Conceptual framework approach for system-of-systems software developments

Caffall, Dale Scott

Monterey, California. Naval Postgraduate School

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

CONCEPTUAL FRAMEWORK APPROACH FOR SYSTEMS-OF-SYSTEMS SOFTWARE DEVELOPMENTS

by

Dale Scott Caffall

March 2003

Thesis Co-Advisor:	James Bret Michael
Thesis Co-Advisor:	Man-tak Shing

REPORT DOCUMENTATION PAGE	*Form Approved OMB No. 0704-0188*

1. AGENCY USE ONLY ***(Leave blank)***	**2. REPORT DATE** March 2003	**3. REPORT TYPE AND DATES COVERED** Master's Thesis

4. TITLE AND SUBTITLE: Conceptual Framework Approach for System-of-Systems Software Developments	**5. FUNDING NUMBERS**
6. AUTHOR(S) Dale Scott Caffall	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A	**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited	**12b. DISTRIBUTION CODE**

13. ABSTRACT ***(maximum 200 words)***

The Department of Defense looks increasingly towards an interoperable and integrated system-of-systems to provide required military capability. Non-essential software complexity of a system-of-systems can have a greater negative impact in system behavior than a single system. Our current systems-of-systems tend to require a great deal of software maintenance and to be intolerant of even the most minor of changes with respect to negative perturbations in system behavior.

In this thesis, we explore the benefits of developing a conceptual framework as the basis for the system-of-systems development. We examine the application of accepted software engineering practices for single-system developments to the more complex problem of system-of-systems development. Using the Ballistic Missile Defense System as a case study, we present an abstract framework from which we can reason about the system-of-systems. We develop a conceptual software architecture that represents a logical organization of proposed software modules. We map the functionality of the system to conceptual software components with coordination and data exchanges handled by conceptual connectors. Finally, we assess our work to determine the feasibility of applying the conceptual framework techniques described in this thesis to system-of-systems acquisitions with the objective of reducing accidental complexity and controlling essential complexity.

14. SUBJECT TERMS System-of-Systems, Software Complexity, Domain Analysis, Software Architecture	**15. NUMBER OF PAGES** 99
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UL

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

THIS PAGE INTENTIONALLY LEFT BLANK

CONCEPTUAL FRAMEWORK APPROACH FOR SYSTEM-OF-SYSTEMS SOFTWARE DEVELOPMENTS

Dale Scott Caffall
Civilian, Missile Defense Agency, Washington, D.C.
B.S., University of Arizona, 1986

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN SOFTWARE ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
March 2003

Author:	Dale Scott Caffall
Approved by:	James Bret Michael Co-Thesis Advisor
	Man-Tak Shing Co-Advisor
	Peter Denning Chairman, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

The Department of Defense looks increasingly towards an interoperable and integrated system-of-systems to provide required military capability. Non-essential software complexity of a system-of-systems can have a greater negative impact in system behavior than a single system. Our current systems-of-systems tend to require a great deal of software maintenance and to be intolerant of even the most minor of changes with respect to negative perturbations in system behavior.

In this thesis, we explore the benefits of developing a conceptual framework as the basis for the system-of-systems development. We examine the application of accepted software engineering practices for single-system developments to the more complex problem of system-of-systems development. Using the Ballistic Missile Defense System as a case study, we present an abstract framework from which we can reason about the system-of-systems. We develop a conceptual software architecture that represents a logical organization of proposed software modules. We map the functionality of the system to conceptual software components with coordination and data exchanges handled by conceptual connectors. Finally, we assess our work to determine the feasibility of applying the conceptual framework techniques described in this thesis to system-of-systems acquisitions with the objective of reducing accidental complexity and controlling essential complexity.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

I would like to thank Professor Bret Michael, Professor Man-Tak Shing, and Professor Richard Riehle for the outstanding support and superior instruction offered to me during my time of study at the Naval Postgraduate School. These folks worked tirelessly to enhance my understanding of software engineering. I would like to offer special thanks to Professor Michael who spent many hours of his time in shaping my understanding and charting my future studies. I would like to thank the Naval Postgraduate School for offering world-class instruction in software engineering. I am very appreciative of the opportunity that this institution afforded to me. I would like to thank my mother and father who provided me with the inspiration to seek higher-level education through personal example and sacrifice. Most of all, I would like to thank the greatest wife in the world who happens to be married to me. Without question, Trudy went “above and beyond” the call to ensure that the fire was stoked and the home front was secure. I love her dearly.

THIS PAGE INTENTIONALLY LEFT BLANK

I. SOFTWARE COMPLEXITY

Get to your places!' shouted the Queen in a voice of thunder, and people began running about in all directions, tumbling up against each other; however, they got settled down in a minute or two, and the game began. Alice thought she had never seen such a curious croquet-ground in her life; it was all ridges and furrows; the balls were live hedgehogs, the mallets live flamingoes, and the soldiers had to double themselves up and to stand on their hands and feet, to make the arches.

The chief difficulty Alice found at first was in managing her flamingo: she succeeded in getting its body tucked away, comfortably enough, under her arm, with its legs hanging down, but generally, just as she had got its neck nicely straightened out, and was going to give the hedgehog a blow with its head, it would twist itself round and look up in her face, with such a puzzled expression that she could not help bursting out laughing: and when she had got its head down, and was going to begin again, it was very provoking to find that the hedgehog had unrolled itself, and was in the act of crawling away: besides all this, there was generally a ridge or furrow in the way wherever she wanted to send the hedgehog to, and, as the doubled-up soldiers were always getting up and walking off to other parts of the ground, Alice soon came to the conclusion that it was a very difficult game indeed.

– Alice in Wonderland

Complexity is the great destroyer of software. Over the lifecycle of the software, the complexity will gradually increase through the inclusion of software patches and enhancements. Just as gradually increasing blood pressure can eventually lead to strokes and aneurisms, gradually increasing software complexity can lead to failed developments and software decay. While some degree of complexity is essential, the accidental and uncontrolled complexity is what we desire to purge from our system software.

Software complexity and size are increasing dramatically in our delivered products. Customers are demanding more features in their systems in less time than in any time in history. Under the demands of management, software developers scurry to coding with limited requirements elicitation and software architecture reasoning. As a result of this mad rush to the goal line, software developers stumble along - oftentimes fumbling the ball - and rarely score a touchdown. Sadly, software developers are building software products that have about a 26% chance of completing on time and on budget. For Government software projects, developers have about an 18% chance of completing their projects on time and on budget. Moreover, the delivered products will have fewer features and functions that originally desired by the customer [15]. Of great alarm to the Department of Defense, only 2% of the software was usable as delivered [11].

So, knowing what we know and armed with a great deal of advice from the wise, grizzled veterans of many software campaigns, why can we not develop better software products? Why is it that we know a great deal about the consequences of software complexity yet we continue to develop system software with a high degree of complexity?

A basic principle in all engineering disciplines is to “keep it simple.” The best that we can do in software engineering is to minimize “accidental complexity” and control “essential complexity.” Accidental complexity occurs due to a mismatch of paradigms, methodologies, and application tools [12]. Essential complexity is a fact of software engineering in that software engineers use complex data structures and algorithms to realize elaborate system features. Consequently, software engineers may not easily comprehend all of the possible behaviors of a software system as determined by the reachable system states. We discover time and again that software development is partly art and partly engineering.

Software complexity can occur in the extension of intended system use beyond the engineering-design space. As system software is used in ways never envisioned by the developers, operators are demanding extensions in their system software. Software will perform as it is designed to do; however, in the context of a system and operational environment in which it is operating, the system may perform the wrong action. Software may function as desired until a user attempts to apply it in an unexpected manner. Software may fail intermittently as sporadic environmental conditions come and go [9]. Additionally, software may function well for years until a particular operating condition changes and produces undesired behavior in the system software.

Two aspects of software complexity that engineers can control are the degree of cohesion and coupling in the software structure. Cohesion is the number of functions included within a single software component whereas coupling is the degree of intricacy of the relationships among software components [4]. A system of low cohesion (i.e., numerous functions in individual software components) and high coupling (i.e., highly intricate linkages among software components) is oftentimes hard to understand, hard to reuse, hard to maintain, and easily affected by change [16]. As such, system software will require a significant effort to implement enhancements and correct deficiencies that occur as a result of high complexity. Furthermore, increased software complexity decreases our ability to reuse software given that software engineers must work through the labyrinths of component linkages and bloated software components that spawn a multitude of functionalities in highly coupled and low cohesion system software.

The potential consequences of software complexity are significant in system software built for military use. Non-essential software complexity of a system-of-systems can have a greater negative impact in system behavior than a single system. A system-of-systems comes about from the assemblage of legacy systems for the purpose of providing a greater military capability than these systems operating autonomously could provide to the warfighters. In some system-of-systems, we integrate new developments into a given combination of legacy systems. (Note: For the purposes of this thesis, we will define a system-of-systems as the amalgamation of legacy systems and developing systems.)

The Department of Defense (DoD) looks increasingly towards an interoperable and integrated system-of-systems to provide required military capability. While the need for networked capability has exploded in military warfare, DoD has yet to develop a system-of-systems acquisition methodology for acquiring system-of-systems that will yield effective, interoperable, and robust systems, and provide enhanced military capability in a the full spectrum of the intended battlespace. Our current systems-of-systems tend to require a great deal of software maintenance and to be intolerant of even the most minor of changes with respect to negative perturbations in system behavior.

As examples of system-of-systems, consider the joint air defense system-of-systems environment and joint information transfer system-of-systems. DoD has invested many millions of dollars in providing effective, interoperable, and robust systems; however, these systems require significant software maintenance and numerous software patches to limp along in support of our warfighters. Many identified interoperability and integration deficiencies have plagued these systems for years; however, due to the considerable software complexity of these systems, our approach to solving these problems is oftentimes the development of a software patch to suppress undesired system behavior.

Undoubtedly, software applications are the most complex entities that humans can build. As the size and complexity increase, the system software design and organization become increasingly more significant than the selection of algorithms and data structures [13]. In the current DoD environment of rapidly paced acquisitions, senior management tends to place intense pressure on delivering desired features in seemingly unrealistic timeframes. In this type of acquisition environment, systems may be developed with little thought about the organization and behavior of the software. In the rush to deliver something quickly, system developers maintain a vision as far as the next line of code or the next aspect of detailed design [3]. This is more evident in the acquisition of a system-of-systems than in a single system acquisition.

While we cannot address all of the issues that negatively impact system software development in this thesis, we will examine the issue of requirements specification and software architectures for system-of-systems. Typically, detailed system specifications address merely the leaves of the system decomposition tree [17]. Software engineers cannot develop and deliver effective, interoperable, and robust military capability from just the detailed system specifications. To increase their understanding of the intended system software behavior, software engineers require layers of abstraction that begin at the top layer of abstraction that models overall system context and expand in definition and depiction with the detailed software specifications. It is at the upper layers of abstraction in which software engineers reason about the system and make architectural and design decisions.

In this thesis, we explore the benefits of developing a conceptual framework as the basis for the system-of-systems development. We examine the application of accepted software engineering practices for single-system developments to the more complex problem of system-of-systems development. Using the Ballistic Missile Defense System (BMDS) as a case study, we present an abstract framework from which we can reason about the system-of-systems. We develop a conceptual software architecture that represents a logical organization of proposed software modules. We map the functionality of the system to architectural elements called conceptual software components with coordination and data exchanges handled by conceptual components called connectors. Finally, we assess our work to determine the feasibility of applying the conceptual framework techniques described in this thesis to system-of-systems acquisitions with the objective of reducing accidental complexity and controlling essential complexity.

THIS PAGE INTENTIONALLY LEFT BLANK

II. SYSTEM-OF-SYSTEMS

Dorothy: "Now which way do we go?"

Scarecrow: “Pardon me. That way is a very nice way.” (pointing one direction)

Dorothy looks around quizzically: "Who said that?"

Toto barks at a stuffed Scarecrow.

Dorothy: "Don't be silly, Toto. Scarecrows don't talk!"

Scarecrow: "It's pleasant down that way too!...(pointing in another direction)."

Dorothy: "That's funny. Wasn't he pointing the other way?"

Scarecrow: "Of course, people do go both ways.” (pointing in two opposite directions)

Scarecrow: “That's the trouble. I can't make up my mind. I haven't got a brain. Only straw.”

Dorothy: “How can you talk if you haven't got a brain?

Scarecrow: “I don't know. Some people without brains do an awful lot of talking, don't they?”

- The Wizard Of Oz

Unlike other development and construction efforts, software developers oftentimes are seemingly quite content without a roadmap that depicts how software components are organized in a system, how these components fit together, how these components interact, and how these components fulfill the system requirements. Such a roadmap for software development is a software architecture that defines the system in terms of computational components and the interactions among those components. Additionally, the software architecture bridges the gap between system requirements and realization, thereby documenting the rationale for design decisions [13]. Although users will quickly know whether something is wrong when they select a menu item and the system crashes, software engineers frequently cannot identify if the problem is in the "foundation" (e.g., operating system), in the "plumbing" (e.g., network or middleware), or in an appliance (e.g., word processing application).

While a software architecture will not guarantee that a system meets its requirements, a poorly designed or ill-defined software architecture makes it nearly impossible for the software developers to realize a system that meets its requirements [8]. Typically, the system architecture is little more than a "sticks-and-circles" diagram in which the circles represent the various systems in the system-of-systems and the sticks represent the communication links among the systems. More often than not, this type of architectural view represents the totality of a system architecture effort in DoD organizations. Unfortunately for the developers who require information models that faithfully represent the operational battlespace, the circles of the sticks-and-circles diagram do not define the behavior of the systems and the sticks reveal little of the connectors that these lines represent. The information model formed by the sticks-and-circles diagram is a weak information model.

Much too often, we initiate coding from a reasoning about the "sticks-and-circles" diagrams. During the development, we add new layers of features and functional enhancements to the system software without clear insight into the organization of the system software. Inevitably, the basic software organization that seemed so reasonable at the beginning begins to break apart under the weight of the system software revisions [3]. Regrettably, the software development becomes another casualty to report in future studies as to why software developments are not successful.

Oftentimes, we compare engineering disciplines to seek points of commonality. Designing and constructing a new building is a common metaphor from which we attempt to draw lessons learned from other engineering disciplines. In designing a skyscraper, a journeyman architect will elicit requirements from the client and translate those requirements into various views of the proposed structure. A civil engineer will develop structural drawings and construction plans to build the new structure. Without some type of framework from the architect, the civil engineer could not construct a skyscraper that would support its own weight as well as the weight of the occupants, and their associated furniture and work materials.

The accepted framework for skyscraper design and construction is the set of blueprints for the proposed building. This set of blueprints has many views to include the physical construction of the building, more detailed construction views of each floor, views of the heating and cooling systems, views of the plumbing, and so forth. In essence, the skyscraper's architect develops this set of blueprints to provide a conceptual framework of the proposed building that all stakeholders can read and understand.

Imagine constructing such a building without a framework. Suppose that our civil engineer was somehow able to construct the skyscraper to the height of three floors whereupon which he loses confidence in constructing additional floors. Consider that the civil engineer now wants to add electrical wiring, communications cables, heating ducts, water and waste drainage plumbing, gas lines, etc. Now, the various craftsmen must drill holes; run wiring, pipes, and ducts; and connect appliances to this skyscraper without the benefit of any visual framework. We can only imagine the hodgepodge of wires, cables, tubing, pipes, ducts, and appliances that would result in this construction.

It is important to emphasize that the set of blueprints is a collection of many views of the new building. Each view is important to the overall construction of the building. Without the full set of views, some portion of the new building will suffer in the construction phase.

Consider the classic novel of the Civil War: The Killer Angels by Michael Shaara. This novel presents the story of the Battle of Gettysburg as told from the viewpoints of the battle's participants from the Union and Confederate armies: General Robert E. Lee, General James Longstreet, Colonel Joshua Chamberlain, Colonel John Buford, General Lewis Armistead, and English Colonel Arthur Freemantle.

Shaara had many options for how he might develop this novel. He could have written a chronological listing of events for each man and allotted a chapter in the book for each man's chronological listing of events. In this case, the integration would be left to the reader. He could have divided the book into two Union and Confederate chapters and provided a chronological listing of events from the perspectives of first the Union and then the Confederacy. Again, the integration of the writings would be left to the reader. In these two approaches, would the author have provided a clearer visualization of the Battle of Gettysburg than his published work?

Instead, Shaara chose to frame the perspectives of these men around the events of the Battle of Gettysburg. That is, the common framework was the battle. He added snippets of perspectives from the participants at the appropriate points in the battle. This approach provided the reader with a common framework as well as many views of the battle from the perspectives of the participants. The Killer Angels provides the reader with insight into the battle by presenting it through the different views of the participants. This approach enhances the reader's understanding of the Battle of Gettysburg beyond a pure chronological listing of events and facts.

We could work through similar examples of other activities. In chemistry, we use the Periodic Table and balanced equations to develop different views of chemical reactions. These views help the chemist visualize the products and side effects of combining various compounds and elements to form the new products. In the design of a new automobile, we develop a set of views to visualize the required integration efforts of many physical systems to the fuel flow mechanisms, environmental sensors, microprocessors, and wiring. In a criminal trial, the district attorney weaves together the physical evidence, motive, time of events, and defendant accessibility to the crime scene to present a visual picture to the jury. Can you imagine the confused and complex picture that the prosecuting attorney would present if he/she could not link the physical evidence to the motive, time of events, and plaintiff accessibility? If the prosecuting attorney cannot successfully present an integrated storyline, then the jury cannot visualize a plausible picture of the crime.

The points of commonality in the above anecdotes are the need for a conceptual framework and the need for multiple views of the problem that we are attempting to solve. We can see that the conceptual framework is essential for reasoning about the problem. In the absence of such a framework, we can easily predict the degree of success in the above endeavors. We can see that the multiple views add richness to our conceptual view of the problem and potential solutions.

While keeping in mind the idea that multiple views of a system can increase our understanding of system behavior, we will apply this concept to the system-of-systems problem. Consider the following hypothetical missile defense system-of-systems that we use to illustrate both the inadequacies of the “sticks-and-circles” system architecture view and the value of the software architecture views. Let us define a proposed missile defense system-of-systems as four sensors of differing type, four battle management systems with organic sensors of differing type, and four weapons launchers of differing type as depicted in the below diagram of Figure 1. The challenge to our system engineer is to integrate these twelve systems into a single system-of-systems.

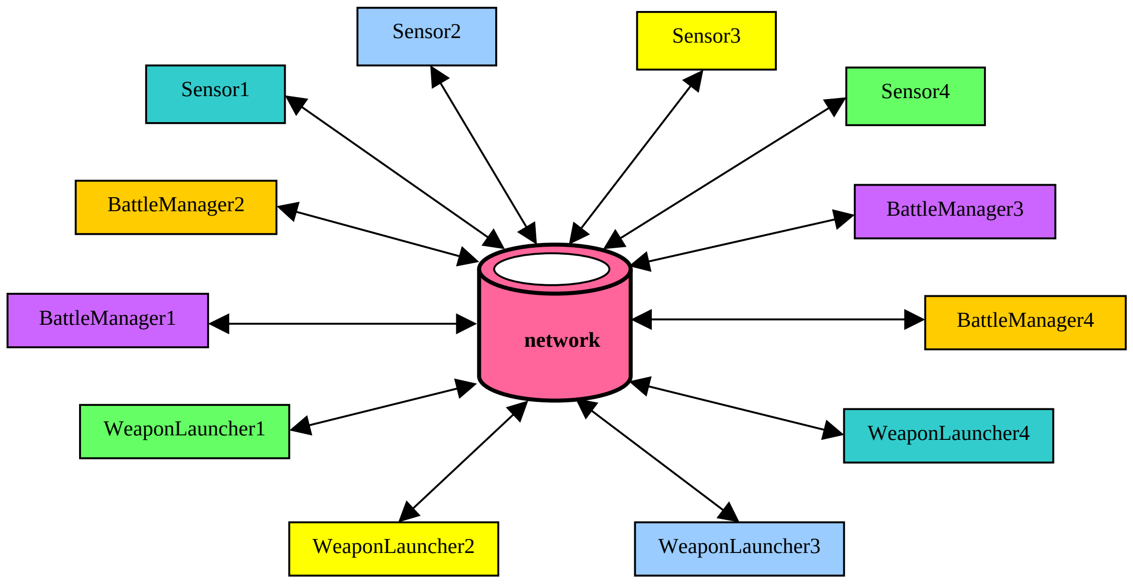


Figure 1. Hypothetical Missile Defense System-of-Systems

Let us outline the strikes against this system engineer as he/she steps into the integration batter's box. At the onset of the system development, the program manager for each system probably did not have a requirement for inclusion of that system into a system-of-systems environment. Thus, these twelve systems were developed in isolation from the other systems and each system design was developed in a different manner than the others. Most importantly to interoperability and integration, the realized software organization differs among the systems.

The traditional solution is to apply a communications solution for interoperability and integration. That is, the "stick" will be a means of information transfer, a messaging protocol, and, perhaps, a translator box to translate the messaging format from one system to another. Traditionally, this methodology has failed to achieve an interoperable and integrated system-of-systems. With each new failure, the system engineers attempt to "tighten up" the protocol standard; however, the system-of-systems did not achieve the desired degree of interoperability and integration. The end-state was a collection of systems that are tightly coupled with a realized protocol standard that only served to increase the system-of-systems software complexity.

System software critical interactions increase as the complexity of highly integrated systems increases. In the complex system-of-systems, these possible combinations are practically limitless. System "unravelings" have an intelligence of their own as they expose hidden connections, neutralize redundancies, bypass firewalls, and exploit chance circumstances for which no system engineer might plan [7]. A software fault in one module of the system software may coincide with the software fault of an entirely different module of the system software. This unforeseeable combination can cause cascading failures within the system-of-systems.

For the development of a system-of-systems in some ideal acquisition world, we might elicit system requirements from the users and develop various software architecture artifacts that define the system behavior. We might design conceptual software components that reflect the required functionality of the system, identify the interfaces between the conceptual components, identify software modules for these components, design hierarchical layers that group similar modules, and complete the software design based on the identified modules and layers.

Because architectural decisions are usually made early in the lifecycle, these decisions are the hardest to change, and hence the most critical and far-reaching. Without a software architecture that faithfully models desired system behavior, it is difficult to achieve the satisfaction of the original performance and behavioral requirements, and it is probably impossible to accommodate major design changes. Software architectures serve as a planning tool for allocating system requirements as well as promote the construction of subsystems from architectural components – not the other way around. Furthermore, problems with the requirements and the architecture will ensconce requirements and design problems via refinement into lower-level system artifacts such as detailed software designs, code, and documentation. It is imperative that we make good decisions early in the lifecycle, and uncover problems in the requirements and architecture as the architecture and engineering artifacts are developed.

We do not always have the luxury of beginning a system-of-systems development from scratch. We must work with the evolving systems that are in some phase of development and legacy systems that are in operational use. We cannot begin anew so we must find other methods to apply software architecture and software engineering techniques to the system-of-systems acquisition. We must develop a framework from which we can reason about a system-of-systems so that we can conduct trade studies on system cost, schedule, and performance, and make decisions from an aggregate system view rather than individual systems. Given our revised system-of-systems framework, we can develop an implementation plan for refactoring the existing system software organization.

THIS PAGE INTENTIONALLY LEFT BLANK

III. BALLISTIC MISSILE DEFENSE SYSTEM

Col Jessep (Jack Nicholson): *You want answers?*

Kaffee (Tom Cruise): *I think I'm entitled to them.*

Jessep: *You want answers?*

Kaffee: *I want the truth!*

Jessep: *You can't handle the truth! Son, we live in a world that has walls. And those walls have to be guarded by men with guns. Who's gonna do it? You? You, Lt. Weinberg? I have a greater responsibility than you can possibly fathom. You weep for Santiago and you curse the Marines. You have that luxury. You have the luxury of not knowing what I know: that Santiago's death, while tragic, probably saved lives. And my existence, while grotesque and incomprehensible to you, saves lives...You don't want the truth. Because deep down, in places you don't talk about at parties, you want me on that wall. You need me on that wall. We use words like honor, code, loyalty...we use these words as the backbone to a life spent defending something. You use 'em as a punchline. I have neither the time nor the inclination to explain myself to a man who rises and sleeps under the blanket of the very freedom I provide, then questions the manner in which I provide it! I'd rather you just said thank you and went on your way. Otherwise, I suggest you pick up a weapon and stand a post. Either way, I don't give a damn what you think you're entitled to!*

- *A Few Good Men*

The Department of Defense (DoD) plans to develop a layered ballistic missile defense to defend the forces and territories of the United States, its Allies, and friends against all classes of ballistic missile threats. The Missile Defense Agency (MDA) will accomplish this mission by developing a layered defense that employs complementary sensors and weapons to engage threat targets in the boost, midcourse, and terminal phases of flight, and incrementally deploying that capability. The Ballistic Missile Defense (BMD) program will pursue a broad range of activities in order to develop and evaluate technologies for the integration of land, sea, air, and space-based platforms to counter ballistic missiles in all phases of their flight. In parallel, sensor suites and battle management and command and control will be developed to form the backbone of the Ballistic Missile Defense System (BMDS).

The objective of the BMDS is to employ a layered defense that provides multiple engagement opportunities along the entire flight path of a ballistic missile. The BMDS will provide a ballistic missile defense to the forces and territories of the United States, its Allies, and friends against all classes of ballistic missile threats.

While the end of the Cold War has signaled a reduction in the likelihood of global conflict, the threat from foreign missiles has grown steadily as sophisticated missile technology becomes available on a wider scale. The proliferation of weapons of mass destruction and the ballistic and cruise missiles that could deliver them pose a direct and immediate threat to the security of U.S. military forces and assets in overseas theaters of operation, our allies and friends, as well as our own country.   Since 1980, ballistic missiles have been used in six regional conflicts.

All ballistic missiles share a common, fundamental element - they follow a ballistic trajectory that includes three phases (reference Figure 2 on page 20). These phases are the boost phase, the midcourse phase, and the terminal phase. The boost phase is the portion of a missile's flight in which it is thrusting to gain the acceleration needed to reach its target. This phase usually last between 3-5 minutes based. During this phase the rocket is climbing against the earth's gravity and either exiting the earth's atmosphere, or in the case of shorter-range missiles, only reaching the fringes of outer space. Once the missile has completed firing its propulsion system, it begins the longest part of its flight, which is known as the mid-course phase. During this phase the missile is coasting, or freefalling towards it target. This phase can last as long as 20 minutes in the case of intercontinental ballistic missiles (ICBMs). Most missiles that leave the atmosphere shed their rocket motors by this time in order to increase the range that the missile's weapon, known as a warhead, can travel. For medium and long-range missiles this phase occurs outside the earth's atmosphere. The final phase of a missile's flight is the terminal phase. During this phase the missile's warhead reenters the earth's atmosphere at incredible speeds, some at over 2,000 mph. This phase last approximately 30 seconds for ICBM class missiles.

There are advantages and challenges to set up engagement opportunities against a threat missile in each of these phases. The capability to defend against an attacking missile in each of these phases is called a layered defense, and it may be expected to increase the chances that the missile and its payload will be destroyed. By attacking the missile in all phases of flight, we exploit opportunities that could increase the advantage of the defense. A capability to intercept a missile in the boost phase, for example, can destroy a missile regardless of its range or intended aim-point and provide a global coverage capability. A midcourse intercept capability can provide wide coverage of a region or regions, while a terminal defense reduces the protection coverage considerably to a localized area. When we then add shot opportunities in the midcourse and terminal phases of flight to boost phase opportunities, we increase significantly the probability that we will be successful. Improving the odds of interception becomes critical when ballistic missiles carry weapons of mass destruction. When possible, for the global coverage and protection against more lethal payloads it can provide, a capability to intercept a missile near its launch point is always preferable to attempting to intercept that same missile closer to its target.

DoD will develop technologies, and deploy systems promising an effective, reliable, and affordable missile defense system. The BMD program is designed to develop effective systems over time by developing layered defenses that employ complementary sensors and weapons to engage threat targets in the boost, midcourse, and terminal phases of flight, and to deploy that capability incrementally.

Mobility in our sensor and interceptor platforms and the capability to do boost phase and/or midcourse phase intercept must be central features in our architecture if we are to provide effective territorial protection at home and abroad. Placing sensors forward, or closer to the target missile launch point, either on land, at sea, in the air, or in space, will expand the battle space, improve discrimination of the target complex, and increase engagement opportunities. We will develop complementary elements in different combinations in order to afford the system a high degree of synergism and effectiveness.

The BMDS will feature a uniform battle management and command and control network and leverage, where possible, other Department communication channels to integrate elements of the BMDS. Because the system must act within minutes or even seconds to counter ballistic missiles, the information we receive on threats must be accurately received, interpreted, and acted upon rapidly. The information network must be seamless and allow information to be passed quickly and reliably among all the elements of the system.

At the direction of the Secretary of Defense, we have developed a research, development and test program that focuses on missile defense as a single integrated system that no longer differentiates between theater and national missile defense.

Over the next three to five years we will pursue parallel technical paths to reduce schedule and cost risk in the individual efforts. We will explore and demonstrate kinetic and directed energy kill mechanisms for potential sea-, ground-, air-, and space-based operations to engage threat missiles in the boost, midcourse, and terminal phases of flight. In parallel, sensor suites and battle management and command and control (BMC2) will be developed to form the backbone of the BMDS.

Unlike the conventional build-to-requirements acquisition process of the past, we adopted a capability-based approach that recognizes that changes will occur along two separate axes. On the one axis, the threat will evolve and change over time based on the emergence of new technologies, continued proliferation of missiles worldwide, and operational and technical adjustments by adversaries (including the introduction of countermeasures) to defeat our BMDS. On the other axis lie changes we will experience. These include improving technologies, incremental system enhancements, evolving views of system affordability, and out-year decisions expanding coverage, potentially including the territory and populations of our Allies and friends.

Specific system choices and timelines will take shape over the next few years through our capability-based, block approach. We will increase our capability over time through an evolutionary process as our technologies mature and are proven through testing. The block approach allows us to put capable technologies "in play" sooner than would otherwise be possible. We have organized the program with the aim of developing militarily useful capabilities in biannual blocks, starting as early as the 2004-2006 timeframe. These block capabilities could be deployed on an interim basis to meet an emergent threat, as an upgrade to an already deployed system, or to discourage a potential adversary from improving its ballistic missile capabilities.

Figure 2. Ballistic Missile Kill Chain

IV. DOMAIN ANALYSIS

The hostile response didn't seem to faze Smoking Man, who calmly took another puff from his cigarette. He blew out the smoke in Skinner's direction and responded, "I have no idea what you are referring to."

"Isn't that just typical. Is that what they teach you up there?" Skinner said, sounding more agitated. "There was a traffic accident last night. Mulder and Scully were involved."

That revelation seemed to cause a slight reaction in Smoking Man, though it was barely discernible from the typical bland expression on his face. "What happened?" Smoking Man asked.

Skinner couldn't stand the way Smoking Man toyed with him. Did he really think that Walter Skinner, Assistant Director of the FBI for chrissake, had a chain long enough to yank? Sure, play dumb. But I know what you and your friends are up to. I beat you once, and I'll beat you again.

"I'll tell you... not because you don't already know... but because I want you to know that I know. Apparently, Mulder and Scully, on their way back from an investigation, crashed into a tanker loaded with industrial solvent on Kings Highway just outside of D.C. It took them nearly four hours to get the flames out. Regrettably, it looks like neither agent survived the crash."

"That is unfortunate."

"Yes, that is unfortunate. I told you what would happen if anyone involved suffered an accident..." Skinner said, referring to the warning he had used to guarantee the safety off all those involved in the incident with the classified data tape.

Suddenly, Smoking Man became quite animated, leaning forward in the chair. "Now wait a minute here. I know what the deal was. To my knowledge, nothing has been done to either Agent Mulder or Scully. What happened last night was a freak occurrence. They just ran out of luck, that's all," he said, gesturing forcefully at the FBI man.

"I have a team going over the forensics of the crash at this moment. If they find any evidence that they 'ran out of luck' because of your doing, there'll be hell to pay. I promise you."

"Are you finished?" Smoking Man asked defiantly.

Skinner didn't dignify the question with an answer. He merely leaned back and stared at his adversary. Seeing that the meeting was over, Smoking Man got out of the chair and walked out of Skinner's office, silently. He would have to meet with the committee this afternoon and figure out what exactly was going on.

- X-Files

A good system model is the basis for sound system development decisions. Conversely, we cannot expect to make sound system development decisions armed with a poor or nonexistent system model. Using various artifacts from the Unified Modeling Language (UML), we will develop our system-of-systems model because humans tend to grasp and understand graphical representations easier than written descriptions. This phenomenon becomes truer as the complexity of a system increases.

We propose that applying the Unified Modeling Language (UML) and object oriented design (OOD) techniques to the system-of-systems requirements analysis offers a new model for reasoning about complex system-of-systems developments. Rather than disparate reasoning about the individual systems of a proposed system-of-systems, we propose that we develop a sound model for reasoning about the system-of-systems as a single, functional entity.

The greatest source of system software faults will occur in the integration of the various systems. With respect to our case study, the hypothetical missile defense systems will be a complex product that will contain many discrete software packages within each system. As a rule, these software packages will be developed independent of each other and programmed in many different languages. Additionally, the hypothetical missile defense system will include legacy systems that are currently in operation. The means of integrating these elements and legacy systems are intricate tactical data links that support the message transfer within the system-of-systems.

The object-oriented paradigm offers a new system-of-systems requirements and design methodology that can minimize accidental complexity and control essential complexity through the object-oriented concepts of decentralized control flow, minimal messaging between classes, implicit case analysis, and information-hiding mechanisms. While the hypothetical missile defense system will not be a pure object-oriented design, we can incorporate many of the principles of object-oriented technology to decrease the complexity of the system-of-systems. We believe that software engineers of system-of-systems can use this object-oriented paradigm to produce a sound design for the system-of-systems rather than the traditional federation of systems through a highly coupled communication medium.

In our approach, we will begin the domain analysis at the top level of abstraction and work downward to the details. We will develop use cases to help us understand the goals and functionality of the missile defense system. We will develop other artifacts to capture system behavior from various perspectives. During the BMDS domain analysis, we will identify issues that surface for future architectural and design considerations.

As a means of dividing the problem into manageable pieces, we will view the missile defense problem via the functional requirements. We will use the ballistic missile kill chain to describe what functions that the BMDS must perform. The kill chain functions will be as follows: Detect, Track, Assign Weapon, Engage, and Assess Kill. In the Detect function, the BMDS will use the received data from various sensors, and either develop a new track file or update existing track files. In the Track function, the BMDS will apply feature recognition applications to identify and type-classify each ballistic missile track. In the Assign Weapon function, the BMDS will assign a weapon to each ballistic missile track based upon the BMDS' evaluation of the ballistic missile against its estimated impact point, defended asset list, weapon availability, and interceptor inventory. In the Engage function, the BMDS will develop the firing solution and authorize the assigned weapon to engage the ballistic missile. In the Assess Kill function, the BMDS will use the received data from various sensors to determine whether the interceptor negated the ballistic missile. If true, then the BMDS will cease monitoring that ballistic missile track. If not true, then the BMDS will repeat the five kill chain functions for that ballistic missile threat.

BMDS Use Cases. We developed the BMDS Use Cases based upon the five functional goals of the Kill Chain. Rather than identify each possible ballistic missile threat, sensor and weapon (e.g. SCUD-B, Shahab-4, CSS-8, DSP satellite, X-band radar, PATRIOT, Navy Theater Wide, etc.), we will use the superclasses of these missile defense elements: Threat Missile, Sensor, BMC2, Weapon, and Interceptor. The five use cases are presented in the following order: Detect, Track, Assign Weapon, Engage, and Assess Kill. We will employ this level of abstraction throughout the domain analysis.

Use Case: Detect Threat Ballistic Missile

Context of Use: The goal of this use case is to detect threat ballistic missiles, and either update an existing track file or create a new track file.

Level: User goal.

Primary Actors: Threat ballistic missile, Sensor, BMC2

Stakeholders and Interests: Area Air Defense Commander

Preconditions: Sensor is in search mode.

Success Guarantee: BMDS detects threat ballistic missile.

Trigger: Adversary launches threat ballistic missile.

Main Success Scenario:

1. Sensor records initial "hit" of a missile launch or a flying object.
2. BMC2 detect feature receives initial track data from sensor.
3. BMC2creates new track file and initiates <u>track threat ballistic missile</u>.

Extensions:

*a.At any time inorganic sensors fail to detect threat ballistic missile: Ensure weapon has permissions for "weapons free" engagement upon determination of threat ballistic missile entering minimum engagement zone above area of assigned defended assets.

3a.Track file exists for track data.

BMC2 updates existing track file.

Technical and Data Variations List: None

Use Case: Track Threat Ballistic Missile

Context of Use: The goal of this use case is to identify and type-classify the threat ballistic missile, and develop a fire-quality track for an engagement solution.

Primary Actors: Threat ballistic missile, Sensor, BMC2

Stakeholders and Interests: Area Air Defense Commander:

Preconditions: Sensor is tracking threat ballistic missile.

Success Guarantee: BMC2 develops fire-quality track in terms of position, velocity, covariance, sigma; missile type, predicted impact point (IPP), launch point estimate (LPE), and re-entry vehicle (RV) type.

Triggers:
1. Detect feature creates track file for BMC2 tracking feature OR
2. BMC2 determines previously engaged threat ballistic missile is not negated.

Main Success Scenario:
1. Sensor provides amplifying track data to BMC2 tracking feature.
2. BMC2 discriminates track from counter-measures and debris, identifies the track as a threat ballistic missile, and type-classifies the track.
3. BMC2 provides track information to assign weapon feature.

Extensions:

*a.At any time inorganic sensors fail to detect threat ballistic missile: Ensure weapon has permissions for "weapons free" engagement upon determination of threat ballistic missile entering minimum engagement zone above area of assigned defended assets.

Technical and Data Variations List:
1. BMC2 will have electronic access to established ROEs.
2. BMC2 will have electronic access to defend assets list (DAL).
3. BMC2 will have electronic access to intelligence profiles of threat ballistic missiles.

Use Case: Assign Weapon

Context of Use: The goal of this use case is to assign a weapon to negate the threat ballistic missile.

Primary Actors: Threat ballistic missile, Sensor, Weapon, BMC2

Stakeholders and Interests: Area Air Defense Commander

Preconditions:

Sensor continues to provide track data.
BMC2 tracking feature develops fire-quality track information.

Success Guarantee: BMDS tasks weapon in sufficient time for interceptor to negate threat ballistic missile at safe stand-off altitude and distance from defended assets.

Trigger: Tracking feature provides fire-quality track information to BMC2 assign weapon feature.

Main Success Scenario:

1. BMC2 compares threat ballistic missile IPP to defended asset list, establishes target priority, and determines target engagement sequence.
2. BMC2 evaluates target engagement sequence against availability information: launcher availability, missile inventory, and defended asset list.
3. BMC2 assigns a weapon to the target and initiates the engage feature.

Extensions:

*a.At any time inorganic sensors fail to detect threat ballistic missile: Ensure weapon has permissions for “weapons free” engagement upon determination of threat ballistic missile entering minimum engagement zone above area of assigned defended assets.

Technical and Data Variations List:

BMC2 will have electronic access to established ROEs.
BMC2 will have electronic access to defended asset list.

Use Case: Engage

Context of Use: The goal of this use case is to engage threat ballistic missile.

Primary Actors: Threat ballistic missile, Sensor, Weapon, Interceptor, BMC2

Stakeholders and Interests: Area Air Defense Commander

Preconditions:

Sensor continues to provide track data.
Assign weapon feature has assigned weapon to target.

Success Guarantee: Interceptor negates threat ballistic missile.

Trigger: BMDS assigns weapon to target.

Main Success Scenario:

1. BMC2 computes intercept point, time to launch, and last-time to launch.
2. BMC2 validates weapon launcher readiness and issues command to fire.
3. Weapon activates interceptor.
4. Interceptor engages threat ballistic missile.

Extensions:

*a. At any time inorganic sensors fail to detect threat ballistic missile: Ensure weapon has permissions for “weapons free” engagement upon determination of threat ballistic missile entering minimum engagement zone above area of assigned defended assets.

Technical and Data Variations List:

1. BMC2 will have electronic access to established ROEs.
2. BMC2 will have electronic access to defend assets list (DAL).

Use Case: Assess Kill

Context of Use: The goal of this use case is to determine the kill status of the threat ballistic missile.

Primary Actors: Threat ballistic missile, Sensor, BMC2

Stakeholders and Interests: Area Air Defense Commander:

Preconditions: Sensor is in search mode.

Success Guarantee: BMDS determines threat ballistic missile is negated and reports kill.

Trigger: Interceptor engages threat ballistic missile.

Main Success Scenario:

1. Sensor provides tracking data to BMC2.
2. BMC2 applies feature recognition process, discriminates objects in debris cloud, and compares tracked objects to intelligence profiles.
3. BMC2 determines that threat ballistic missile is negated and issues kill assessment report.

Extensions:

*a. At any time inorganic sensors fail to detect threat ballistic missile: Ensure weapon has permissions for "weapons free" engagement upon determination of threat ballistic missile entering minimum engagement zone above area of assigned defended assets.

2a. BMC2 cannot discriminate objects.

2a1. Organic weapon sensor searches debris cloud and discriminates objects.

3a. BMC2 cannot determine that threat ballistic missile is negated.

3a1. BMC2 continues to carry track as active threat.

3a2. Organic weapon sensor searches debris cloud and discriminates objects.

3b. BMC2 determines that threat ballistic missile is not negated.

3b1. BMDS repeats cycle: Detect, Track, Assign Weapon, Engage, Assess Kill.

Technical and Data Variations List:

1. BMC2 will have electronic access to established ROEs.
2. BMC2 will have electronic access to defend assets list (DAL).
3. BMC2 will have electronic access to intelligence profiles of threat ballistic missiles.

BMDS Class Diagram. Let us propose a model of the BMDS from the information in the BMDS Use Cases. We will develop a class diagram with abstract classes for the major components of the system-of-systems. We will reason about the class diagram in our attempt to develop subclasses to which we can begin to allocate requirements and analyze system capabilities and limitations. Additionally, we will identify message requirements and message flow in our attempt to reduce coupling in the system-of-systems by developing requirements for simplified interfaces between the components. We will propose a reassignment of methods to increase the cohesion of the components [2]. We will use the following five classes:

- **Threat Missile:** The threat missile class is the enemy missile that contains warhead of mass destruction: nuclear, chemical, or high explosive munitions. The adversary will launch the threat missile within the confines of his state. The missile will climb into the exo-atmospheric region that constitutes up to 80% of the missile flight. The missile will re-enter the atmosphere over our forces or defended assets at which time it will impact at its aim point.

- **Sensor:** The sensor class is the object that detects the threat missile. Sensor is an abstraction of two subclasses: infrared class and radar class.

- **BM/C2:** The Battle Manager/Command and Control (BM/C2) class processes track data from the sensor. The BM/C2 monitors the threat missile, develops firing solution to negate the threat missile, and directs a weapon to launch its interceptor with the BM/C2-provided firing solution. The BM/C2 class is an abstraction for all system echelons of battle management.

- **Weapon:** The weapon class develops firing solutions, calculates the probability of kill, and implements the BM/C2 authorization to engage the threat missile.

- **Interceptor:** The interceptor class is the engagement mechanism that negates the threat missile. The interceptor class is the abstraction for both directed and kinetic energy intercepts of the threat missile.

Given these classes and the BMDS Use Cases, we can construct a class diagram of the BMDS on the following pages.

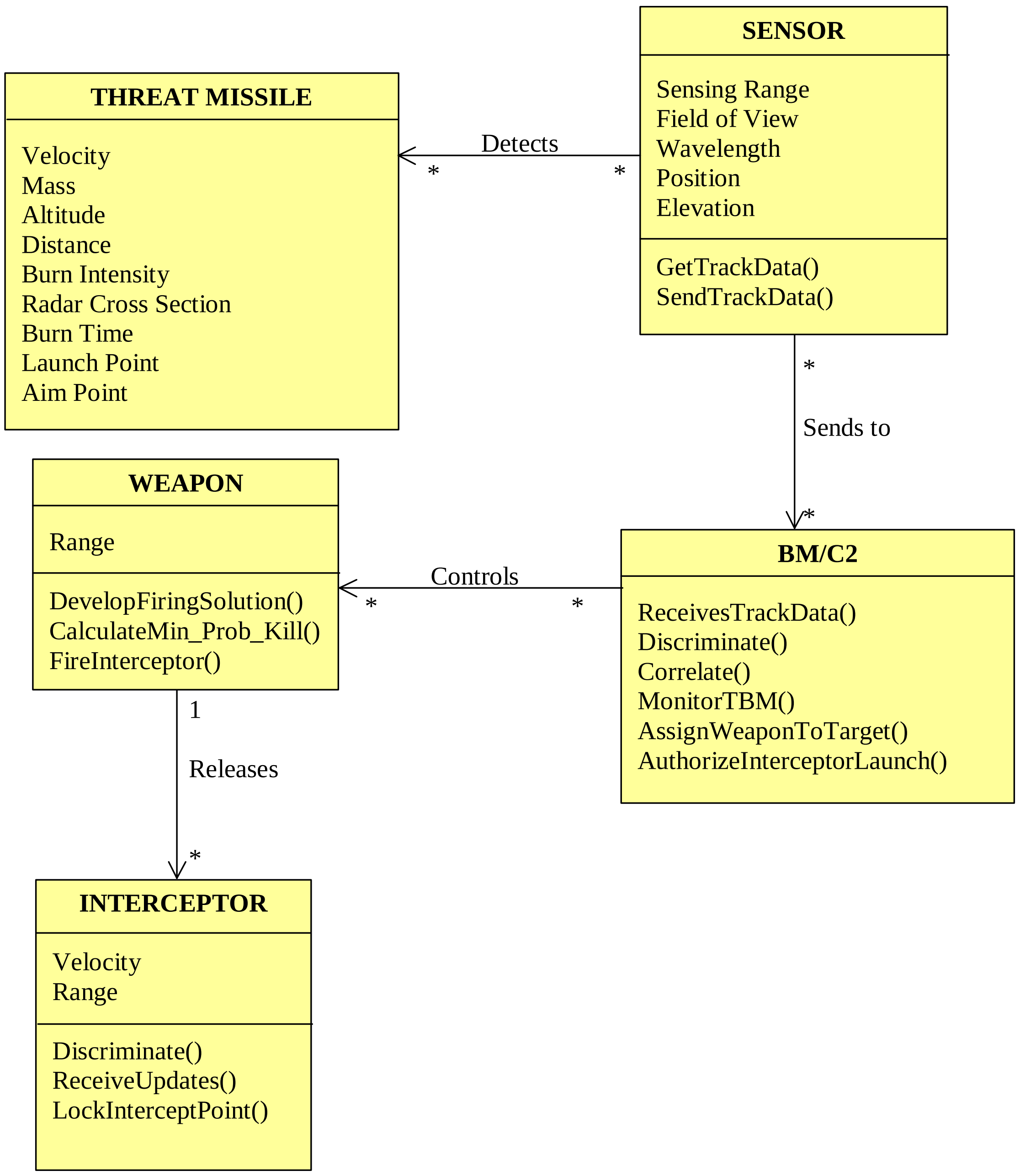


Figure 3. Class Diagram of Hypothetical Missile Defense System-of-Systems

Note that the message requirements in the above class diagram are very specific as compared to the single, large network interface of the sticks and circles diagram. Through this class diagram, we can easily determine the messaging requirements of each class. For example, the sensor class wants to determine the attributes of the threat missile class. The BM/C2 class wants formed track data from the sensor class. The weapon class waits for control data from the BM/C2 class. The interceptor class waits for the interceptor release command from the weapon class.

From this class diagram, we can begin to define abstract interfaces between the classes. Rather than the largely unmanageable and complex network interface of the sticks and circles diagram, we can begin to develop very specific interface requirements from the class diagram approach.

Let us add detail to the threat missile class as this is the point of reference for our hypothetical missile defense system. We can develop subclasses (i.e. short range, intermediate range, and long range threat missiles) of the threat missile class as depicted below in Figure 4.

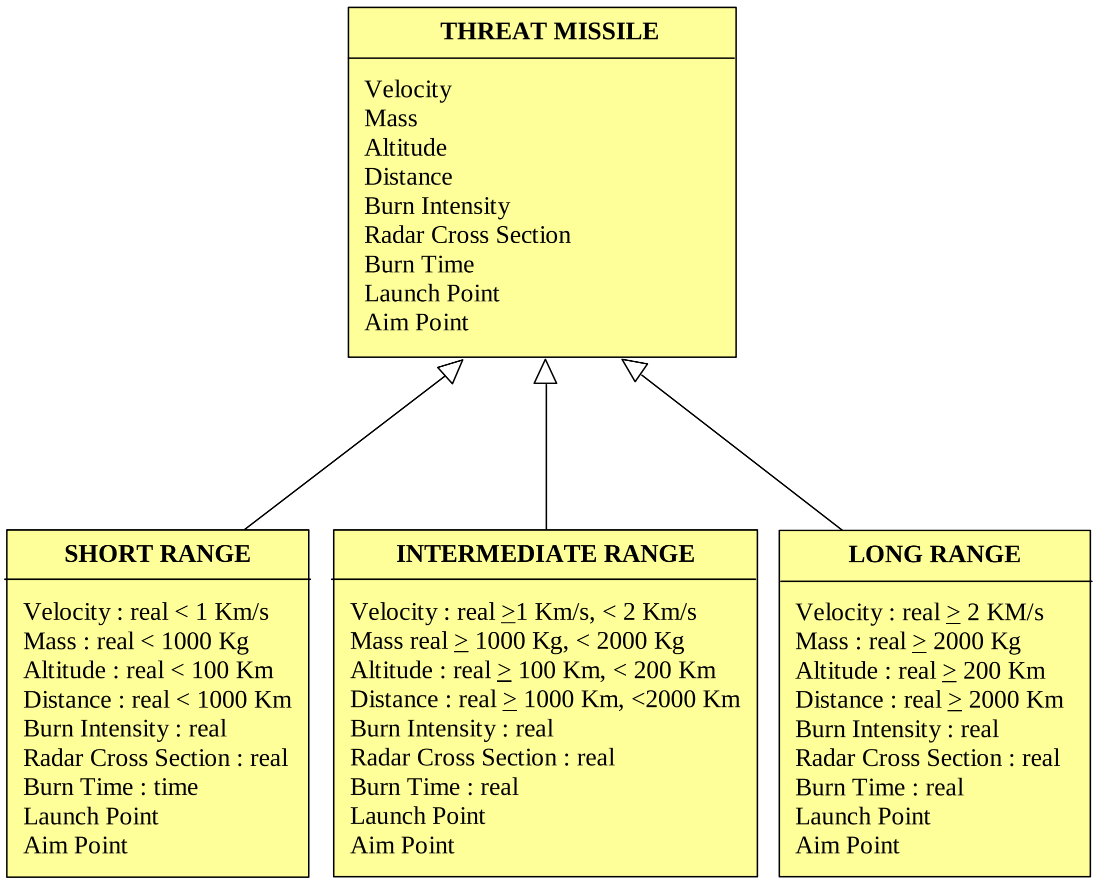


Figure 4. Subclasses of Threat Missile Class*

*Note: All attribute values listed in subclasses are fictitious and do not represent real threat missile data.

In our definition of the subclasses, we have assigned attribute values that represent fictitious data so that our example remains out of the classified regime. These subclasses with the assigned attributes will form the basis for our reasoning about the hypothetical missile defense system.

The sensor class is responsible for detecting the threat missile class so let us develop subclasses that can detect the threat missile subclasses that we have defined. The subclasses for the sensor class are depicted below in Figure 5.

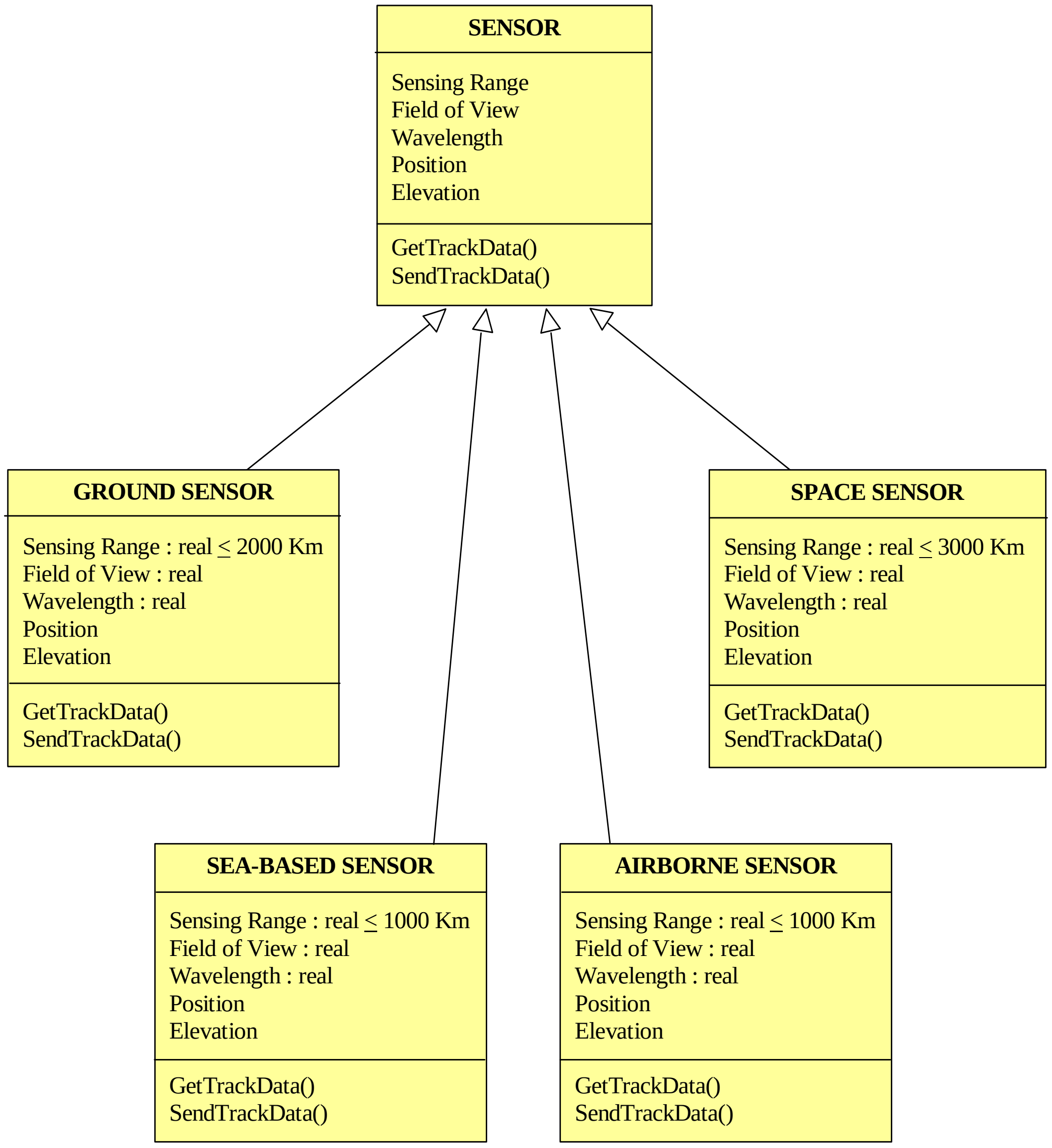


Figure 5. Subclasses of Sensor Class*

*Note: All attribute values listed in subclasses are fictitious and do not represent real sensor data.

By considering the subclasses of the threat missile class, we can design a sensor framework for which we can attain overlapping coverage of our sensor subclasses to greatly increase our opportunities for the detection of the threat missiles. Additionally, we can develop additional requirements to bolster our detection capability. For example, after considering the threat missile subclasses for a potential adversary, we may desire to increase the sensing range of the Sea-Based Sensor to extend our coverage into an adversary's territory into which a Ground Sensor solution is not feasible. We can now levy this requirement change on the Sea-Based Sensor subclass.

After we have detected a Threat Missile object, then we must develop a firing solution and engage the threat missile. As depicted in Figure 2, the BM/C2 class handles these functions and several other important functions. While these functions are related, the incorporation of these methods in a single class lessens the cohesion of the class. Rather than a single BM/C2 class, we might develop the BM/C2 class as an aggregate of several classes as depicted below in Figure 6.

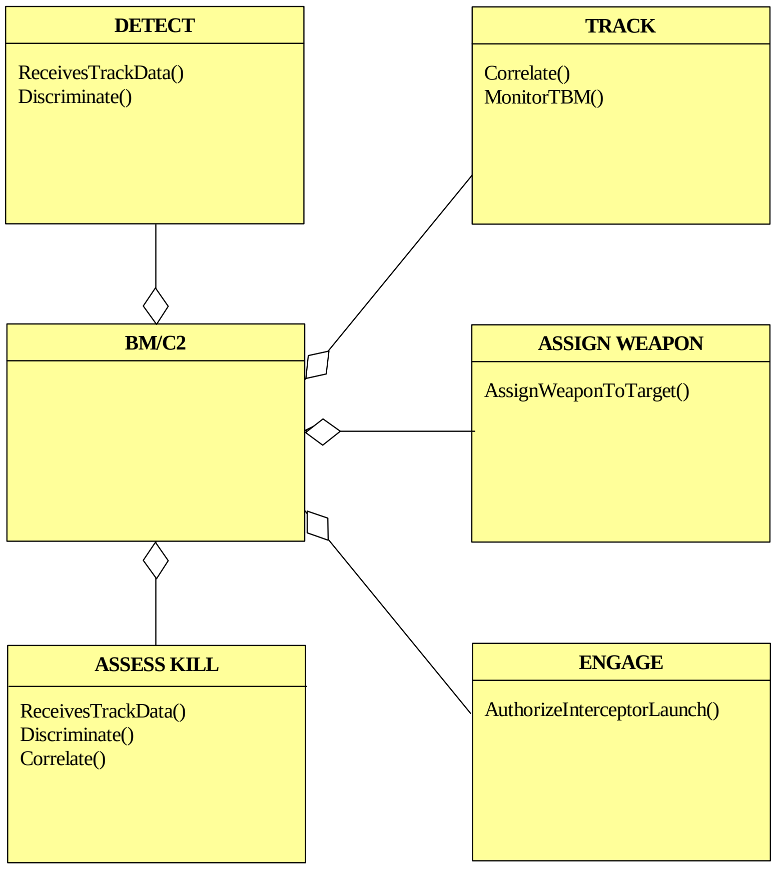


Figure 6. BM/C2 Class as an Aggregate

As depicted in Figure 3, we separated the methods for developing a firing solution from the BM/C2 class and assigned these methods to the Weapon class. These methods are similar in function so the cohesion of this class is high. This separation is important as the realizations of the BM/C2 class and the Weapon class may physically reside on different hardware platforms. So, in addition to increasing the cohesion, we reduce the coupling by substituting more interfaces that are small and better defined for the larger interface required for data flow and messaging of the sticks and circles architecture depicted in Figure 1 (reference Chapter II). The Weapon class and subclasses are shown below in Figure 7.

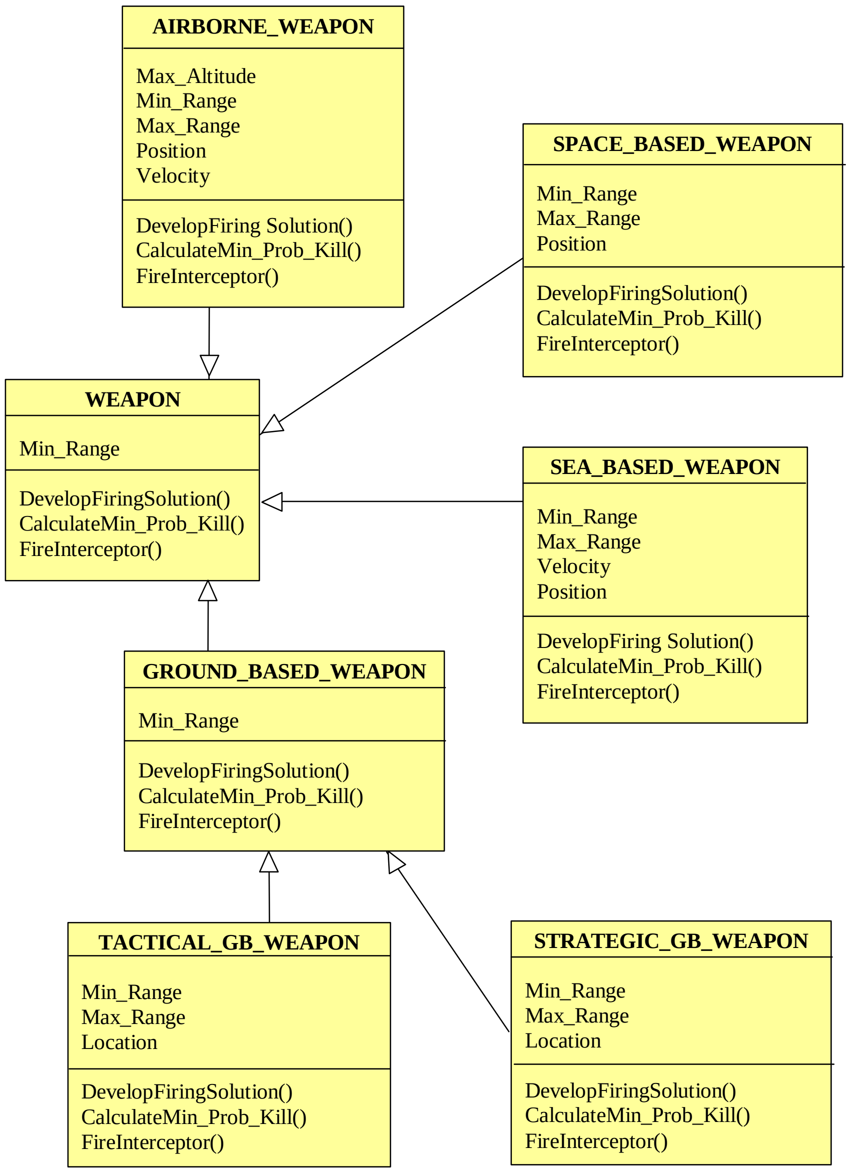


Figure 7. Subclasses of Weapon Class

Finally, we consider the Interceptor class. Given the attributes of the Threat Missile class as well as potential deployment of our hypothetical missile defense system, we can develop the attributes and associated requirements for the Interceptor class. For example, the velocity of the Intermediate Range subclass of the Threat Missile class ranges between 1 Km/second and 2 Km/second and the distance of this same subclass ranges from 1000 Km to 2000 Km. As we consider the minimum altitude in which we must negate the threat missile to ensure minimal ground effects of the resulting debris, we can determine minimum velocities for our three subclasses of the Interceptor class. These subclasses are depicted below in Figure 8.

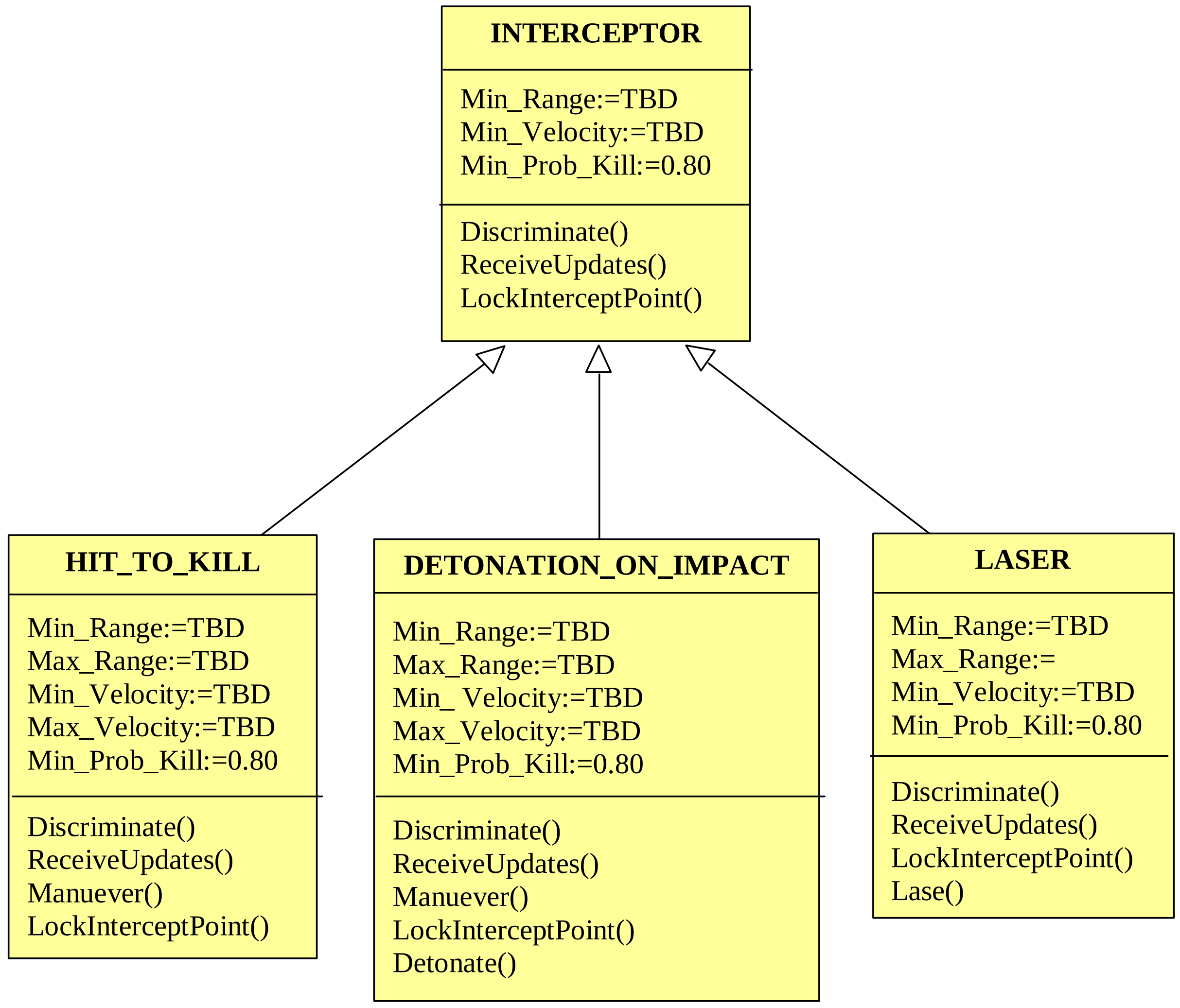


Figure 8. Subclasses of Interceptor Class

So, what can we glean from the above system-of-systems class diagram?

Minimal Messaging Between Classes. As we reason about the classes and subclasses of our missile defense system, we can see that we will develop many interfaces in the realization that replaces the single, large network interface of the sticks-and-circles diagram (reference Figure 1 in Chapter II). This is important to us in that we can manage a larger number of small, well-defined interfaces; however, the single, large network interface is much too unwieldy and complicated to manage effectively. We can reduce the messaging requirements of the large network interface to only that which is necessary for realizing the subclasses of our system-of-systems. Because the interface requirements are now manageable and known to all of the system developers, we have enhanced our ability to effectively integrate these systems into a system-of-systems.

Additionally, by treating the missile defense components as classes and developing concise interfaces that implement the minimum level of information sharing among the classes, we can define a data structure that implements data hiding. That is, by reducing the message traffic among classes to only that which is necessary to complete the missile defense missions and functions, we can prevent external programs from inadvertently modifying the state of a given class or injecting superfluous message traffic that may cause undesired system-of-systems behavior.

By defining a data-only interface strategy, we can greatly reduce the coupling of the missile defense components. A data-only interface design will result in a data-only integration realization. That is, each system within the missile defense system-of-systems will provide data that is suitable for transport and use by another system. Thus, the missile defense system-of-systems will exhibit the following properties

- More likely to work with legacy software code
- No build-time coupling in any system
- Missile defense systems are not required to share a common platform
- Missile defense systems can share a database to store exchanged data

A final benefit of realizing many small, well-defined interfaces rather than a single large interface will be the flexibility for incorporating future changes in a given class without negatively affecting the other classes. By data hiding and minimal message traffic, the software within a missile defense class is effectively independent in structure and realization than the other classes. As such, an internal software change to any single missile defense class should not affect any other class given that the interfaces among the classes remain unchanged [3].

Inheritance and Decentralized Control Flow. As we define the class and subclass attributes, the concept of inheritance becomes important in that the allocation of requirements through attributes and methods ensures consistency in the realization of the subclasses in our developments. Each system developer will know the minimum set of requirements that must be implemented and each developer knows what requirements the other developers will realize.

By careful assignment of methods to each class, we can avoid the creation of the so-called "god class" that performs the bulk of the work within the system-of-systems [4]. Typically, we overload the battle manager function with the vast majority of the work. More often than not, the battle manager software contains many dissimilar tasks and requires a complex messaging network. Rather than primarily exchanging control or triggering messages among several classes, the typical battle manager requires the continual transport of great amounts of data that results in more complex rules of messaging and bandwidth requirements. By employing the aforementioned UML and OOD techniques, we can reassign methods to other classes in which these methods are better suited.

For example, consider the discriminate method listed in the BM/C2 class in Figure 2. This requires that the Sensor class send a great deal of data to the BM/C2 class. Perhaps we might reason that the Sensor class should contain the discriminate method and send a much smaller, refined track file to the BM/C2 class for prosecution. This would greatly reduce the messaging requirements and greatly simplify the interface between the Sensor class and the BM/C2 class.

Encapsulation. As we reason about the classes and subclasses of the hypothetical system, we find that we can modify the methods to maximize the benefits of data hiding within the appropriate class. In the large sticks-and-circles network of Figure 1, nearly all data is public by definition of the single, large interface to each system. By developing appropriate methods for each class, we can begin to hide data within its class.

For example, consider the development of a firing solution for a given threat missile. In the large sticks-and-circles network, the firing solution uses public data that is visible to all other systems. Because the data is public and the network connects each system to all other systems, it is difficult for software designers to understand the impact on system behavior as it is not readily apparent what system functionality is dependent on the public data.

On the other hand, we can determine the data requirements for the development of the firing solution in the Weapon class in Figure 6, and understand that the software developers should hide that data within the Weapon class. While this data hiding may be more difficult in procedural software, the public data issue is more readily apparent in the class views of the system-of-systems than in the large sticks-and-circles network diagram.

Activity Diagram. (Please refer to the BMDS Activity Diagram that is depicted on pages 46 & 47.) The BMDS Activity Diagram provides a visual representation of the sequencing of missile defense events from the perspective of a single threat ballistic missile as described in the BMDS Use Cases. While the diagram depicts a single sequence of events, each detected threat object follows this process concurrently without any correlation to the other processes of the other threat objects.

In looking for inefficiencies, it is worth noting that the differences in tracking a threat object and assessing the kill of a threat object are minimal. While additional redundancy may be revealed in the development of future design and architectural artifacts, we have established a return path from the kill assessment sequence of events back to the detect and track sequence of events. This efficiency should reduce the amount of code and effort required to realize the design.

Of particular interest in the activity diagram are the two branches in the diagram which are discussed as follows:

After the completion of the Correlate TrackData activity, the diagram branches into two paths. The guard condition [Correlate to Existing Track] allows the activity to proceed down this path given that the BMDS associated the newly received track data to an existing track file. This ensures that the BMDS updates the existing threat ballistic missile characteristics in the track file rather than creating a new track file for each newly received set of track data. If [Correlate to Existing Track] is not true, then the BMDS will assign a new track number to the track data and develop a new track file.

After the BMDS has assessed the newly received track data from a reported intercept, the path divides into two paths. The guard condition [Target Negated] opens the path that represents the case of the BMDS determination that the interceptor destroyed the threat ballistic missile. This path terminates with the completion of the diagram. The other path represents two possible situations: (1) the BMDS determines that the threat ballistic missile was not destroyed or (2) the BMDS cannot make a determination on the destruction of the threat ballistic missile. This path merges just prior to the Apply Feature Recognition activity. This will cause the BMDS to repeat the cycle so that the weapon can initiate the events required to destroy the threat ballistic missile.

Sequence Diagrams. (Please refer to the BMDS Sequence Diagrams that are depicted on pages 48-50.) We developed three sequence diagrams that depict the interactions among the classes. For the sequence diagrams, we identified an infrared sensor and a radar sensor as the mission differences between the two sensors are significant. Infrared sensors have the primary responsibility for detecting a missile launch while the radars have the primary responsibility for developing a fire quality track for the weapon. Although the BMDS BMC2 will be a composite of many objects, we will treat it as a single object at this time. This will make the initial iterations of the analysis easier to follow.

As we view the BMDS sequence diagrams, one subtle aspect of the BMDS comes to light. While the sensors are depicted as a class generalization, the subclasses of the

Sensor class will not transmit track data in a synchronous fashion with respect to each other. In other words, the Sensor subclasses will have varying sweep rates and data transfer rates. As a means of simplifying the implementation of the BMDS, it would seem wise to consider a ReceiveTrackData module that resides in each Sensor subclass. The ReceiveTrackData module would be responsible for getting the track data from the Sensor subclass by formatting the track data to a common BMDS protocol and transferring the track data to the BMDS BMC2. The ReceiveTrackData module could be standardized across the BMDS to facilitate ease of information transmission. Additionally, this approach reduces the amount of transmitted data given that raw radar traffic from multiple sensors would incur a huge network bandwidth requirement.

The Sensor subclasses will transmit track data to the BMDS BMC2 continuously. As such, the BMDS BMC2 must work all its functions concurrently so that the BMDS can react rapidly and effectively. Also, the sequence diagrams identify numerous BMDS evaluations that will require persistent data: intelligence profiles of threat ballistic missiles, weapon availability, interceptor characteristics, phenomenology data of adversary missile booster flames, and phenomenology data of exploding adversary warheads.

Statechart Diagrams. (Please refer to the BMDS Statechart Diagrams that are depicted on pages 51-56.) Recall that we determined that the BMDS BMC2 must work all its functions concurrently so that the BMDS can react rapidly and effectively. This will require orthogonal states in the BMDS BMC2 statechart diagram. For the initial iterations of the statechart diagrams, we will expand the activities in the activity diagram to provide a greater level of detail of the activities within each substate.

Within the BMDS state, we have identified six orthogonal states: BMDS On/Off, Detect, Track, Assign Weapon, Engage, and Assess Kill. The latter five states represent the five major functions along the threat ballistic missile kill chain.

We incorporated timers in many substate regions to ensure that the substate is exited regardless whether the activities are completed. This is important as the BMDS BMC2 cannot cease to function if a specific activity is hung. It is more important to cease the BMDS BMC2 activities for a single threat ballistic missile for the greater cause of negating the remainder of the detected threat ballistic missiles vice halting all BMDS BMC2 operations while waiting for the BMDS BMC2 to clear a single, hung activity.

Note that the BMDS can assign up to 1000 track numbers (i.e. 000-999) in the Track Region. It is essential that the BMDS have the capability to track a significant number of detected objects. Consider the situation in which a weapon destroys a threat ballistic missile. This missile will break up into many smaller pieces – some of which will have mass and velocity. The BMDS must monitor these pieces until it can determine that the pieces are missile debris rather than threat ballistic missiles.

The statechart also suggests a requirement for access to stored data. Note that in the Detect region (substate S1_R2.3) that we want to apply feature recognition to the newly received track data. To identify the track as a threat object, the battle management function must have direct access to stored data of threat ballistic missile characteristics that can be compared against track data. This requirement will be true in the Track region (S2_R2) in that the battle management function must have access to stored data of intelligence data so that the battle management function can determine the type and the known characteristics of the threat object.

The statechart also reveals a requirement to develop two critical algorithms that are essential to employ for multiple sensors that are tracking multiple threat objects. The algorithm for data fusion is critical for a multiple sensor environment given that each sensor will report track data to the battle management function which must fuse the multiple track reports on a single threat object to form a single track report for that threat object. Additionally, given that the battle management function will concurrently track multiple objects, we must develop a correlation algorithm so that the incoming track reports are associated with the appropriate track files. If a track report cannot be associated with a currently active track file, then the battle management function must assign a new track number to the track report and initiate a new track file.

In this iteration of the statechart logic, note that we employed a brute force method for updating each track file regardless whether it contains active track data. In later iterations of the statechart, a more efficient method of accessing and reviewing track files will be necessary to avoid unnecessary processing and delaying the prosecution of the threat ballistic missiles.

Broker Pattern. (Please refer to the BMDS Broker Pattern diagrams that are depicted on pages 57-58.) We chose to employ the Broker Pattern for the BMDS problem for two reasons: (1) the Broker Pattern offers good decoupling of the Sensor subclasses from the BMDS and the Weapon subclasses from the BMDS, and (2) the BMDS will not know the location of the Sensor subclasses and Weapon subclasses at compile time. The BMDS can be deployed anywhere on Earth so the number, subclasses, and location of the Sensor and Weapon classes cannot be determined until an adversary launches a ballistic missile attack. With the Broker Pattern, the Sensor and Weapon subclasses register with the BMDS Broker during the boot process of those subtypes. Of additional benefit to the BMDS is that subclasses can enter, leave, and re-enter the network at any time. Finally, we can implement the Broker Pattern in such a way that the BMDS Broker stores the received track data from the Sensor subclasses and forwards the track data to the BMDS Detect Feature at a programmed data transfer rate.

The Broker Pattern also provides for the integrating of a Receive Track Data module within the Sensor subclasses. This particular pattern fits the BMDS problem very well.

Conclusions. Before we develop the conceptual software architecture views, let us review what we have learned in the domain analysis.

- We identified five major functions that compose the missile defense kill chain: Detect, Track, Assign Weapon, Engage, and Assess Kill.
- The BM/C2 will control the BMDS messaging and activities.
- We can identify specific messaging requirements among the classes.
- We should employ data hiding techniques to decrease the coupling issue.
- The sensor class will continuously transmit data to the BM/C2.
- The sweep rates of the sensors are independent of the update rate from the sensors to the BM/C2.

- Given that the Sensor superclass will result in various Sensor subclasses that continually transmit data to the BMDS BMC2, the BMDS software designers will need to develop track fusion algorithms.
- The BM/C2 will require concurrent processing activities.
- Incoming track data will either update an existing track file or initiate a new track file.
- The BM/C2 must confirm a threat missile kill before dropping the track.
- Given that sensors, BM/C2 components, and weapons may enter and leave the network, we should consider the broker pattern for the subscription of services into the network.

Additionally, we have developed additional questions that must be addressed prior to the system design:

- What will be the update rate from the sensor class to the BMDS BMC2?
- What will be the maximum number of objects that the BMDS BMC2 must track?
- What is the range in number of sensors that will feed into the BMDS BMC2?
- Will the BMDS battle management be automated or manual?
- Will the BMDS battle management be centralized or decentralized?
- What BMDS battle management overrides are required to prevent undesired interceptor launches?
- What BMDS battle management software interlocks are required to minimize the occurrence of an inadvertent launch?
- With respect to dynamically extending missile defense coverage, what should the BMDS BMC2 do if a weapons platform is either negated or its magazine is empty?

· With respect to threat identification and classification, what should the BMDS BMC2 do if a new track does not match the profiles in the persistent data?

· Consider the engage function. What will be the trigger to cease track refinement and to authorize the launch of an interceptor?

· Consider debris clouds formed as a result of previously negated threat missiles. What is the impact of the debris to the sensors and reporting of threat objects?

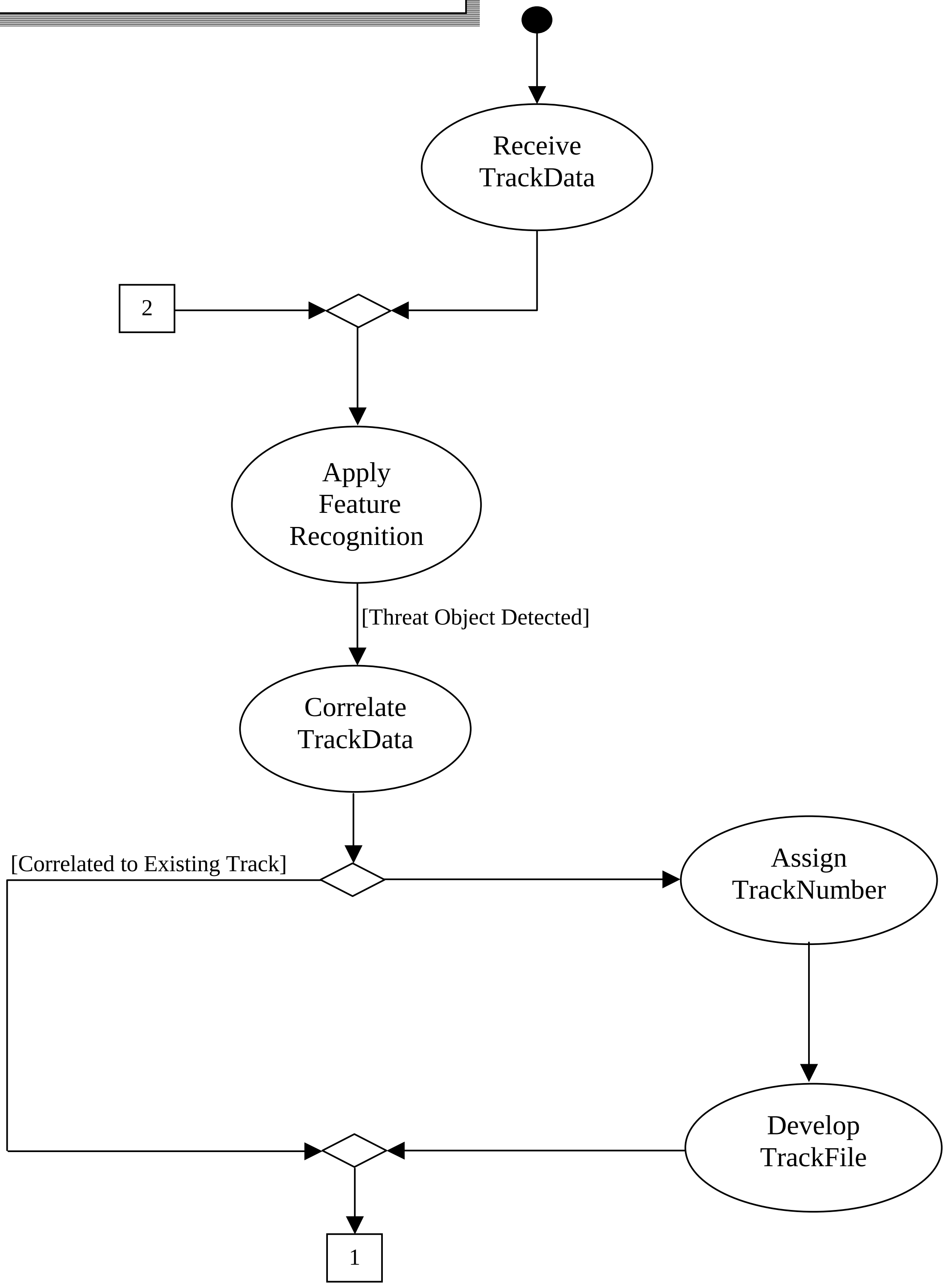
· Consider a track lost in a debris cloud. Should the BMC2 propagate the track with predictive information and continue to report the track or should the BMC2 issue a drop track message and notify potentially impacted defended assets of the lost track?

· Consider a forward-based sensor that reports to the battle management function. What are the timing requirements (i.e., latency) for the track reports from the sensor to the battle management function?

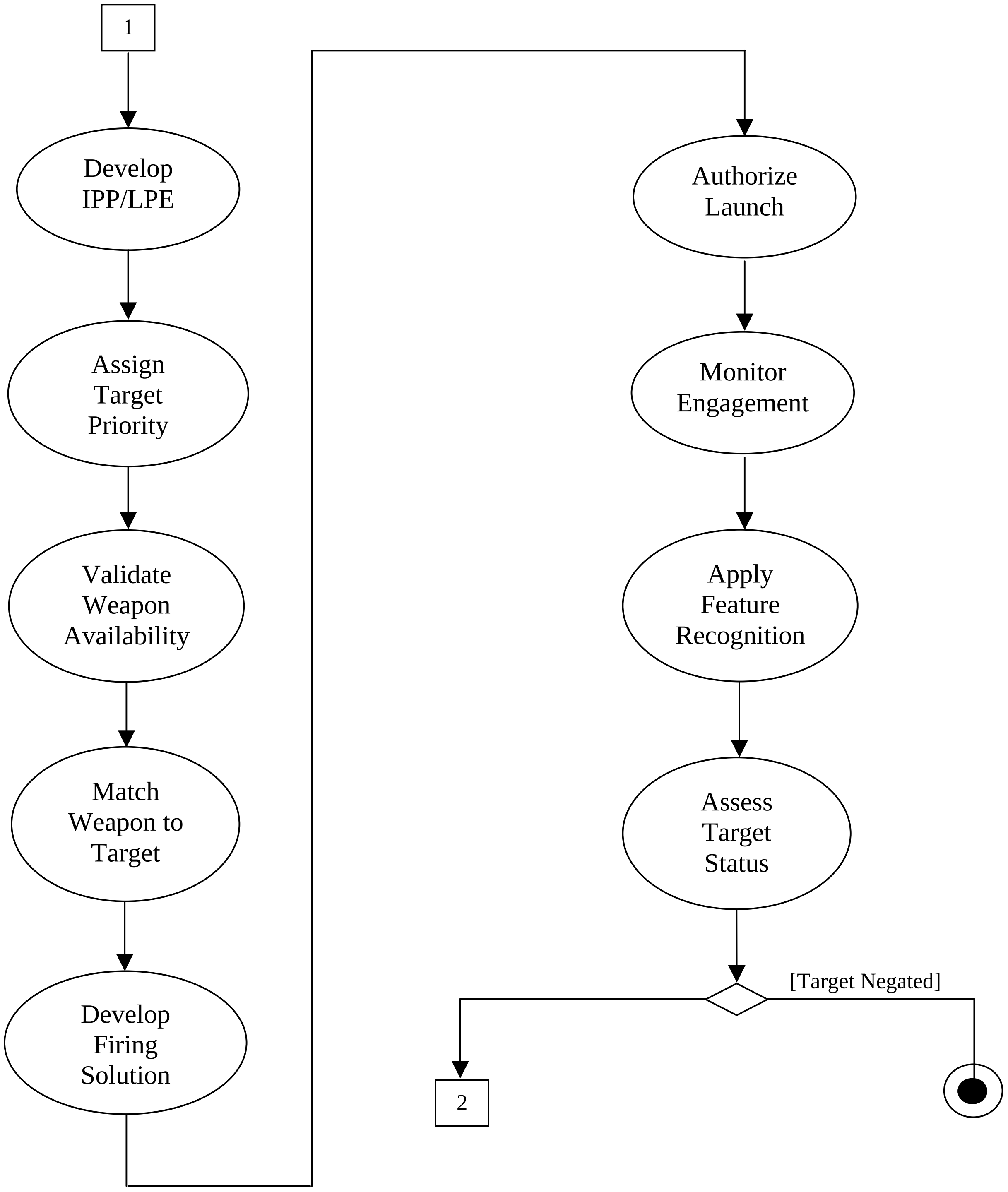
· Consider weapons that use remote tracks for firing solutions. What are the timing requirements (i.e, latency) and accuracy requirements (i.e. fire quality track) for the weapon?

· Consider establishing the correlation and fusion requirements. What is the solution for a common timing source and a common geodetic reference source for the BMDS classes?

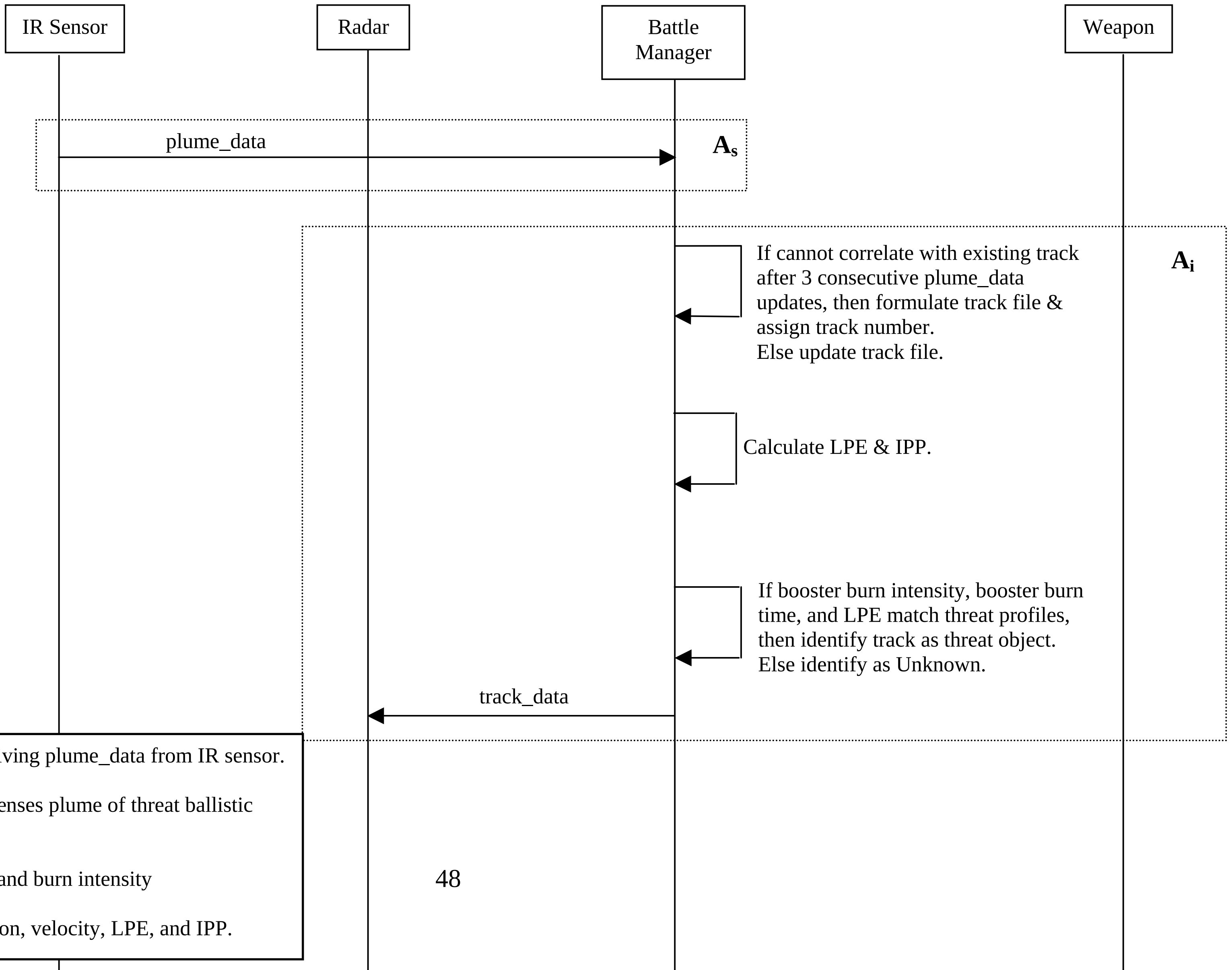
Battle Management Activity Diagram
Page 1 of 2



Battle Management Activity Diagram
Page 2 of 2



Battle Management Sequence Diagram
IR Detect & Cue: Boost Phase
Page 1 of 3


IR Sensor
Radar
Battle Manager
Weapon
plume_data
A_s
A_i
If cannot correlate with existing track after 3 consecutive plume_data updates, then formulate track file & assign track number.
Else update track file.
Calculate LPE & IPP.
If booster burn intensity, booster burn time, and LPE match threat profiles, then identify track as threat object.
Else identify as Unknown.
track_data

Correlate track data with existing track files.

Compare booster burn intensity and time of burn stored in track file to stored threat profiles.

Compare developed IPP/LPE to threat intelligence data.

Cue radar with track data.

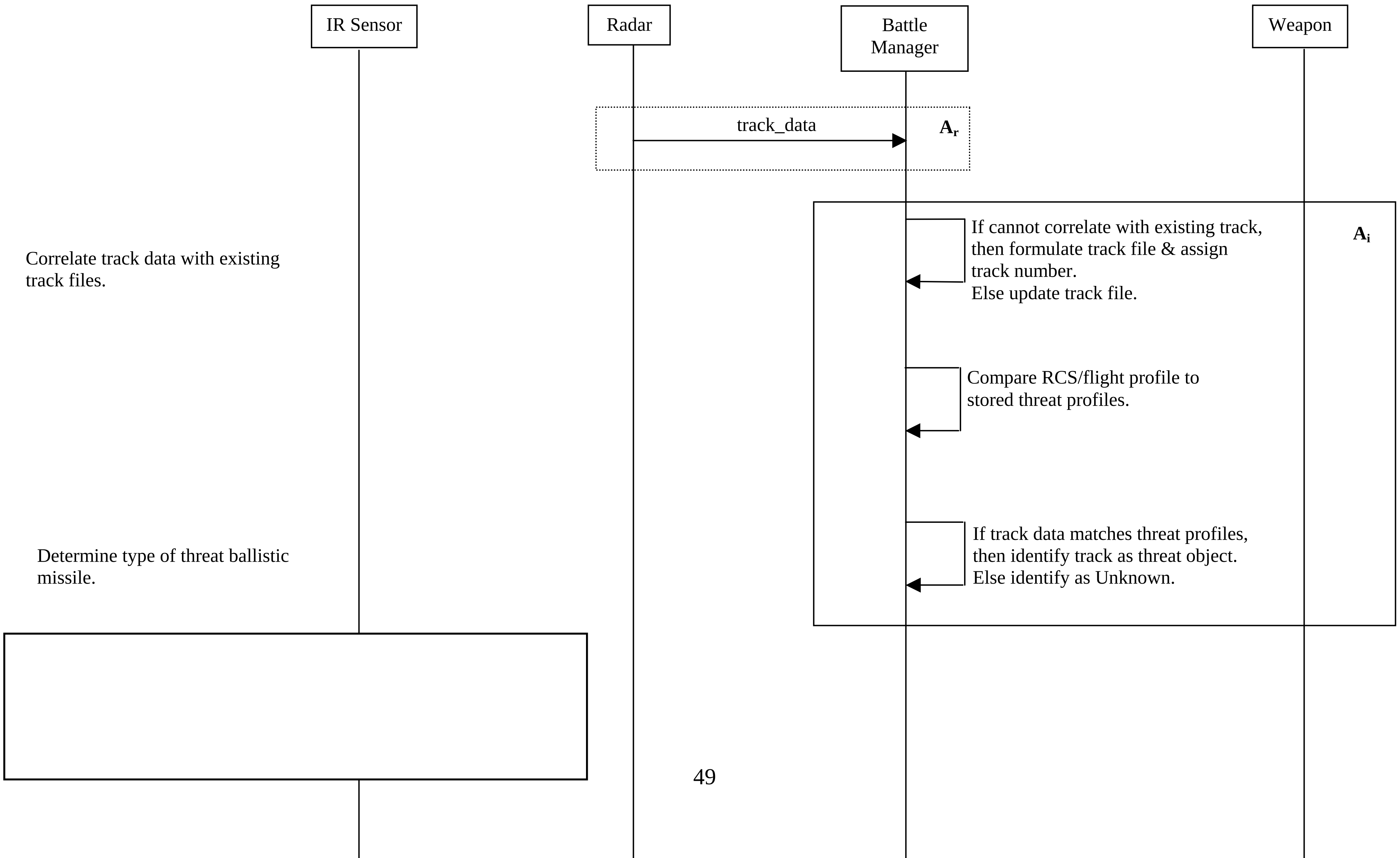
Note 1: A_i continually repeats while receiving plume_data from IR sensor.

Note 2: A_s continually repeats while IR senses plume of threat ballistic missile. A_s is independent of A_i.

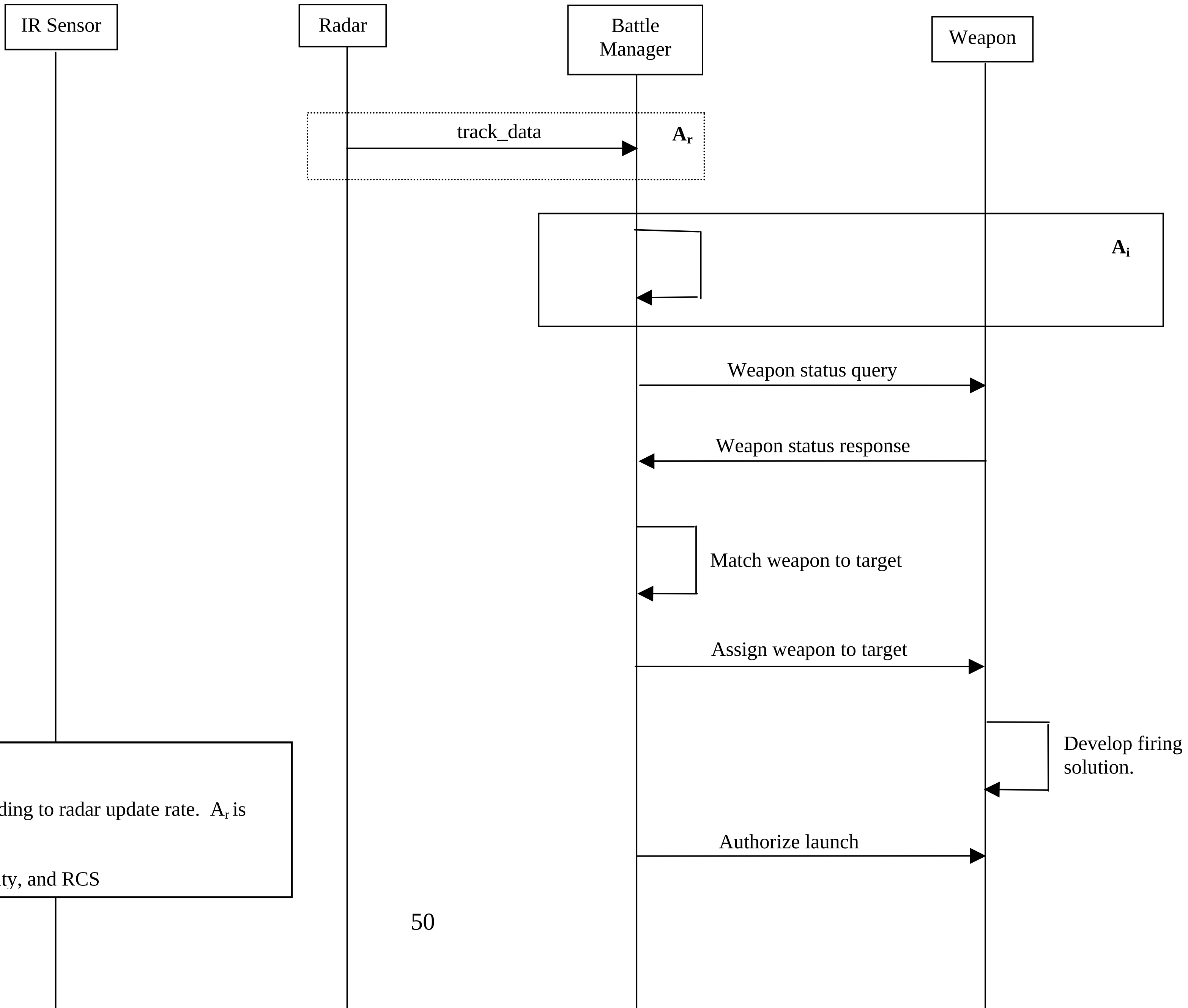
Note 3: plume_data = position, velocity, and burn intensity

Note 4: track_data = track number, position, velocity, LPE, and IPP.

Battle Management Sequence Diagram
Radar Tracking: Mid-Course Phase
Page 2 of 3



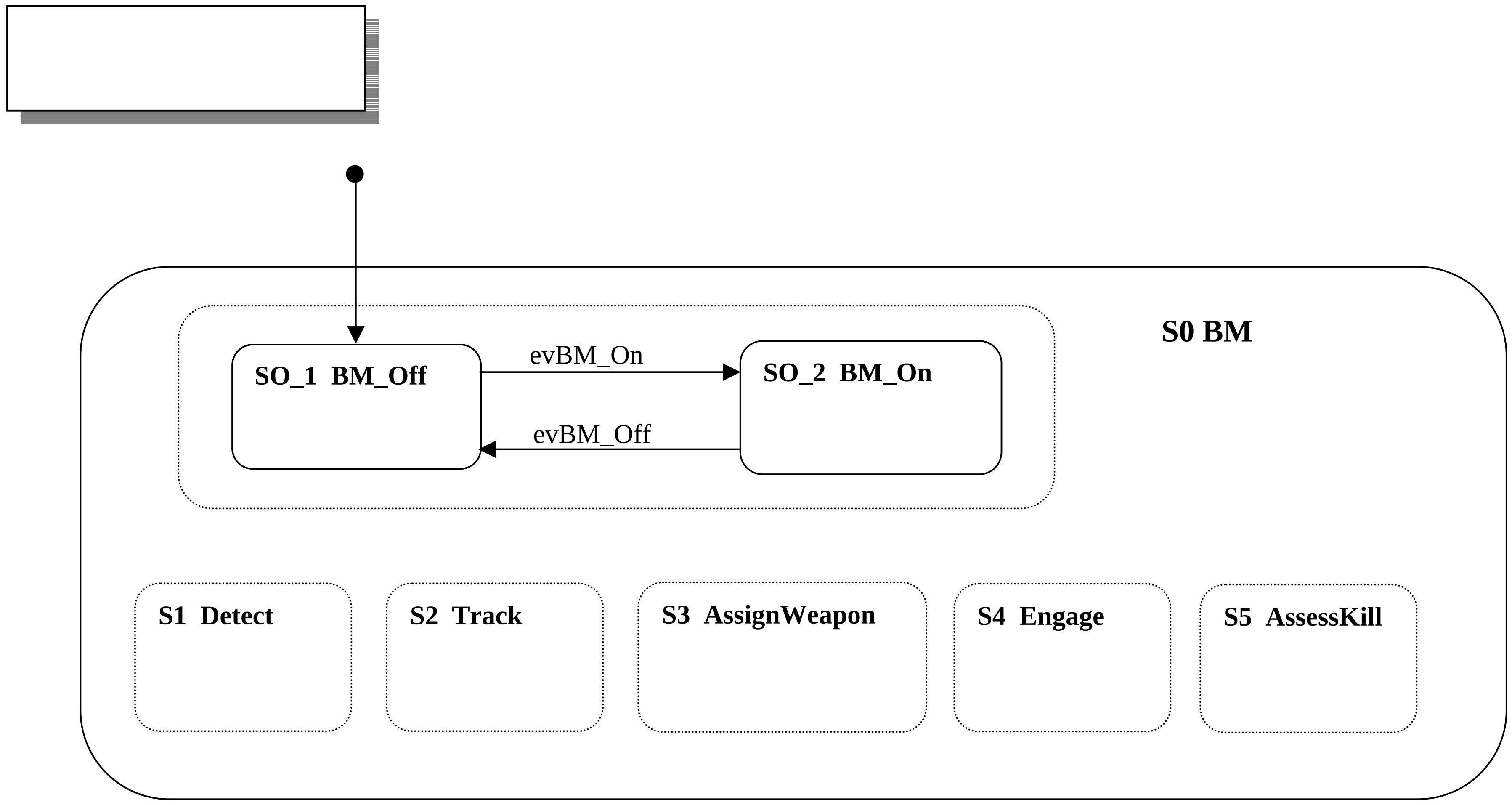
IR Sensor
Radar
Battle Manager
Weapon
track_data
A_r
A_i
If cannot correlate with existing track, then formulate track file & assign track number. Else update track file.
Compare RCS/flight profile to stored threat profiles.
If track data matches threat profiles, then identify track as threat object. Else identify as Unknown.
Correlate track data with existing track files.
Determine type of threat ballistic missile.

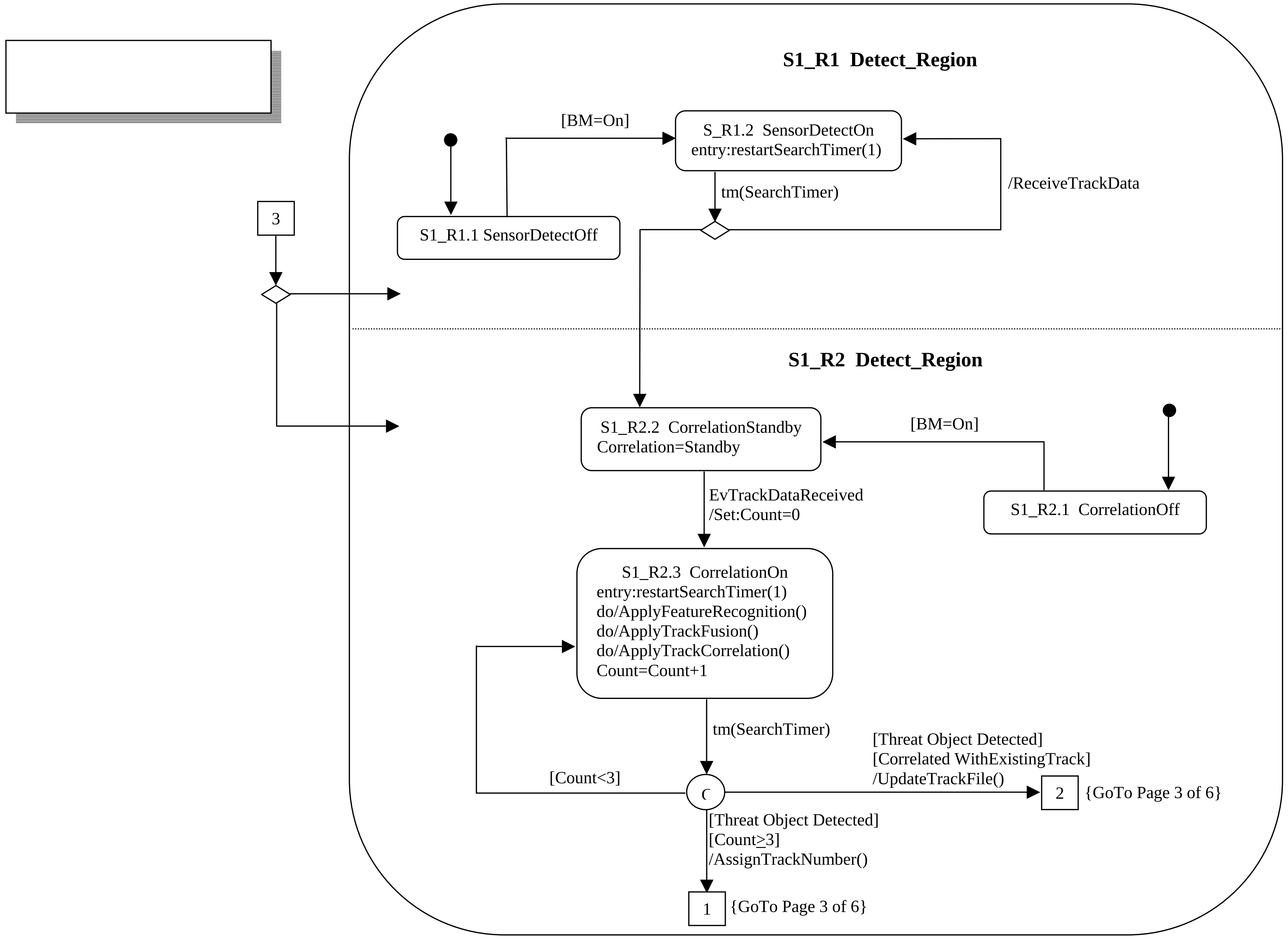


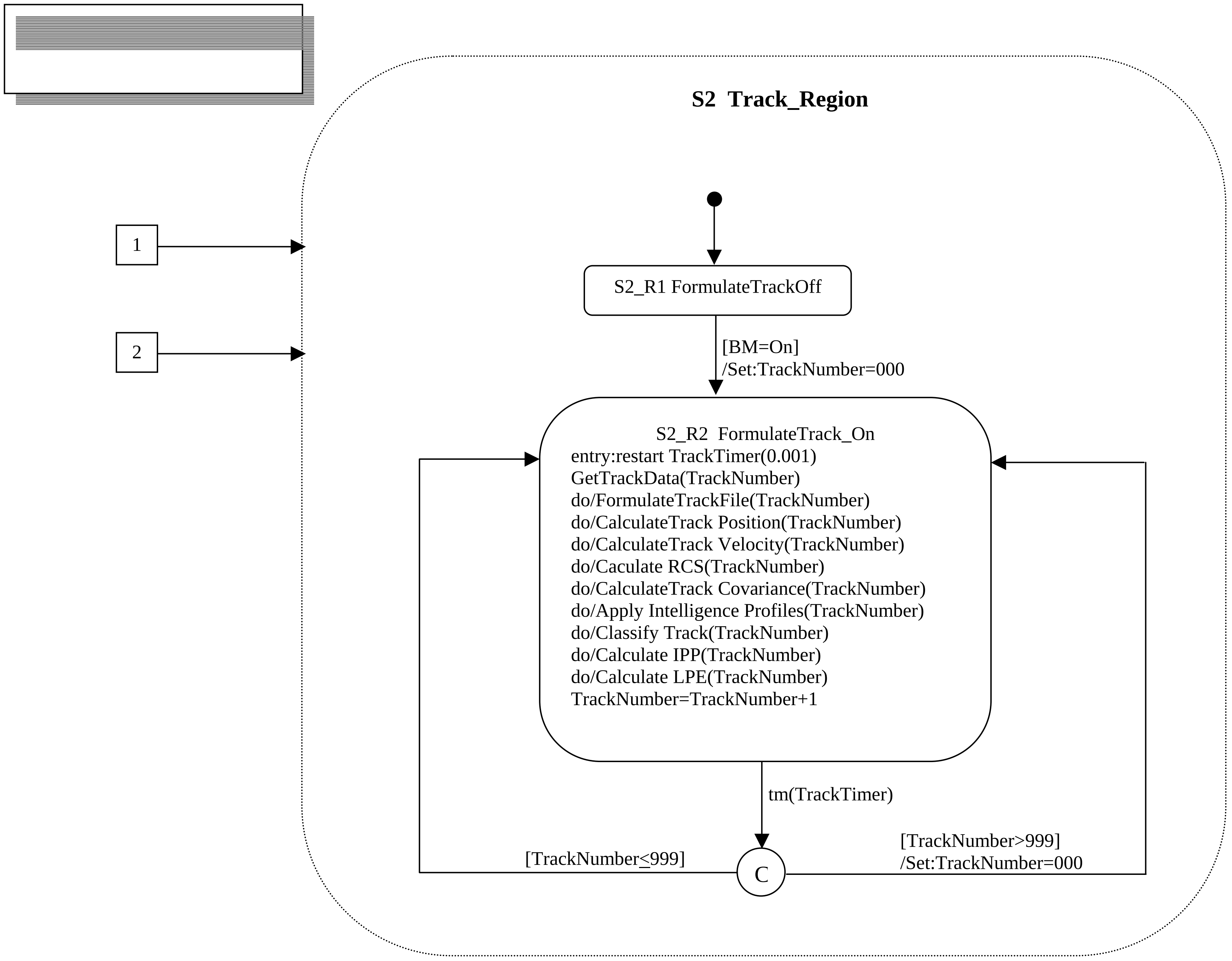
Note 1: A_i continually repeats.

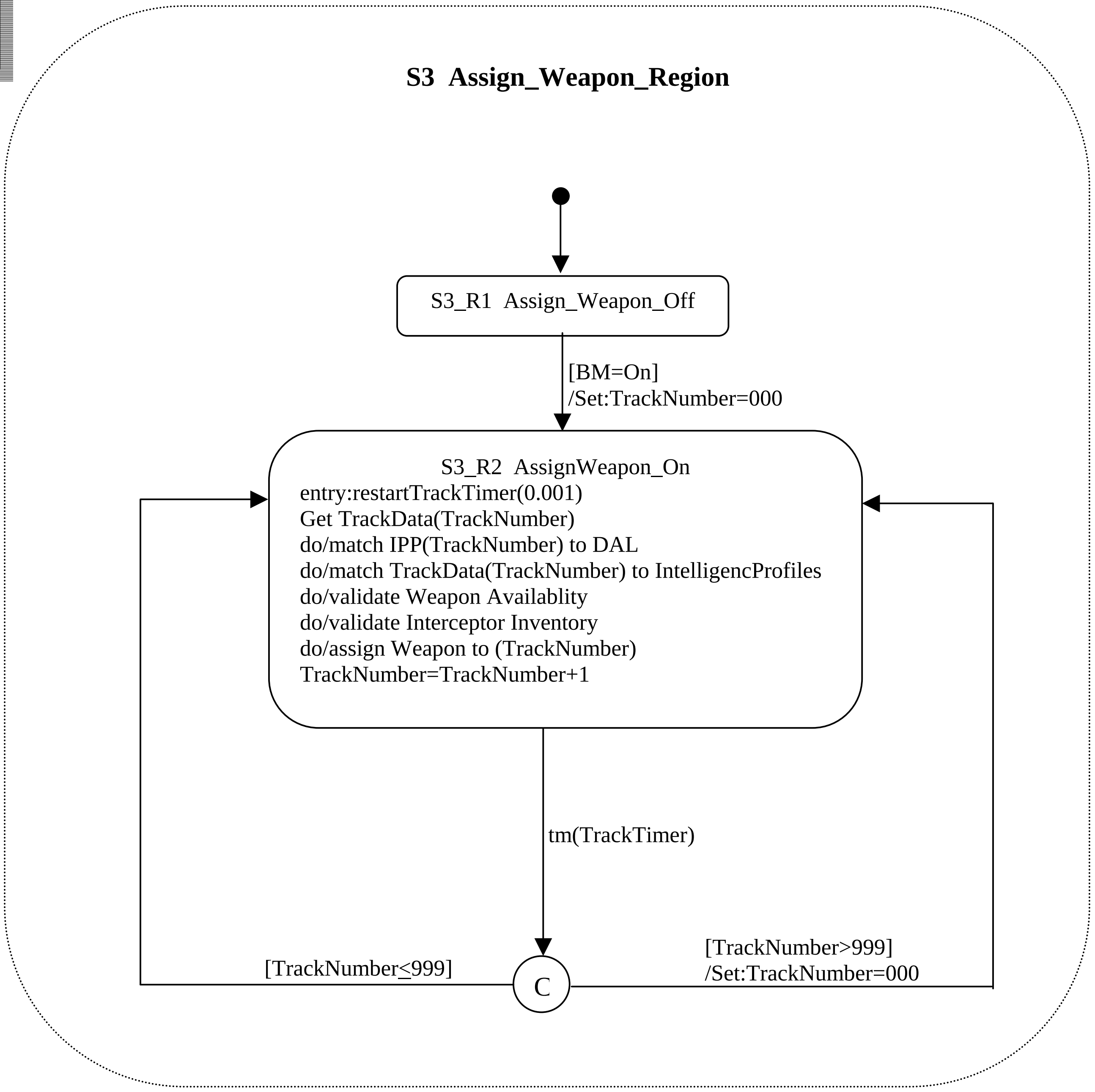
Note 2: A_r continually repeats according to radar update rate. A_r is independent of A_i.

Note 3: track data = position, velocity, and RCS

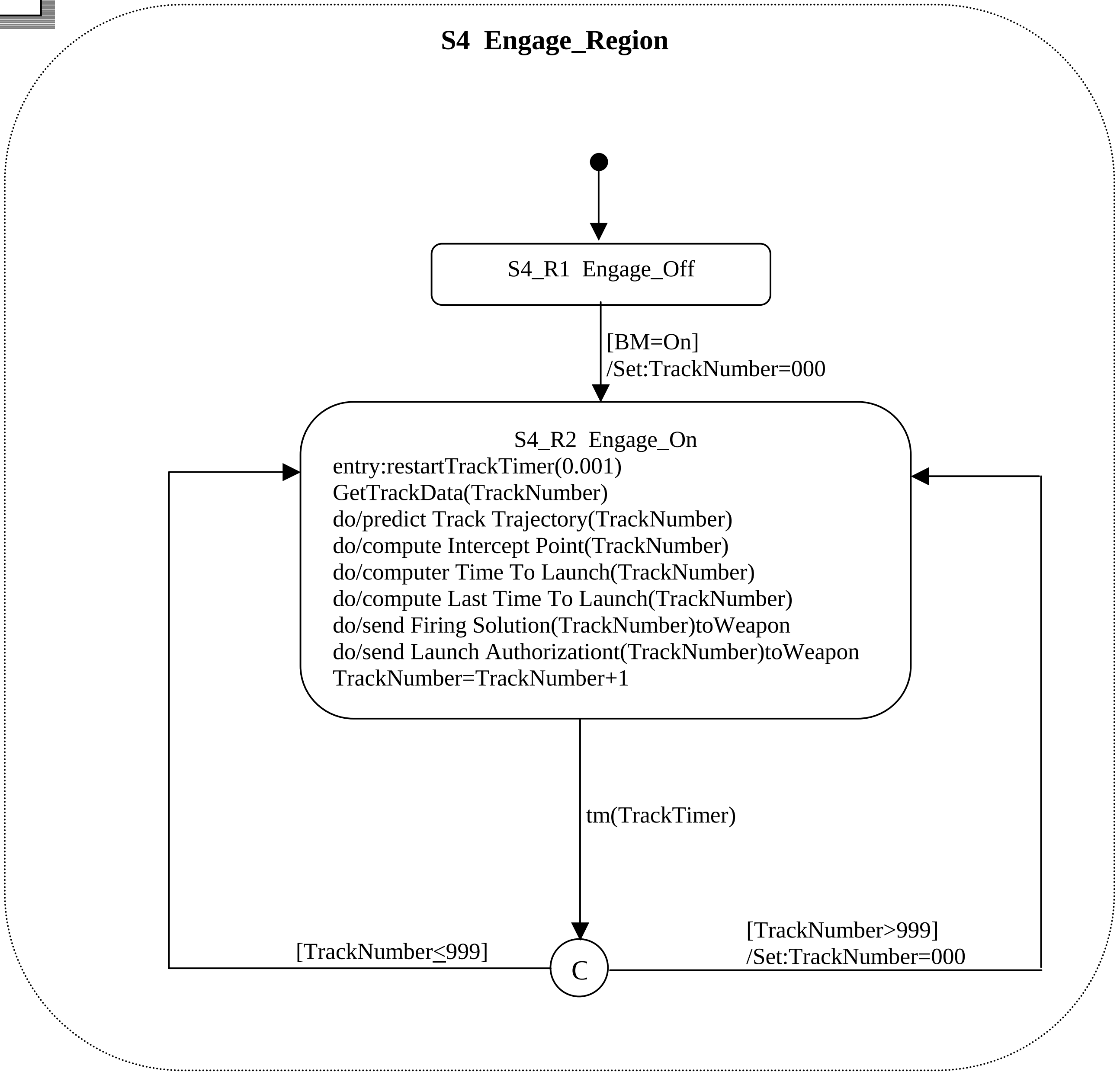

S0 BM
SO_1 BM_Off
evBM_On
SO_2 BM_On
evBM_Off
S1 Detect
S2 Track
S3 AssignWeapon
S4 Engage
S5 AssessKill

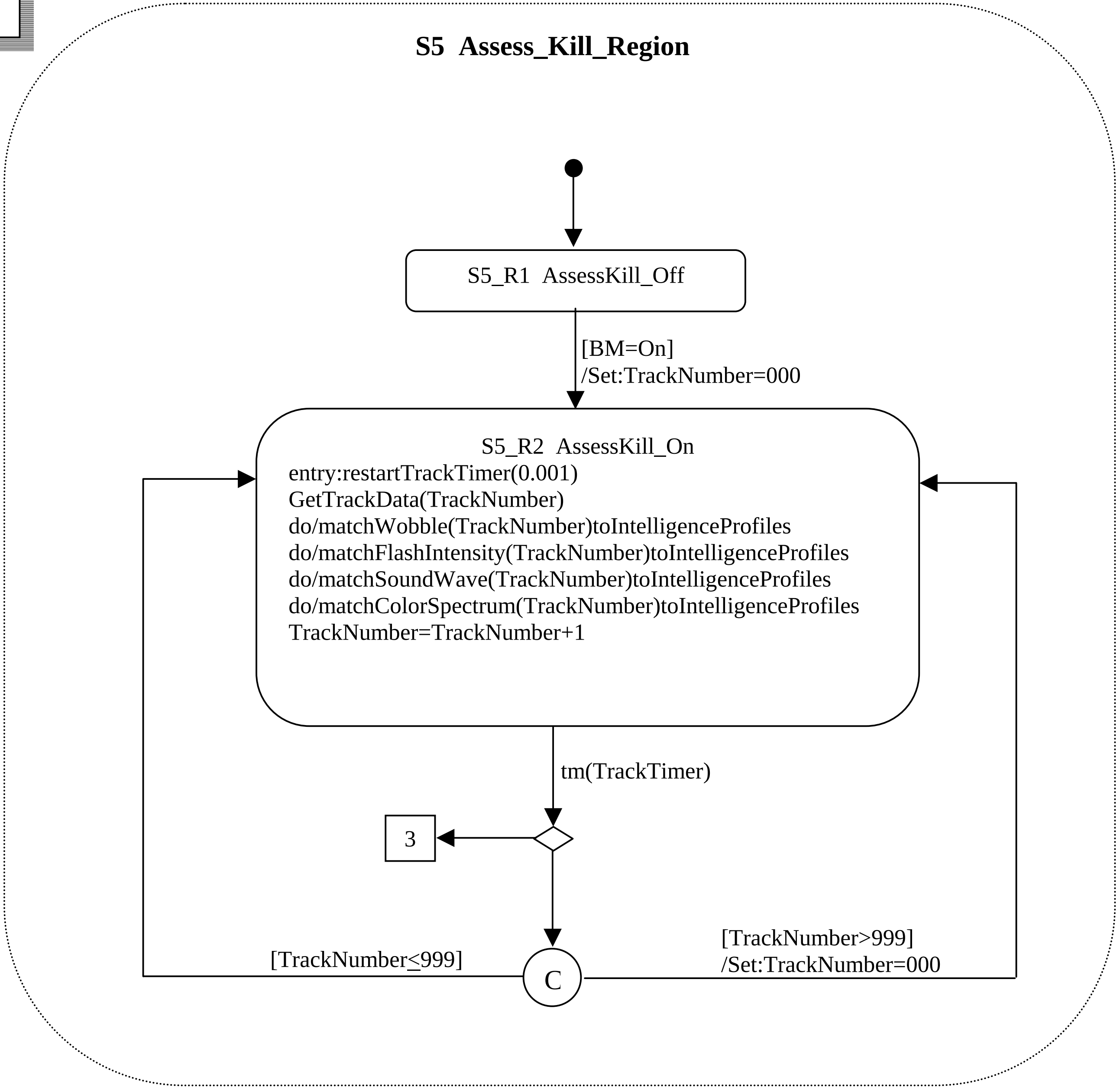

S1_R1 Detect_Region
[BM=On]
S_R1.2 SensorDetectOn
entry:restartSearchTimer(1)
/ReceiveTrackData
tm(SearchTimer)
3
S1_R1.1 SensorDetectOff
S1_R2 Detect_Region
S1_R2.2 CorrelationStandby
Correlation=Standby
[BM=On]
EvTrackDataReceived
/Set:Count=0
S1_R2.1 CorrelationOff
S1_R2.3 CorrelationOn
entry:restartSearchTimer(1)
do/ApplyFeatureRecognition()
do/ApplyTrackFusion()
do/ApplyTrackCorrelation()
Count=Count+1
tm(SearchTimer)
[Threat Object Detected]
[Correlated WithExistingTrack]
/UpdateTrackFile()
[Count<3]
C
2
{GoTo Page 3 of 6}
[Threat Object Detected]
[Count≥3]
/AssignTrackNumber()
1
{GoTo Page 3 of 6}


S2 Track_Region
1
2
S2_R1 FormulateTrackOff
[BM=On]
/Set:TrackNumber=000
S2_R2 FormulateTrack_On
entry:restart TrackTimer(0.001)
GetTrackData(TrackNumber)
do/FormulateTrackFile(TrackNumber)
do/CalculateTrack Position(TrackNumber)
do/CalculateTrack Velocity(TrackNumber)
do/Caculate RCS(TrackNumber)
do/CalculateTrack Covariance(TrackNumber)
do/Apply Intelligence Profiles(TrackNumber)
do/Classify Track(TrackNumber)
do/Calculate IPP(TrackNumber)
do/Calculate LPE(TrackNumber)
TrackNumber=TrackNumber+1
tm(TrackTimer)
[TrackNumber≤999]
C
[TrackNumber>999]
/Set:TrackNumber=000

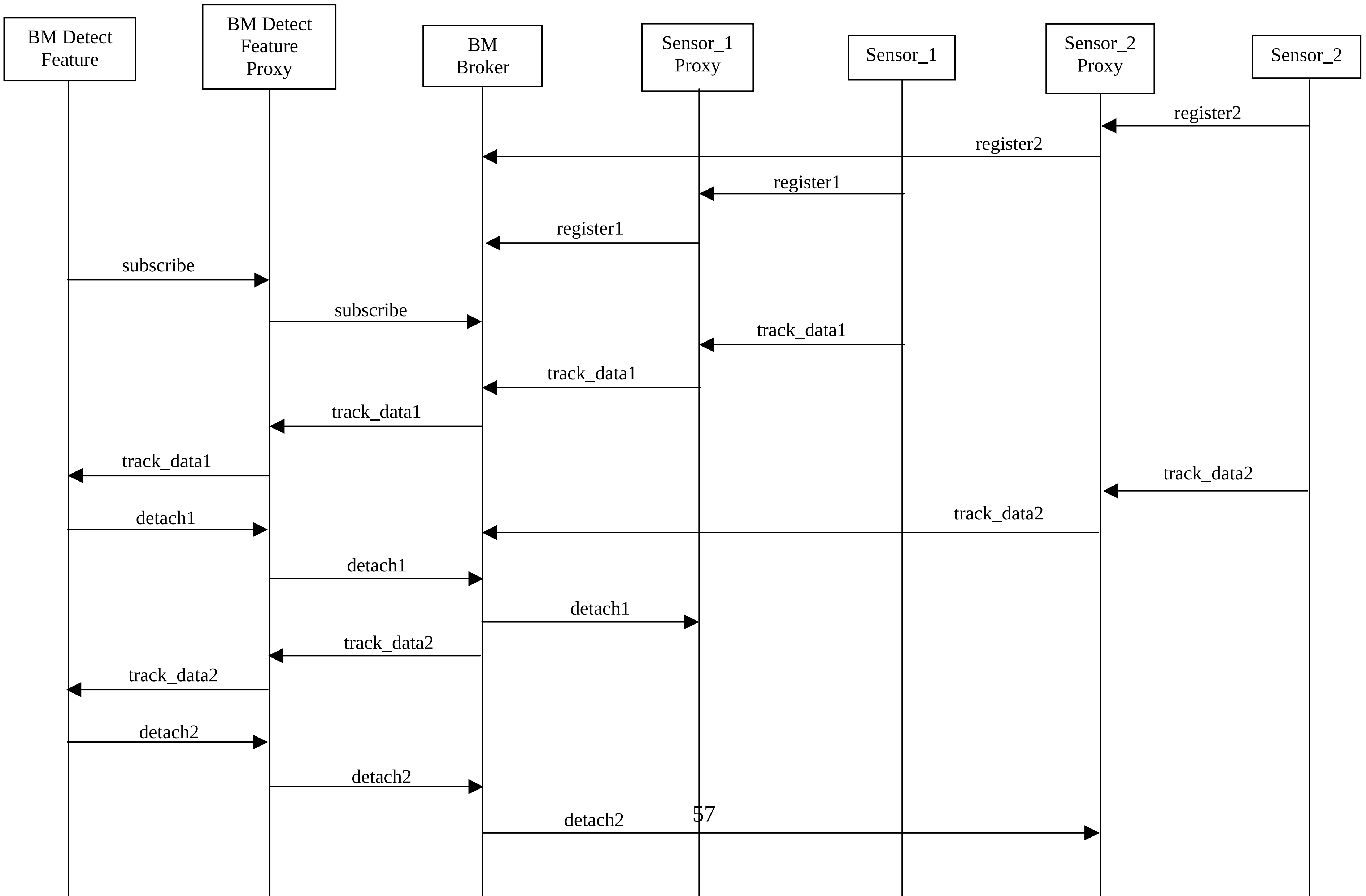


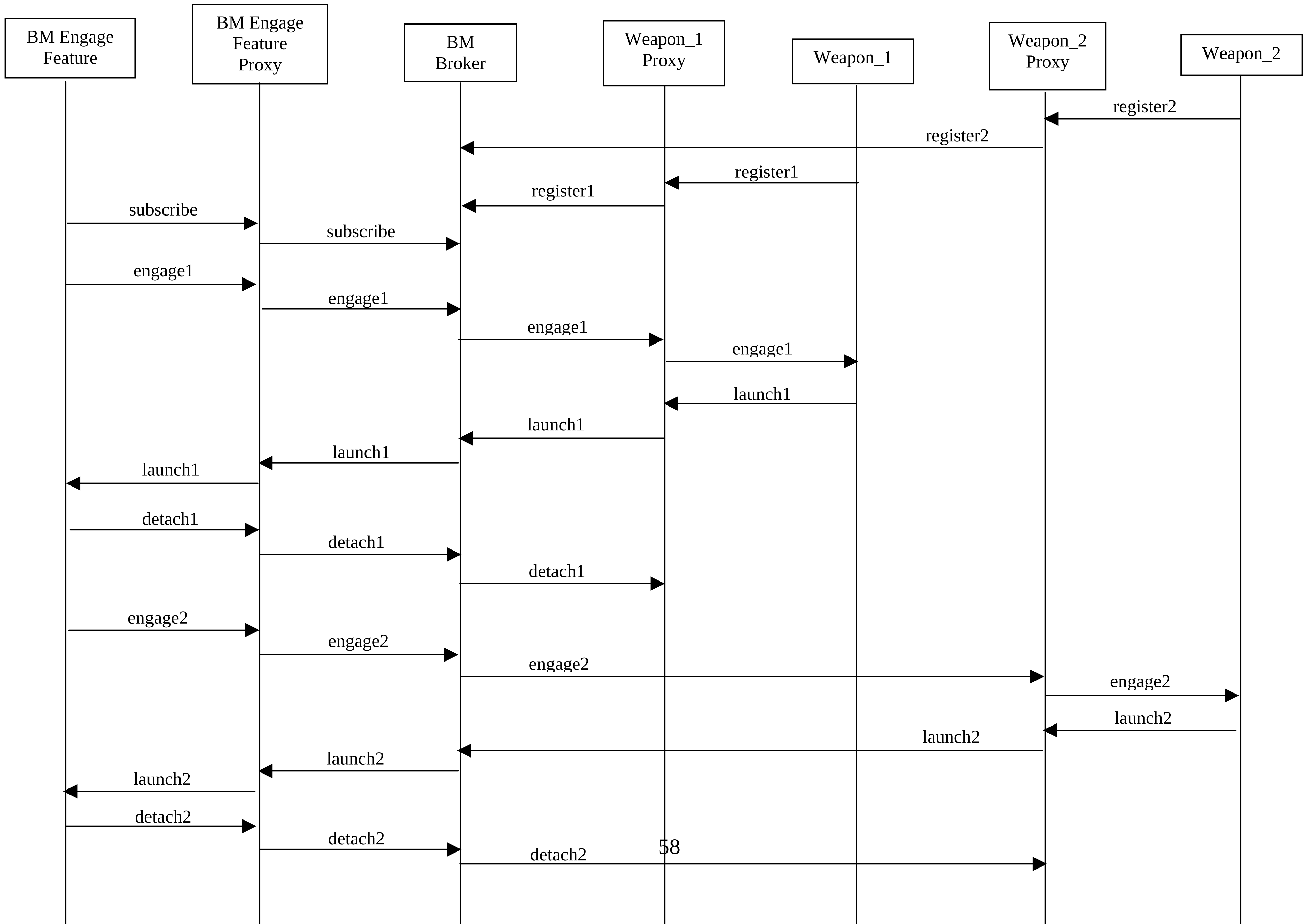
S3 Assign_Weapon_Region
S3_R1 Assign_Weapon_Off
[BM=On]
/Set:TrackNumber=000
S3_R2 AssignWeapon_On
entry:restartTrackTimer(0.001)
Get TrackData(TrackNumber)
do/match IPP(TrackNumber) to DAL
do/match TrackData(TrackNumber) to IntelligencProfiles
do/validate Weapon Availablity
do/validate Interceptor Inventory
do/assign Weapon to (TrackNumber)
TrackNumber=TrackNumber+1
tm(TrackTimer)
C
[TrackNumber≤999]
[TrackNumber>999]
/Set:TrackNumber=000


S4 Engage_Region
S4_R1 Engage_Off
[BM=On]
/Set:TrackNumber=000
S4_R2 Engage_On
entry:restartTrackTimer(0.001)
GetTrackData(TrackNumber)
do/predict Track Trajectory(TrackNumber)
do/compute Intercept Point(TrackNumber)
do/computer Time To Launch(TrackNumber)
do/compute Last Time To Launch(TrackNumber)
do/send Firing Solution(TrackNumber)toWeapon
do/send Launch Authorizationt(TrackNumber)toWeapon
TrackNumber=TrackNumber+1
tm(TrackTimer)
[TrackNumber≤999]
C
[TrackNumber>999]
/Set:TrackNumber=000


S5 Assess_Kill_Region
S5_R1 AssessKill_Off
[BM=On]
/Set:TrackNumber=000
S5_R2 AssessKill_On
entry:restartTrackTimer(0.001)
GetTrackData(TrackNumber)
do/matchWobble(TrackNumber)toIntelligenceProfiles
do/matchFlashIntensity(TrackNumber)toIntelligenceProfiles
do/matchSoundWave(TrackNumber)toIntelligenceProfiles
do/matchColorSpectrum(TrackNumber)toIntelligenceProfiles
TrackNumber=TrackNumber+1
tm(TrackTimer)
3
[TrackNumber≤999]
C
[TrackNumber>999]
/Set:TrackNumber=000

Battle Management Pattern Diagram
Detect Feature
Broker Pattern
Page 1 of 2




BM Engage Feature
BM Engage Feature Proxy
BM Broker
Weapon_1 Proxy
Weapon_1
Weapon_2 Proxy
Weapon_2
register2
register2
register1
register1
subscribe
subscribe
engage1
engage1
engage1
engage1
launch1
launch1
launch1
launch1
detach1
detach1
detach1
engage2
engage2
engage2
engage2
launch2
launch2
launch2
launch2
detach2
detach2
detach2

V. SOFTWARE ARCHITECTURE

1. *And the whole earth was of one language, and of one speech.*
2. *And it came to pass, as they journeyed from the east, that they found a plain in the land of Shinar; and they dwelt there.*
3. *And they said one to another, Go to, let us make brick, and burn them thoroughly. And they had brick for stone, and slime had they for mortar.*
4. *And they said, G to, let us build us a city and a tower, whose top may reach unto heaven; and let us make us a name, lest we be scattered abroad upon the face of the whole earth.*
5. *And the LORD came down to see the city and the tower, which the children of men builded.*
6. *And the LORD said, Behold, the people is one, and they have all one language; and this they begin to do: and now nothing will be restrained from them, which they have imagined to do.*
7. *Go to, let us go down, and there confound their language, that they may not understand one another's speech.*
8. *So the LORD scattered them abroad from thence upon the face of all the earth: and they left off to build the city.*
9. *Therefore is the name of it called Babel; because the LORD did there confound the language of all the earth: and from thence did the LORD scatter them abroad upon the face of all the earth.*

Book of Genesis – Chapter 11

Dealing with the complexity of large-scale systems is a tremendous challenge for even the most experienced software designers and developers. Large software systems contain millions of components that interact to achieve the system capabilities. The interaction of these components is far from obvious – especially true given the typical artifacts that are created for a software project. These artifacts are critical to achieve a successful system acquisition as new team members are added at different phases of the project. Even more challenging, the behavior of the components must be well understood and modified as the system evolves. One prerequisite for do this correctly is an understanding of how the software components interact as well the underlying principles of the design [6].

If we are to develop system-of-systems that exhibit the desired system behavior, then we should develop a software organizational structure that defines the behavior and characteristics of both the conceptual software components and connectors that provide the interaction between two components. Otherwise, we may introduce spurious or incorrect software components and connectors. As we have experienced time and again, this approach results in a tangled web of connectivity in which messages are passed all about and all data must remain globally visible because we have not defined messaging requirements among the components. The accidental complexity can then increase with each new added feature and enhanced interface [3].

Unfortunately, humans are ill equipped to manage complexity. Human short-term memory can typically hold between five and nine items simultaneously. Discussing the complexity of a system can be difficult when humans use imprecise language to do so.

Architecture-based development is often recommended as a technique for handling the complexity of large-scale software projects. For this thesis, we will define software architecture as the fundamental organization of a system embodied in its components, their relationships to each other and to the environment, and the principles guiding its design and evolution. Additionally, we will define an architectural view as a presentation of a particular system or part of a system from a particular perspective [5]. We will develop the conceptual view and the module view of the BMDS software components. Our objective in the conceptual and module views of the BMDS software is to understand the behavior of the conceptual components.

Conceptual View. (Please refer to Conceptual Views on pages 67-70.) The conceptual view describes the system in terms of its major design components and the relationships among the components. The conceptual view is tied most closely to the application domain. In this view, the functionality of the system is mapped to architectural elements called conceptual components with coordination and data exchanges handled by components called connectors. In the conceptual view, problems and solutions are viewed primarily in domain terms. The problems and solutions should be independent of any particular software and hardware solutions. The engineering concerns addressed by the conceptual view include the following:

· How does the BMDS fulfill the functional requirements?

· How might the BMDS functionality be partitioned?

· How are the legacy components integrated with the new components with respect to the functional requirements?

· How are product lines supported with respect to BMDS elements?

· How can we minimize impact of changes to the domain with respect to operational aspects such as performance, information transfer, availability, fault tolerance, safety, effectiveness, new systems, etc.?

From the domain analysis, we developed a high-level configuration of the BMDS (see Level One Conceptual View). We developed conceptual components and connectors to depict the BMDS system-of-systems software conceptual organization as defined in the BMDS BMC2 statechart (i.e., Detect, Track, Assign Weapon, Engage, and Assess Kill). Note that we have identified the connectors between the conceptual components as either a data component or a control component. Because the eventual realization of this architecture might well be on numerous hardware platforms, our initial efforts will be to maintain separation by identifying these five main conceptual components.

In our refinement of the high-level BMDS configuration (see Level Two Conceptual View), we identified conceptual subcomponents within the five main conceptual components to include a conceptual subcomponent to accept external sensor data. These sub-components are cohesive in nature and are connected to each other with data interfaces that reduce the level of coupling in the system. Given that components will continually provide a visual display to the user, we know that the system will require a graphical user interface (GUI). We developed this as a layer in the Module View rather than develop a large number of conceptual subcomponents to handle the user display. The rationale for this decision was to isolate the GUI from the Domain Logic so that these two layers can be developed independently of each other. The data Domain Logic will pass information to the GUI via the interface between the Presentation Layer and the Domain Logic Layer.

Let us follow the logic in the Level Two Conceptual View to ensure that we were consistent and complete with respect to the domain analysis by using a systematic manual analysis technique. Sensor data is accepted by the conceptual Detect component. The data is processed through the initial filters into a track file which is sent to the user display and the conceptual Track component. In the conceptual Track component, the track data in the track file is further processed to correlate the track data to existing tracks as well as to identify and classify the track. Given this data processing, the conceptual Track component develops the launch point prediction of the threat missile and the predicted impact point of the threat missile. This information is provided to the user display as well as the conceptual Assign Weapon component. In the conceptual Assign Weapon component, the track file is evaluated to determine whether the threat missile will impact within the area that contains the predefined defended assets list. If true, then the conceptual Assign Weapon component matches a weapons platform to the threat missile. This information is provided to the user display as well as the conceptual Engage component. In the conceptual Engage component, the weapons platforms releases the interceptor and provides corrections to the interceptor. As the interceptor approaches the threat missile, flight control is transferred from the BMDS BMC2 to the interceptor which engages in the endgame negation. This information is provided to the user display as well as the conceptual Assess Kill component. In the conceptual Assess Kill component, the BMDS accepts sensor data, and compares the received data with persistent profiles and feature recognition data. If the conceptual Assess Kill component determines that the threat missile was destroyed, then this component updates the user display and drops the track from the active track processing. If the conceptual Assess Kill component determines that the threat missile was not destroyed, then the track is retained in the active processing and the BMDS repeats the cycle. The logic in the Conceptual View is consistent and complete as compared to the domain analysis.

Given that the connectors must be realized in the design, we developed the protocol diagrams for the Conceptual View components and subcomponents (reference Protocol diagrams). The importance of these diagrams is that these views tell the software designer that incoming data types, outgoing message types, and valid message exchange sequences must all be specifically defined and designed. Also, the software designer must take into account the amount of incoming data as well as the track file update sequencing and timing.

Module View. (Please refer to Module Views on pages 71 & 72.) The module view begins the shift from conceptual design towards the realization of the system. The module view is the architectural view in which application functionality, control functionality, adaptation, and mediation are mapped to modules. In the module view, the components and connectors from the conceptual view are mapped to subsystems and modules. These modules interact by invoking services declared in the associated interfaces or wait to be invoked by other modules [8].

The engineering concerns addressed by the module view include the following:

· How are the modules mapped to the conceptual BMDS software platforms?

· What support services are required?

· How can dependencies between modules be minimized?

· How can reuse of modules and subsystems be maximized?

· What techniques can be used to insulate the components from changes in other software components, software platforms, or BMDS requirements?

· How can testing be supported?

As previously mentioned, the software architecture defines the system software organization in terms of computational components and the interactions among those components. We employed a layered organization in which each layer provides services to the layer above it and acts as a client to the layer below it. We chose layers as we want to leave open the possibility of realizing layers on different platforms in different physical locations. For example, we may choose to realize the sensor services layer in the Sensor class vice incorporating the sensor services in the BMC2 class. Layering is one of the most common techniques that software designers employ to decompose a complicated software system.

By decomposing the BMDS into layers, we can reap a number of benefits:

· We can understand a single layer as a coherent whole without knowing much about the other layers.

· We can substitute layers with alternative implementations of the same basic services.

· We can minimize dependencies between layers.

· Layers can make good places for standardization.

We chose to allocate the BMDS conceptual components into five layers: Presentation, Domain Logic, Sensor Services, System Services, and Data Source. As we assigned software components and modules to the layers, we considered coupling, cohesion, and the likelihood of future changes [1]. We minimized the coupling by separating the user display and user transaction requests from the domain logic components and the sensor services. As the domain logic components are modified in the future, this will not impact the components in either the presentation layer or the sensor services layer. Additionally, we increased the cohesion of the components by separating the functions of battle management and sensing through the establishment of two layers: domain logic and sensor services.

The logic for the five layers was as follows:

1. *Presentation Layer:* This is the information presented to the BMDS users by the Domain Logic components.

2. *Domain Logic Layer:* This is the layer that contains the actual “work” components of the system.

3. *Sensor Services Layer:* We chose to separate the sensor components from the applications for two reasons: (1) decouple the sensor logic from the application logic to reduce the dependency of the applications to the sensors and to reduce the complexity of the interfaces and (2) ease of replacement of the sensors at a future date.

4. *Data Source Layer.* This layer contains persistent BMDS data storage.

5. *System Services Layer:* We chose to separate the communications component from the components in the other layers to increase the independence of applications and sensors from the communication method.

Following the development of the layers, we mapped the conceptual elements to module elements as depicted in a Mapping Conceptual Elements to Module Elements Table. For this effort, we grouped the ports and connectors into modules separate from the component modules. The intent was to isolate the interfaces from the components. Given that several modules had similar functions, we grouped several of the modules (now sub-modules) into a more general module. These general modules are used in the remainder of this effort.

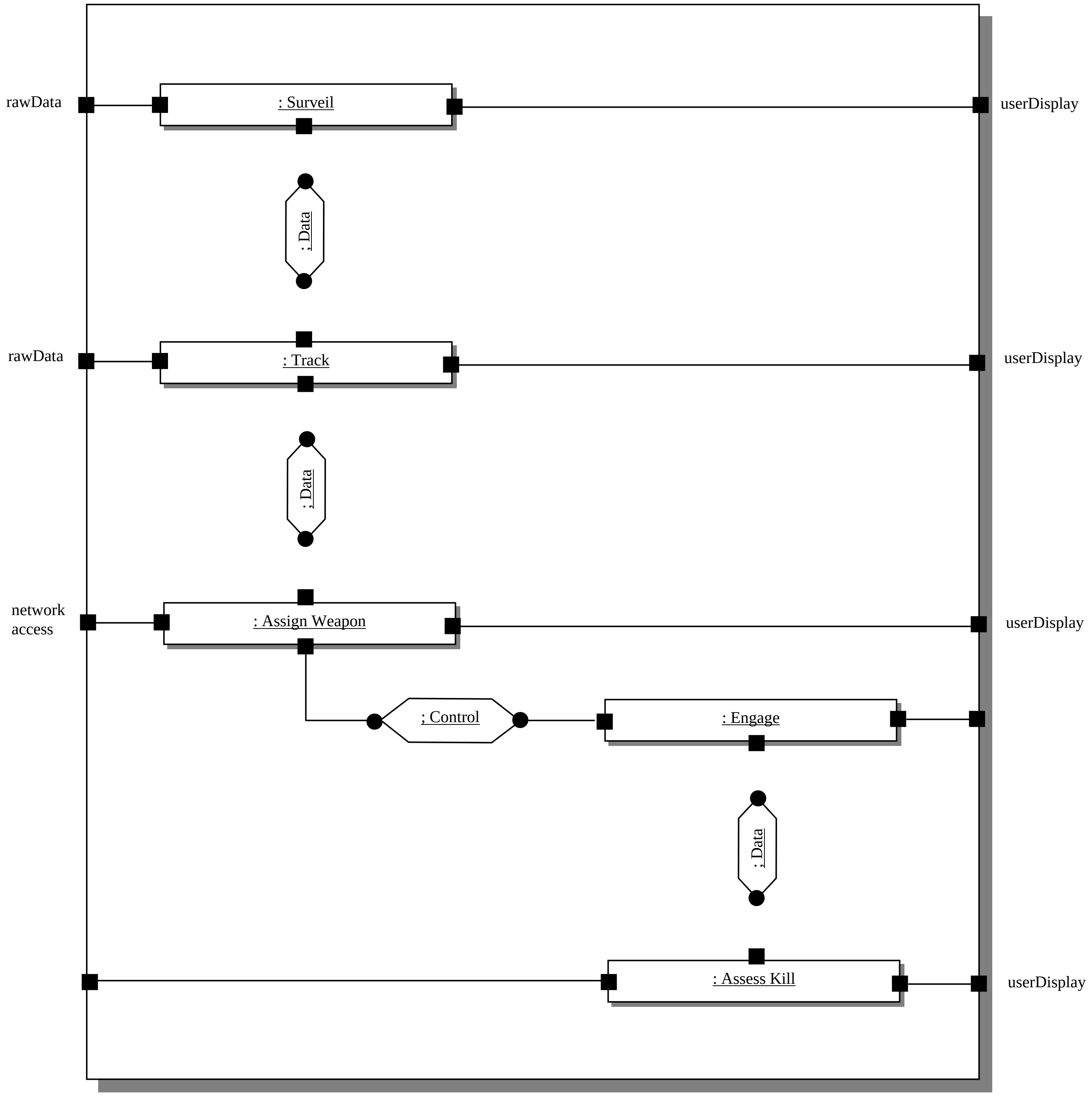
In the *Layers Connectivity* diagrams, we presented the layers and the dependencies of each layer to the others. The dependencies identified in this diagram represent the identified interfaces among the layers. Note that the System Services and Data Source layers are not visible at the functional level; however, we need: (1) communications services to transport information in the BMDS and (2) data storage and retrieval for persistent system data requirements.

Recall in Chapter IV that we discussed avoiding the creation of a BM/C2 “god class” that performs the bulk of the work in the BMDS. As we observe the conceptual components of the Domain Logic Layer, we might consider identifying three major modules in this layer: MBMDS Network module, MLocal Domain module, and MDomainLogicData Manager.

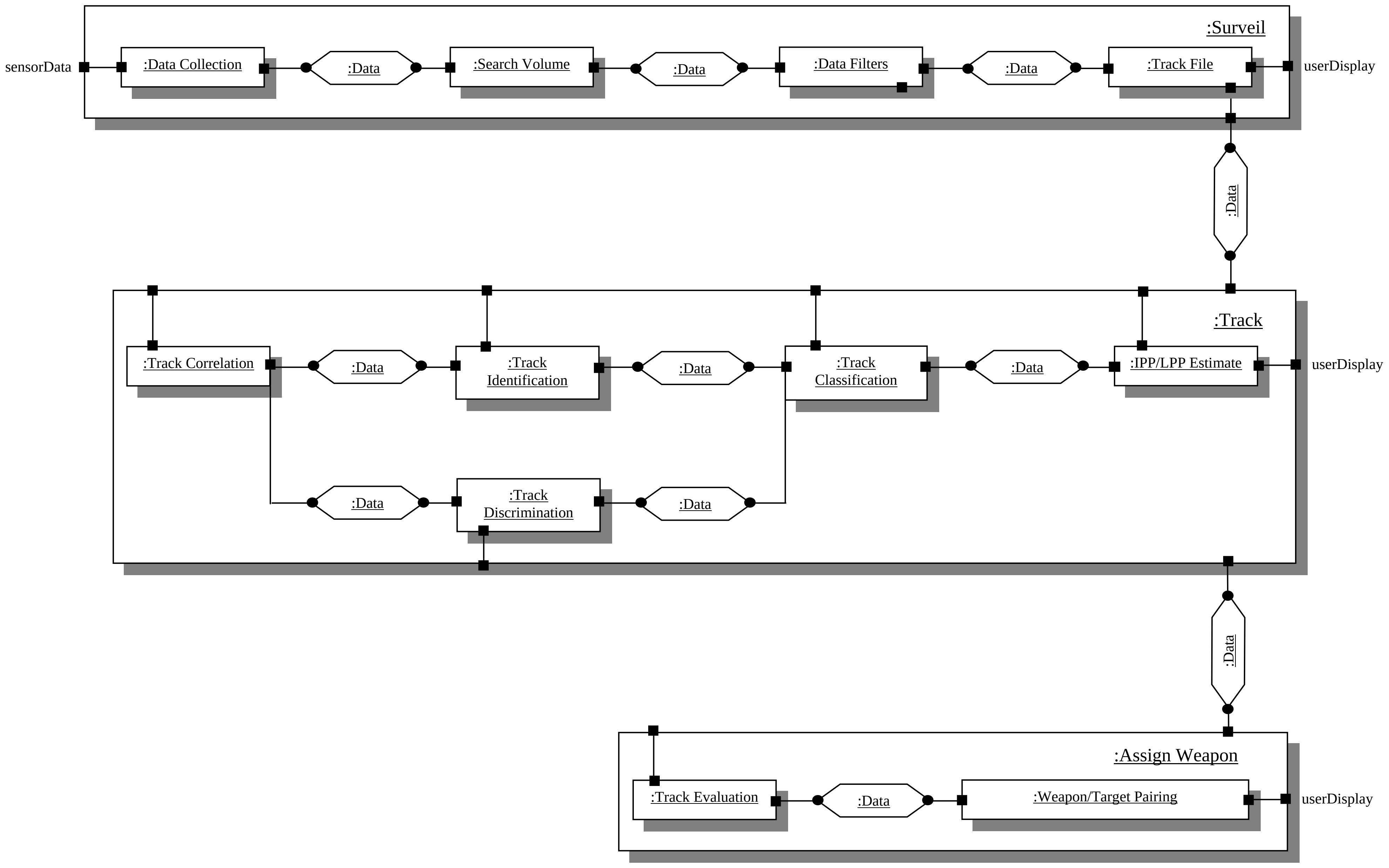
The MBMDS Network module will contain the conceptual components that are allocated to the BM/C2 elements that are networked together in the BMDS. This module will focus on sharing track information among all BMDS elements as well as upper level military management for situational awareness. The real-time information requirements are not critical in this module as its primary objective is providing situational awareness of the BMDS battlespace.

The MLocal Domain module will contain the conceptual components that are allocated to the local sensors and associated sensor ground-stations. This module will focus on local sensor tasks such as discrimination, sensor resource management, and non-organic track file integration. The real-time processing information requirements are essential to accomplish MLocal Domain's primary objective of supporting the development of providing a fire-quality track for the weapons class.

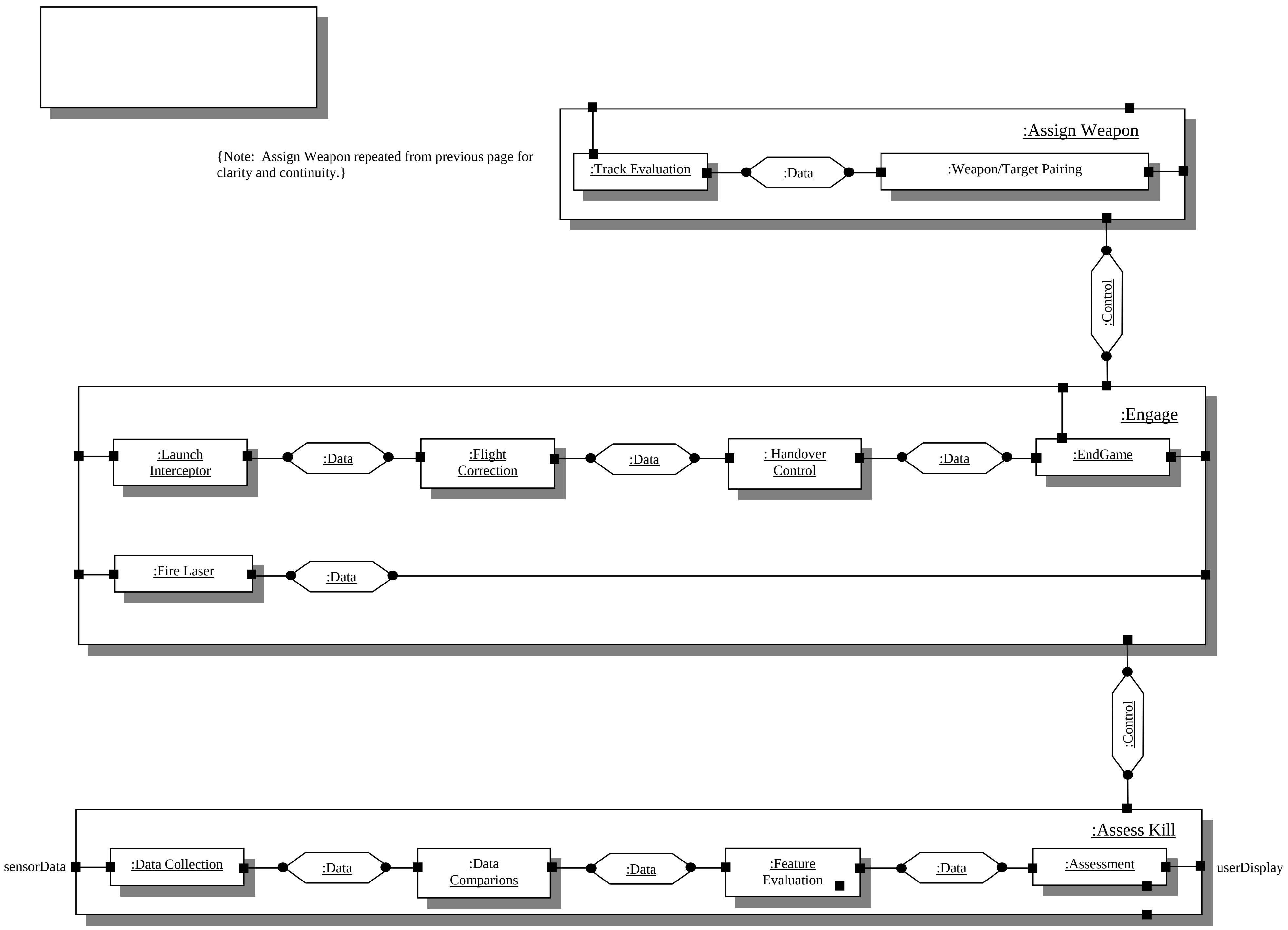
BMDS High-Level Architecture
Conceptual View
21FEB03

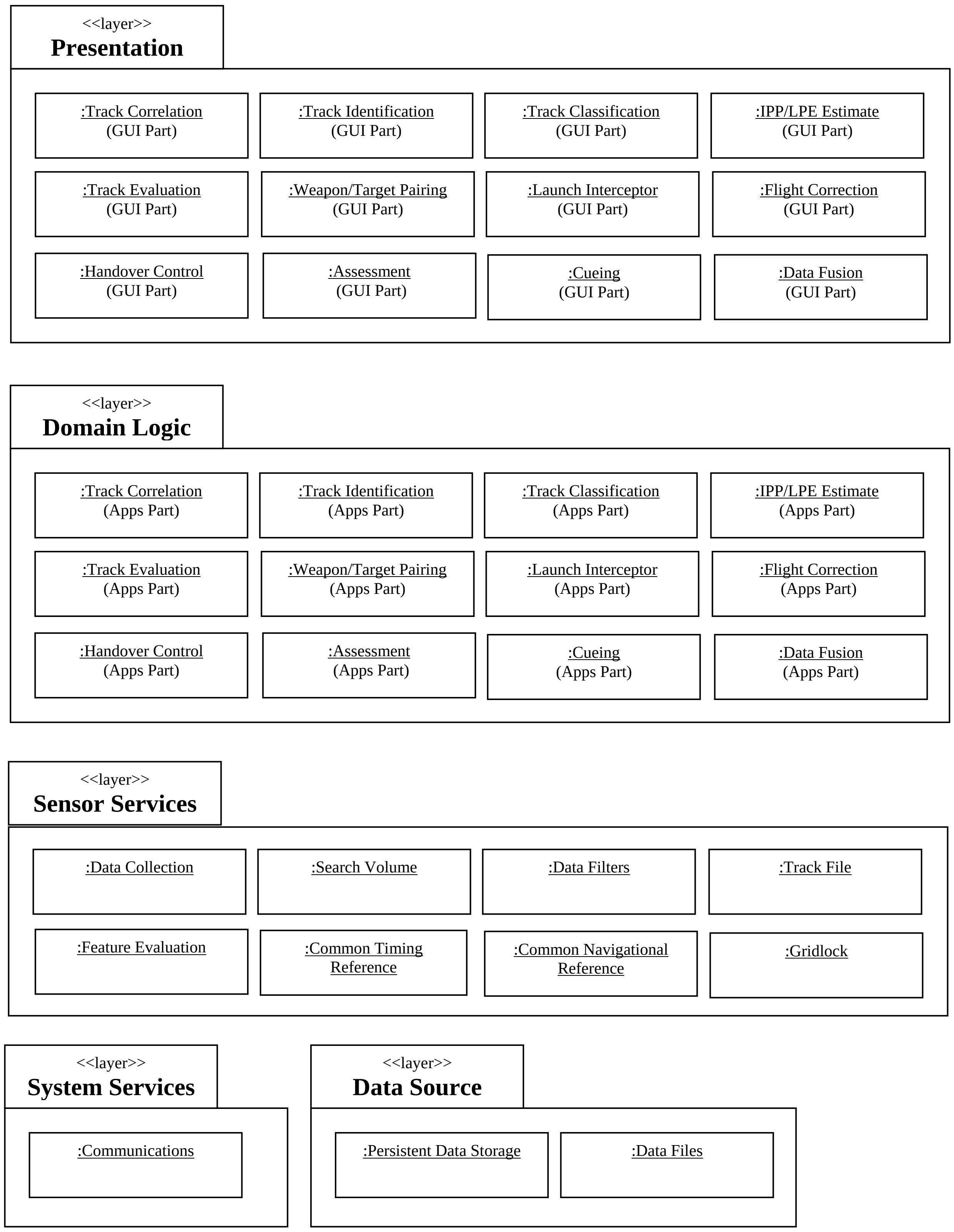


BMDS Architecture
Second Level Conceptual View
21FEB03
Page 1 of 2



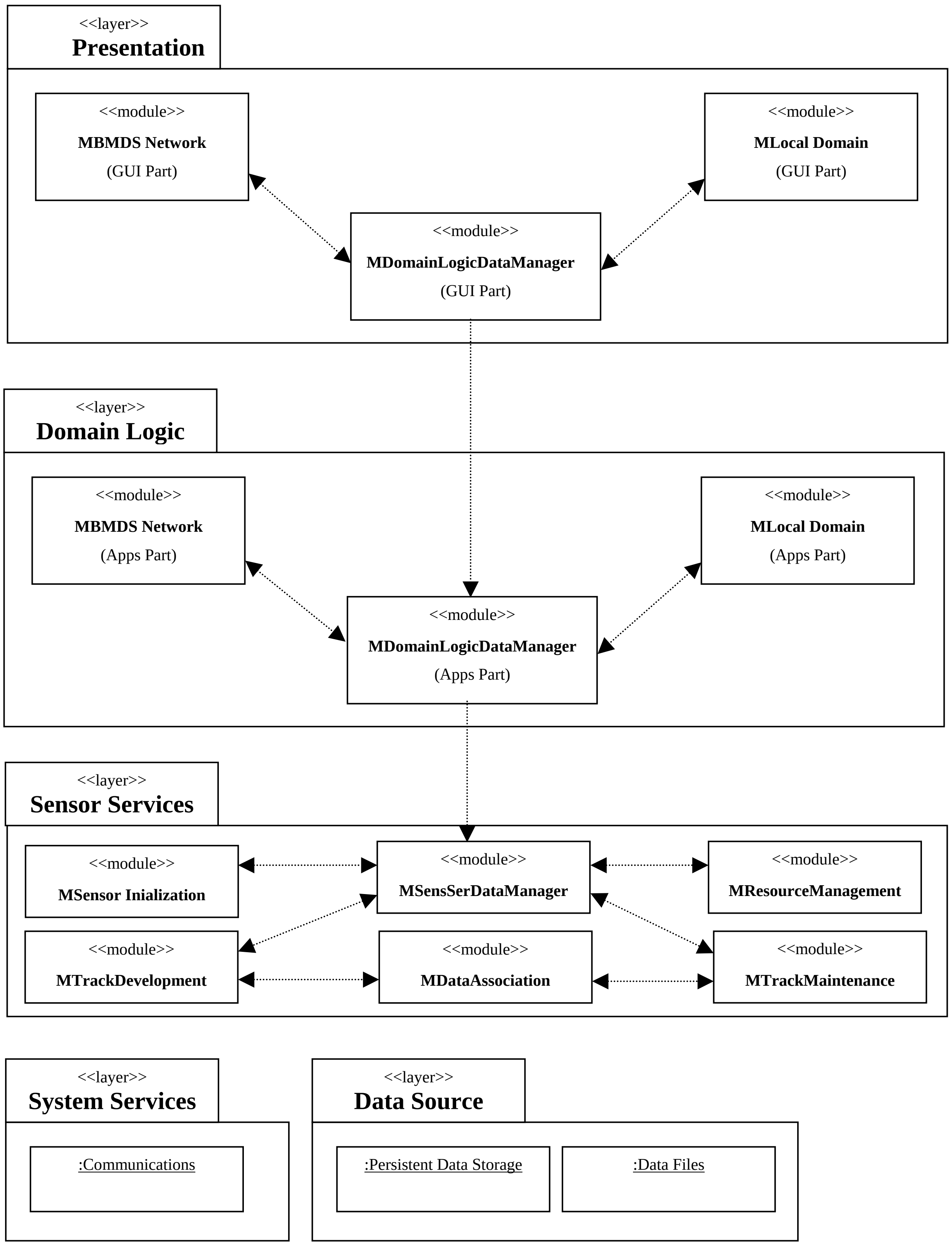
{Note: Assign Weapon repeated from previous page for clarity and continuity.}



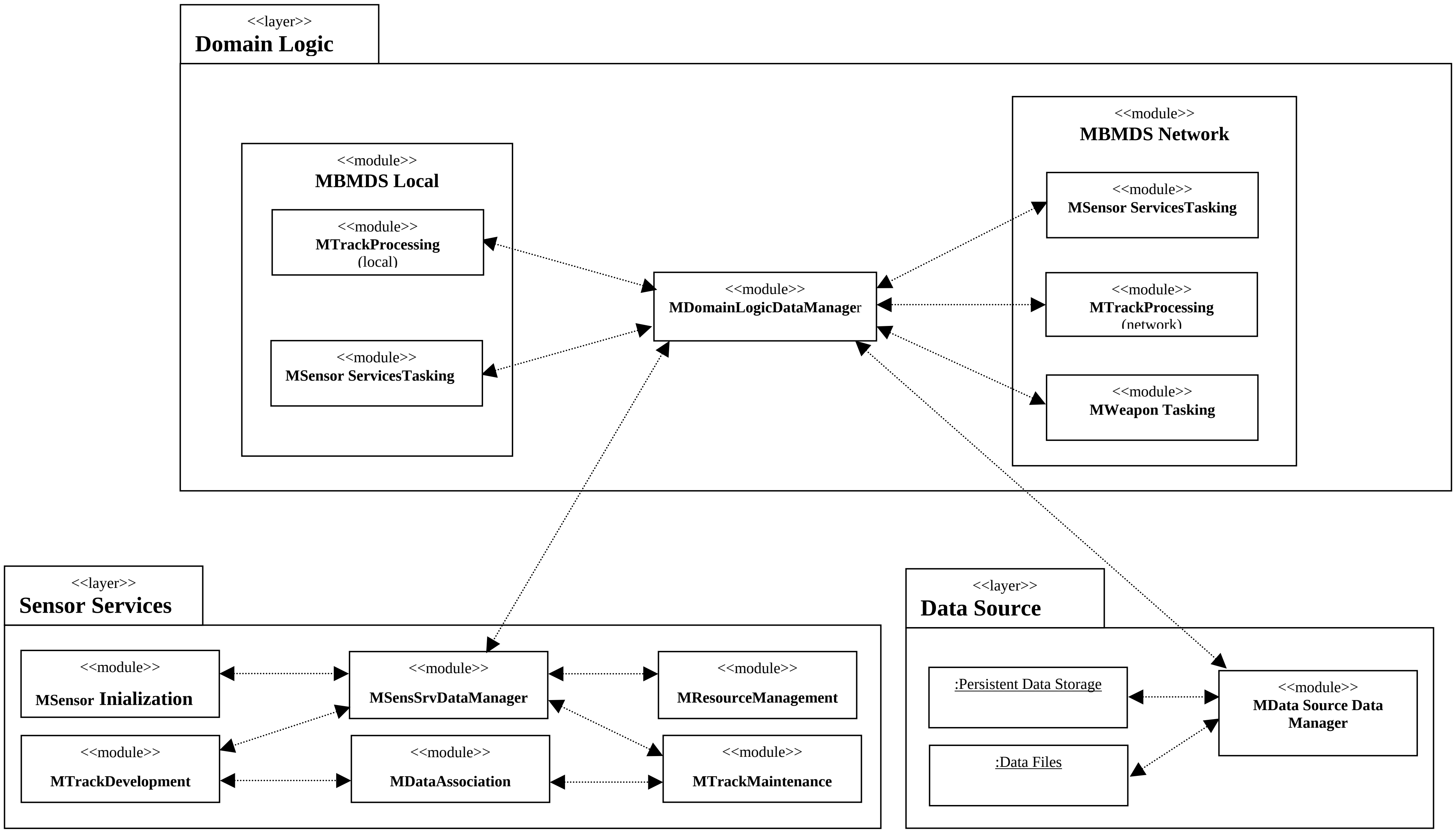


<<layer>>
Presentation
:Track Correlation (GUI Part)
:Track Identification (GUI Part)
:Track Classification (GUI Part)
:IPP/LPE Estimate (GUI Part)
:Track Evaluation (GUI Part)
:Weapon/Target Pairing (GUI Part)
:Launch Interceptor (GUI Part)
:Flight Correction (GUI Part)
:Handover Control (GUI Part)
:Assessment (GUI Part)
:Cueing (GUI Part)
:Data Fusion (GUI Part)
<<layer>>
Domain Logic
:Track Correlation (Apps Part)
:Track Identification (Apps Part)
:Track Classification (Apps Part)
:IPP/LPE Estimate (Apps Part)
:Track Evaluation (Apps Part)
:Weapon/Target Pairing (Apps Part)
:Launch Interceptor (Apps Part)
:Flight Correction (Apps Part)
:Handover Control (Apps Part)
:Assessment (Apps Part)
:Cueing (Apps Part)
:Data Fusion (Apps Part)
<<layer>>
Sensor Services
:Data Collection
:Search Volume
:Data Filters
:Track File
:Feature Evaluation
:Common Timing Reference
:Common Navigational Reference
:Gridlock
<<layer>>
System Services
:Communications
<<layer>>
Data Source
:Persistent Data Storage
:Data Files

BMDS Architecture
Layers Connectivity: Presentation to Domain Logic to Sensor Services
20FEB03



BMDS Architecture
Layers Connectivity: Domain Logic to Sensor Services & Data Source
20FEB03



VI. OBSERVATIONS AND CONCLUSIONS

Doc: Wyatt, just in time. Pull up a chair.

Wyatt: Doc, you been hittin' it awful hard, haven't you?

Doc: Nonsense. I've not yet begun to defile myself.

Wyatt: I was wondering if maybe you wouldn't wanna go on over to the Crystal Palace.

Doc: I will not be pawed at, thank you very much.

Kate: That's right. Doc can go on day and night and then some. That's my lovin' man. Have another one, my lovin' man.

Player: I'm in

Ike: Hey, lovin' man. You been called.

Doc: Ooops.

Ike: What is that now? That's 12 hands in a row, Holliday? Son of a bitch, nobody is that lucky.

Doc: Why, Ike, whatever do you mean?

Virgil: Take it easy boys.

Doc: Maybe poker's just not your game, Ike. I know, let's have a spelling contest.

-Tombstone

Based on this research, The Standish Group estimates that in 1995 American companies and government agencies will spend $81 billion for canceled software projects. These same organizations will pay an additional $59 billion for software projects that will be completed, but will exceed their original time estimates. Risk is always a factor when pushing the technology

envelope, but many of these projects were as mundane as a driver's license database, a new accounting package, or an order entry system.

On the success side, the average is only 16.2% for software projects that are completed on-time and on-budget. In the larger companies, the news is even worse: only 9% of their projects come in on-time and on-budget. And, even when these projects are completed, many are not more than a mere shadow of their original specification requirements. Projects completed by the largest American companies have only approximately 42% of the originally-proposed features and functions. Smaller companies do much better. A total of 78.4% of their software projects will get deployed with at least 74.2% of their original features and functions. [14]

The opportunities for project failure are legion. Large-scale software development efforts today are conducted in complex, distributed IT environments. Development occurs in a fragile matrix of applications, users, customer demands, laws, internal politics, budgets, and project an organizational dependencies that change constantly. …Underestimating project complexity and ignoring changing requirements are basic reasons why projects fail. Under these conditions, software project management is almost an oxymoron. [15]

In summary, this data demonstrates two things:

1. ***Requirements errors are likely to be the most common class of error.***

2. ***Requirements errors are likely to be the most expensive errors to fix.***

Given the frequency of requirements errors and multiplicative effect of the "cost to fix" factor, it's easy to predict that requirements errors will contribute the majority –often 70 percent or more – of the rework costs. And since rework typically consumes 30%-50% of a typical project budget, it follows

that requirements errors can easily consume 25%-40% of the total project budget." [10]

The above quotes are but a few of the many testimonials on our seemingly inability to acquire sound software systems. From study after study, we fail to learn the painful lessons of other software developers and follow the desperate path of shattered dreams of those that had the best of intentions of developing good software.

Much too often, we initiate coding from a reasoning about the "sticks-and-circles" diagram. During the development, we add new layers of features and functional enhancements to the system software without clear insight into the organization of the system software. Inevitably, the basic software organization that seemed so reasonable at the beginning begins to break apart under the weight of the system software revisions. Unfortunately, the software development becomes another casualty to report in future studies as to why software developments are not successful.

A good system model is an important basis for system development decisions. Conversely, we cannot expect to make sound system development decisions armed with a poor or nonexistent system model. Using various artifacts from the UML, we will develop our system-of-systems model because humans tend to grasp and understand graphical representations easier than written descriptions. This phenomenon becomes truer as the complexity of a system increases.

While a software architecture will not guarantee that a system meets its requirements, a poorly designed or ill-defined software architecture makes it nearly impossible for the software developers to realize a system that meets its requirements [8]. Typically, the system architecture is little more than a "sticks-and-circles" diagram in which the circles represent the various systems in the system-of-systems and the sticks represent the communication links among the systems. More often than not, this type of architectural view represents the totality of a system architectural effort in defense organizations. The circles of the sticks-and-circles diagram do not define the behavior of the systems and the sticks reveal nothing of the connectors that these lines represent. Software engineers cannot make effective design decisions that will result in successful acquisitions from this very limited system view.

The greatest source of system software faults will occur in the integration of the various systems. With respect to our case study, the missile defense systems will be a complex product that will contain many discrete software packages within each system. As a rule, these software packages will be developed independent of each other and programmed in many different languages. Additionally, the missile defense system will include legacy systems that are currently in operation. The means of integrating these elements and legacy systems are intricate tactical data links that support the message transfer within the system-of-systems.

As outlined in this thesis, we applied the UML and OOD techniques to the system-of-systems requirements analysis as a means for reasoning about the very complex BMDS development. Rather than disparate reasoning about the individual systems in the BMDS system-of-systems construct, we developed a system-of-systems model for reasoning about the system-of-systems as a single, functional entity.

The object-oriented paradigm offers a new system-of-systems requirements and design methodology that can minimize accidental complexity and control essential complexity through the object-oriented concepts of decentralized control flow, minimal messaging between classes, implicit case analysis, and information-hiding mechanisms. While the missile defense system will not be a pure object-oriented design, we can incorporate many of the principles of object-oriented technology to decrease the complexity of the system-of-systems. We believe that software engineers of system-of-systems can use this object-oriented paradigm to produce a sound design for the system-of-systems rather than the traditional federation of systems through a highly coupled communication medium.

In our approach, we developed a domain analysis at the top level of abstraction and worked downward into the details of the missile defense system. We developed use cases to help us understand the goals and functionality of the missile defense system. We developed other artifacts to capture system behavior from various perspectives. During the BMDS domain analysis, we identified issues that surface for future architectural and design considerations.

As a means of dividing the problem into manageable pieces, we viewed the missile defense problem via the functional requirements. We used the ballistic missile kill chain to describe what functions that the BMDS must perform. We developed the domain analysis by developing an abstract class diagram and developed various artifacts that provided insight into the BMDS behavior. We observed the following in the domain analysis:

- We identified five major functions that compose the missile defense kill chain: Detect, Track, Assign Weapon, Engage, and Assess Kill.
- The BM/C2 will control the BMDS messaging and activities.
- We can identify specific messaging requirements among the classes.
- We should employ data hiding techniques to decrease the coupling issue.
- The sensor class will continuously transmit data to the BM/C2.
- The sweep rates of the sensors are independent of the update rate from the sensors to the BM/C2.
- Given that the Sensor superclass will result in various Sensor subclasses that continually transmit data to the BMDS BMC2, the BMDS software designers will need to develop track fusion algorithms.
- The BM/C2 will require concurrent processing activities.
- Incoming track data will either update an existing track file or initiate a new track file.
- The BM/C2 must confirm a threat missile kill before dropping the track.
- Given that sensors, BM/C2 components, and weapons may enter and leave the network, we should consider the broker pattern for the subscription of services into the network.

Additionally, we developed additional questions that must be addressed prior to the system design:

- What will be the update rate from the sensor class to the BMDS BMC2?

- What will be the maximum number of objects that the BMDS BMC2 must track?

- What is the range in number of sensors that will feed into the BMDS BMC2?

- Will the BMDS battle management be automated or manual?

- Will the BMDS battle management be centralized or decentralized?

- What BMDS battle management overrides are required to prevent undesired interceptor launches?

- What BMDS battle management software interlocks are required to minimize the occurrence of an inadvertent launch?

- With respect to dynamically extending missile defense coverage, what should the BMDS BMC2 do if a weapons platform is either negated or its magazine is empty?

- With respect to threat identification and classification, what should the BMDS BMC2 do if a new track does not match the profiles in the persistent data?

- Consider the engage function. What will be the trigger to cease track refinement and to authorize the launch of an interceptor?

- Consider debris clouds formed as a result of previously negated threat missiles. What is the impact of the debris to the sensors and reporting of threat objects?

- Consider a track lost in a debris cloud. Should the BMC2 propagate the track with predictive information and continue to report the track or should the BMC2 issue a drop track message and notify potentially impacted defended assets of the lost track?

- Consider a forward-based sensor that reports to the battle management function. What are the timing requirements (i.e. latency) for the track reports from the sensor to the battle management function?

· Consider weapons that use remote tracks for firing solutions. What are the timing requirements (i.e. latency) and accuracy requirements (i.e. fire quality track) for the weapon?

· Consider establishing the correlation and fusion requirements. What is the solution for a common timing source and a common geodetic reference source for the BMDS classes?

As we continued to develop our conceptual framework, we initiated the development of the software architecture by constructing the conceptual and module views of the BMDS. From a functional perspective, we derived conceptual subcomponents for each of the five major conceptual components of the BMDS software architecture. In the module view, we chose to allocate the BMDS conceptual components into five layers: Presentation, Domain Logic, Sensor Services, System Services, and Data Source. By decomposing the BMDS into layers, we reaped the following benefits:

· We can understand a single layer as a coherent whole without knowing much about the other layers.

· We can substitute layers with alternative implementations of the same basic services.

· We can minimize dependencies between layers.

Future Research Considerations. The work presented in this thesis can be expanded in several areas such as the following:

· Is this approach applicable to other systems-of-systems?

· Is this approach sufficiently general to transcend UML? That is, could software engineers use other modeling languages for this approach?

· Can we develop automated tools to support detailed requirements elicitation and conceptual architecture validation?

· Can we develop a cost/benefit analysis from a software engineering perspective to contrast the approach of continued software maintenance as frequently practiced against the refactoring of system-of-systems software using the techniques outlined in this thesis?

· Can we use formal methods (e.g., assertions) in the development of component, module, and layer interfaces to increase the confidence in displaying the desired behaviors of the system-of-systems? If true, could we develop a system-of-systems test methodology for the operational system-of-systems to evaluate proposed software enhancements?

Conclusion. Based on the results of the analysis from our missile defense case study, we believe that system-of-systems software engineers can develop a conceptual framework that will serve as a sound basis for system-of-systems development. We can apply many of the accepted software engineering practices for single system developments to the more complex problem of system-of-systems development. We can develop various artifacts that help software engineers understand the system-of-systems behavior rather than depending on voluminous system requirements specifications of the written word. We can develop an abstract framework from which we can reason about the system-of-systems. We can develop a conceptual software architecture that describes a logical organization of proposed software modules. We believe that software engineers can apply the conceptual framework techniques described in this document to system-of-systems acquisitions with the objective of reducing accidental complexity and identifying essential complexity.

If we choose to develop such a conceptual framework for our system-of-systems development, then we can hope to improve on the dismal record of our software developments. More importantly, we can hope to provide more useful software products to our customers while reducing the time and cost to develop our products. As in all human endeavors of any consequence, keen insight, careful reasoning, and deliberate planning are the keys to a successful outcome. In the absence of these activities, we are surely doomed to failure.

Now the general who wins a battle makes many calculations in his temple ere the battle is fought. The general who loses a battle makes but few calculations beforehand. Thus do many calculations lead to victory, and few calculations to defeat: how much *more no calculation at all! It is by attention to this point that I can foresee who is likely to win or lose.*

- Sun Tzu on the Art of War

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

[1] Bachman, Felix, et al, “Software Architecture Documentation in Practice: Documenting Architectural Layers,” Carnegie Mellon Software Engineering Institute, March 2000.

[2] Caffall, Dale Scott and Michael, J. Bret, “System-of-Systems Design From An Object-Oriented Paradigm,” Monterey 2002 Workshop Proceedings: Radical Innovations of Software and Systems Engineering in the Future, October 7-11, 2002.

[3] Constantine, Larry L., *The Peopleware Papers: Notes on the Human Side of Software*, Upper Saddle River, New Jersey, Prentice-Hall, 2001.

[4] Darcy, David P. and Kemerer, Chris F., “Software Complexity: Toward a Unified Theory of Coupling and Cohesion,” Friday Workshops, Management Information Systems Research Center, Carlson School of Management, University of Minnesota, February 8, 2002.

[5] Fowler, Martin, *Patterns of Enterprise Application Architecture*, Boston, Massachusetts, Addison-Wesley, 2003.

[6] Garland, J. and Anthony, R. Large-*Scale Software Architecture: A Practical Guide to Using UML*, New York: John Wiley & Sons, Ltd., 2002.

[7] Greenfield, Michael A., “Mission Success Starts With Safety,” presentation to 19th International System Safety Conference, Huntsville, Alabama, September 11, 2001.

[8] Hofemeister, Christine; Nord, Robert; and Soni, Delip; *Applied Software Architecture*, Boston, Massachusetts, Addision-Wesley, 2000.

[9] Knutson, Charles and Carmichael, Sam, “Safety First: Avoiding Software Mishaps,” Embedded.com, http://www.embedded.com/2000/0011/0011feat1.htm, November 2001.

[10] Leffingwell, Dean and Widrig, Don, *Managing Software Requirements – A Unified Approach*, Boston, Massachusetts, Addison-Wesley 2000.

[11] Lesishman, Theron R. and Cook, David A., “Requirements Risks Can Drown Software Projects,” CrossTalk, Volume 15, Number 4, April 2002.

[12] Riel, Arthur J., *Object-Oriented Design Heuristics*, Reading, Massachusetts, Addison-Wesley, 1996.

[13] Shaw, Mary and Garlan, David, *Software Architecture: Perspectives on an Emerging Discipline*, Upper Saddle River, New Jersey, Prentice-Hall, 1996.

[14] Standish Group, “CHAOS,” The Standish Group, 1995.

[15] Standish Group, “CHAOS: A Recipe for Success,” The Standish Group International, 1999.

[16] Storey, Neil, *Safety-Critical Computer Systems*, New York, Addison-Wesley 1996.

[17] Weber, Matthias and Weisbrod, Joachim, “Requirements Engineering in Automotive Development: Experiences and Challenges,” IEEE Software, Volume 20, Number 1, January/February 2003.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Fort Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Bret Michael
   Naval Postgraduate School
   Monterey, California

4. Man-Tak Shing
   Naval Postgraduate School
   Monterey, California

5. Richard Riehle
   Naval Postgraduate School
   Monterey, California

6. Drew Hamilton
   Auburn University
   Auburn, Alabama

7. COL Kevin Greaney
   Missile Defense Agency
   Washington, D.C.