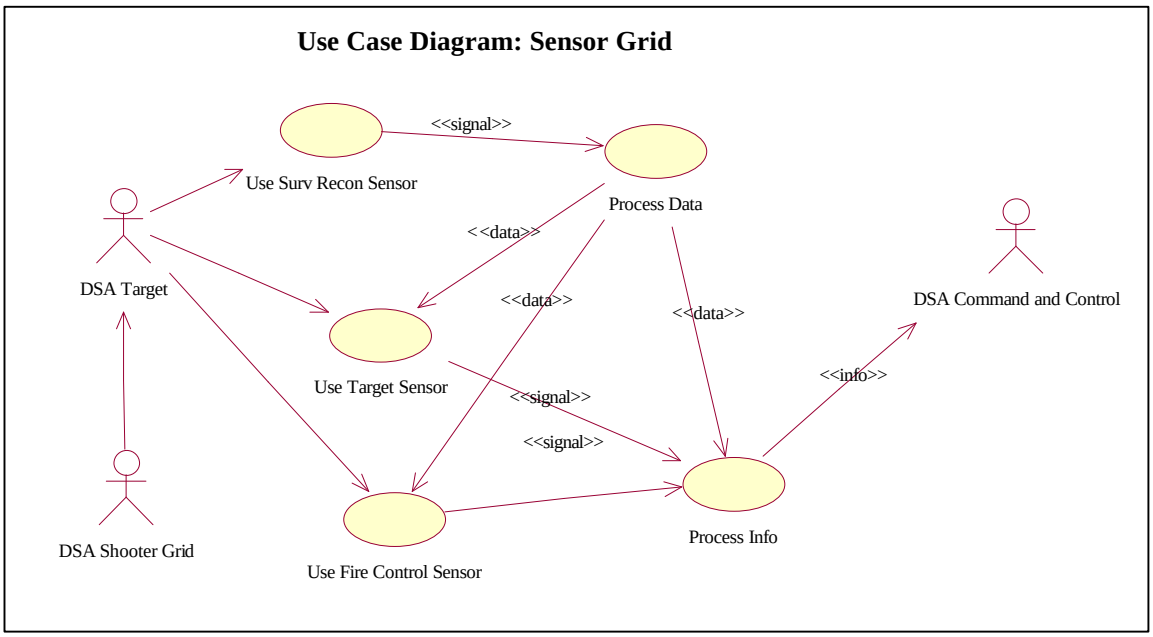

Fig 3.1: Use case diagram for the sensor grid

As the diagram shows, the SOE of the use case is triggered by the target as an actor external to the system. The target starts the "Use Surv Recon Sensor" use case, which in turn triggers the "Process Data" use case. After the latter finishes processing the data from the surveillance and reconnaissance, it detects the target. Target information (bearing, range, and velocity) are then sent to the "DSA Command and Control" to process the information into knowledge to recognize the potential target. The information from the "Process Data" use case are sent to the "Use Target Sensor" and "Use Fire Control Sensor" to direct them to the location of the target so that they can return higher resolution data (images) about the target.

[1] DSA is a prefix that stands for Digital Ship's Actor.

[2] Actors in UML are rendered as stickmen.

Each use case of the system has textual description, which represents the tasks to be achieved by it and the flow of events in response to a request from the actor. This description is modeled by activity diagrams, which give a guideline for the interaction diagrams to be used later to specify the interaction domain of the system. Some use cases represent different scenarios; each of which has its own textual description and interaction diagram. These scenarios are labeled as "main flow of events" and "exceptional flow of events" to refer to the common scenarios that happen typically and those that happen due to an exceptional action by the actor, respectively. The description should state clearly how the use case is triggered and how it ends.

3.2.1 The "Process Data" Use Case

The behavior of the "Process Data" use case is described textually next.

Main flow of events: This use case is responsible for processing the data from the surveillance and reconnaissance sensors to obtain information about a potential target. This information includes the bearing, range, and speed of the target. The use case ends by sending the resulting information to the command and control system, target sensors, and fire control sensors. Goal: Detect Target.

Accordingly, the use case only describes what needs to be done with no reference to how to do it. The end user would utilize the use case to calculate the range and bearing of a potential target whenever it is triggered by data arriving from the surveillance and reconnaissance sensors.

This textual behavior provided by the domain experts needs to be described in a more understandable graphical diagram. Activity diagrams are suitable for showing this behavior. Figure 3.2 shows the activity diagram of the "Process Data" use case; it explains the activities that happen when processing data.

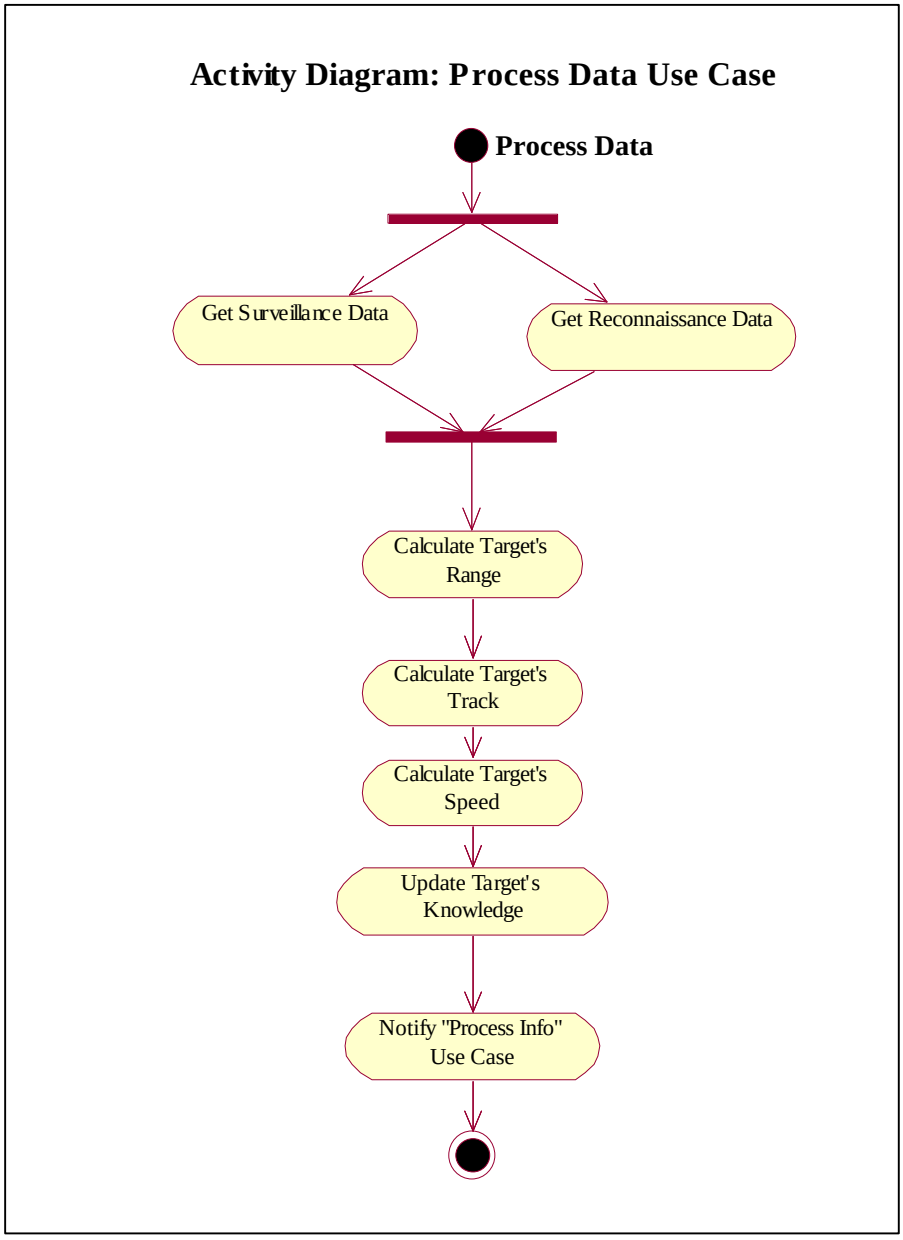

Fig 3.2: Activity diagram for “Process Data” use case

3.2.2 The “DSA Target” Actor

This actor represents any potential target that can be detected by the sensors on the ship. The target is physically external to the ship; hence, it is external to the shooter grid. This is why it is modeled as an actor. It triggers the sensors’ use cases: “Use Surv Recon Sensor”, “Use

Target Sensor", and "Use Fire Control Sensor". Therefore, after the target starts interacting with the system, these use cases are started.

3.2.3 The "DSA Command and Control" Actor

The "DSA Command and Control" actor represents the link to the command and control grid, which is part of the combat system, but not part of the sensor grid. Each of the "Process Data", "Use Target Sensor", and "Use Fire Control Sensor" triggers some use cases in the command and control grid (subsystem), this explains why the earlier use cases interact with the "DSA Command and Control" actor.

3.2.4 The "Use Surv Recon Sensor" Use Case

The "Use Surv Recon Sensor" explains how the surveillance and reconnaissance sensors are used to gather data about the target. These sensors survey the sky to gather data for each object in their scan range. The radar is an example of a surveillance sensor. Surveillance sensors are low-resolution sensors; they only return signals enough to detect the potential targets. The processed information about the target cannot tell more than the range, bearing, and sometimes target speed. The surveillance and reconnaissance sensors generate electrical signals; these signals need to be processed in order to obtain useful data. These data are sent to the "Process Data" use case for processing.

The textual description of the "Use Surv Recon Sensor" use case is:

Main flow of events: This use case keeps polling periodically[1] returned signals from surveillance sensors and radars for any potential targets. The use case ends if no targets are detected.

Exceptional flow of events: If the returned signals from the sensors indicate the existence of a potential target, the use case filters these signals to obtain only the relevant data about the target. These data are sent to the "process data" use case. The use case ends after the data are sent to the "Process Data" and when there are no further targets to detect.

[1] This process needs to run periodically in order to have up to date data from sensors.

Figure 3.3 shows the activity diagram for the exceptional flow of events for the “Use Surv Recon Sensor” use case.

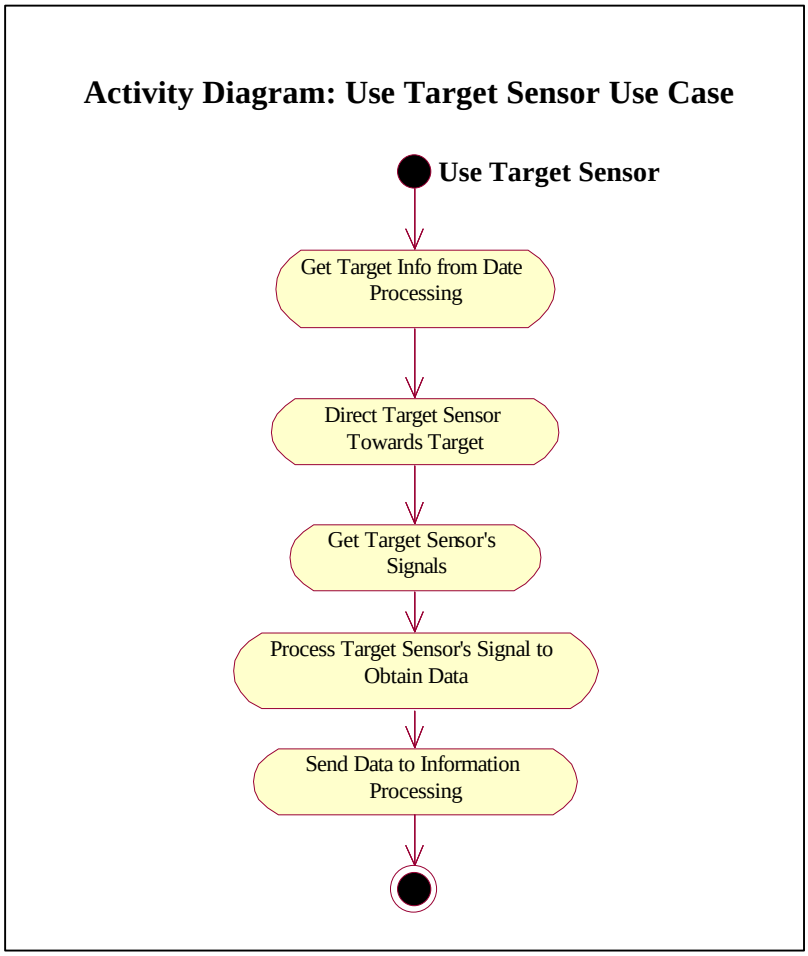

Fig 3.3: Activity diagram for the “Use Surv Recon Sensors” use case

3.2.5 The “Use Target Sensor” Use Case

The “Use Target Sensor” use case is triggered when the target is detected by the “Process Data” use case. The “Process Data” use case sends the information about the target to this use case. The information received deals with the target range and bearing. The target sensor can

then obtain a higher resolution image of the target. The image data of the target sensor are used by a "Process Info"[1] use case inside the command and control use case.

The textual description of the "Use Target Sensor" use case is:

Main flow of events: This use case starts when a potential target is detected by the "Process Data" use case. The input to this use case is information about the target range and bearing from the "Process Data" use case. This information is used to obtain a higher resolution data about the target. The use case ends when the high-resolution data is sent to "Process Info" use case inside the " DSA Command and Control" subsystem.

The activity diagram for the "Use Target Sensor" use case is shown in Figure 3.4.

[1] More about "Process Info" use case later in Command and Control use case diagram.

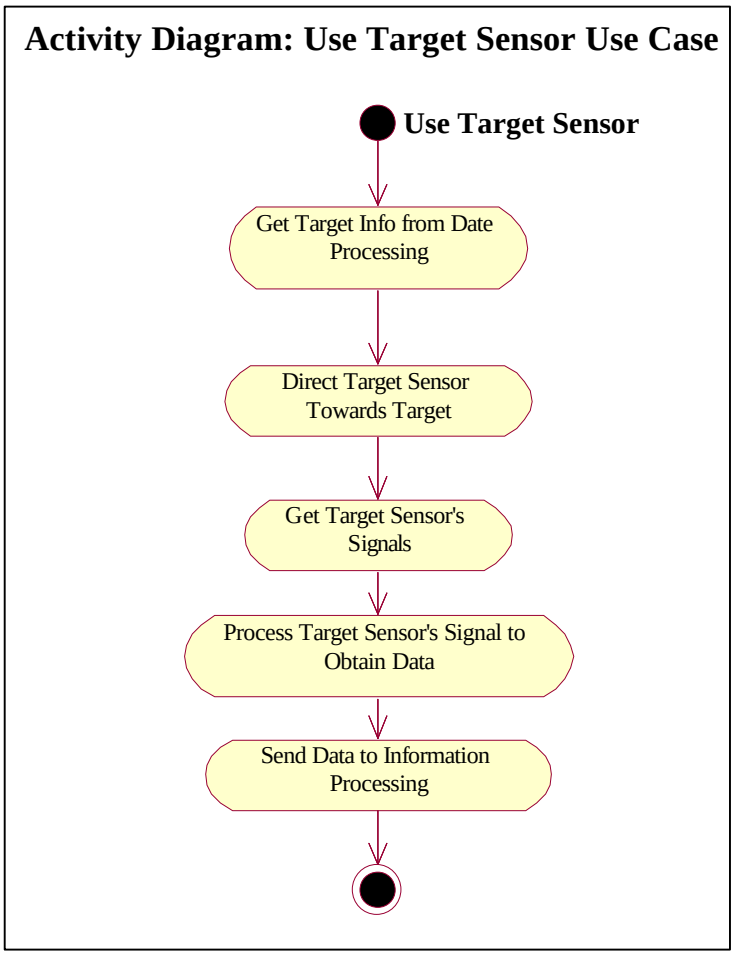

Fig 3.4: Activity diagram for the "Use Target Sensor" use case

3.2.6 The "Use Fire Control Sensor" Use Case

The "Use Fire Control Sensor" use case is similar to the "Use Target Sensor" use case in its task and in the way it is triggered. However, this use case is responsible for initiating the fire control sensors, which have higher resolution than the target sensor. The "Use Fire Control Sensor" use case sends its output to the "Process Knowledge"[1] use case in the "DSA Command and Control" subsystem.

[1] More about "Process Knowledge" use case later in the Command and Control use case diagram.

Main flow of events: The “Use Fire Control Sensor” use case starts when a potential target is detected by the “Process Data” use case. The input to the “Use Fire Control Sensor” is information about the target range and bearing from the “Process Data” use case. It uses this information to obtain higher resolution data about the target. The use case ends when the high-resolution data are sent to “Process Info” and “Process Knowledge” use cases in the “DSA Command and Control” subsystem.

Figure 3.5 shows the activity diagram for the “Use Fire Control Sensor” use case.

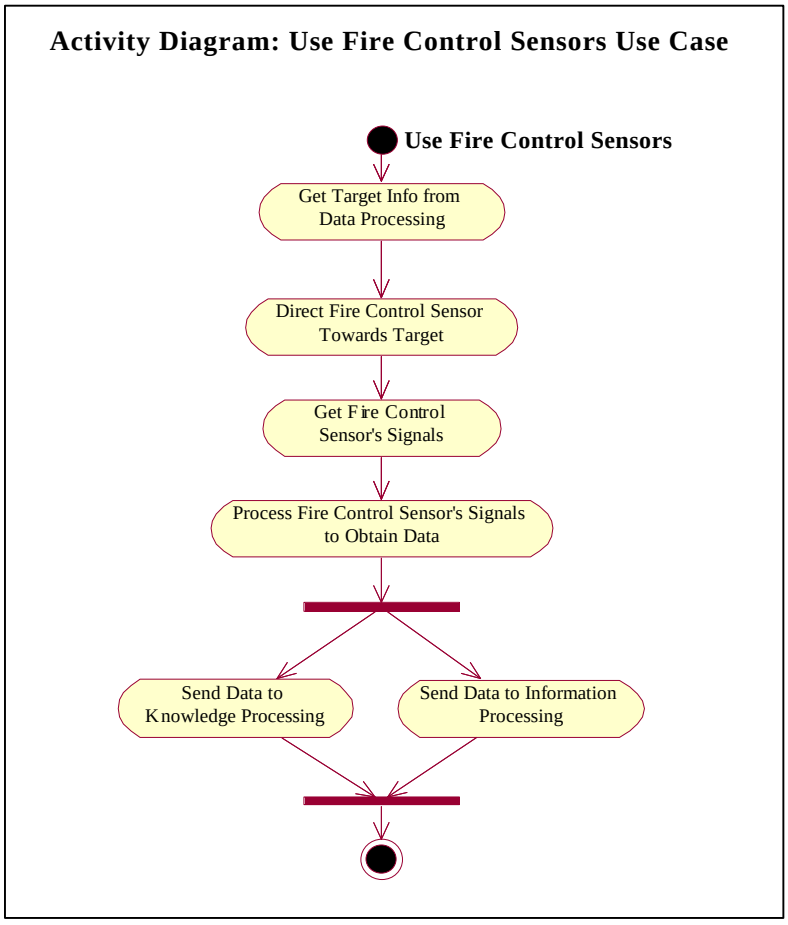

Fig 3.5: Activity diagram for “Use Fire Control Sensor” use case

3.3 Command and Control Grid

The command and control grid processes the data to recognize and identify the target. The identification occurs on different layers. First, the information coming from the "Process Data" use case triggers the information processing use case "Process Information" along with the high-resolution data from the "Use Target Sensor" use case. The "information processing" layer will result in knowledge that recognizes the target. Knowledge about the targets can indicate its type (aircraft, missile, or another ship) from target geometry, image, and intent. This knowledge is fed in turn to the knowledge processing use case: "Process Knowledge" along with the high-resolution data from the "Use Fire Control Sensor" use case. The "Process Knowledge" use case identifies the target, including its exact model and capability. Having identified the target, the "Process Knowledge" use case triggers the decision making process in the "DSA Shooter Grid" subsystem. Figure 3.2 shows the use case diagram for the command and control system.

The command and control grid has two use cases and four actors. The use cases represent the functionality of the Command and Control system. The actors in this diagram are "DSA Data Processing Unit", "DSA Target Sensor", "DSA Fire Control Sensor" and "DSA Shooter Grid". The "DSA Data Processing Unit", "DSA Target Sensor" and the "DSA Fire Control Sensor" actors are part of the Sensor grid subsystem. Now that they are external to the command and control subsystem, they are modeled as actors. The "DSA Shooter Grid" actor represents the shooter grid subsystem, which is external to the command and control subsystem.

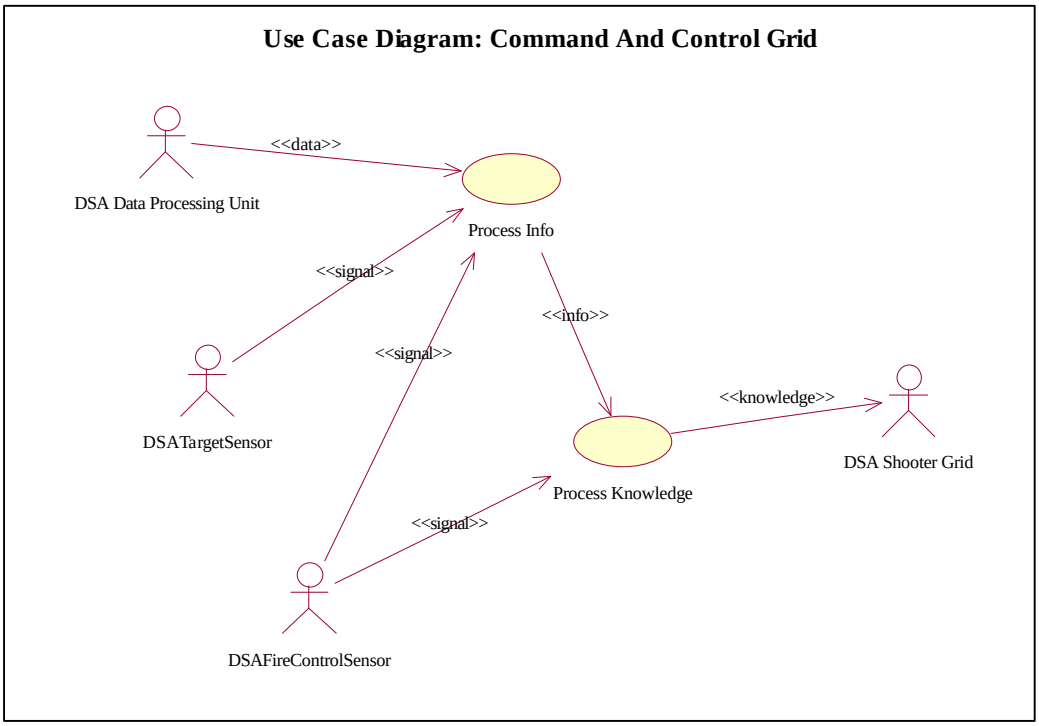

Fig 3.6: Use case diagram for the command and control grid

3.3.1 The "Process Info" Use Case

In the "Process Info" use case, the information from the "Data Processing" is processed again with the aid of data from the target and fire control sensors. In this use case, the geometry and intent of the target are identified. The target intent is estimated based on its initial trajectory; if the target were moving towards the ship, the initial estimate for its intent would be "threat". This piece of information is provided by the sensor grid. The target geometry is known from the data received from the target sensor.

The textual description of the "Process Info" use case is:

Main flow of events: This use case starts after the processing of the data ends at the "Process Data" use case. The "Process Info" use case queries the "Use Target Sensor" use case for high-resolution data. Then, it builds a knowledge base about the target geometry and intent, which are used to recognize it. This use case ends after it passes this information to the "Process Knowledge" use case.

Figure 3.7 shows the activity diagram describing the behavior of the “Process Info” use case.

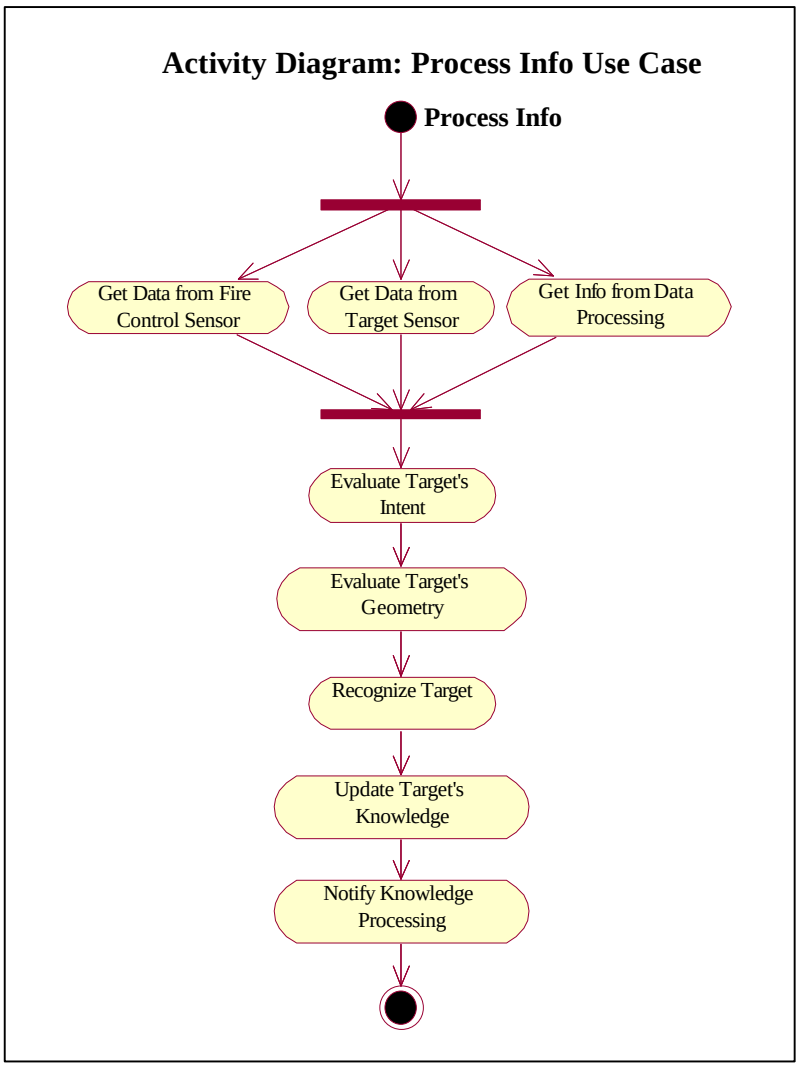

Fig 3.7: Activity diagram for “Process Info” use case

3.3.2 The “Process Knowledge” Use Case

In the “Process Knowledge” use case, the knowledge resulting from processing the information is processed again with the aid of data from the fire control sensor. The difference among data, information, and knowledge is as follows

- Data is defined by “DoD Dictionary of Military Terms” as:

"Representation of facts, concepts, or instructions in a formalized manner suitable for communication, interpretation, or processing by humans or by automatic means. Any representations such as characters or analog quantities to which meaning is or might be assigned."

This implies that the data is only a formal representation of facts. It does not carry any meaning or interpretation by itself. Data needs to be processed to obtain information.

- Information is defined by "DoD Dictionary of Military Terms" as:

  "The meaning that a human assigns to data by means of the known conventions used in their representation"

  This means that information is the interpretation of data into meanings. These interpretations are based on previous experience, training, or knowledge. False data or false interpretations could lead to false information. Different people or machines could have different interpretations of the same piece of data; this leads to different inferred information.

- Knowledge is generated by gathering information from different sources and collaborating and coordinating this information to reach a common understanding of the initial facts interpreted by these sources. Since information coming from different sources might be different, knowledge processing should be able to regenerate the facts interpreted by information processors by processing the information from all of the sources.

To process data and obtain information, one needs to understand the data format. In the case of radar, one needs to know the radar equation to detect the existence of a target based on the reflected radar signals. A target that might seem hostile to one ship might seem friendly to another. Hence, one needs to coordinate with all of the partners and exchange information about the current situation in order to be able to get a clear idea about the hostility of the target. Even though data is a representation of facts, it is not useful to take action unless a meaning is

added to them. Adding meaning to data will result in information that might not be accurate. Information is not sufficient for decision support, since it is not trust worthy, and hence, the resulting actions might not be appropriate. Knowledge is the best way to guarantee a more trustful action. Hence, information processing would be needed before an action is taken.

The command and control builds knowledge of its environment by collaborating with neighboring partners. If needed, the ship then updates its status based on the current situation. At this stage, the command and control manages to recognize and identify the target. Identifying the target requires knowledge of its exact model and capabilities. Techniques for identifying targets are left for future research. If the current target qualifies as a threat, the "Process Knowledge" use case triggers the engagement loop in the shooter grid.

The textual description of the "Process Knowledge" use case is as follows:

Main flow of events: The "Process Knowledge" use case starts when the information processing ends. The system uses the high-resolution data from the fire control sensor along with knowledge obtained from the "Process Info" use case to recognize and identify the target. The command and control system then coordinates with partners in the battlefield in order to learn about the surrounding environment. The command and control system updates the ship status based on the new situation. The system evaluates the capabilities of the ship in its current situation. This use case ends when the command and control system notifies the shooter grid subsystem for initiating the engagement decision.

Figure 3.8 shows the activity diagram describing the "Process Knowledge" use case.

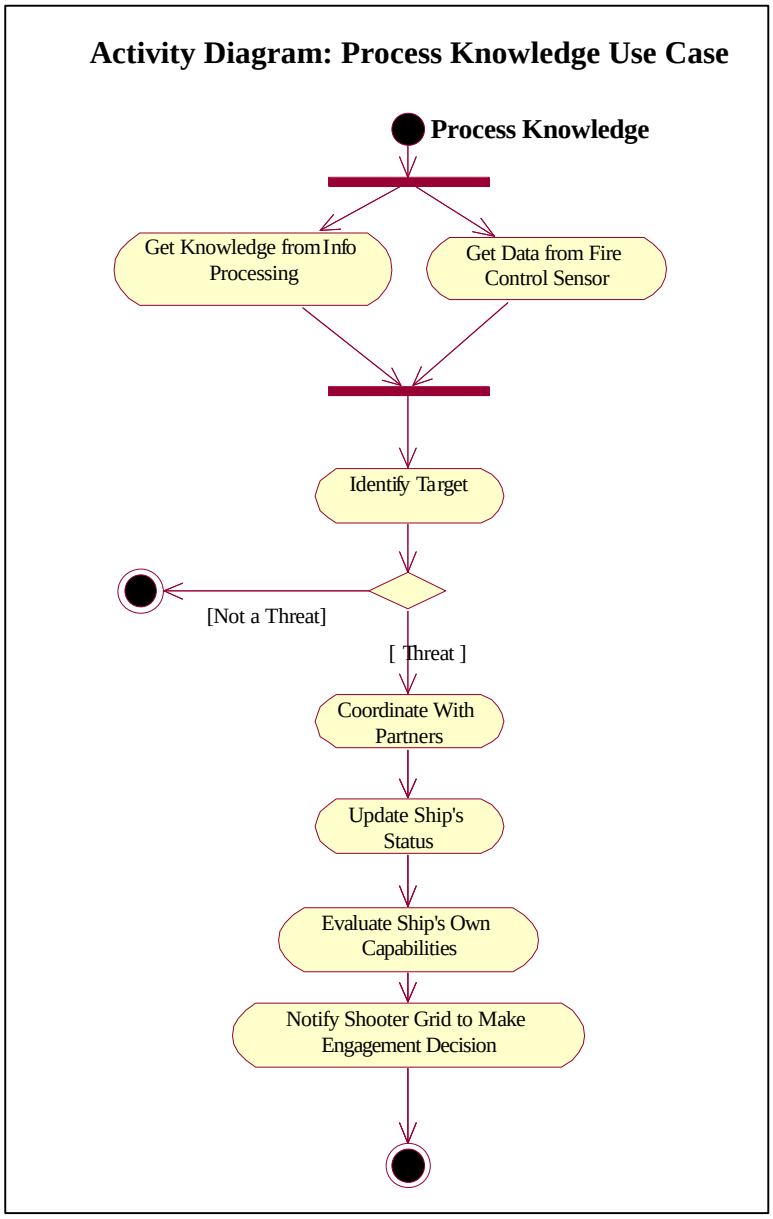

Fig 3.8: Activity diagram for “Process Knowledge” use case

3.4 Shooter Grid

The shooter grid is the engagement grid for the combat system. The process of decision making, effect processing (damage assessment), and outcome evaluation occurs in this grid. Whenever the shooter grid is notified by the command and control grid of the existence of a threat, the shooter grid goes into the decision making process to evaluate the needed capabilities to destroy the target, as well as the probability to destroy it. If an engagement decision is taken, an interceptor is launched against the target. The damage assessment process starts after the effect processing stage. In the effect processing stage, the target's damage and draw down[1] are estimated. The next stage involves the evaluation of the outcome. In this stage, the level of damage occurred to the target is compared to the needed damage level for the current situation. The needed level of damage is based on available ammo, weapons, and supplies; the amount of danger the target might cause, and more mission-specific tasks. If further actions are needed, another iteration of decision-making, effect processing, and outcome evaluation is started. Figure 3.9 shows the use case diagram representing the functionality of the shooter grid.

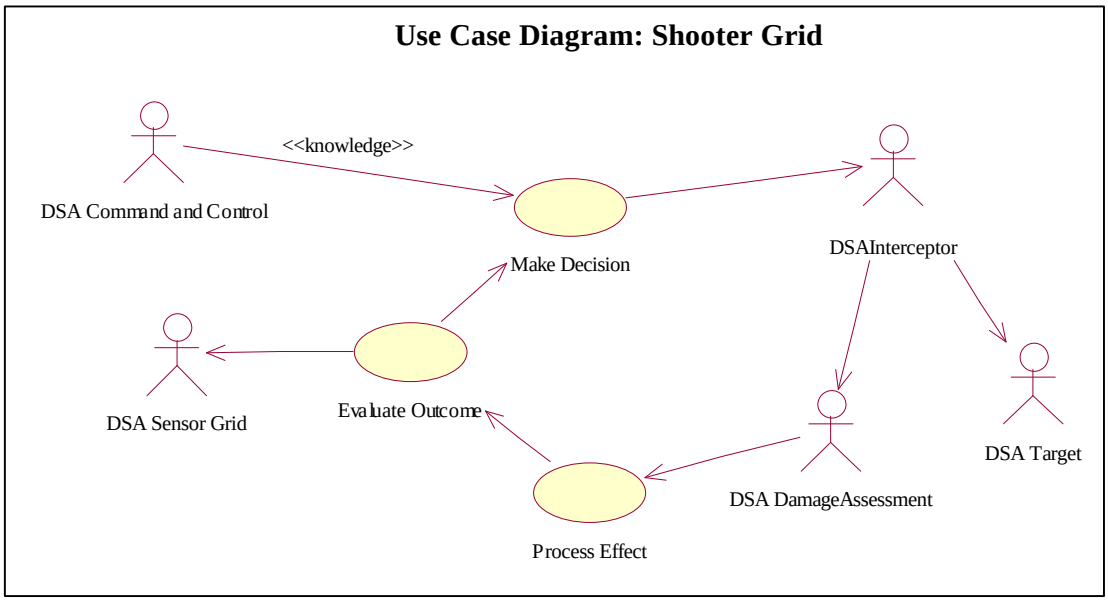

Fig 3.9: Use case diagram for the shooter grid

[1] Draw down is an indication of the degradation in the abilities of the target after being hit by an interceptor.

3.4.1 The "Make Decision" Use Case

The "Make Decision" process is responsible for making the decision of firing the interceptor based on the knowledge gained from previous use case. This use case has two passes. The first pass is initiated by the command and control grid. This is done to make a decision on the first action to be taken against the target. The second pass is invoked iteratively whenever the damage assessment and the outcome evaluation show the target destruction was less than the needed level. In both passes, the use case evaluates the capability to destroy the target based on the knowledge of the target. The capability (weapons) needed to destroy a ship is different from that needed to stop a missile. After these passes, this use case evaluates the possible outcome if engagement is taken. This outcome is evaluated as a probability to destroy target (PDT) based on its own capability and needed capability. If the PDT is greater than or equal to a specified value <MED>, the action to engage is taken and the interceptor is launched. The activity diagram for the "Make Decision" use case is shown in Figure 3.10 below.

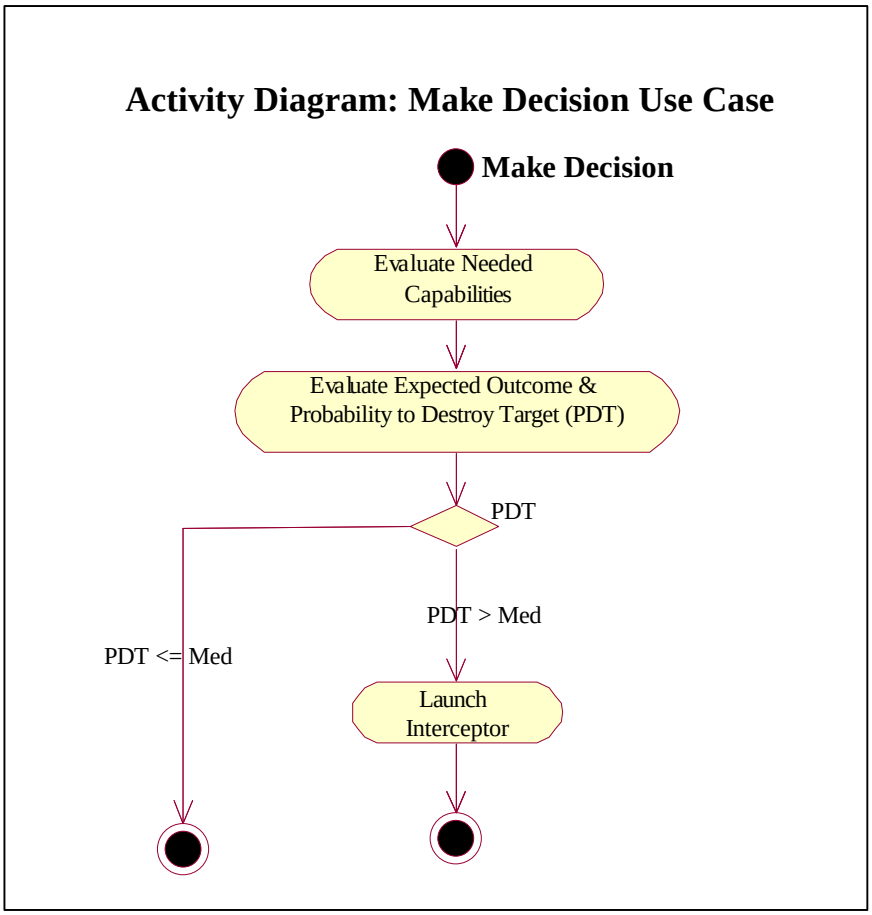

Fig 3.10: Activity diagram for "Make Decision" use case

3.4.2 The "Process Effect" Use Case

The "Process Effect" use case starts after the "Make Decision" use case has launched the interceptor. In this use case, the damage assessment returns information about the target's damage. The shooter grid system then updates the target's knowledge object with the new information. The system evaluates the damage that occurred to the target due to the interceptor. This use case ends after the "Evaluate Outcome" use case is notified about the results of this use case. The activity diagram for the "Process Effect" use case is shown in Figure 3.11.

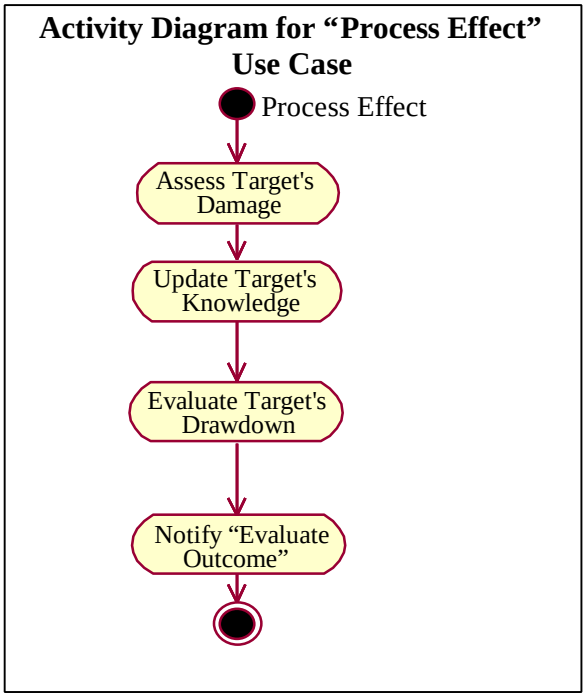

Fig 3.11: Activity diagram for the “Process Effect” use case

3.4.3 The “Evaluate Outcome” Use Case

The “Evaluate Outcome” use case is started when it is notified by the “Process Effect” use case to evaluate the damage of the target. The system compares the target’s damage to the needed level of damage (NLD), which is both mission and target specific. If the NLD is achieved by the current interceptor, then the engagement is marked as a success. Otherwise, the need for further action (NFA) is evaluated. If there is no need for further actions, the engagement is marked as successful; otherwise, the engagement is marked as a failure and the system notifies the “Decision Making” use case to make further engagement decisions. This use case ends after the “Decision Making” use case is notified. The activity diagram describing “Evaluate Outcome” use case is shown in Figure 3.12.

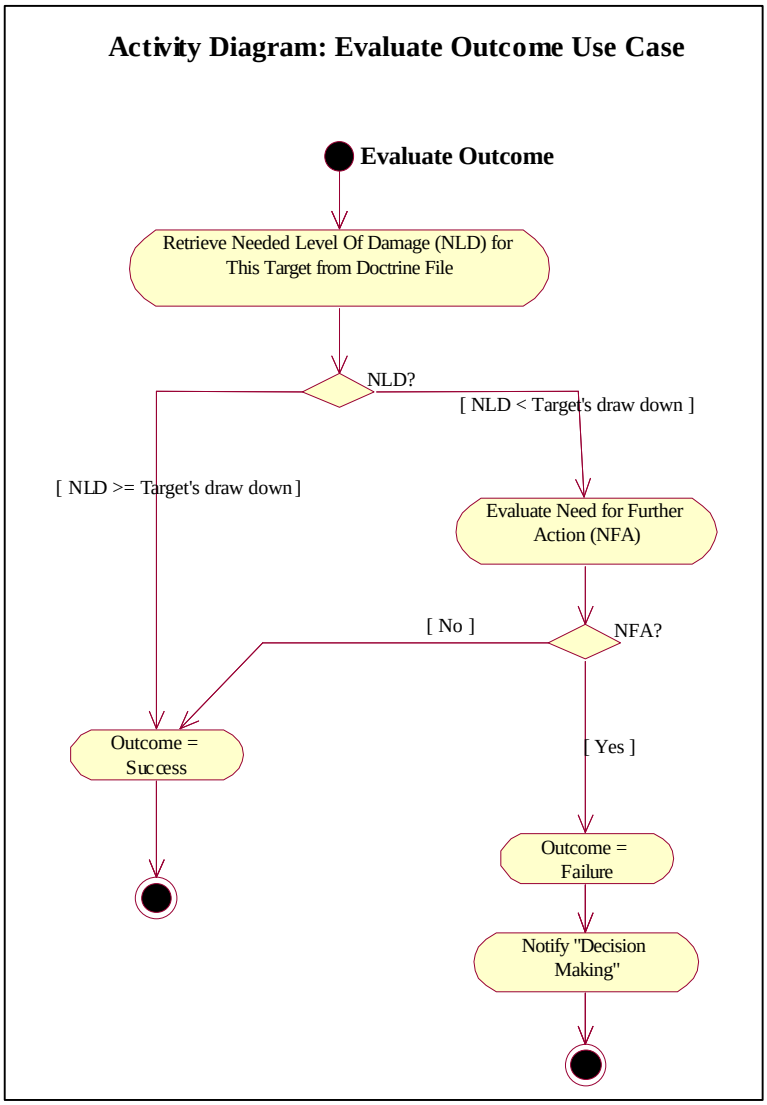

Fig 3.12: Activity diagram for “Evaluate Outcome” use case

3.4.4 The “DSA Interceptor” Actor

This actor represents the interceptor that would be fired by the shooter grid. Since the interceptor is external to the shooter grid, it was rendered as an actor. Interceptors represent all available ammo, missiles, and rockets on board the ship. Interceptors can be classified based on their speed, warhead size, and the targets it is designed to engage.

3.5 Next Step

After building the use case diagrams and the corresponding activity diagrams, the system's requirement will be determined. The next step is building the logical view of the system; this view contains the structural, interaction, and operational domains of the system. The structural domain is modeled by the structural diagrams (class and package diagrams), which are needed to carryout and realize the use case diagrams. Chapter 4 describes the package and class diagrams used to specify the static structure of the system.

Chapter 4:

LOGICAL VIEW: STRUCTURAL DOMAIN OF THE COMBAT SYSTEM

This chapter discusses the development of the structural domain for the logical view of the combat system. The structural domain describes the structure of the system at both the package and class levels. Unlike the previous chapter (use case view), this domain is more tightly coupled with the source code. For an incremental system development, the structural domain describes how the previously mentioned use cases are implemented. In this chapter, the package diagrams of the digital ship and combat system are introduced. Then the class diagrams that represent the system's structure are discussed.

4.1 Package Diagram of the Digital Ship

This section discusses the breakdown of the digital ship's system into subsystems. The digital ship's organization is broken into five major subsystems: combat systems, engineering, supplies, weapons, and operations. Package diagrams are used to show this breakdown. Packages are used to group related things in UML; in this case, packages are used to group related classes and to show the different subsystems of the digital ship. Figure 4.1 shows the breakdown of the digital ship's system into different subsystems.

The package diagram of the digital ship contains one class: "DSC Status,"[1] which encapsulates the ship's status information. The "DSC Status" is visible to all ship organizations; this will allow all of the ship organizations to modify and retrieve information from this class. For example, when the ship's command and control system evaluates the ship's own capabilities, it needs to retrieve information about the weapon systems and supplies. This "DSC Status" might evolve later in this project into an object oriented or relational database entry. The "DSC

[1] DSC: a prefix for Digital Ship Class

Status" contains several operations to simulate the actions taken by the ship under different states.

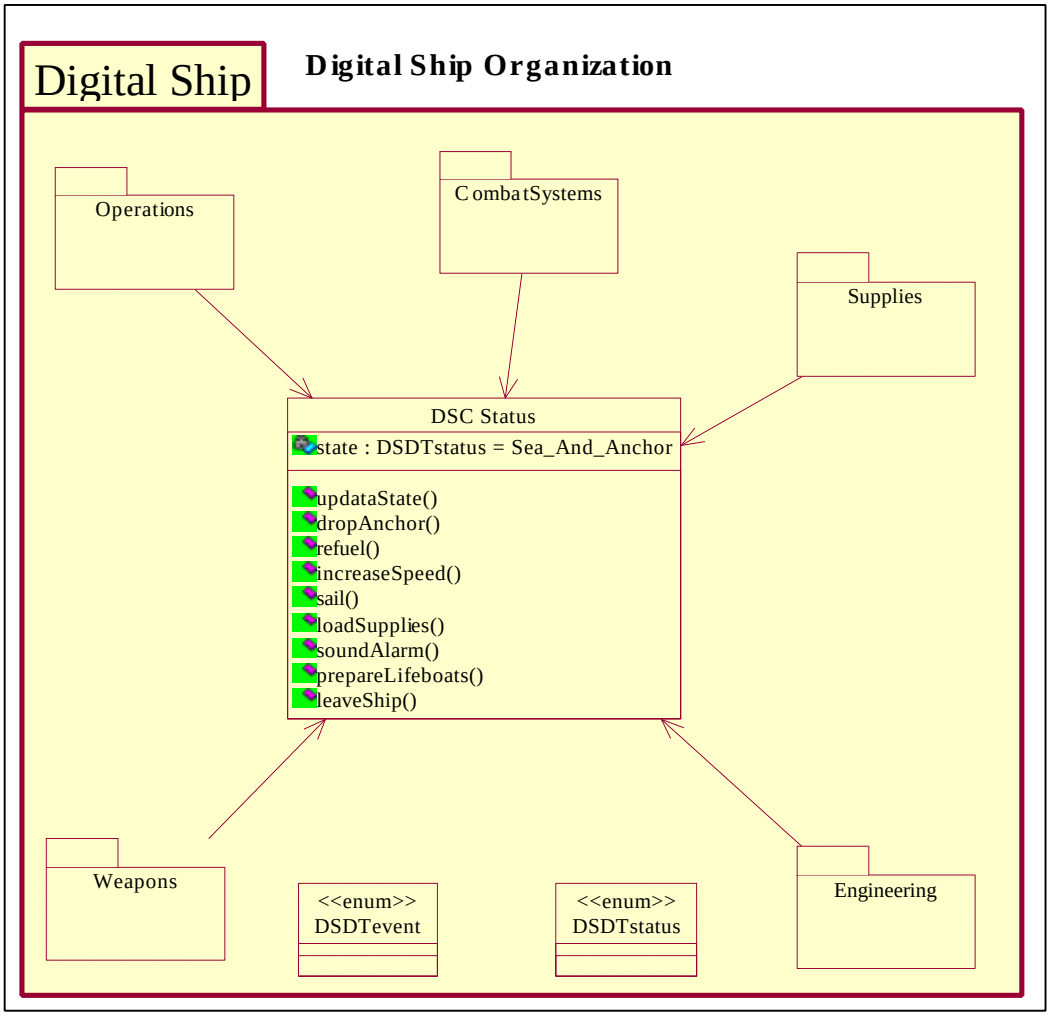

Fig 4.1: Digital ship's organization

The "DSDTevent"[1] and "DSDTstatus" represent enumeration data types. In UML, these data types are modeled as classes with <<enum>> stereotype[2].

[1] DSDT: a prefix for Digital Ship Data Type

[2] The stereotypes in UML are strings of text in << >> marks. They indicate additional information about the thing or relationship that is modeled by the UML notation.

Another package diagram was built for the combat system to show its subsystems. Three subsystems, one per grid, are needed to model and simulate the combat system (Sensor, Command and Control, and Shooter). Figure 4.2 shows this breakdown.

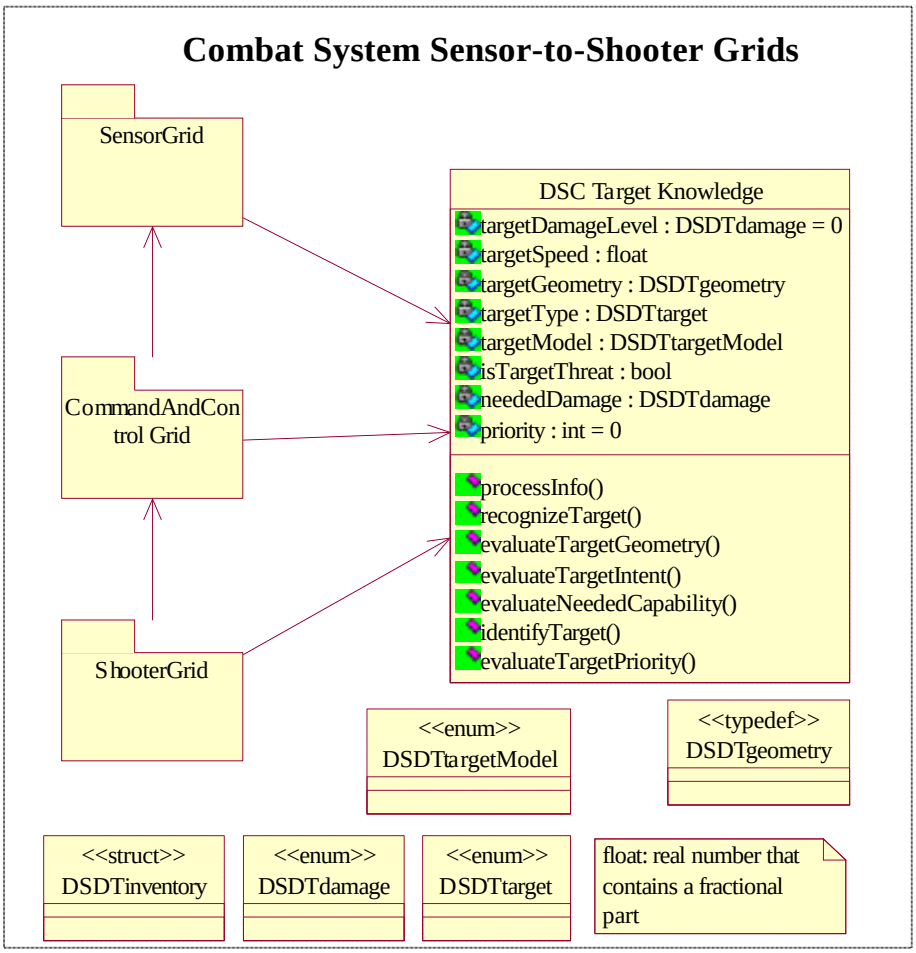

Fig 4.2: Combat system's sensor-to-shooter grids

Similarly, the "DSC Target Knowledge" is visible to all the grids of the combat system. An object of this class is created for each detected target. These instances are terminated when the target is destroyed or when the target is out of the range of the ship sensors. Both package diagrams show the dependency relationships between the different packages. All of the digital ship organizations depend on the "DSC Status" for their operation. All of the three grids of the combat system depend on the "DSC Target Knowledge" for their operation. Likewise, the

"Shooter Grid" package is dependent on the "Command and Control" package, which, in turn, is dependant on the "Sensor Grid" package.

4.1.1 The " DSC Target Knowledge"

The "DSC Target Knowledge" is used to build the knowledge base, which the combat system gains about each target. An object is created for every detected target, which contains attributes about the target's speed, geometry, type, model, priority, and damage level. This class has operations for evaluating the target's geometry, intent, the needed capabilities to destroy it, some operations for identifying and recognizing it, one operation for processing information, and another for evaluating its priority. The target's priority indicates how dangerous the target is to the ship compared to other targets; for example, an anti-ship missile typically has higher priority as a threat than hostile aircraft. This class is placed in the "Combat System" package in order to be visible to and can be used by all the combat grids. The "DSC Target Knowledge" is transient; that is, it will be created at run time[1] whenever a target is detected, and it will be terminated after the target is destroyed. The "processInfo" operation is called by the "DSC Info Proc"[2] class to handle the processing of information obtained by the "DSC Data Proc" and data from the target sensor; it calls the local operations of "evaluateTargetGeomenty", "evaluateTargetIntent", and "recognizeTarget". The "evaluateNeededCapability" operation is called by the "DSC Decision Making"[3] class to evaluate the assets needed to destroy the target; these capabilities are described in the form of type and number of missiles needed to destroy the target. Capable missiles should have the required speed, warhead size, and range for this specific target.

4.2 Class Diagrams of the Combat System

[1] Run time in software is used to refer to the time when the executable file of an application is running. At run time, some objects could be created and destroyed. Compile time, on the other hand, refers to the time when the executable file of the application is generated. Objects created during compile time are typically persistent. These objects persist until the application is terminted.

[2] More about "DSC Info Proc" in the command and control class diagrams <4.2.2>

[3] More about "DSC Decision Making" in the shooter grid class diagrams <4.2.3>

Class diagrams in UML show the static structure of the system. They specify the relationships between the classes whose instances (objects) build the behavior of the system. Choosing the correct classes is an iterative process. The developer starts with a set of suggested classes, and as the design evolves, classes might be added or removed. The relationships between classes include inheritance, dependency, aggregation, and association.

4.2.1 Class Diagram for the Sensor Grid

Based on the use case diagram for the sensor grid, the class diagrams for this grid were developed. Figure 4.3 shows the class diagram for the sensor grid. This diagram consists of two classes, four actors, and three association classes. The classes are "DSC Data Processing" and "DSC Target Tracking Unit". The actors are "DSA Sensor System", "DSA Surv Recon Sensor", "DSA Target Sensor", and "DSA Fire Control Sensor"; they are rendered as actors because they are considered external to the sensor grid. They are used by the sensor grid to achieve its functionality. The association classes used are "DSC Information", "DSC Notify Info Proc", and "DSC Data from Surv Recon Sensor". The association classes are a special kind of class used to model the nature of an association linking two other classes. For example, the association class "DSC Information" models the information, which is transmitted between "DSC Target Tracking Unit" and "DSC Target Knowledge".

4.2.1.1 The "DSC Data Processing"

This class realizes the data processing unit. The instances of this class are persistent. One object of this class is sufficient for each combat system. The only attribute of the "DSC Data Processing" is the "DSC Target Tracking Unit". This containment is emphasized by the aggregation relationship between them. This aggregation relationship shows that one object of the "DSC Data Processing" would contain zero or more objects of the "DSC Target Tracking Unit". The object of the "DSC Data Processing" creates one instance of the "DSC Target Tracking Unit" per potential target.

The "DSC Data from SurvRecon Sensor" association class has two operations: one is used by the "DSC Data Processing" for getting data from the surveillance and reconnaissance sensors "getData", and the other is used for sending data from the sensors "sendData", called by the "DSC Surv Recon Sensor" when the new data is available.

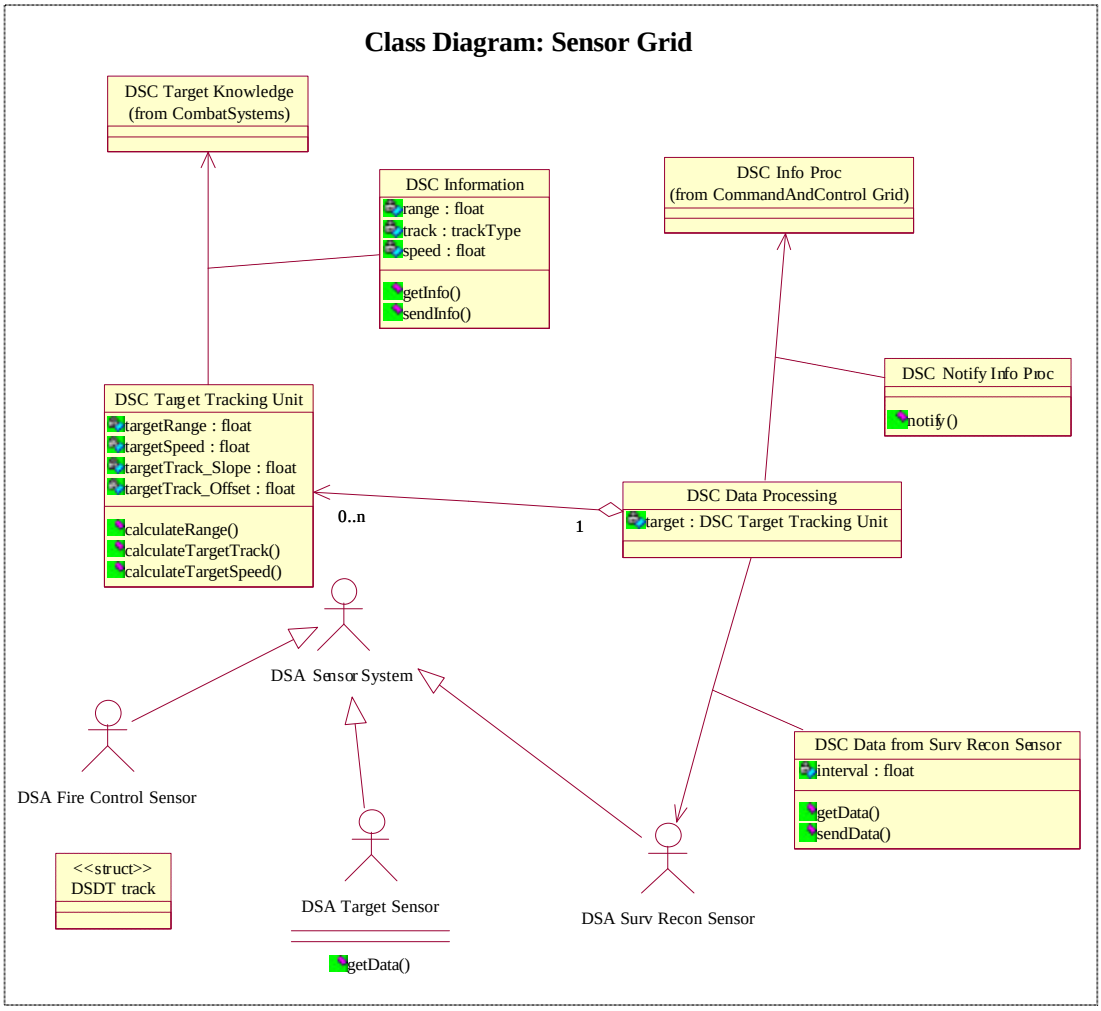

Fig 4.3: Class diagram for the sensor grid

4.2.1.2 The "DSC Target Tracking Unit"

This class is responsible for processing the data coming from one specific target. The objects of this class are transient; that is, the objects are created at run time whenever a potential target is detected, and they are destroyed whenever the object completes its job of calculating the range, bearing, and in some cases speed of the target. This class has three operations: "calculateRange", "calculateTargetTrack", and "calculateTargetSpeed". This information is sent to the target knowledge class in the command and control grid for further processing.

4.2.1.3 The "DSA Sensor System"

This class is rendered as an actor because it is external to the sensor grid. However, it still needs to be modeled in the sensor grid, because it is the base class of all generic sensor systems, which in turn interact with the sensor grid. In the current phase of the project, three kinds of sensors are considered: Surveillance/Reconnaissance, target, and fire control sensors. The "DSA Sensor System" is the parent for these three sensor systems. This is modeled using the inheritance relationships shown in Figure 4.3. The "DSA Sensor System" contains the common attributes and operations that exist between all generic sensor systems.

4.2.2 Class Diagram for the Command and Control Grid

To fulfill the behavior of the command and control grid, one needs two classes with persistent objects: "DSC Info Proc" and "DSC Knowledge Proc". These classes interact with the "DSC Target Knowledge". Figure 4.4 shows the class diagram for the command and control grid.

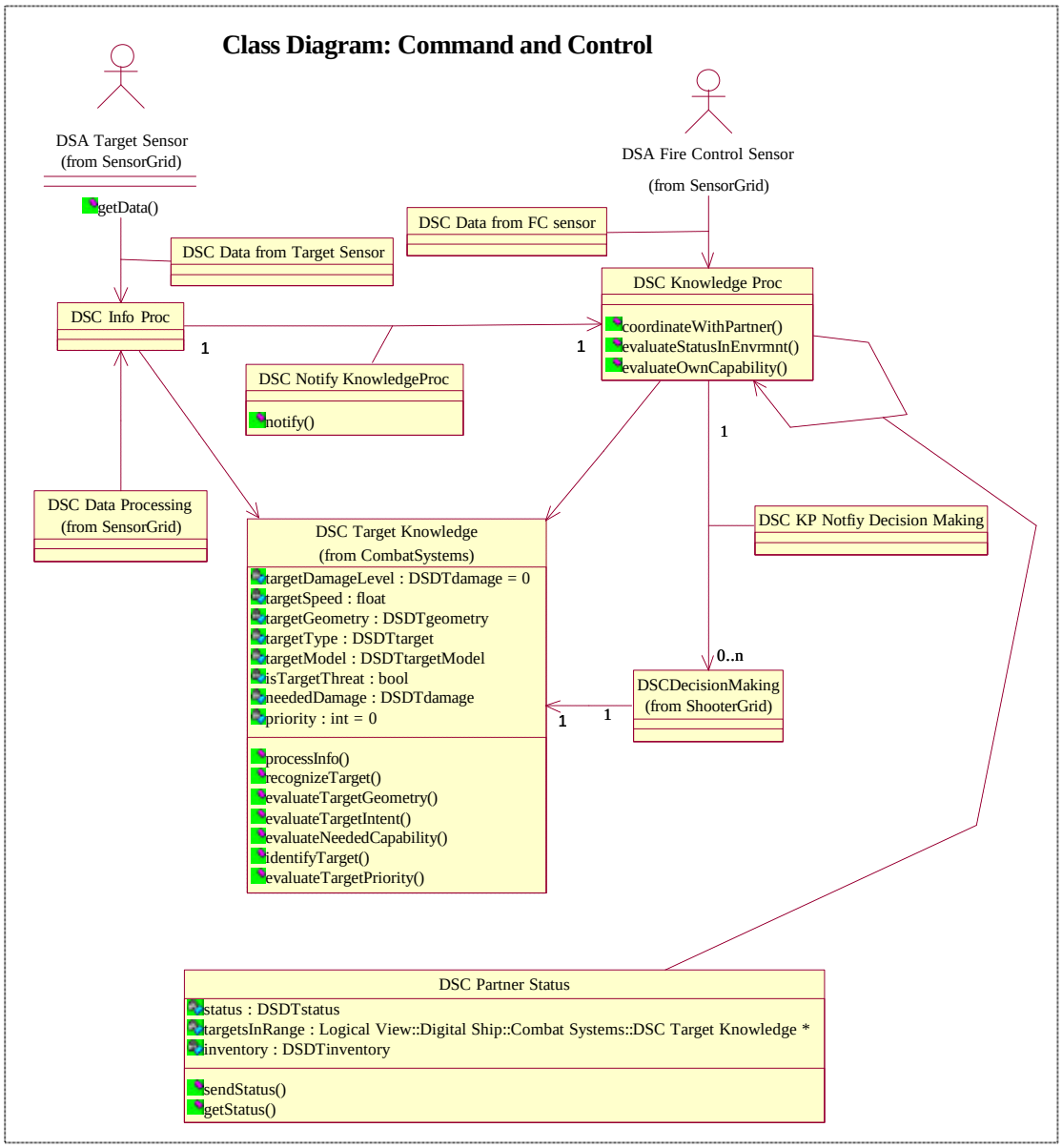

Fig 4.4: Class diagram for the command and control grid

4.2.2.1 The "DSC Info Proc"

The "DSC Info Proc" has one persistent object per combat system. This class is notified by the sensor grid of any potential targets. This class is responsible for triggering the information processing procedure in the "DSC Target Knowledge" for every detected target. All of the

actual functions for processing the information are implemented in the “DSC Target Knowledge”.

4.2.2.2 The “DSC Knowledge Proc”

The “DSC Process Knowledge” has one persistent object per combat system. This object is notified by the “DSC Process Info” about every detected target. This class is responsible for coordinating with peers in partner ships, aircraft, submarines, or satellites to build a global environmental knowledge about the surrounding situation. This class has three operations: “evaluateOwnCapabilities”, “coordinateWithPartner” and “evaluateStatusInEnvrmnt”. The “evaluateOwnCapabilities” operation is responsible for building knowledge about the ship capabilities. They are based on ship’s own assets and weapons and satellite information available to the ship commander. The “coordinateWithPartner” operation is responsible for exchanging information with partners. This information is typically about local capabilities, surrounding environment, and targets detected by each partner. The “evaluateStatusInEnvrmnt” operation is responsible for building the knowledge base about the environment surrounding the fleet and all possible threats inside and outside the scope of the ship sensors and estimating the potential of partners in assisting the ship to destroy its immediate threats.

4.2.3 Class Diagram for the Shooter Grid

The shooter grid is the engagement grid of the combat system. To fulfill the behavior of the use case of the shooter grid, we used three classes: One class with persistent objects: “DSC Decision Making”, and two classes with transient objects: “DSC Effect Processing” and “DSC Evaluate Outcome”. Each of these classes is used to realize one use case in the shooter grid’s use case diagram. Figure 4.5 shows the class diagram for the shooter grid.

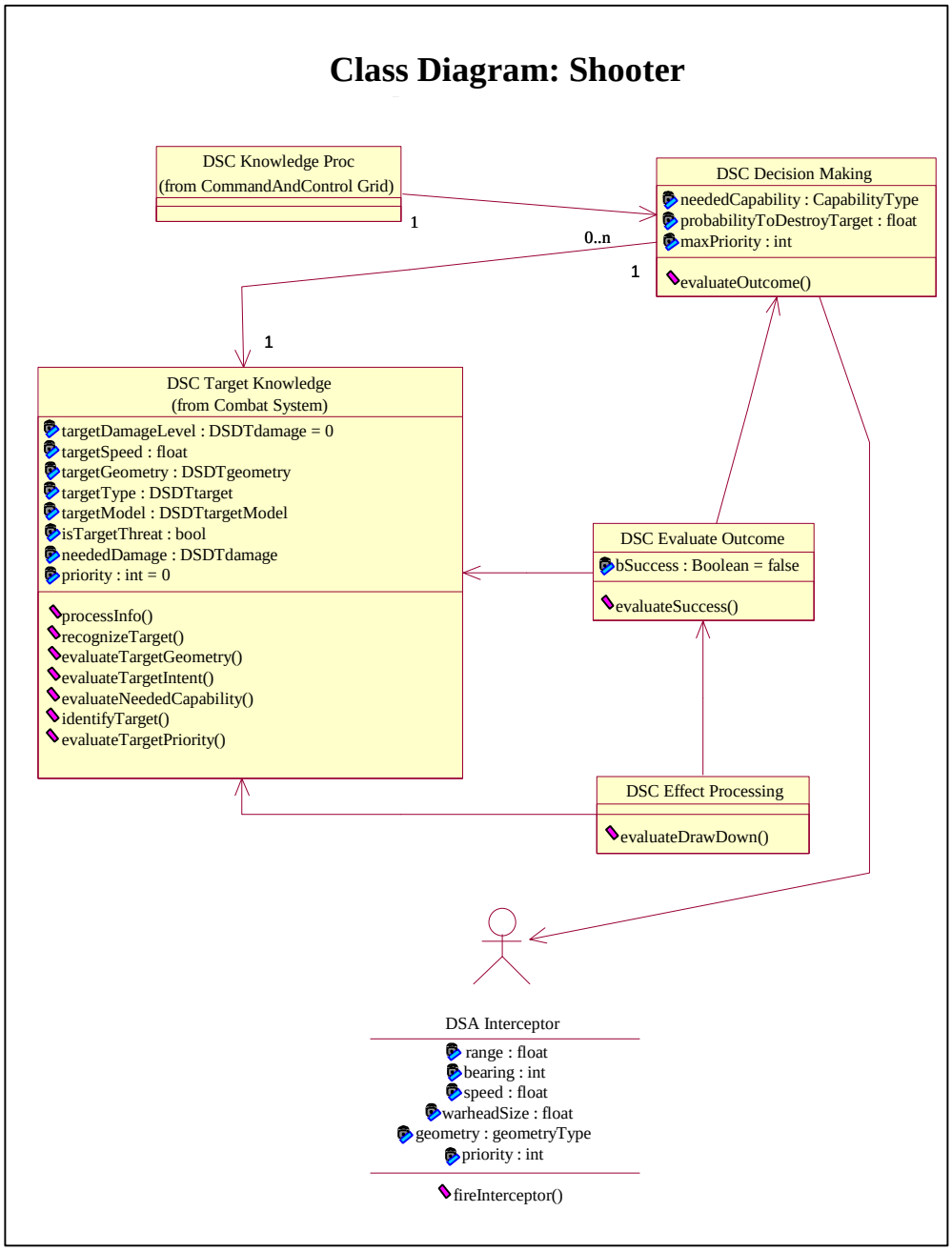

Fig 4.5: Class diagram for the shooter grid

It shows that one instance of the “DSC Knowledge Proc” is associated with several instances of the “DSC Decision Making”. One instance of the “DSC Decision Making” is responsible for making decisions about each detected target.

4.3 Next Step

Now that the static structural diagrams are specified, the interaction diagrams need to be designed. These interaction diagrams are drawn to show the possible scenarios for each use case. Interaction diagrams build the interaction domain for the software system. Chapter 5 discusses these diagrams.

Chapter 5

LOGICAL VIEW: INTERACTION DOMAIN OF THE COMBAT SYSTEM

This chapter describes the interaction diagrams[1] of the combat systems. Interaction diagrams are drawn for each use case of the system to model the possible scenarios, which it can handle. The interaction diagrams show the message passing between different objects to perform the behavior required by the system. There are two kinds of interaction diagrams: sequence and collaboration diagrams[2].

5.1 Sequence Diagram for "Process Data" Use Case

Figure 5.1 shows a sequence diagram representing the dynamic behavior (scenario) of the "Process Data" use case. It shows the sequence of messages sent between instances of classes to achieve the data processing scenario. A persistent object of the class "DSC Data Processing" keeps getting data from an object of the "DSC Surv Recon Sensor" class. For each detected target, the object creates a transient object "<u>: DSC Target Tracking Unit</u>"[3] of the class "DSC Target Tracking Unit", this object in turn creates an object of the "DSC Target Knowledge". Afterwards, the "DSC Target Tracking Unit" object calculates the target's range for several data samples. Using the range information of the target, the "DSC Target Tracking Unit" object plots the target's track. The target's track is used later to estimate the target's intent. Afterwards, the "DSC Target Tracking Unit" object calculates the target's speed and updates the "DSC Target Knowledge" object of the new information. The "<u>: DSC Target Tracking Unit</u>" object notifies the "<u>: DSC Data Processing</u>" object when it is finished. The "<u>:</u>

[1] Interaction diagrams are discussed in more details in Chapter 2

[2] Sequence and collaboration diagrams are discussed in more details in Chapter 2

[3] The colon '*:*' before the name of the class and the <u>underline</u> is used to refer to an object of that class. If the object has its own name, its name should be written before the colon.

DSC Data Processing" object destroys the ": DSC Target Tracking Unit" object and notifies the command and control system.

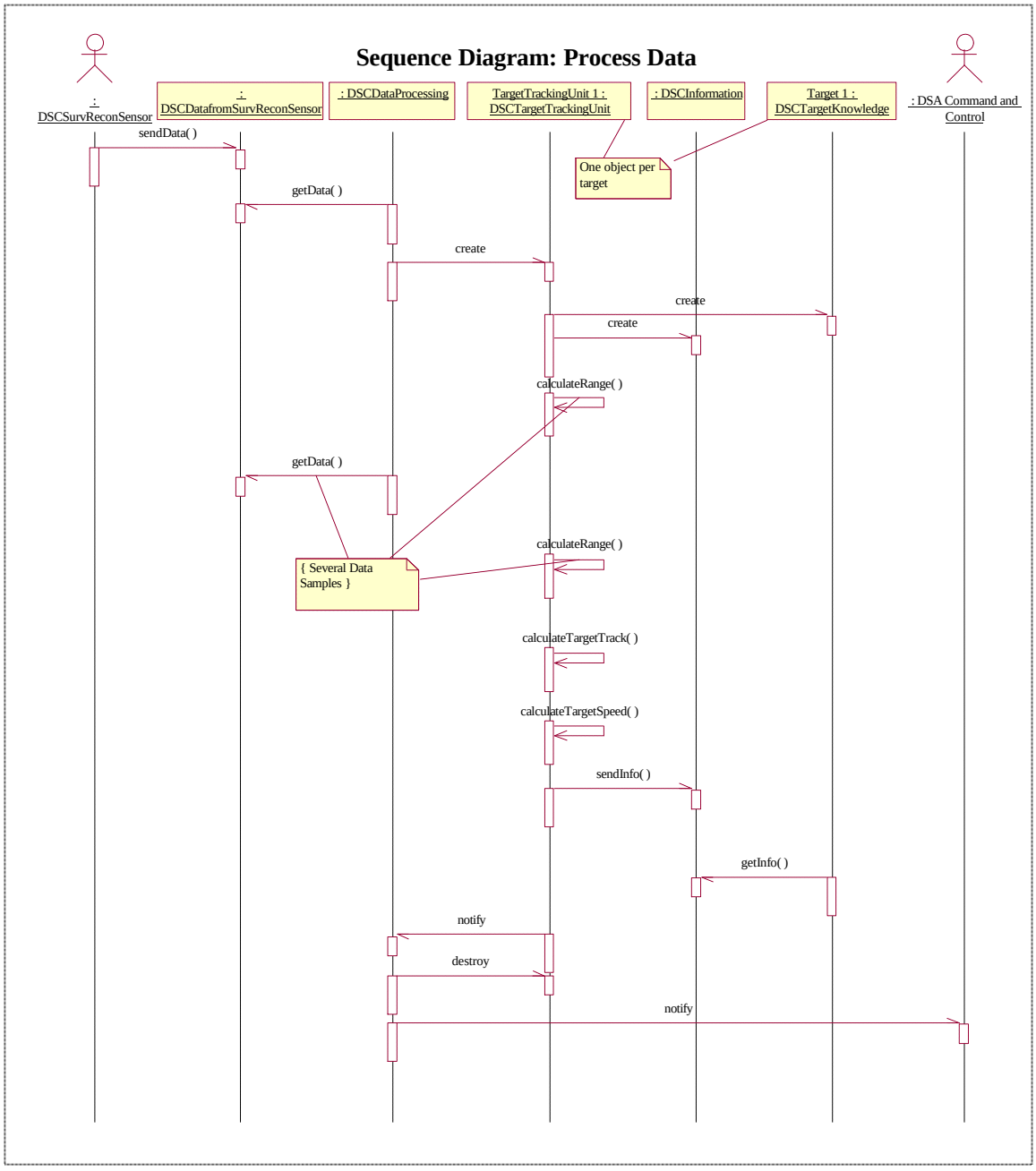

Fig 5.1: Sequence diagram for the "Process Data" use case

5.2 Sequence Diagram for "Process Info" Use Case

Figure 5.2 shows the use case diagram for the "Process Info" use case. When a persistent object of the "DSC Info Proc" is notified by the ": DSC Data Processing Unit" object, it queries information from the object of "DSC Target's Knowledge" and data from the target sensor's object. The "DSC Info Proc" then calls the "processInfo" operation in the object "T1" of the "DSC Target Knowledge". This operation calls locally the appropriate subroutines to achieve the data processing: evaluating the target's geometry and intent and recognizing the target. When the "DSC Info Proc" object finishes processing the information, it notifies the "DSC Knowledge Proc".

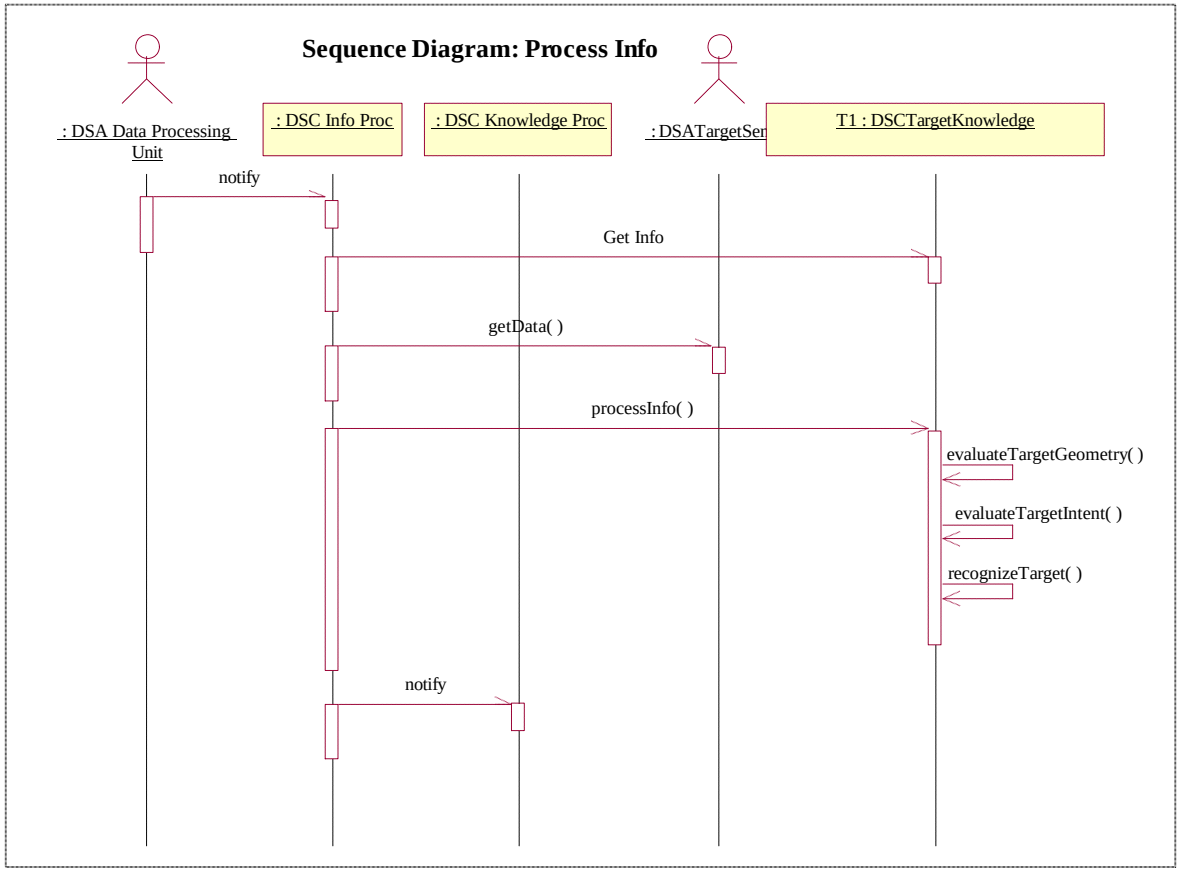

Fig 5.2: Sequence diagram for the "Process Info" use case

5.3 Sequence Diagram for "Process Knowledge" Use Case

This Sequence diagram starts when the ": DSC Info Proc" object notifies the "thisKP: DSC Info Proc" object. The "thisKP: DSC Info Proc"[1] object queries the fire control sensor for high-resolution data about the target and the "T1: DSC Target Knowledge" object about the target's information processed earlier. Afterwards, the "thisKP: DSC Knowledge Proc" object evaluates the capabilities of the ship and coordinates with partners to exchange knowledge about their situation and capabilities. The "thisKP: DSC Knowledge Proc" object evaluates the status of the ship in the current situation and global environmental knowledge and updates the status of the ship in the "this: DSC Status"[2] object. The "thisKP: DSC Knowledge Proc" object then notifies the shooter grid to make the engagement decision.

It is obvious that during the knowledge processing stage, the ship doesn't depend only on its local interpretation of the target's data, but it collaborates with its partners to build a high fidelity knowledge base.

[1] "thisKP" is used here to name the local knowledge processing object of the digital ship, contrary to the "partnerKP" object that is used to refer to the knowledge-processing object in the partner ships.

[2] "this" is used here to name the local ship's status object.

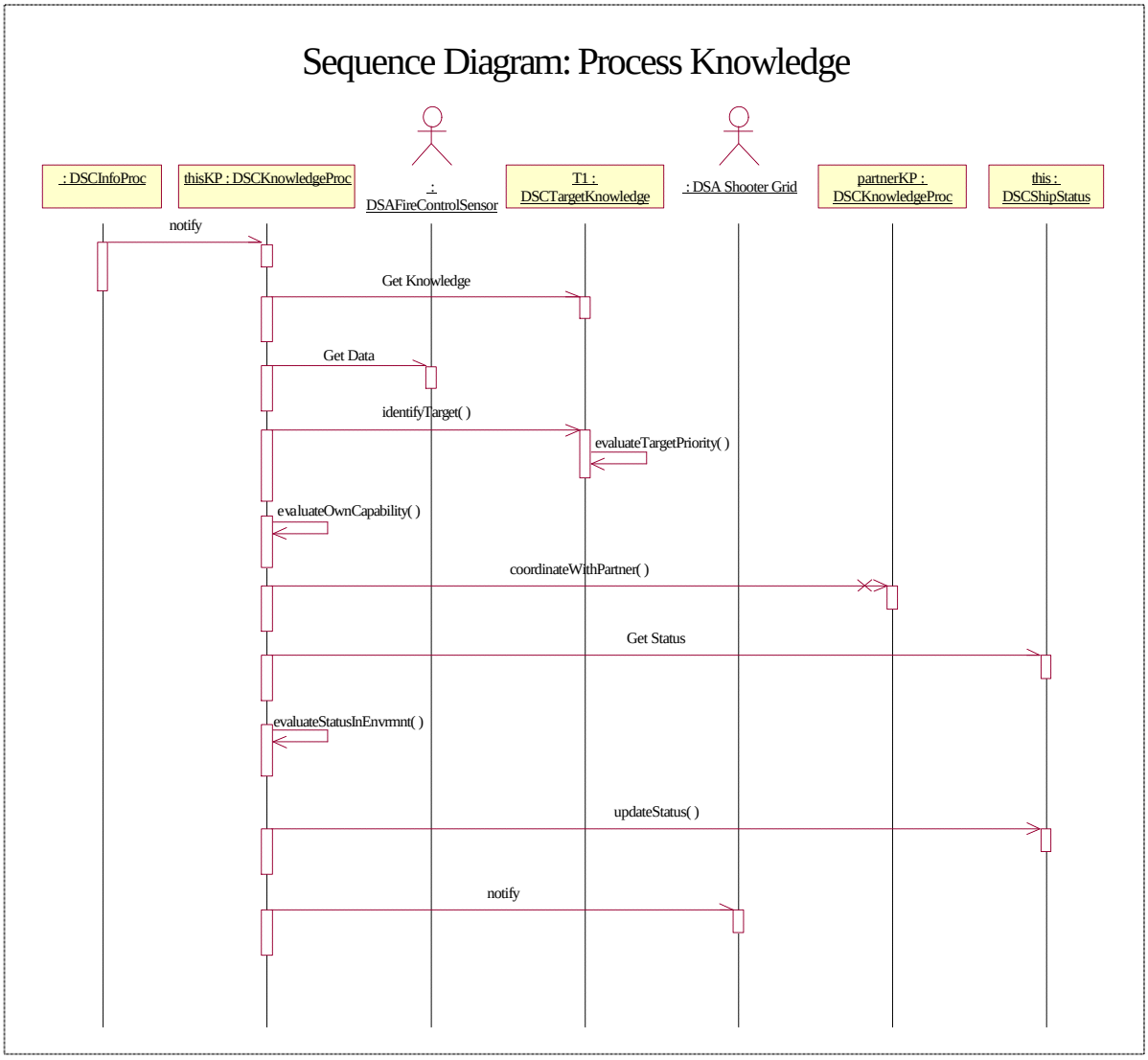

Fig 5.3: Sequence diagram for the “Process Knowledge” use case

5.4 Collaboration Diagrams for the "Make Decision" Use Case

As shown in the use case diagram for the shooter grid in Figure 3.9, the system goes through the decision-making stage from two passes. The first pass starts when the ": DSC Decision Making" object is first notified about the target through the ": DSC Knowledge Proc". The second pass starts for every iteration when the first engagement is reported as a failure.

5.4.1 "Decision Making" First Pass

Figure 5.4 shows the collaboration diagram for the first pass of the "Make Decision" use case. The "thisKP: DSC Knowledge Proc" object in the command and control system creates a transient object of "DSC Decision Making" for each target. This object calls the "evaluateNeededCapability" operation in the ": DSC Target Knowledge" object. Then the ": DSC Decision Making" object evaluates the outcome if the engagement option was chosen as a probability to destroy target (PDT). If PDT is greater than a certain value, the interceptor is fired and transient objects of the "DSC Effect Processing" and "DSC Evaluate Outcome" are created.

5.4.2 "Decision Making" Second Pass

If the resultant outcome is estimated to be a failure after the first iteration of engagement, the corresponding ": DSC Evaluate Outcome" object notifies the ": DSC Decision Making" object. The same scenario of evaluating outcome and needed capability and firing interceptor is repeated. Figure 5.5 shows the second pass of the decision-making use case.

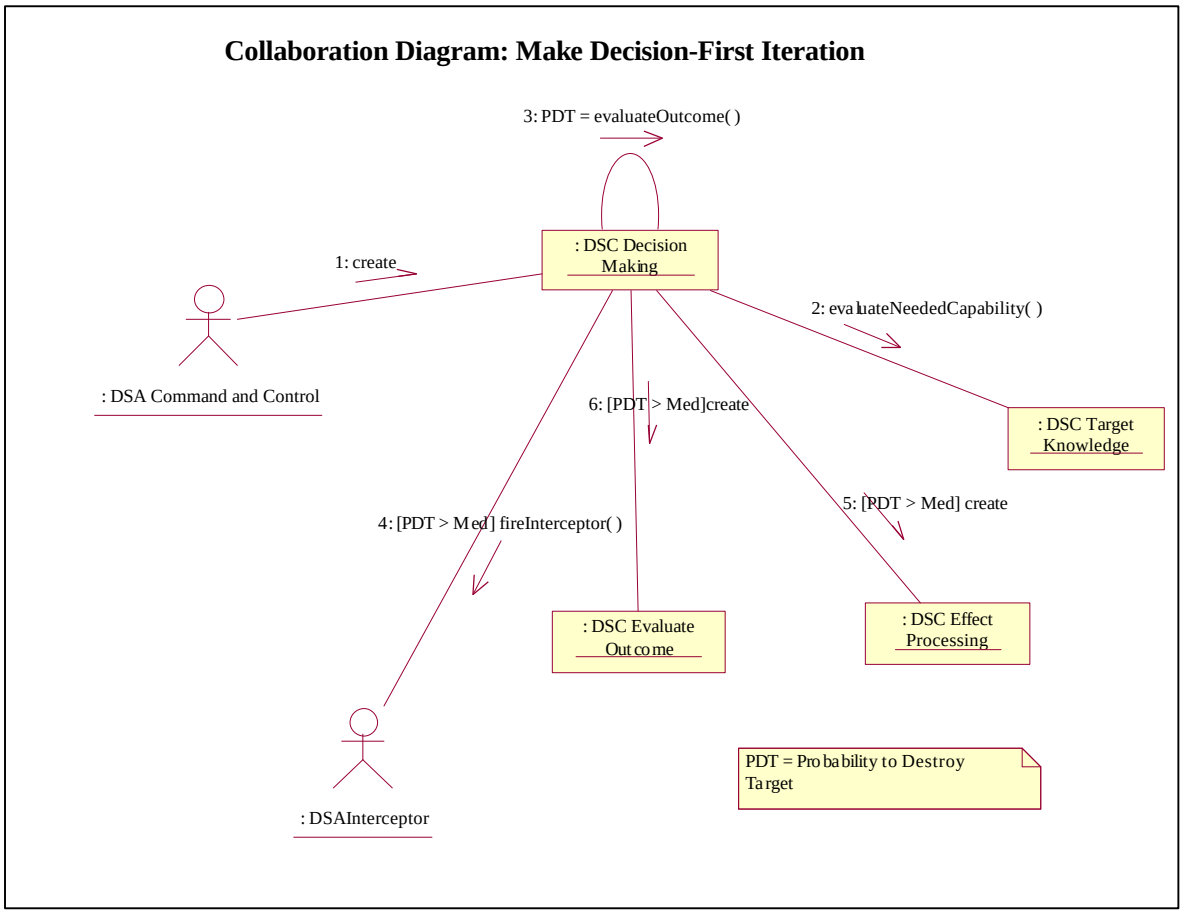

Fig 5.4: Collaboration diagram for "Make Decision" use case, first pass

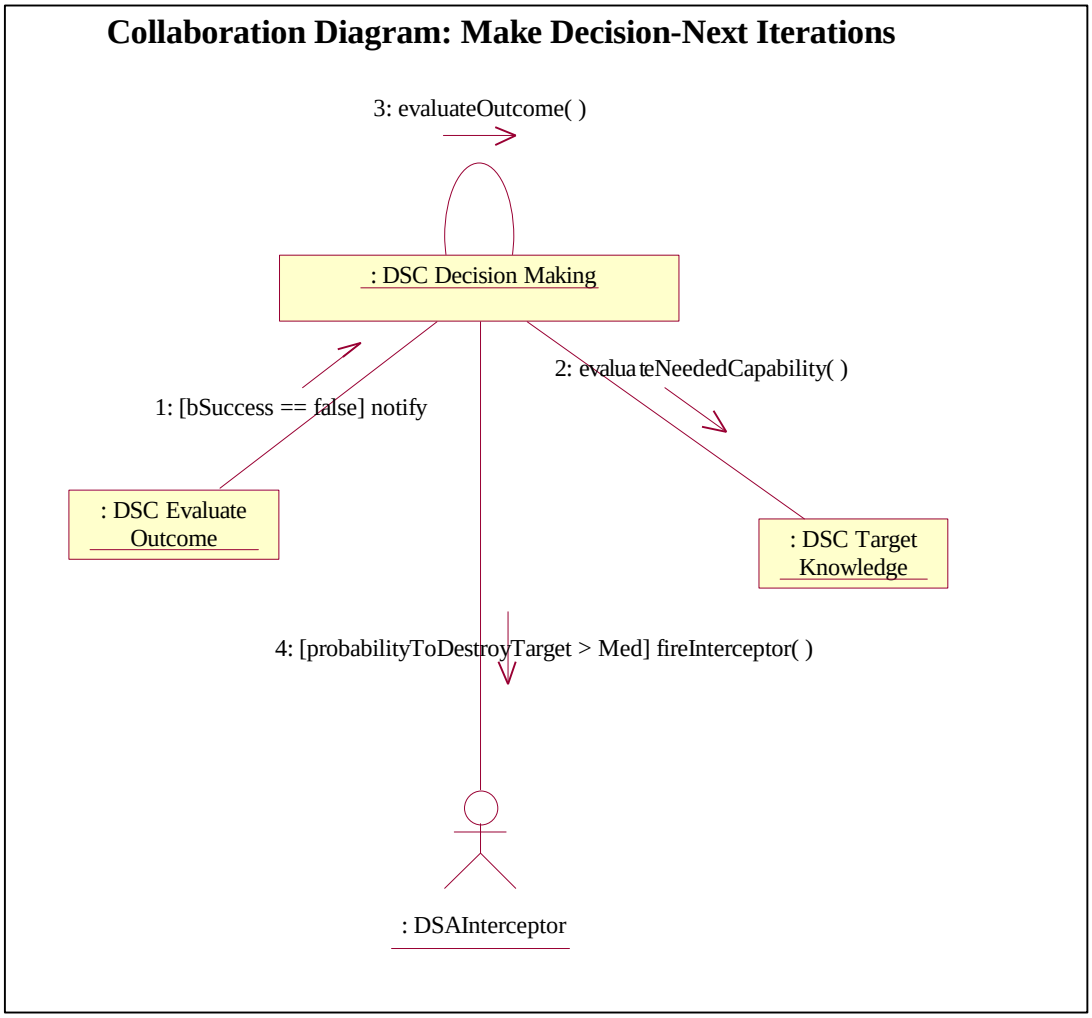

Fig 5.5: Collaboration diagram for “Decision Making” use case, second pass

5.5 Collaboration Diagram for the “Evaluate Outcome” Use Case

Figure 5.6 shows the collaboration diagram for the "Evaluate Outcome" use case. The “: DSC Evaluate Outcome” object is notified by the “: DSC Effect Processing” object. The “: DSC Evaluate Outcome” object evaluates the success of the engagement. If the engagement is successful, the “: DSC Evaluate Outcome” object destroys each of the “: DSC Decision Making”, “: DSC Target Knowledge”, “: DSC Effect Processing” objects and itself. If the

engagement was a failure, the "<u>: DSC Evaluate Outcome</u>" object notifies the "<u>: DSC Decision Making</u>" object to make the next engagement decision.

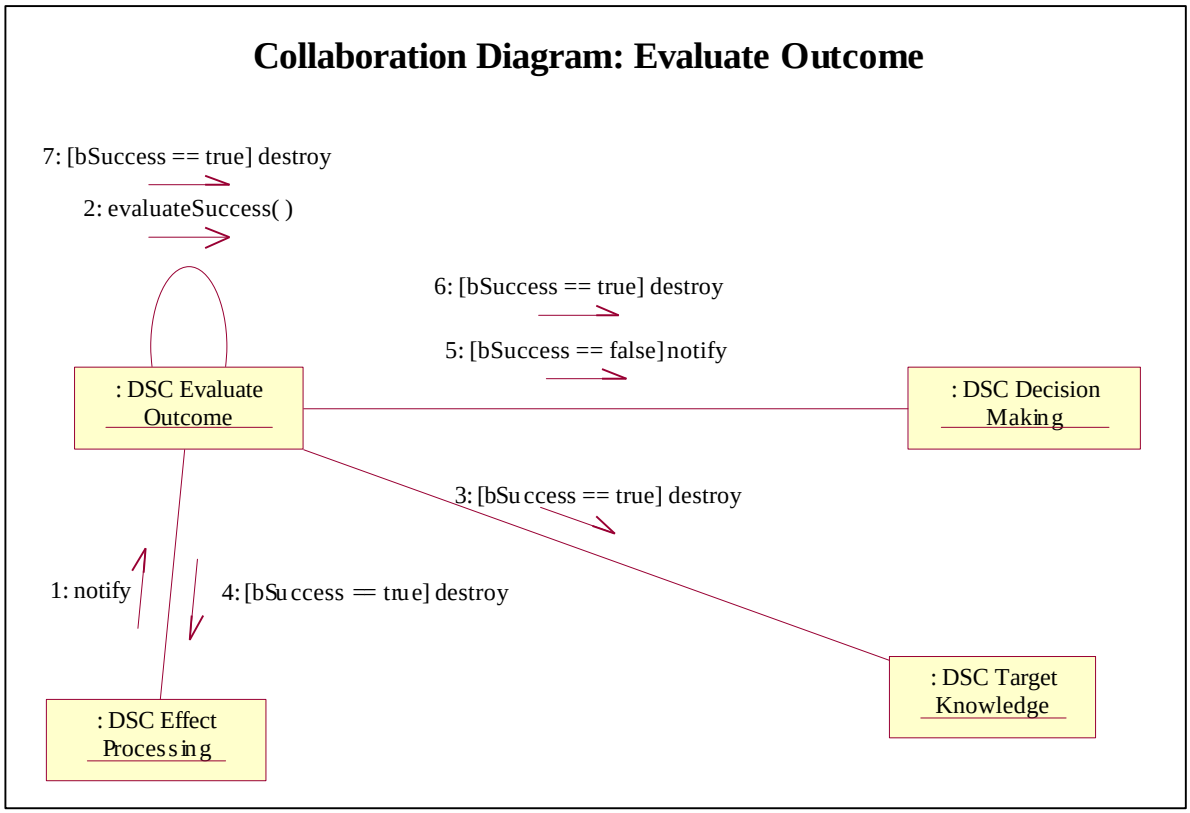

Fig 5.6: Collaboration diagram for "Evaluate Outcome" use case

5.6 Next Step

The structural and interaction domains of the system are specified via building the sequence diagrams. The next step is to build the diagrams for the operational domain of the system. This domain is captured using the state chart and activity diagrams described in Chapter 6.

Chapter 6

LOGICAL VIEW: OPERATIONAL DOMAIN OF THE COMBAT SYSTEM

This chapter discusses the state chart and activity diagrams, which are used to model the behavior and operations of classes in the system. A state chart diagram was drawn for the ship's status class. This diagram shows how the ship reacts in response to different events by changing its status. Several activity diagrams were drawn for different operations of different classes. These activity diagrams range from high level of abstraction to source code level. Low-level activity diagrams were drawn for large activities. In this chapter, the behaviors of these activity diagrams are discussed.

6.1 State Chart Diagram for Ship's Status Class

The ship status class "DSC Status" is responsible for keeping track of the general status of the ship. The ship could operate under different conditions according to its internal situation and external effects. The state chart diagram is shown in Figure 6.1. The diagram shows the different states of the ship and the events that cause the transition from one state to another. For example, if the ship is in "Cold Iron" state and the "Leaving Shore" event is triggered, it moves to "Underway" state. Upon entering the "Underway" state, the ship increases its speed. In addition, as long as the ship is in this state, it sails to the destination. If the "Replenishment Needed" event arrives, the ship enters the "Underway Replenishment" state. Upon entering the "Underway Replenishment" state, the ship drops the anchor and gets supplies. When the replenishment is finished, the event "Replenishment Finished" is triggered and the ship goes back to the "Underway" state.

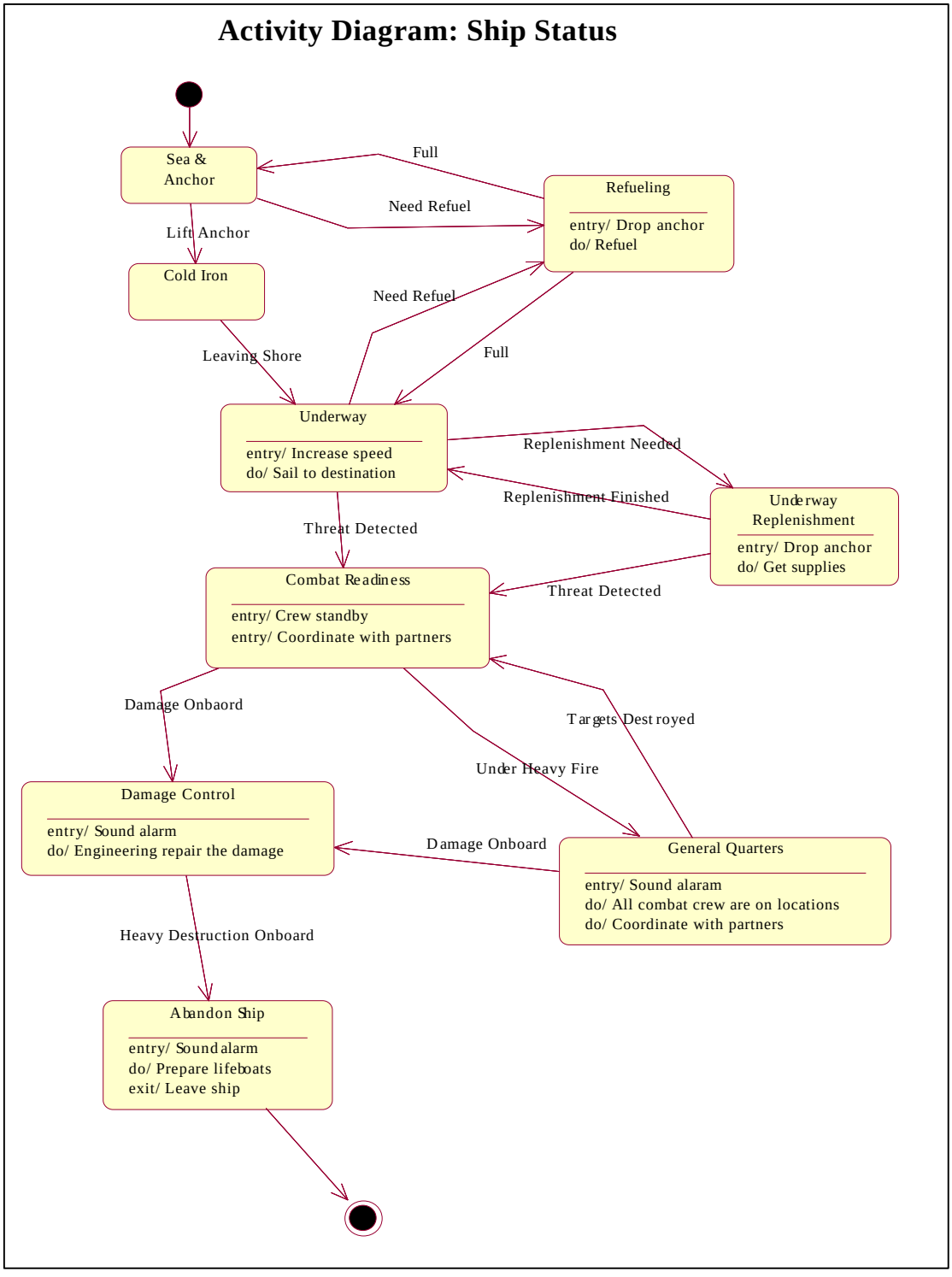

Fig 6.1: State chart diagram for “DSC Status”

6.2 Sensor Grid's Operations

In the sensor grid, classes perform three major calculations: the target's track, speed, and range. Activity diagrams were drawn for calculating the track and speed of the target.

6.2.1 Calculating Target's Track Activity Diagram

As shown in the data-processing sequence diagram in Figure 5.1, the "DSC Target Tracking Unit" obtains several data samples from the surveillance and reconnaissance sensors about the range and bearing of the target. These data are used to fit a best line between these data samples[1]. The target's track is identified by the slope and the offset of that line. Figure 6.2 shows the activity diagram for the "calculateTargetTrack" operation.

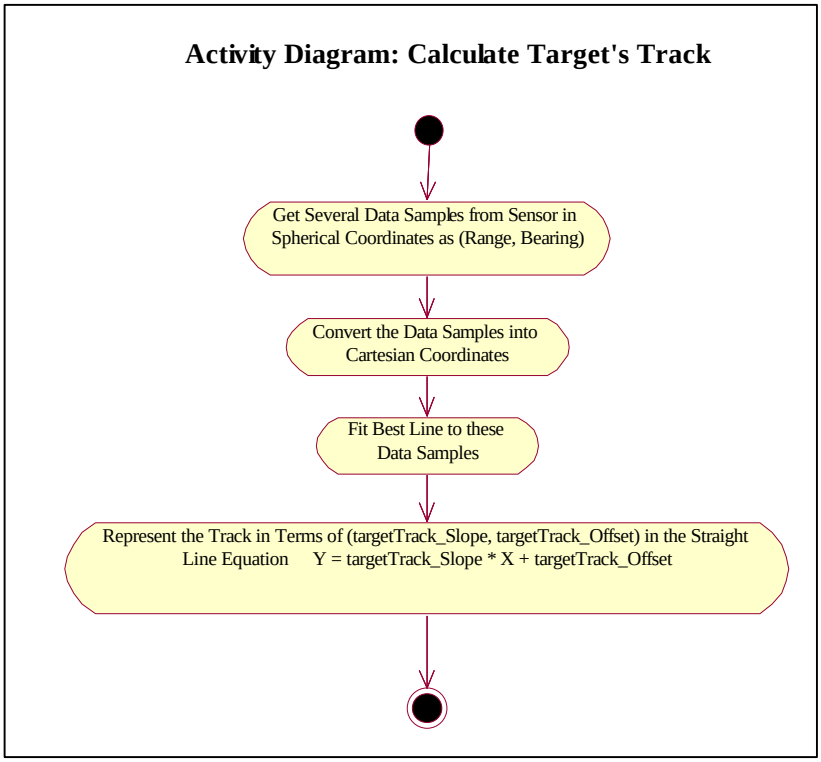

Fig 6.2: Activity diagram for calculating the target's track

[1] This assumes that the trajectory is a straight line

6.2.2 Calculating Target's Speed Activity Diagram

To obtain data about the target's speed, a Doppler Pulse Radar would be needed. The Doppler shift is used to estimate the target's speed. The Doppler frequency of the return signals from a moving object is a function of the object's speed. This phenomenon can be observed when a moving train passes by a person. The sound of the train's whistle changes as the train moves by the person. The Doppler shift is give by

$$Df = f_{tans} - f_{rec} = 2 * V_{trgt} * f_{tans} / C$$

where ? f is the Doppler frequency shift, f_{tans} is the transmitted frequency of the radar, and f_{rec} is the received frequency of the reflected signal from the target. Figure 6.3 shows the activity diagram for calculating the target's speed.

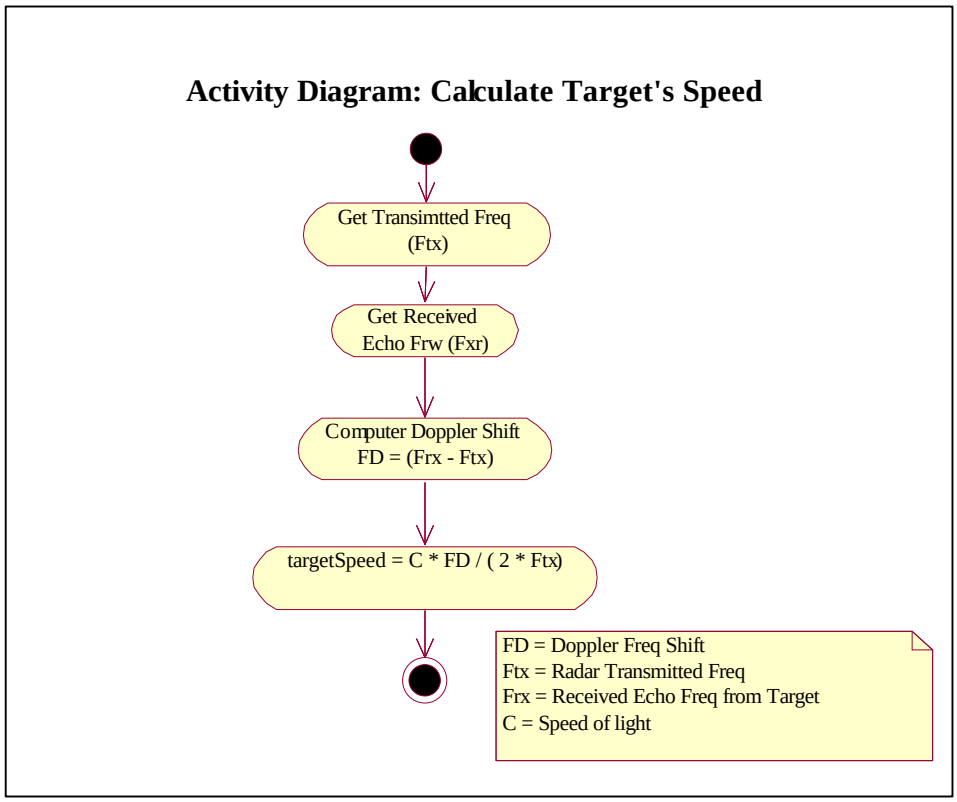

Fig 6.3: Activity diagram for "calculateTargetSpeed" operation

6.3 Command and Control Grid's Operations

The command and control's operations occur at two levels: information processing and knowledge processing. Activity diagrams are drawn for operations at both levels. This section discusses these operations and their activity diagrams.

6.3.1 Information Processing Operations

At the information processing level, the command and control grid performs four operations: "processInfo", "evaluateTargetGeometry", "evaluateTargetIntent", and "recognizeTarget". Activity diagrams were drawn for these operations. All of these operations are implemented as a part of the "DSC Target Knowledge".

6.3.1.1 Activity Diagram for "processInfo" Operation

Figure 6.4 shows the activity diagram for the "processInfo" operation. This operation calls three local operations inside the other three operations in the "DSC Target Knowledge". The interaction diagram for processing information in Figure 5.2 indicates that the information processing class only needs to call the "processInfo" operation in the "DSC Target Knowledge".

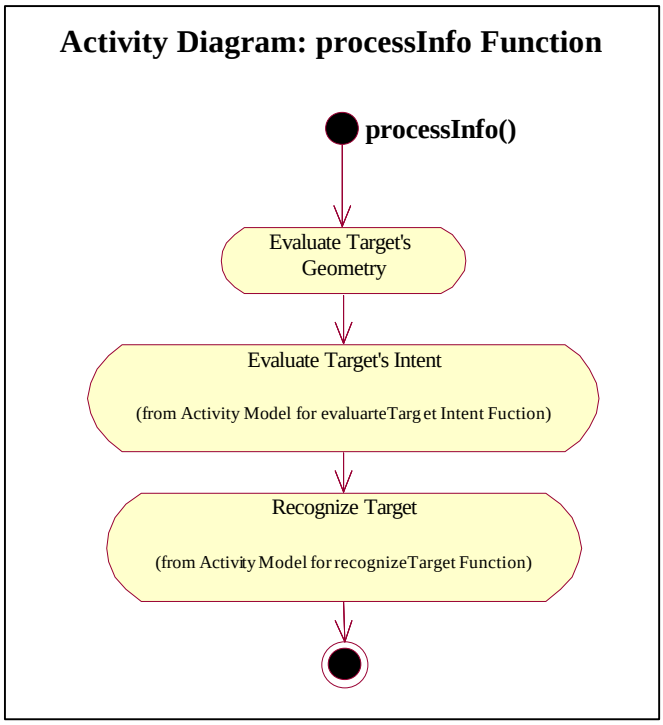

Fig 6.4: Activity diagram for “processInfo” operation

6.3.1.2 Activity Diagram for Evaluate Target’s Intent Operation

Evaluating the intent of the target is initially based on the track of the target. If the target’s track is directed towards the ship, hostile intentions are assumed. Since that target’s track is expressed in terms of slope and offset, the track will pass through the origin if the offset is zero. Figure 6.5 shows the relation between the track’s offset and the target’s intent.

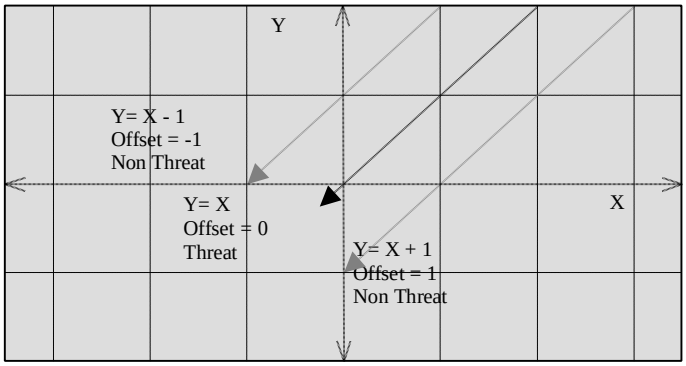

Fig 6.5: Evaluating target's intent based on the offset of its track

Figure 6.6 shows the activity diagram for evaluating the target's intent.

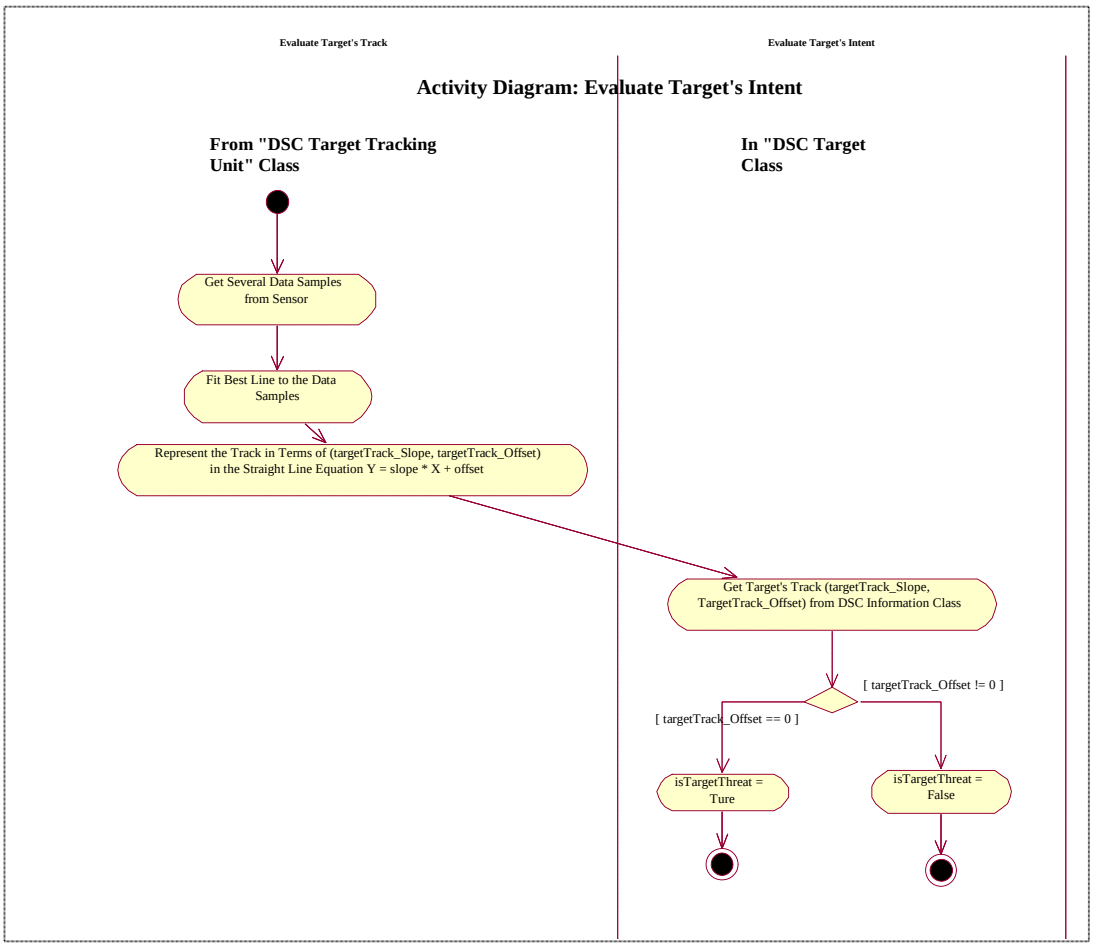

Fig 6.6: Activity diagram for "evaluateTargetInent" operation

6.3.1.3 Activity Diagram for Recognizing the Target

Recognizing a target means determining its type, (e.g. aircraft, ship, missile). At this step of the project, recognition of the target is based on the target's speed. The target's speed is a major factor in recognizing the target's type. The speed for a missile is significantly higher than that of an aircraft. An aircraft's speed is significantly higher than that of a ship. Figure 6.7 shows the activity diagram for the "recognizeTarget" operation.

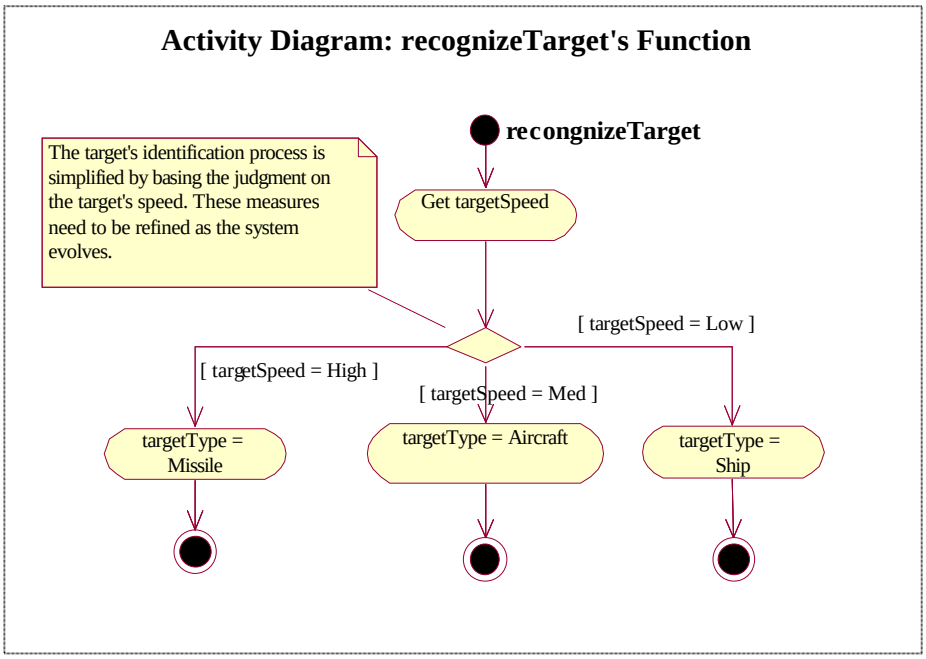

Fig 6.7: Activity diagram for "recognizeTarget" operation

6.3.2 Knowledge Processing Operations

Most of the knowledge-processing operations are high-level operations that involve knowledge of the different models of targets and their characteristic. At this step of the project, one operation was modeled: "evaluateTargetPriority". This operation is called by the local "identifyTarget" operation.

6.3.2.1 Activity Diagram for "identifyTarget" Operation

Figure 6.8 shows the activity diagram for target's identification operation. This operation has three activities: it starts by checking the target's model; then it evaluates its priority; and

finally, it evaluates the needed level of damage for that target. Figuring out the target's model is a sophisticated operation that requires knowledge of the different kinds of aircraft and missiles. One of the methods to check the target's model uses the fact that the engines of an airplane modulate the transmitted radar signal. The reflected radar signal can be compared to information from the ship's database about different targets and their effect on the radar signals. Other methods involve image processing of data from the target and fire control sensors. These sensors can give high-resolution images of the targets. Computer vision theories can be helpful in this regard. Unfortunately, this process is beyond the scope of this thesis. These operations might be considered later in future steps of this project.

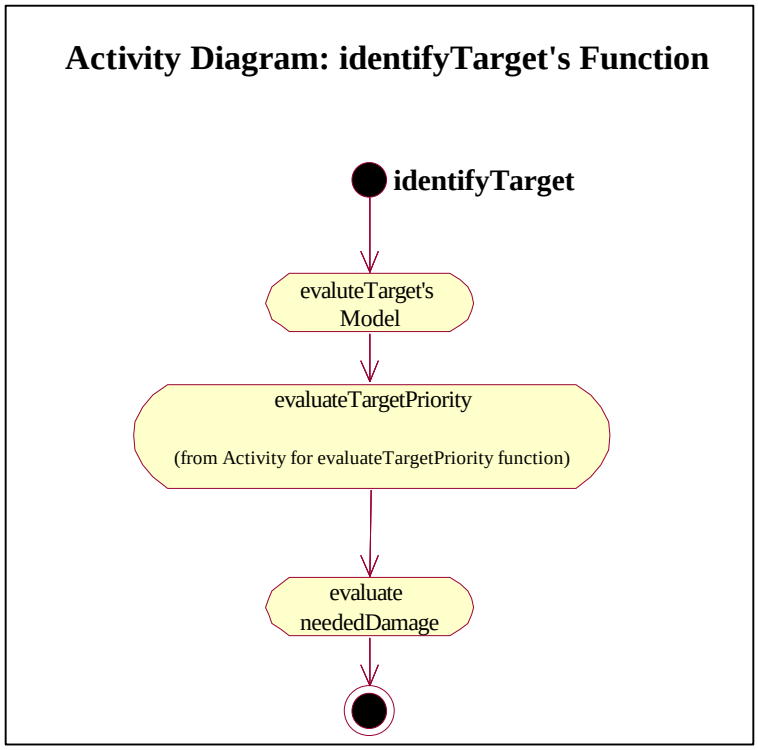

Fig 6.8: Activity diagram for "identifyTarget" operation

6.3.2.2 Activity Diagram for Evaluating Needed Damage Level

Figure 6.9 shows the activity diagram for evaluating the needed damage level. The needed level of damage is a key point in the ship command. Deciding when to stop shooting at a

certain target can affect the whole performance of the fleet. The commander should make his decision based on the target's type and model and on the ship's mission, supplies, and state. Naval clients can bring their input to bear on the needed level of damage. This input is represented by a doctrine file. This file has six entries; each entry represents the needed damage level for one of the six cases shown in Figure 6.9.

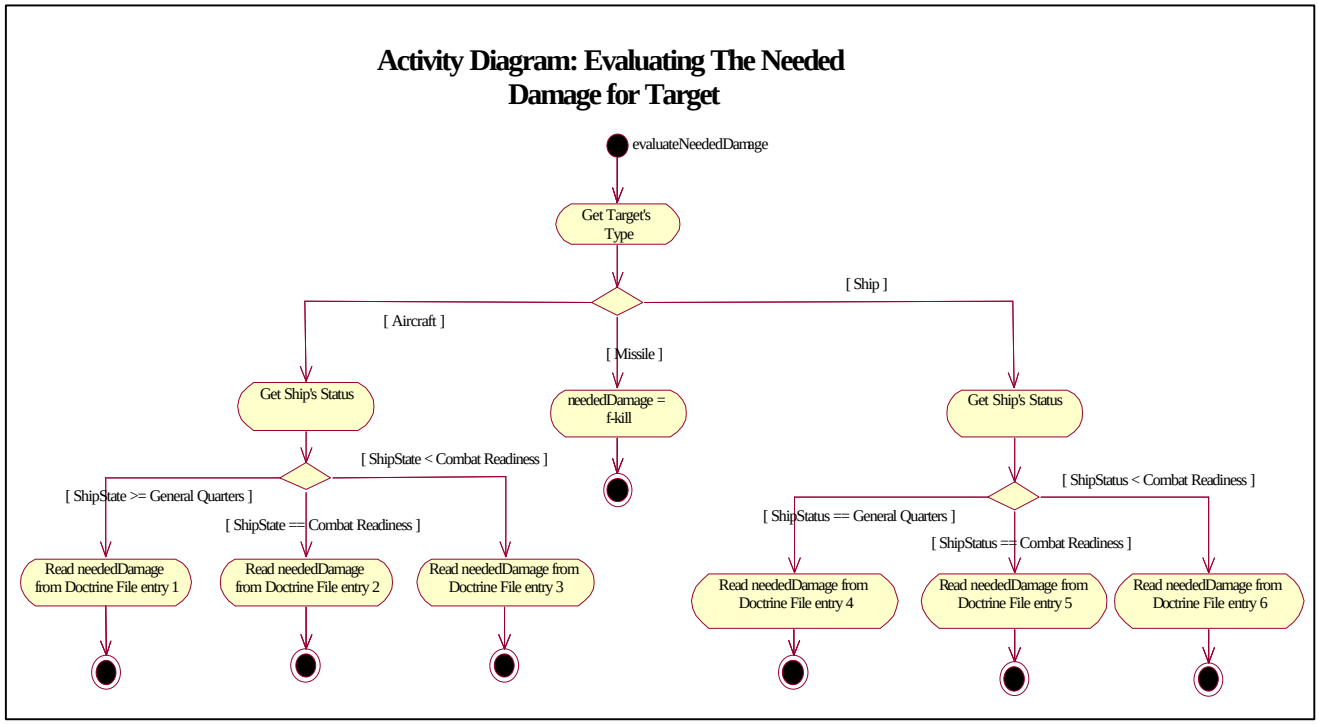

Fig 6.9: Activity diagram for "evaluateNeededDamage" operation

6.3.2.3 Activity Diagram for Evaluating the Target's Priority

Evaluating the target's priority is a key point in assigning interceptors to targets, maintaining the target's knowledge, and assessing the needed damage level. Making decisions about prioritizing the targets is another field which can benefit from input from the Naval clients. Figure 6.10 shows the activity diagram used to evaluate the target's priority. However, in this diagram some suggested priority values for the different situations were hardcoded[1]. From the

[1] Hardcoded is said of a data value or behavior written directly into a program, possibly in multiple places, where it cannot be easily modified. <http://ww.dictionary.com/>

diagram, one can see that the highest priority is given for a missile heading towards the ship. The second highest priority is given for an aircraft threatening the ship. The third priority is given for a missile not threatening the ship itself. Actually, this missile could be threatening a partner, and then it could get a higher priority. There are many other situations covered in the diagram. One can modify these hardcoded values so that they can be read from a doctrine file.

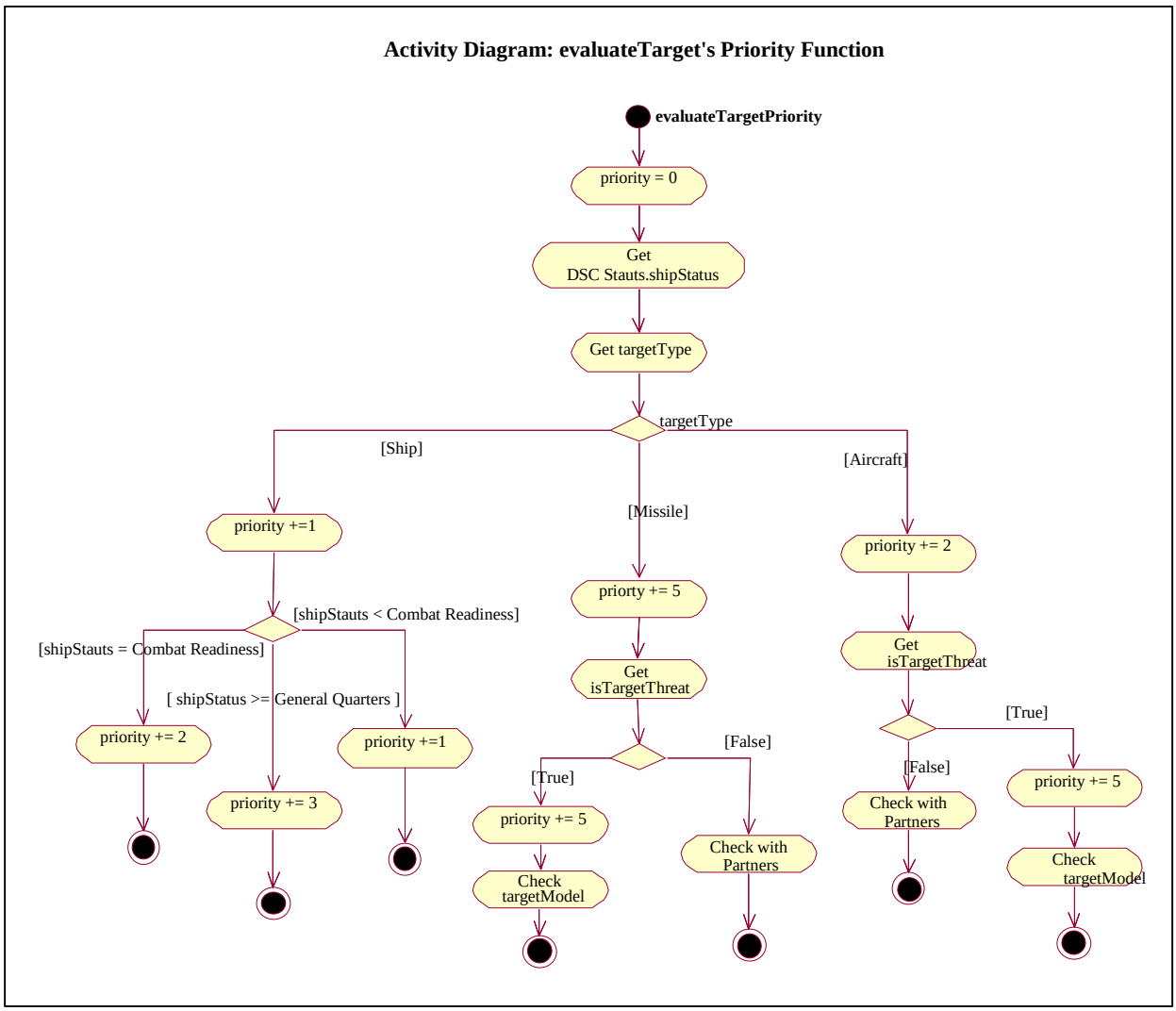

Fig 6.10: Activity diagram for “evaluateTargetPriority” operation

6.4 Shooter Grid's Operations

The shooter grid contains three major operations for evaluating the expected outcome of the engagement: when making the engagement decision, when evaluating the damage of the target, and when evaluating the success or failure of the engagement.

6.4.1 Activity Diagram for Evaluating Expected Outcome

The "evaluateOutcome" operation is defined in the "DSC Decision Making" and is responsible for evaluating the outcome of the engagement. This operation starts by checking the target's type and the inventory of the suitable defense interceptors. Figure 6.11 shows the activity diagram for evaluating the outcome.

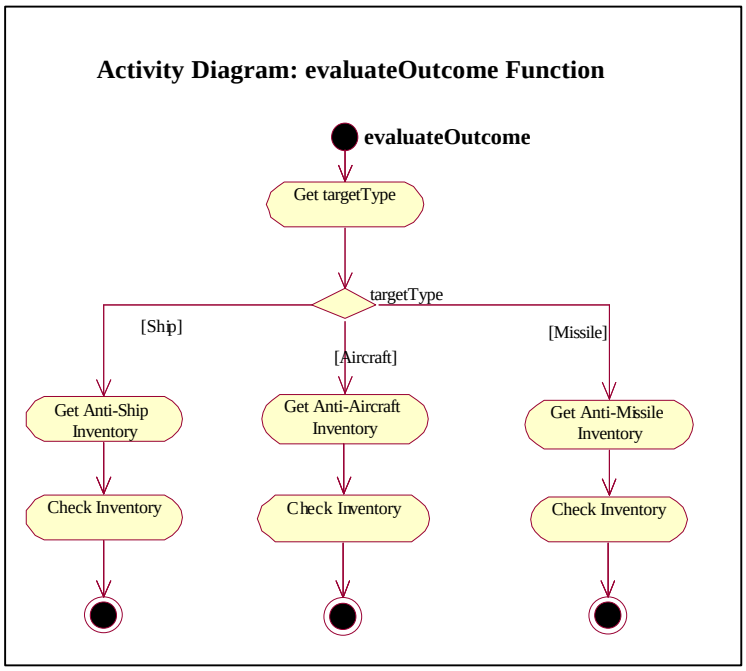

Fig 6.11: Activity diagram for "evaluateOutcome" operation

6.4.1.1 Activity Diagram for "Check Inventory" Activity

The "Check Inventory" activity is a higher-level non-atomic[1] activity that needs to be specified by its own activity diagram. Figure 6.12 shows the activity diagram for the "Check Inventory" activity.

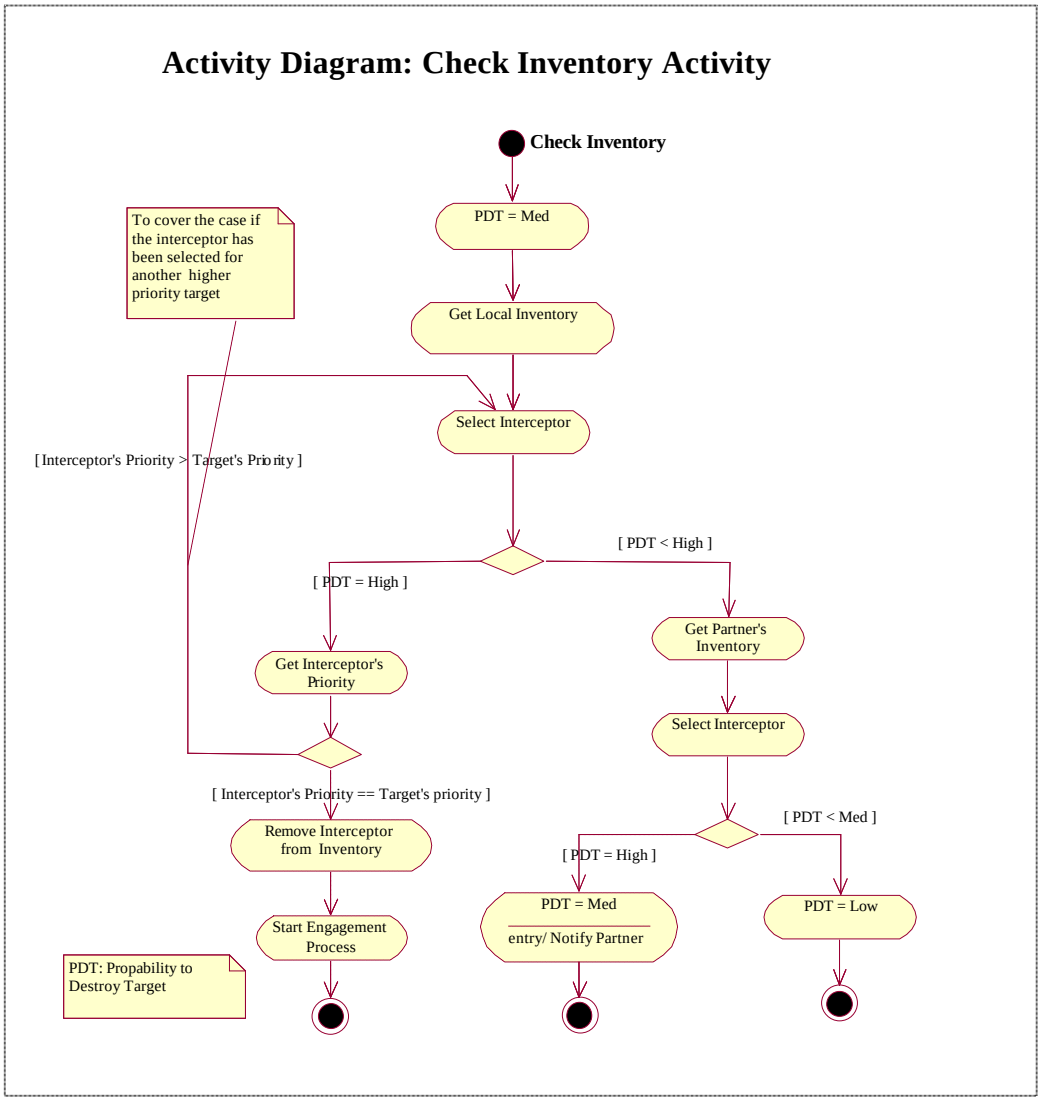

Fig 6.12: Activity diagram for "Check Inventory" activity

[1] Non-atomic activity is the one that can be broken down into another activity diagram.

The “Check Inventory” activity starts by setting the Priority to Destroy Target (PDT) to an intermediate value “Med”. Then the activity selects an interceptor from the local inventory. The “Select Interceptor” activity is another high-level non-atomic activity. It will set the PDT to either a high or a low value, depending on whether or not the inventory contains enough and suitable interceptors. If the PDT is high, the priority of the interceptor is checked again to make sure that the same interceptor was not selected for anther higher priority target. If the priority has not been changed, the interceptor is removed from the inventory, and the engagement process is started. The engagement process is described by the interaction diagrams of the shooter grid.

If the initial estimate of the PDT is Low, the operation checks the partner’s inventory for suitable interceptors. As before, the PDT could be set to either low or high. If the PDT is set to high, the partner will be able to destroy the target. At this point, the PDT is down graded to med; setting the PDT to med is used as a guide and not to take action against the target and leave this work to the partner.

6.4.1.2 Activity Diagram for “Select Interceptor” Activity

In selecting an interceptor for a specific target, it is important to differentiate between the priority of the target and that of the interceptor. The target’s priority is a measure of its danger to the ship, while the interceptor’s priority is the priority of the target to which the interceptor is assigned. The interceptor’s priority is used to check whether the current interceptor is already assigned to another target; if so, the interceptor’s priority is used to compare the priority of the new target to that of the interceptor. Interceptors are selected in favor of the target with higher priority when the resources are scarce.

Figure 6.13 shows the activity diagram for “Select Interceptor” activity. The activity starts by searching the inventory for a capable interceptor with a zero priority (not assigned to any other targets). Hence, the interceptor is first selected from among those interceptors not already been selected. If an appropriate interceptor is found in the inventory, it is selected by setting its priority equal to the target’s priority and setting the PDT to high. Otherwise, the activity searches the inventory for other interceptors that are already selected by lower priority targets. If such an interceptor is found, it is set equal to the priority of the target and the PDT is set

equal to high. Such actions are the reason why each interceptor needs to be checked before being used by a target in Figure 6.12. Finally, if none of the interceptors available in the inventory matches the above criteria, the PDT is set equal to low.

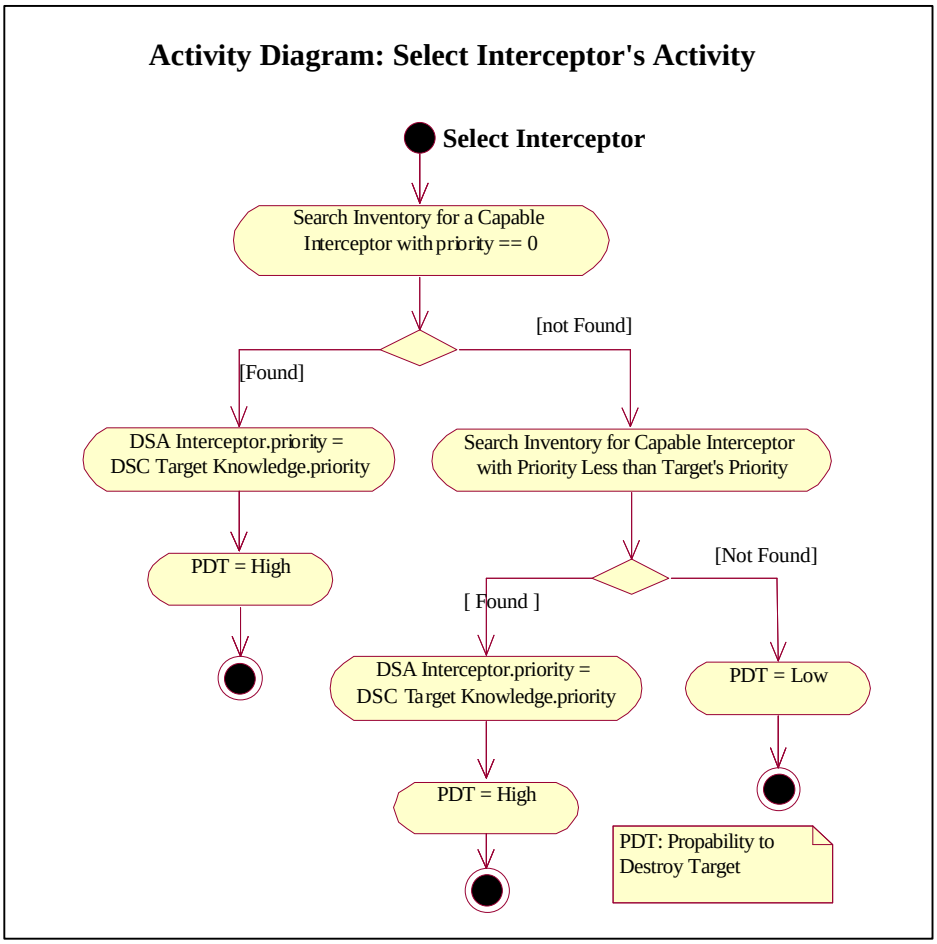

Fig 6.13: Activity diagram for "Select Interceptor" activity

6.4.2 Activity Diagram for "evaluateSuccess" Operation

The operation for evaluating success is implemented in the "DSC Evaluate Outcome". This operation compares the target's damage level with the needed damage level for this specific target. If the achieved damage due to engagement is greater than or equal to the needed damage level, the engagement is considered a success. Otherwise, the engagement is

considered a failure, and further actions will be taken. Figure 6.14 shows the activity diagram for the “evaluateSuccess” operation.

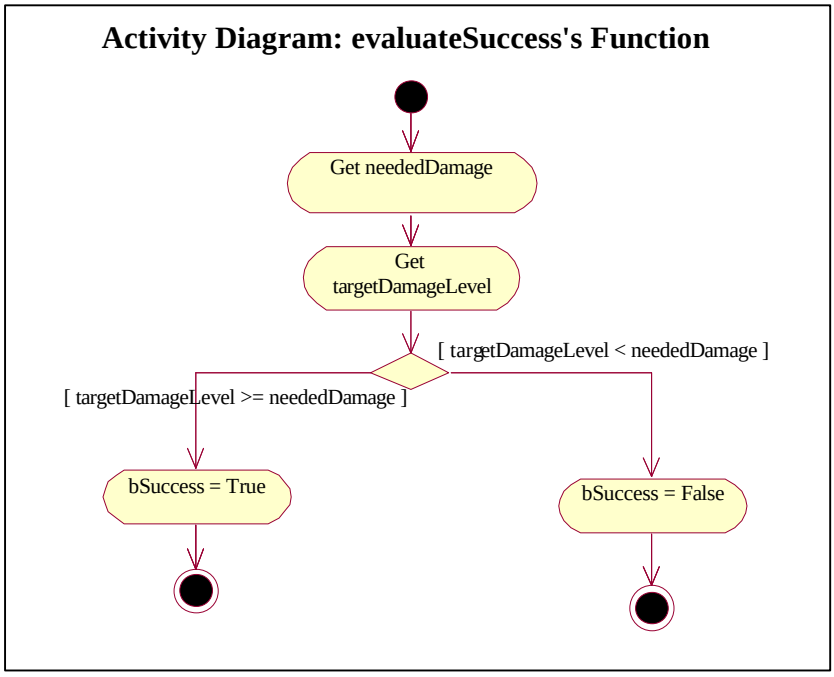

Fig 6.14: Activity diagram for “evaluateSuccess” operation

6.5 Next Step

At this point, the system is completely specified in all three domains. The next step is to incrementally generate the code for this system. A sample code generation for the system is covered in Chapter 7.

Chapter 7

PHYSICAL VIEW: COMPONENT DIAGRAMS AND CODE GENERATION

This chapter introduces the component diagrams and their use in the code generation process for the UML models. A sample C++ code of the "DSC Status" is discussed and listed later in this chapter.

Automated tools[1] were used to generate a skeleton[2] code of the classes and their operations. Since the UML models are independent of any programming language and since they were not linked to any existing library, these models are highly flexible and easily portable. In this chapter, the process of writing the source code from the state chart diagrams of the digital ship's "DSC Status" is introduced. Writing the code from activity diagrams is rather a straightforward process; activity diagrams are similar to the flow chart diagrams used for modeling sequential codes[3]. The source code is generated for both C++ and Java programming languages. The Java code generation was carried out by Surendranath Ramasubbu, a graduate student at Virginia Tech. I worked on the generation of the C++ code; examples of this code are discussed in this chapter.

7.1 Component Diagrams for the Combat System

Component diagrams for the logical diagrams were built to guide the code generation process. These component diagrams enabled the automated code generation tool to generate the source code for the different classes in files and to build the directory structure for the system.

[1] Rational Rose ®

[2] A skeleton code is the term used to refer to the prototypes of the functions and classes

[3] A sequential code refers to non-OOP code that is used to fill the body of some procedures. A sequential code is made of simple instructions and loops.

Visibility of the different files was specified using the dependency relationships. Unlike dependency relationships between classes, component dependency relationships are transitive; that is, if component A is dependent on component B, and B is dependent on component C, implies that component A is dependent on component C.

It is important to understand the compiler's behavior at this point. When the compiler tries to compile a dependent file of the source code, it starts by compiling all of its guardians[1]. In other words, the compiler follows the dependency graph (component diagram) of the source code. Hence, one should be careful not to make any loops in the component diagrams; such loops would cause "infinite iteration" errors at the compile-time of the program. If component A is dependent on component B and component B depends on component A, then when A is compiled, B will be compiled, and vice versa. This will lead to the compiler being trapped between compiling A for its dependence on B and B for its dependence on A.

On the other hand, the visibility of each component requires that before any dependent component is compiled, all of the components it depends on must be compiled. A tree structure would be required to meet these two conditions.

Figure 7.1 shows the component diagram for the digital ship's system. This diagram contains the digital ship's package and one component modeling a file for the user defined data types called "Data Types".

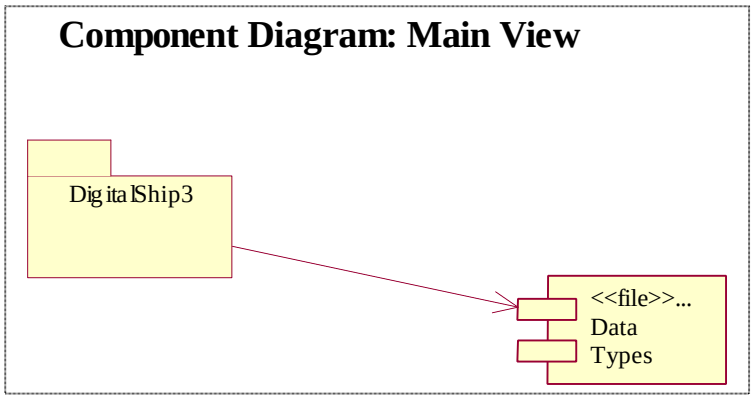

Fig 7.1: The component diagram of the digital ship's system

[1] Guardian is the independent member of the dependency relationship. If A is dependent on B, B is the guardian of A.

The digital ship's package is modeled with another package diagram shown in Figure 7.2. The diagram contains one package, "Combat Systems", and two components: "MainProgram" and "ShipStatus". The "MainProgram" component represents the main file where the function "main ()" is implemented. This function is responsible for creating instances of the needed classes. The "ShipStatus" component models the file that implements the "DSC ShipStatus".

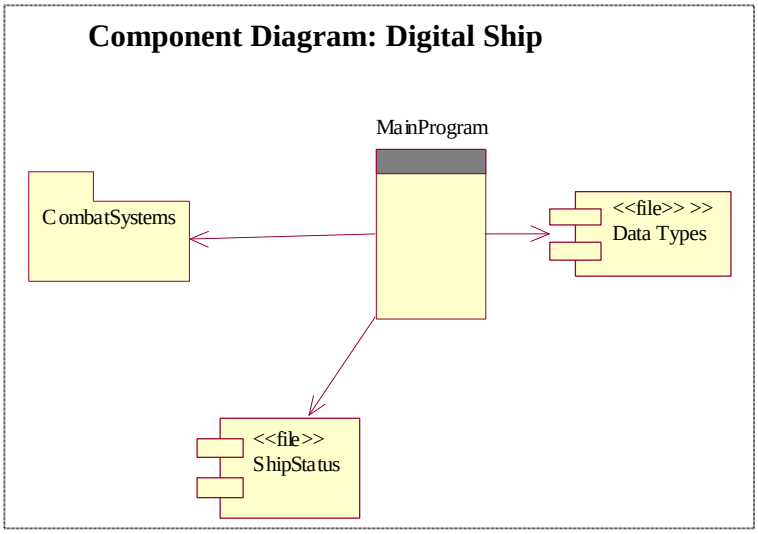

Fig 7.2: The component diagram for the digital ship's package

Figure 7.3 shows the component diagram for the package contents of the combat system. From the diagram, one can see that a tree structure is implemented by the dependency relationships between the packages.

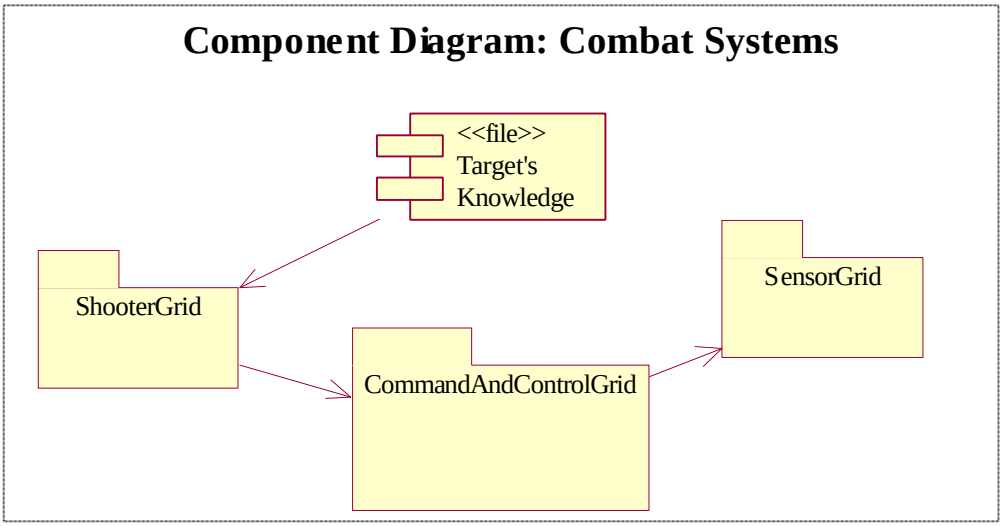

Fig 7.3: The combat system's component diagram

Starting from the component diagram for the "Sensor Grid" package (Figure 7.4), one can observe that this diagram contains the leaf components[1], whose compilation would be needed for any other component to compile correctly. These components need to be visible to all elements of the system, starting from the "Data Processing Unit". Two files are introduced in this diagram: "Sensors" to implement the classes for the sensors of the combat system and "Data Processing Unit" to implement the needed classes for processing the data from the sensors (see Figure 4.3). The "Info Proc" file from the command and control package shows the link between the two packages. From the transitivity of the dependency relationship, one can notice that the "Info Proc" file is dependent on the "Data Types" file.

[1] A leaf in a graph is an end vertex that has only one edge connecting it to its parent. In the dependency graph, a leaf component refers to the component that is independent of all of the other components and other components depend on it.

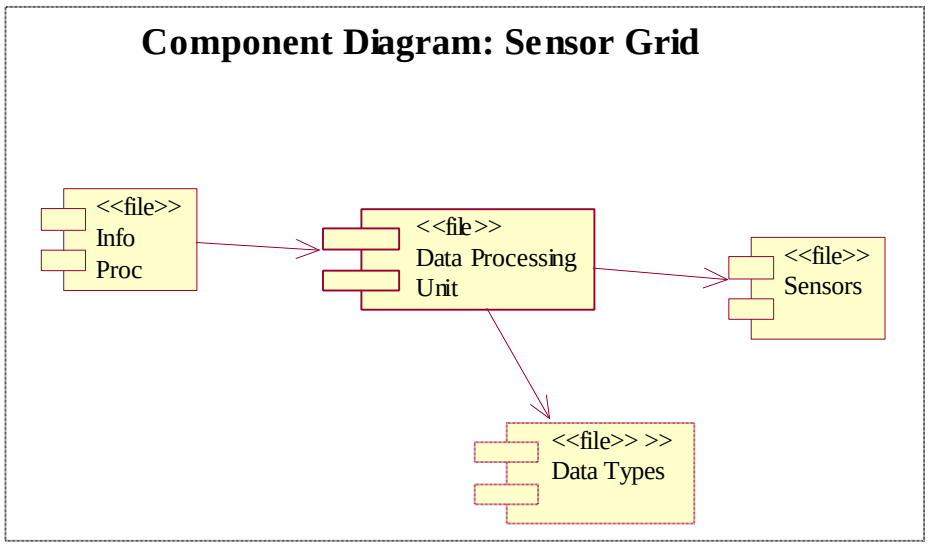

Fig 7.4: The component diagram for the Sensor grid

The component diagram for the command and control's package is shown in Figure 7.5. This diagram contains three files (components): "Info Proc" to implement the classes used for processing information[1], "Knowledge Processing" to implement the knowledge processing classes, and "Partner's status" to implement the "DSC PartnerStatus". The "Decision Making" file from the shooter grid's package shows the link between the package of the shooter and the command and control grids.

[1] See Figure 4.4 for details about the command and control grid

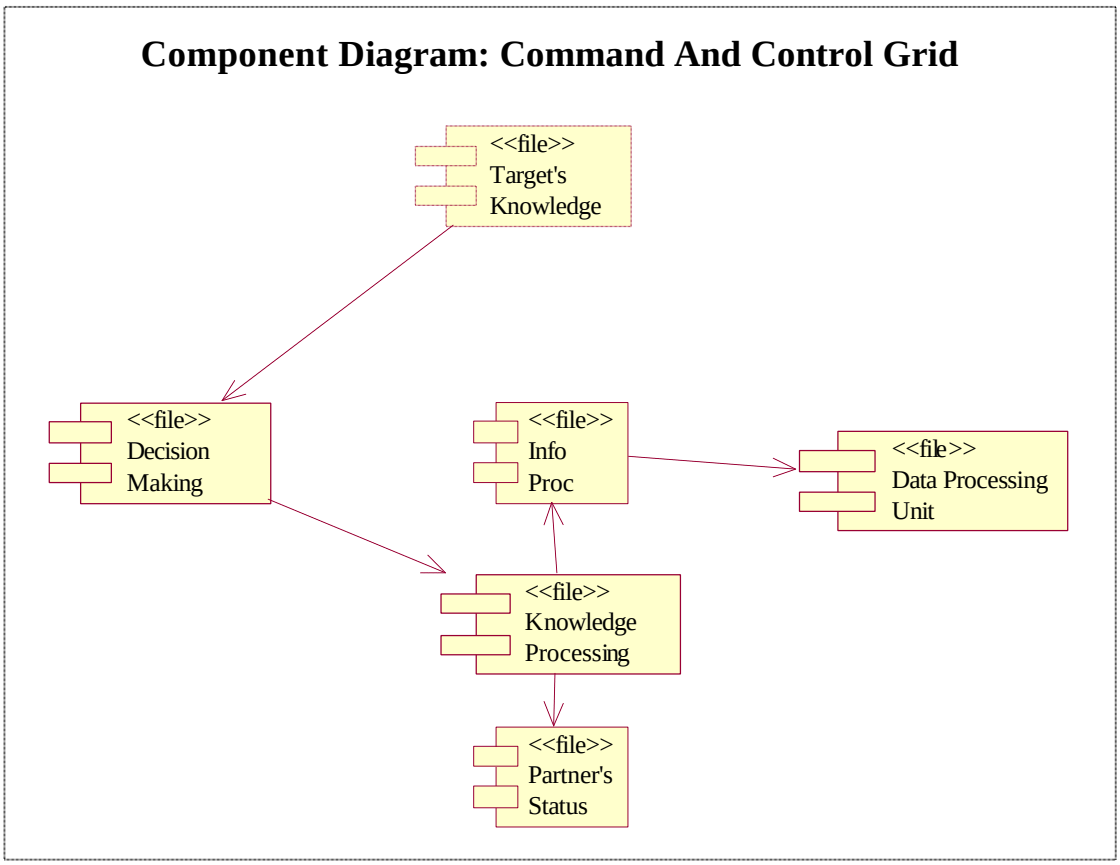

Fig 7.5: The component diagram for the command and control package

Figure 7.6 shows the component diagram for the shooter grid. Each class in the shooter grid[1] is implemented in one file (component). The "Target's Knowledge" component is dependent on the "Decision Making" component; hence, it is dependent on all of the components of the sensor-to-shooter grid.

[1] See Figure 4.5 for details about the class diagram for the shooter grid

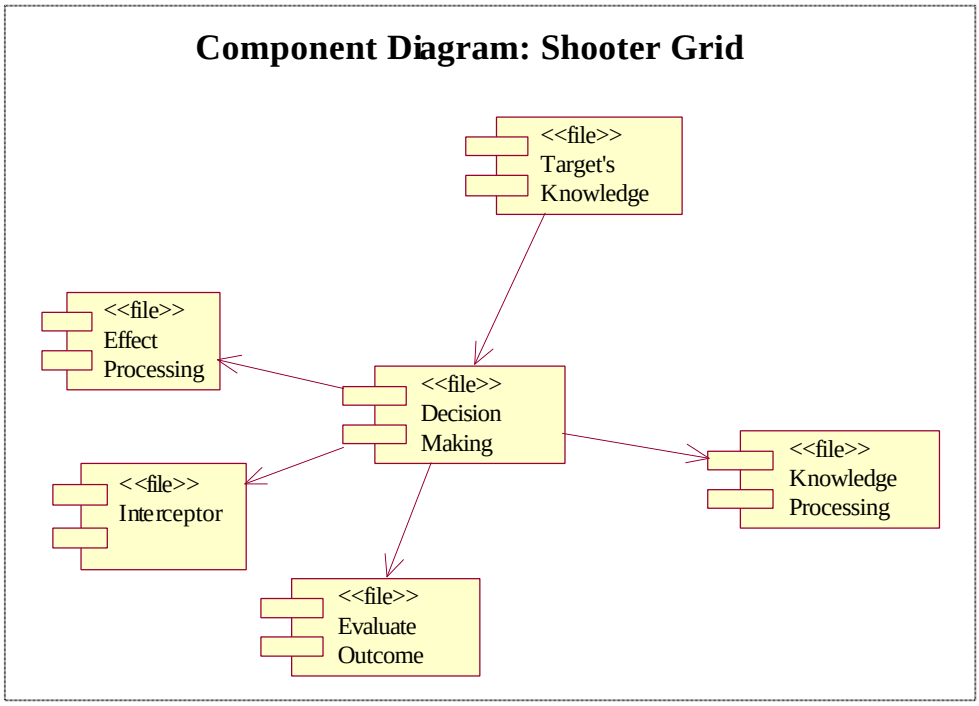

Fig 7.6: The component diagram for the shooter grid

The automated tool (Rose) was used to generate the skeleton code for these files (components). The next section introduces the steps taken towards generating the code for the “Ship Status” component.

7.2 C++ Code Generation

The C++ skeleton code was generated for the combat system and the “DSC Status”. The automated tool generated the prototypes of the class along with the prototypes of its operations and attributes. The required (Get / Set) operations are automatically generated also. The “Get” and “Set” are used for reading and writing the values of the attributes, respectively. The actual implementation of the operations had to be written by hand in light of the associated activity and state chart diagrams. However, the code generation tools are not perfect yet; hence, further modifications of the automatically generated code might be needed. When the code is

generated in C++, the main function has to be written by hand. However, if the generated code is written in Java or MFC[1], the main functions will be part of the classes themselves.

In each class, the automated tools generate the skeleton code for the following:

- "Constructor" and "destructor" operations
- Operators: assignment '=', equality '==' and inequality '!='
- User defined operations and attributes

When writing the code for the "DSC Status", one should fill out the body of the functions whose prototypes are generated automatically. The body for the constructor[1] function with an initialization object parameter is built by setting the state of the initializing object to the state of the created object, as shown below:

```
this->set_state(right.get_state());
```

The body for the assignment operation '=' function is built by creating an object of the class using the constructor's operation. The new object is created using the right side of the '=' operation as the initialization object. This operation returns the created object, as shown below:

```
return DSC_Ship_Status(right);
```

The body for the equality operator "==" is built by checking if the sate of the current object is equal to that of the right side object. This operator is called as follows:

```
if ( ourObject == rightObject)
{
// The objects are equal
}
```

The body of the '==' operation is implemented as follows:

[1] Microsoft Foundation Classes®

```
if ( this->get_state() == right.get_state() )
   return 1;
else
   return 0;
```

The “updataState(DSDT_eventType event)” operation is generated from the state chart diagram (Figure 6.1). The code was implemented as a finite state machine. The body of the “updataState ()” operation has a “switch” statement that checks the value of the “event” parameter of the “DSDT_eventType” type. If the “event” was “Need Refill”, the current state of the ship is stored in the “previousState” variable, and ship “state” is set to “Refilling” and the “refill ()” operation is called to simulate the refilling process.

Moreover, the code generated accounts for the case when the ship is in “Underway Replenishment” and the “need Refill” event occurs; in this case, the ship should keep track of both previous states: “Underway Replenishment” and the state before “Underway Replenishment”. Another variable “firstPreviousState” was used for this purpose. When the “Need Refill” event is triggered, the “updataState ()” operation checks if the current state is “Underway Replenishment”. If so, the “firstPreviousState” is set to “previousState”, the “previousState” is set to the current state, and “previousState” is set to “Refuling”. This part of the code is shown below:

```
   case Need_Refill:
         if (state == Underway_Replenishment)
         {
               firstPreviousState = previousState;
               previousState = state;
         }
         else if ( state != Refuling)
               previousState = state;
         state = Refuling;
         refill();
         break;
```

[1] Constructor is the first function that is called when an object of the class is created. This function is responsible for initializing the attributes of this function. Sometimes the constructor function includes a parameter object of the same type as the created class. In this case, the attributes of the created are set to the attributes of the parameter object.

Appendix A shows the source code listing. Listing A.1 shows the source code for the "shipStatus" component. The manually added code is indicated by ***underlined italic bold*** text. The source code for the "MainProgram" component is shown in listing A.2. In this source code, the "Ship Status" and "Data Types" components are made visible by including the "Ship Status.h" and "Data Types.h" files.

Listing A.3 shows the automatically generated code for the "Data Types" component. This component is implemented in the "Data Types.h" file.

Chapter 8

CONCLUSIONS AND RECOMMENDATIONS FOR FURTHER RESEARCH

The UML is shown to be a great aid in building software intensive systems and modeling the software in an easy to comprehend way. The modeling was the tool that led the software development process. Going through the different stages of this process, we found that UML helped visualize, specify, construct, and document these models.

UML enabled the specification of the system at different levels of abstraction and guided the design to build and model a distributed Object-Oriented simulation. Modifications to the UML models were much easier than changes to the code. Modifications to the code should be reflected in the UML models, and vice versa.

Explaining the ideas with use case diagrams was easier than with natural language. Generating codes from the well-written UML models was a straightforward job. However, the UML diagrams need to be specified at a low level of abstraction (source code level); otherwise, additional effort would be needed to code these ideas, and additional effort would be needed to reverse-engineer the code back to UML models for documentation and validation. This process of getting the code from the UML models and the UML models from the code, and vice versa, is known as round trip engineering.

It was clear during the research that the abstraction had aided a great deal in concentrating the attention of the reader in the relevant aspects he needs to know, keeping the diagrams simple and manageable for the reader to understand, and building a good documentation of the generated code. There is always a trade off between the level of details in the models and the effort that is left for code generation. Simpler models are easier to understand, but leave more choice to the code generator; however, low-level models guarantee exact matching between the code and the specification, but makes the models more difficult to understand. An intermediate

solution can be to facilitate building structured diagrams by explaining highly abstract elements with separate diagrams, which can be referenced by the reader if he needs them.

Dependency is a transitive relationship between components in the component diagrams. Thus, the component diagrams should have a tree structure to prevent infinite loops for the compiler.

Surendranath Ramasubbu was able to start writing the code in Java with minimum communication with either the domain experts or myself. This shows the power of using UML diagram as documentation for legacy applications when new members join the team.

8.1 Contribution of this Thesis

A digital model of the combat and control system of a ship was built using UML. The work presented in this thesis has aided in introducing the UML as a means for specifying and building a well-documented software artifacts to be used in the simulation of the digital ship. The UML diagrams for the digital ship were built and a sample code was generated to illustrate the process of code generation.

The process of building the models of the system started by meeting with the domain expert, Dr. Habayeb. Those meeting aided in building the use case diagrams, which were later validated by the domain expert. Each use case of these diagrams was explained with an activity diagram. The use case and activity diagrams were used to build the logical view of the system. Building the logical view was based on the three stages introduced by Savio-Vazquez et al [13] for building the structural domain (class diagrams). The class diagrams were used in building the interaction domain (interaction diagrams), which described the message passing behavior between classes to achieve the behavior of the use case diagrams. The interaction and class diagrams were used to build the activity diagrams that specify the behavior of the operations of the classes. After modeling the logical view, the physical view of the system was built using component diagrams. A sample source code was generated for the Digital Ship

Class Status based on all of the component, class, and state chart diagrams, which describe that class.

8.2 Recommendations for Future Work

The process used in this thesis can be extended and used to cover a reverse-engineering path to pass modifications in the source code to the existing UML models. This modification will guarantee having an up to date documentation for the code. This process is known in UML as the round-trip-engineering process.

Building models for the rest of the ship's organizations would be needed; these models need to be integrated later to achieve a suitable test bed for the command and control system. The same process that was followed for generating the code for the "DSC Status" need to be followed to generate the source code for all of the UML models of the combat systems and the digital ship.

Finally, building the deployment diagrams would be helpful. The deployment diagrams specify the distribution of the components of the system into different machines and servers. These diagrams will be needed when the system evolves to a complex distributed system to aid in debugging and maintaining the distributed software. Building the deployment diagrams will be the last step needed after building the complete system models and before running the complete system code.

APPENDIX A

SOURCE CODE LISTING

Listing A.1: The Source Code for the "Ship Status.h" File

```
//## begin module%1.3%.codegen_version preserve=yes
//   Read the documentation to learn more about C++ code generator
//   versioning.
//## end module%1.3%.codegen_version

//## begin module%3AD478C101F4.cm preserve=no
//  %X% %Q% %Z% %W%
//## end module%3AD478C101F4.cm

//## begin module%3AD478C101F4.cp preserve=no
//## end module%3AD478C101F4.cp

//## Module: ShipStatus%3AD478C101F4;
//## Subsystem: Digital Ship3%3AD47896000F
//## Source file: D:\Program Files\Rational\Rose\C++\source\Digital
//## Ship3\ShipStatus.h

//## begin module%3AD478C101F4.additionalIncludes preserve=no
//## end module%3AD478C101F4.additionalIncludes

//## begin module%3AD478C101F4.includes preserve=yes
//## end module%3AD478C101F4.includes

//## begin module%3AD478C101F4.declarations preserve=no
//## end module%3AD478C101F4.declarations

//## begin module%3AD478C101F4.additionalDeclarations preserve=yes
```

```
//## end module%3AD478C101F4.additionalDeclarations


//## begin DSC_Ship_Status%3A7B232E029F.preface preserve=yes
//## end DSC_Ship_Status%3A7B232E029F.preface

//## Class: DSC Status%3A7B232E029F
//## Category: Digital Ship3%3A7B22A002FD
//## Subsystem: Digital Ship3%3AD47896000F
//## Persistence: Transient
//## Cardinality/Multiplicity: n



class DSC_Ship_Status
{
  //## begin DSC_Ship_Status%3A7B232E029F.initialDeclarations preserve=yes
  //## end DSC_Ship_Status%3A7B232E029F.initialDeclarations

  public:
    //## Constructors (generated)
      DSC_Ship_Status();

      DSC_Ship_Status(const DSC_Ship_Status &right);

    //## Destructor (generated)
      ~DSC_Ship_Status();

    //## Assignment Operation (generated)
      DSC_Ship_Status & operator=(const DSC_Ship_Status &right);

    //## Equality Operations (generated)
      int operator==(const DSC_Ship_Status &right) const;

      int operator!=(const DSC_Ship_Status &right) const;
```

```
  //## Other Operations (specified)
    //## Operation: updataState%3AC7BE4100AB
    void updataState (DSDT_eventType);

    //## Operation: dropAnchor%3AD3D3A60399
    void dropAnchor ();

    //## Operation: refill%3AD3D3AB03D8
    void refill ();

    //## Operation: increaseSpeed%3AD3D3B20399
    void increaseSpeed ();

    //## Operation: sail%3AD3D4D602DE
    void sail ();

    //## Operation: loadSupplies%3AD3D50300BB
    void loadSupplies ();

    //## Operation: soundAlarm%3AD3D4D900BB
    void soundAlarm ();

    //## Operation: prepareLifeboats%3AD3D4EA004E
    void prepareLifeboats ();

    //## Operation: leaveShip%3AD3D4FA008C
    void leaveShip ();

  // Additional Public Declarations
    //## begin DSC_Ship_Status%3A7B232E029F.public preserve=yes
    //## end DSC_Ship_Status%3A7B232E029F.public

protected:
  // Additional Protected Declarations
    //## begin DSC_Ship_Status%3A7B232E029F.protected preserve=yes
    //## end DSC_Ship_Status%3A7B232E029F.protected
```

```
  public: //private: modified by MSA to enable the reading of

//the state from outside for testing.
    //## Get and Set Operations for Class Attributes (generated)

      //## Attribute: state%3AC7BE5C00CB
      const DSDT_shipStatusType get_state () const;
      void set_state (DSDT_shipStatusType value);

    // Additional Private Declarations
      //## begin DSC_Ship_Status%3A7B232E029F.private preserve=yes
      //## end DSC_Ship_Status%3A7B232E029F.private

  private: //## implementation
    // Data Members for Class Attributes

      //## begin DSC_Ship_Status::state%3AC7BE5C00CB.attr preserve=no
private: DSDT_shipStatusType {U} Sea_And_Anchor
      DSDT_shipStatusType state;
      //## end DSC_Ship_Status::state%3AC7BE5C00CB.attr

    // Additional Implementation Declarations
      //## begin DSC_Ship_Status%3A7B232E029F.implementation preserve=yes
      //## end DSC_Ship_Status%3A7B232E029F.implementation

};

//## begin DSC_Ship_Status%3A7B232E029F.postscript preserve=yes
//## end DSC_Ship_Status%3A7B232E029F.postscript

// Class DSC_Ship_Status

//## Get and Set Operations for Class Attributes (inline)

inline const DSDT_shipStatusType DSC_Ship_Status::get_state () const
{
  //## begin DSC_Ship_Status::get_state%3AC7BE5C00CB.getpreserve=no
  return state;
```

```
  //## end DSC_Ship_Status::get_state%3AC7BE5C00CB.get
}


inline void DSC_Ship_Status::set_state (DSDT_shipStatusType value)
{
  //## begin DSC_Ship_Status::set_state%3AC7BE5C00CB.set preserve=no
  state = value;
  //## end DSC_Ship_Status::set_state%3AC7BE5C00CB.set
}


// Class DSC_Ship_Status


DSC_Ship_Status::DSC_Ship_Status()
  //## begin DSC_Ship_Status::DSC_Ship_Status%3A7B232E029F_const.hasinit
//preserve=no
      : state(Sea_And_Anchor)
  //## end DSC_Ship_Status::DSC_Ship_Status%3A7B232E029F_const.hasinit
  //## begin
//DSC_Ship_Status::DSC_Ship_Status%3A7B232E029F_const.initialization
//preserve=yes
  //## end
//DSC_Ship_Status::DSC_Ship_Status%3A7B232E029F_const.initialization
{
  //## begin DSC_Ship_Status::DSC_Ship_Status%3A7B232E029F_const.body
//preserve=yes
  //## end DSC_Ship_Status::DSC_Ship_Status%3A7B232E029F_const.body
}


DSC_Ship_Status::DSC_Ship_Status(const DSC_Ship_Status &right)
  //## begin DSC_Ship_Status::DSC_Ship_Status%3A7B232E029F_copy.hasinit
//preserve=no
      : state(Sea_And_Anchor)
  //## end DSC_Ship_Status::DSC_Ship_Status%3A7B232E029F_copy.hasinit
  //## begin
//DSC_Ship_Status::DSC_Ship_Status%3A7B232E029F_copy.initialization
//preserve=yes
  //## end DSC_Ship_Status::DSC_Ship_Status%3A7B232E029F_copy.initialization
{
  //## begin DSC_Ship_Status::DSC_Ship_Status%3A7B232E029F_copy.body
//preserve=yes
```

```
this->set_state ( right.get_state () );
  //## end DSC_Ship_Status::DSC_Ship_Status%3A7B232E029F_copy.body
}



DSC_Ship_Status::~DSC_Ship_Status()
{
  //## begin DSC_Ship_Status::~DSC_Ship_Status%3A7B232E029F_dest.body
//preserve=yes
  //## end DSC_Ship_Status::~DSC_Ship_Status%3A7B232E029F_dest.body
}



DSC_Ship_Status & DSC_Ship_Status::operator=(const DSC_Ship_Status &right)
{
  //## begin DSC_Ship_Status::operator=%3A7B232E029F_assign.body
//preserve=yes
return DSC_Ship_Status(right);
  //## end DSC_Ship_Status::operator=%3A7B232E029F_assign.body
}



int DSC_Ship_Status::operator==(const DSC_Ship_Status &right) const
{
  //## begin DSC_Ship_Status::operator==%3A7B232E029F_eq.body preserve=yes
  if ( this->get_state() == right.get_state() )
      return 1;
   else
      return 0;
  //## end DSC_Ship_Status::operator==%3A7B232E029F_eq.body
}

int DSC_Ship_Status::operator!=(const DSC_Ship_Status &right) const
{
  //## begin DSC_Ship_Status::operator!=%3A7B232E029F_neq.body preserve=yes
  if ( this->get_state() != right.get_state() )
    return 1;
  else
```

```
    return 0;
  //## end DSC_Ship_Status::operator!=%3A7B232E029F_neq.body
}



//## Other Operations (implementation)
void DSC_Ship_Status::updataState (DSDT_eventType event)//[1]
{
  //## begin DSC_Ship_Status::updataState%3AC7BE4100AB.body preserve=yes
  static DSDT_shipStatusType previousState;
  static DSDT_shipStatusType firstPreviousState;
  switch(event)
  {
  case Full:
    if ( state== Refuling)
    {
      state = previousState;
     previousState = firstPreviousState;
    }
    else if(state == Underway_Replenishment)
    {
      previousState = firstPreviousState;
    }
    break;
  case Need_Refill:
    if (state == Underway_Replenishment)
    {
      firstPreviousState = previousState;
      previousState = state;
    }
    else if ( state != Refuling)
      previousState = state;
    state = Refuling;
    refill();
```

[1] See the state chart diagram for the updateStauts operation Figure 6.1

```
    break;
  case Lift_Anchor:
    state = Cold_Iron;
    break;
  case Leaving_Shore:
    increaseSpeed();
    state = Underway;
    sail();
    break;
  case Replenishment_Needed:
    if( state == Refuling)
    {
      firstPreviousState = previousState;
      previousState = state;
    }
    else if ( state != Underway_Replenishment)
      previousState = state;
    dropAnchor();
    state = Underway_Replenishment;
    loadSupplies();
    break;
  case Replenishment_Finished:
    if ( state== Underway_Replenishment)
    {
      state = previousState;
      previousState = firstPreviousState;
    }
    else if(state == Refuling)
    {
      previousState = firstPreviousState;
    }
    break;
  case Threat_Detected:
    state = Combat_Readiness;
    break;
  case Damage_Onboard:
    soundAlarm();
```

```
    state = Damage_Control;
    break;
  case Targets_Destroyed:
    state = Combat_Readiness;
    break;
  case Under_Heavy_Fire:
    soundAlarm();
    state = General_Quarters;
    break;
  case Heavy_Destruction_Onboard:
    soundAlarm();
    state = Abandon_Ship;
    prepareLifeboats();
    leaveShip();
    break;
  default:
    state = Underway;
   }
  //## end DSC_Ship_Status::updataState%3AC7BE4100AB.body
}

void DSC_Ship_Status::dropAnchor ()
{
  //## begin DSC_Ship_Status::dropAnchor%3AD3D3A60399.body preserve=yes
  //## end DSC_Ship_Status::dropAnchor%3AD3D3A60399.body
}

void DSC_Ship_Status::refill ()
{
  //## begin DSC_Ship_Status::refill%3AD3D3AB03D8.body preserve=yes
  //## end DSC_Ship_Status::refill%3AD3D3AB03D8.body
}

void DSC_Ship_Status::increaseSpeed ()
{
  //## begin DSC_Ship_Status::increaseSpeed%3AD3D3B20399.body preserve=yes
  //## end DSC_Ship_Status::increaseSpeed%3AD3D3B20399.body
```

```
}

void DSC_Ship_Status::sail ()
{
  //## begin DSC_Ship_Status::sail%3AD3D4D602DE.body preserve=yes
  //## end DSC_Ship_Status::sail%3AD3D4D602DE.body
}

void DSC_Ship_Status::loadSupplies ()
{
  //## begin DSC_Ship_Status::loadSupplies%3AD3D50300BB.body preserve=yes
  //## end DSC_Ship_Status::loadSupplies%3AD3D50300BB.body
}

void DSC_Ship_Status::soundAlarm ()
{
  //## begin DSC_Ship_Status::soundAlarm%3AD3D4D900BB.body preserve=yes
  //## end DSC_Ship_Status::soundAlarm%3AD3D4D900BB.body
}

void DSC_Ship_Status::prepareLifeboats ()
{
  //## begin DSC_Ship_Status::prepareLifeboats%3AD3D4EA004E.body
  //## end DSC_Ship_Status::prepareLifeboats%3AD3D4EA004E.body
}

void DSC_Ship_Status::leaveShip ()
{
  //## begin DSC_Ship_Status::leaveShip%3AD3D4FA008C.body preserve=yes
  //## end DSC_Ship_Status::leaveShip%3AD3D4FA008C.body
}
// Additional Declarations
  //## begin DSC_Ship_Status%3A7B232E029F.declarations preserve=yes
  //## end DSC_Ship_Status%3A7B232E029F.declarations
//## begin module%3AD478C101F4.epilog preserve=yes
//## end module%3AD478C101F4.epilog
```

Listing A.2: The Source Code for the “MainFile.h” File

```
#include <iostream.h>
#include "Digital Ship2\DSC Status.h"
#include "Digital Ship2\Data Types.h"
#include "Digital Ship2\DSDT_eventType.h"

void main()
{
   DSC_Ship_Status Ship;
   cout << (int) Ship.get_state()<< endl;
   int event;
   cin >> event;

  while ( event > -1 && event < 11)
  {
      Ship.updataState((DSDT_eventType) event);
      cout << (int) Ship.get_state()<< endl;
      cin >> event;
  }
}
```

Listing A.2: The Source Code for the “Data Types.h” File

```
/* The automated comments generated by the tool are deleted from this file
by MSA.*/
enum DSDTtargetType
{
   Ship,
   Aircraft,
   Missile
};
// DSDTtargetType;

enum DSDTshipStatusType
{
   Sea_And_Anchor = 0,
   Cold_Iron,
   Refuling,
   Underway,
   Underway_Replenishment,
   Combat_Readiness,
   General_Quarters,
   Damage_Control,
   Abandon_Ship
};
// DSDTshipStatusType;

enum DSDTtargetModelType
{
};// DSDTtargetModelType;

enum DSDTeventType
{
   Full,
   Need_Refill,
   Lift_Anchor,
   Leaving_Shore,
   Replenishment_Needed,
   Replenishment_Finished,
```

```
   Threat_Detected,
   Damage_Onboard,
   Targets_Destroyed,
   Under_Heavy_Fire,
   Heavy_Destruction_Onboard
};
// DSDTeventType;

typedef struct
{
} DSDTgeometryType;


typedef struct
{
   float slope;
   float offset;
} DSDTtrackType;


typedef struct
{
   int priority;
   int speed;
   int warheadSize;
}interceptor;

typedef struct
{
   interceptor * Anti_Air_Interceptor;
   interceptor *Anti_Missile_Interceptors;
   interceptor *Anti_Ship_Interceptors;
} DSDTinventoryType;


enum DSDTdamageType {
   O_kill,
```

```
    b_kill,
    f_kill,
    m_kill,
    k_kill
};
// DSDTdamageType;
```

BIBLIOGRAPHY

[1] Hollenbach J and Misch G, "Linking C4I and modeling and simulation systems," *IEEE Military Communications Conference*, pp. 1126-1128, USA, 1995.

[2] Janowiak T, " A simulation for combat systems development and acceptance testing," *IEEE Simulation Conference*, pp. 210-213, USA, 1990.

[3] Hyer S, Johnston J, and Roe C, "Combat system effectiveness modeling to support development of anti-air warfare tactics," *Johns Hopkins APL Technical Digest*, vol. 16, pp. 69-82, USA, 1995.

[4] Joint Staff, " DoD dictionary of military terms," *Joint Electronic Library*,

<http://www.dtic.mil/doctrine/jel/doddict/> Feb 15, 2001.

[5] Jo K, McGreer M, and O'Brien G, " Global command and control system target architecture modeling," *IEEE MILCOM*, vol. 2, pp. 479-484, USA, 1994.

[6] Allen R, Arkin S, and Lawrence R, " Moving naval C^4I into the next century," *IEEE Communications Magazine*, vol. 33, pp. 96-105, USA, 1995.

[7] Rebbapragada V, " Distributed battle management for command and control," *IEEE MILCOM*, vol. 2, pp. 542-545, USA, 1994.

[8] Holt J, Newman T, Luscombe J, and Mathieson G, " A realistic simulation of human decision making behavior," *IEE International Conference on Information-Decision-Action Systems in Complex Organizations*, no. 353, pp. 20-24, UK, 1992.

[9] Qi Y, "The theory of C^3 countermeasure decision-making supporting system," *IEEE CIE International Conferenceof Radar Proceedings*, pp.586-589, China 1996.

[10] Siliato J, Sudnikovich W, Kennedy L, and Hallgring R, "Evolving modeling and simulation technology into fieldable products," *IEEE MILCOM*, vol. 3, pp. 907-911, USA, 1993.

[11] Schwalm N, Samet M, Silver J, and Novick Y, "Scenario generation for planning and execution monitoring," *IEEE MILCOM*, vol. 3, pp. 853-858, USA, 1994.

[12] Perse R, Callahan K, and Malone T, "Development of an AEGIS combatant integrated survivability management system (ISMS) modeling tool," *Proceedings of the Human Factors Society*, pp. 1201-1205, USA, 1991.

[13] Savino-Vazquez N and Ruigjaner R, "A UML-based method to specify the structural component of simulation-based queuing network performance models," *IEEE Simulation Symposium*, pp. 71-78, USA, 1999.

[14] Yao S, Tang F, and Liu Y, "An object-oriented model for parallel software," *IEEE Technology of Object-Oriented Languages*, pp. 245-250, USA, 1998.

[15] Huang H and Yeh C, "Development of a virtual factory emulator based on three-tier architecture," *IEEE International Conference on Robotics and Automation*, pp. 2434-2439, USA, 1999.

[16] Kortright E, "Modeling and simulation with UML and Java," *IEEE Simulation Symposium*, pp. 43-48, USA, 1997.

[17] Booch G, Rumbaugh J, and Jacobson I, "*The Unified Modeling Language: User Guide*," *Addison Wesley Longman, Inc.*, MA, **USA**, 1999.

VITA

Mohammad S. AL-Aqrabawi

Mohammad was born on August 7, 1976 in Sahab, Jordan. He received a B.S. degree in electrical engineering in 1999 from the University of Jordan, Jordan. He received his M.S. degree in computer engineering in 2001 from Virginia Polytechnic Institute and State University, USA. He research interests are modeling and simulation using UML. He joined the microprocessor design group at Intel Corporation in Portland, OR.