NPS-CS-05-007

NAVAL
POSTGRADUATE
SCHOOL

MONTEREY, CALIFORNIA

Formal Specification and Run-time Monitoring within the Ballistic Missile Defense Project

by

Dale S. Caffall, Thomas Cook, Doron Drusinsky, James Bret Michael,
Man-Tak Shing and Nicholas Sklavounos

August 2005

Prepared for: Missile Defense Agency
7100 Defense Pentagon
Washington, D.C. 20301-7100

This report was prepared for Missile Defense Agency and funded by the Missile Defense Agency.

This report was prepared by:

Man-Tak Shing, Associate Professor
Department of Computer Science

Reviewed by:

Peter J. Denning, Chairman and Professor
Department of Computer Science

Released by:

Leonard A. Ferrari
Associate Provost and Dean of Research

REPORT DOCUMENTATION PAGE	Form Approved OMB NO. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE 8/15/2005	3. REPORT TYPE AND DATES COVERED Technical Report

4. TITLE AND SUBTITLE Formal Specification and Run-time Monitoring within the Ballistic Missile Defense Project	5. FUNDING NUMBERS BMDO0137389189
6. AUTHOR(S) Dale S. Caffall. Thomas Cook, Doron Drusinsky, James Bret Michael, Man-Tak Shing, and Nicholas Sklavounos	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Department of Computer Science. Naval Postgraduate School 833 Dyer Road, Monterey, CA 93943-5118	8. PERFORMING ORGANIZATION REPORT NUMBER NPS-CS-05-007
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) Missile Defense Agency, 7100 Defense Pentagon Washington, DC 20301-7100	10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
The views, opinions and/or findings contained in this report are those of the author(s) and should not be construed as an official Missile Defense Agency position, policy or decision, unless so designated by other documentation.

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

13. ABSTRACT (Maximum 200 words)

This report describes the application of formal specifications and run-time monitoring within the U.S. Ballistic Missile Defense Advanced Battle Manager (ABM) project in an effort that is amongst the most comprehensive application of formal methods to a large-scale safety-critical software application ever reported. This project is unique in the following aspects: (i) formal specification assertions are being developed during the UML modeling phase, that is, before code development, (ii) all specification requirements are simulated under a variety of input and timing scenarios before being deployed, (iii) requirements are written in metric temporal logic with time-series constraints, (iv) run-time monitoring is used as the primary verification technique, and (v) temporal assertions are used for run-time recovery from formal requirement violations.

14. SUBJECT TERMS Missile Defense, Formal Specifications, Temporal Assertions, Run-time Monitoring, System-of-Systems, Safety-critical software	15. NUMBER OF PAGES 19
	16. PRICE CODE

17. SECURITY CLASSIFICATION OR REPORT **UNCLASSIFIED**	18. SECURITY CLASSIFICATION ON THIS PAGE **UNCLASSIFIED**	19. SECURITY CLASSIFICATION OF ABSTRACT **UNCLASSIFIED**	20. LIMITATION OF ABSTRACT **UL**

Formal Specification and Run-time Monitoring within the Ballistic Missile Defense Project

Dale S. Caffall[1], Thomas Cook[1,2], Doron Drusinsky[2,3], James Bret Michael[2], Man-Tak Shing[2] and Nicholas Sklavounos[4]

1. Introduction

In spite of more than a quarter of a century of research in the field, there has been limited application of formal methods to large-scale commercial and defense software projects. This report describes a formal methods-based specification and verification approach to the development of the U.S. Ballistic Missile Defense System (BMDS) battle-management (the *Battle Manager*, BM), an effort that is likely amongst the most comprehensive application of formal methods to a large-scale safety-critical software development effort ever reported[5].

Besides its scale, this effort is unique in the following aspects:

- Formal specification assertions were developed during the requirements elicitation phase before the code development and testing phases.
- Formal specification assertions, written in Metric Temporal Logic with Time-Series constraints (MTL/TLS) [CP, D1] were simulated under a variety of scenarios and real-time constraints using an assertion simulator [DS1]. Simulated scenarios were communicated to the customer (a domain expert) and to the modeling team, and were stored for later use (e.g., inside a test suite).
- Formal specification assertions were associated with UML statechart models using a heterogeneous statechart and temporal logic specification environment [DS2].
- Specification assertions span a wide variety of property categories such as interface safety and liveness assertions, capacity and throughput assertions, and statechart assertions. Run-time Verification (RV) was chosen as the primary verification method [D2].
- Temporal assertions are used for run-time recovery from formal requirement violations [DW].

This paper is organized as follows. Section 2 describes reported applications of formal methods to large commercial and government software projects. Section 3 describes software dependability issues pertaining to the U.S. Department of Defense acquisition process for a complex system-of-systems, such as the Ballistic Missile Defense Systems (BMDS). Section 4 describes the BMDS and Battle Manager (BM), and section 5 describes the BM assertion lifecycle, namely

[1] Missile Defense Agency, 7100 Defense Pentagon, Washington, D.C. 20301-7100, USA.
[2] Department of Computer Science, Naval Postgraduate School, 833 Dyer Rd., Monterey, CA 93943, USA.
[3] Time Rover, Inc. 11425 Charsan Ln, Cupertino, CA, 95014.
[4] The MITRE Corporation, 7515 Colshire Drive, McLean, VA 22102-7508.
[5] The description of the ABM provided in this paper does not necessarily reflect the architecture, technical opinions, or techniques used by of the U.S. government or any of the contractors working on the ABM project or any other project. Project details provided in this paper can be found in [Ca].

from assertion specification, via simulation of assertions and ending in deployment and monitoring. Section 6 describes tools used for specification, simulation, and monitoring within the BMDS, and is followed by conclusions.

2. Background: The Application of Formal Methods in Industry and Government

The use of formal methods has been particularly successful in the computer hardware industry [KG]. For example, Chen *et al.* at Fujitsu used the SMV model checker to debug an error in a VLSI chip responsible for high-speed switching operations. The chip has 111K gates and SMV finds a counter-example identifying the error in 50 clock cycles, which is much less than what it would take to reproduce the abnormal behavior through simulation [CY1]. Computation Tree Logic (CTL) based model checkers have also been used by Raimi *et al.* at the IBM-Motorola Somerset Design Center to debug a design error discovered during hardware testing of the PowerPC 620 microprocessor [RL]. Asgaard *et al.* at Intel Corporation's Strategic CAD Labs used a combination of symbolic trajectory evaluation model checking and lightweight theorem proving techniques to verify the correctness of an instruction-length marker (IM), a large, complex (12K gates and 1100 latches) circuit that detects and marks the boundaries between Intel architecture (IA-32) instructions, against an implementation-independent specification of IA-32 Instruction lengths [AJ]. They discovered a total of eight previously unknown IM errors, four of which were considered by the chip design team as defects that were difficult to find and diagnose with traditional validation techniques. Choi *et al.* at Samsung Electronics used the SMV model checker to verify the RTL implementation of the functional modules in an embedded system-on-a-chip (SOC) composed of the ARM920T processor core and sixteen functional modules [CY2]. Besides finding many environment modeling errors and errors in describing the properties to be verified, they also discovered a few but crucial design errors.

NASA was an early leader of using formal methods in the aerospace industry. There are many examples of using formal methods to design and verify mission-critical software in the 1990s [BC, CD]. In more recent years, the NASA Langley Formal Methods Team applied model checking to verify the presence of the safety properties in the requirements model using the NuSMV Model Checker [TM]. They found many errors in the original English statement of requirements and several errors in the model itself. Glück and Holzman at NASA Jet Propulsion Laboratory used a model extractor (called FEAVER) to create the top-level system models of two modules of the legacy flight software of NASA's Deep Space One mission (one containing 5166 lines of C code and the other containing 1894 lines of C code) to make direct calls to the source code, and applied the SPIN Model Checker to verify the correctness of the software [GH]. In addition to detecting a known error of the launch version of the software, the model checker also discovered a rare race condition that could cause the sequencing module to fail. In another experiment conducted by NASA and Time Rover, Inc., run-time verification was used to verify flight code for the Deep Impact mission [DW] and the Martian Rover software [BD].

Model checking has also been used by the Lucent Technologies to analyze their CDMA call-processing library [CG]. The company created a testing infrastructure using VeriSoft to test the call-processing software by systematically exploring the state space of the model of the wireless network. The more than 1500 runs of the testing infrastructure uncovered several newly found defects in the call-processing software.

Despite these successes, formal methods have always been treated as a supplemental means for quality assurance and have not been an integral part of any large-scale safety-critical software development.

3. Background: Unpredictability of Previous System-of-Systems Acquisition

Acquisition of system-of-systems by the U.S. Government raises a multitude of issues that go beyond the acquisition of a single system, such as the system behavior of the system-of-systems as well as each system within the system-of-systems. We define a system-of-systems as a federation of legacy systems and developing systems that provide an enhanced capability greater than that of any of the individual systems within the system-of-systems. The individual systems making up of a system-of-systems are often both developed for a different context and subjected to a different set of constraints than that of the system-of-systems. For instance, the AEGIS weapon system was originally intended to provide area defense for naval battle groups. When integrated into the BMDS, it is expected to interoperate with other systems within BMDS and in contexts of use that were not envisioned when AEGIS was developed in the 1970s.

As examples to the potential impact of the U.S. Government's current inability to confidently predict acquisition-based system behavior and continued failings in system-of-systems acquisitions, the following examples are offered:

US Central Command (CENTCOM) forces deployed six PATRIOT batteries in the Dhahran area of operations during the Persian Gulf War of 1991. Of those six PATRIOT batteries, CENTCOM forces assigned Alpha Battery the mission of protecting the Dhahran air base. Alpha Battery had been in continuous operations for over one hundred hours on February 25, 1991. Iraqi forces launched a Scud missile at the Dhahran air base that Alpha Battery failed to track and intercept. The Scud missile impacted at an US Army barracks and killed twenty-eight US soldiers. Subsequent investigations into this catastrophe revealed that PATRIOT could not perform sustained operations beyond twenty continuous hours as potential targets would fall outside the range gate – an electronic detection system within the PATRIOT radar that calculates the area in the field of regard where PATRIOT should next look for the threat missile. At one hundred hours of continuous operation, the shift in the range gate would be 687 meters so the PATRIOT could not detect, track, and destroy incoming ballistic missiles [Ga, Pa]. This example is an instance of *insufficient requirements specification and verification.*

From a study of 387 software errors discovered during the integration and testing phase of the Voyager and Galileo spacecraft, Lutz [Lu] observed that the safety-related, functional faults Voyager could be categorized as follows: 50% as behavioral faults, 31% as conditional faults, and 19% as operating faults. For Galileo, the safety-related, functional faults could be categorized as follows: 38% as behavioral faults, 18% as conditional faults, and 44% as operating faults. The author concluded that the primary cause of safety-related, functional faults (62% on Voyager and 79% on Galileo) was due to requirements that had not been identified by the developers. This example is an instance of *incomplete specification of desired system behavior and limited fault tolerance.*

These difficulties are by no means unique to the U.S. Government's acquisition-based system. Wallace and Kuhn [WK] analyzed software faults from 342 medical systems and determined that 43% of the software faults were logic-related errors such as incorrect logic in requirements specifications, unexpected behavior of multiple conditions occurring simultaneously, and improper limits. In addition, they attributed 24% of the software faults to calculation errors to include incorrect limits and ranges of computational variables as well as incorrect implementations of mathematical expressions [WK]. This example is an instance of *logic errors*. Wallace and Kuhn further suggested that software engineers should consider formal methods for highly-complex systems with emphasis on pre- and post-conditions as well as the interaction of system functions.

4. The Ballistic Missile Defense System and Battle Manager

The Department of Defense (DoD) plans to develop a ballistic missile defense (also referred to as a "global shield") 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 is pursuing a broad range of activities in order to aggressively develop and evaluate technologies for the integration of land-, sea-, air-, and space-based platforms to counter ballistic missiles. In parallel, sensor suites, battle management, and command and control will be developed to form the backbone of the BMDS.

The BMDS battle-management (BM) software is a real-time set of system functionality that addresses warfighter usage. Key characteristics of the BM will include the following: (1) a globally-distributed network, (2) an operational battlespace that includes land, sea, air, and space, (3) capability to address multiple targets that can threaten a specific theater of operations or region of the world, (4) management of concurrent battlespace activities, (5) some level of automated decision making regarding the release or hold of lethal weapons, and (6) stringent requirements for high levels of trustworthiness of the systems that provide BMD capabilities due to the fact that the threats to be encountered consist of weapons of mass destruction (WMD). Item number six makes unpredictable system behavior untenable from the public-policy, functional, and safety perspectives.

To achieve the level of desired predictable battle-management behavior, the BM contains a formal representation that captures the desired BM system behavior and is used to test the formal representation against the expected battle-management properties. The BM will be deployed in a manner similar to that depicted in Fig. 1.

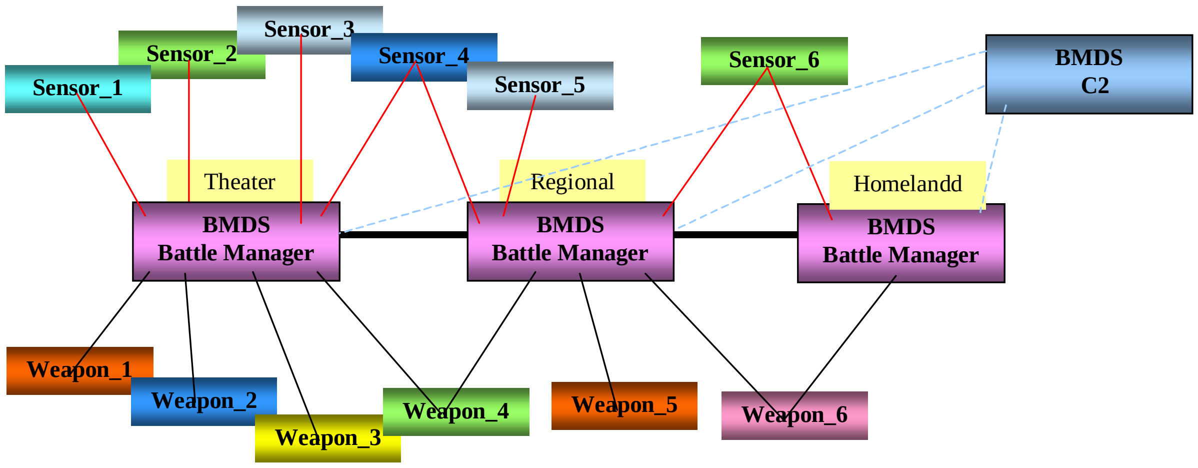

Figure 1. The BM deployment within the BMDS

The battle managers must direct the activities in the battlespace. Typically, multiple engagements are occurring concurrently in the battlespace. Oftentimes, the tasks for killing a threat object need to be executed at such a high operations tempo (OPTEMPO) that humans experience great difficulty in maintaining situational awareness of the entire battlespace.

The battle manager acts as “glueware” between software applications unique to each battle-management domain, and the sensors, Command and Control (C2) systems, and weapon systems in that domain. Battle managers must rapidly make decisions to counter both enemy actions and force movements. Battle managers must correctly cope with the fog-of-war conditions that are ever-present during the prosecution of the war. The success or failure of the battle-management functions will determine the success or failure of joint forces with respect to the achievement of their assigned objectives.

Thus, the BM software is mandated to be trustworthy, real-time and distributed. The BM computing will be accomplished through a network of computers that are connected to sensors and weapons as well as other battle-management computers. The behavior of the battle-management software cannot be predicted with confidence given the actual configuration of weapons, sensors, and battle managers at the moment of battle. Developers cannot test the battle-management software under realistic conditions prior to actual use of the software to determine the system behavior. The duration of the defense engagement will be short: it will not allow for either human intervention or debugging the software to overcome software faults at runtime.

Fig. 2 illustrates a collaboration diagram for the Track Processing (TP) component. The Track Processor abides by the following domain rules:

1. Track Processor polls the Track Data Store every five seconds, possibly resulting in a track object with information about an object in flight that is being tracked.
2. Track Processor then discriminates this track, classifying it as either a threat track or a no-kill track.
3. Track Processor then correlates threat tracks with previous instances of the same track, as detected by the same or by other sensors.

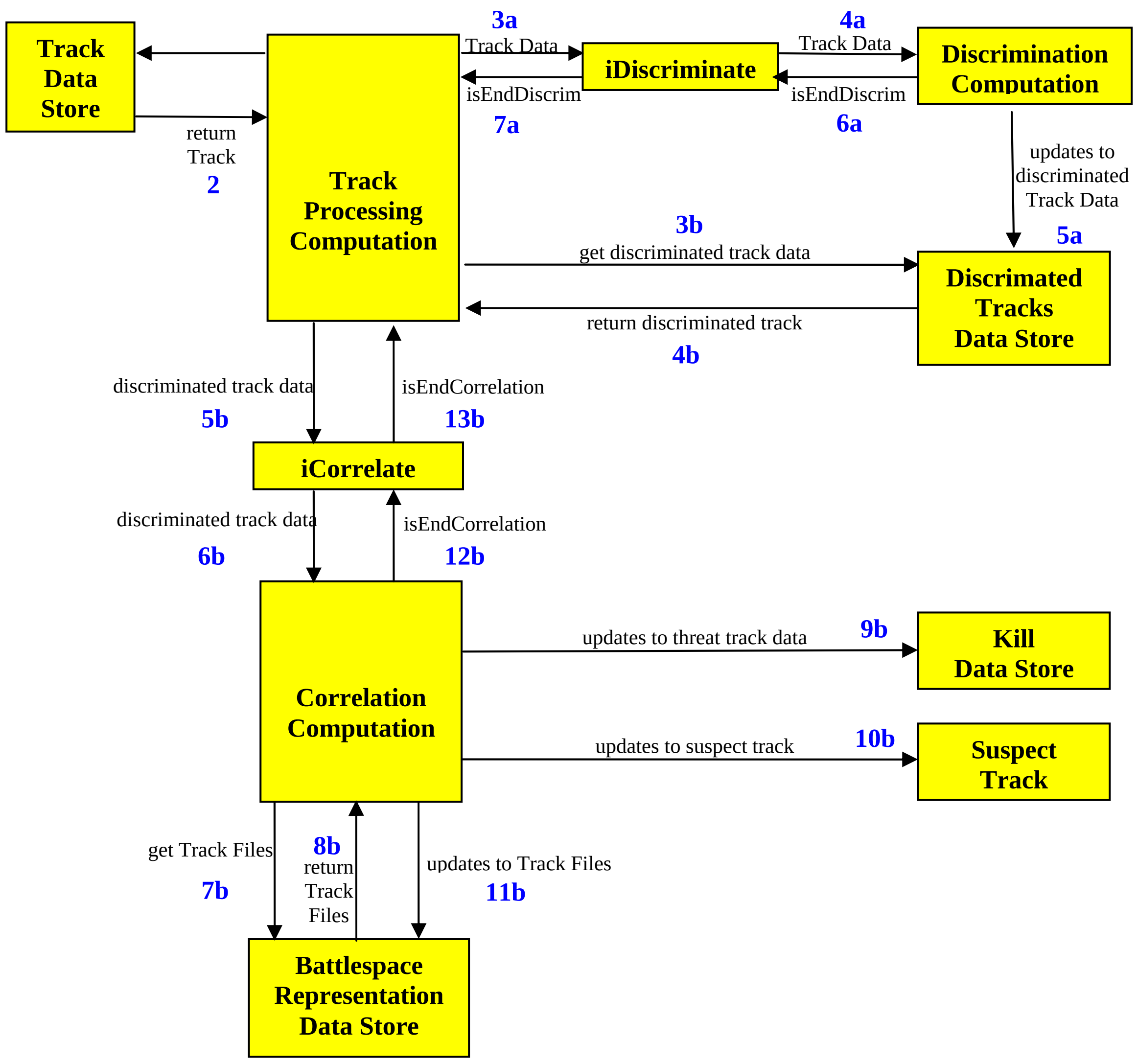

Figure 2, Track Processing Collaboration Diagram

5. ABM Assertion Lifecycle: Formal Specification, Simulation, and Monitoring

An ABM assertion lifecycle process was defined in which specifications were represented formally, refined and validated through simulation exercises, and monitored via code execution under a spectrum of test scenarios.

Formal specification assertions were developed within the early stages of the software development lifecycle. An initial set of assertions was defined as part of the requirements-elicitation phase, independent of the software architecture. Assertions were derived systematically from each of the rules of the domain, as each domain rule elicited relevant claims of safety and liveness.

During the software design phase (UML modeling), additional formal specification assertions were defined and associated with the current software architecture. Assertions were defined as relating to specific states, pertaining to an entire statechart (controller-level), or to a collaboration diagram, as discussed in the sequel. This hybrid of UML statecharts and formal specifications resulted in an extended executable model. The extended executable model was created through the use of automated code generation tools: they produced executable code for the control flow of the state-based model as well as the associated armor-plating assertion-checking code.

The objectives of the extended executable model were as follows:

1. To provide a behavioral prototype early in the design process. This prototype is used for demonstration purposes and for sanity checks. The prototype will be used in the future as a robust oracle for developers and testers.
2. To resolve, early in the design process, possible contradictions between the UML model and the formal specification assertions.
3. To provide a framework (timing model) in which to measure the system's temporal behavior under various test scenarios.

Assertions in the BM armor-plate extend executable UML-statechart models and collaboration diagrams in the following way:

1. Formal specification assertions are capable of explicitly describing *bad*, or illegal, behavior. The statechart model on the other hand tends to capture *good*, or legal, behavior, with the expectation that bad behavior will be precluded by the good behavior according to the model, but without explicitly specifying such illegal behavior. For example, the TP component performs correlation after discrimination. Bad behavior to be explicitly stated is that no correlation of a track is ever performed before discrimination is performed for that track.
2. Safety-related requirements require armor-plating to increase the trustworthiness of the software. The battle manager must ensure that its safety-related functions are correctly executed in the BMDS. In all likelihood, a ballistic-missile attack will involve multiple missiles; in other words the battle manager will be controlling the engagements of multiple, concurrent engagements. The battle-management software must provide a degree of trustworthiness that is commensurate with the critical functions of battle management. The consequences for failed software include, for example, massive civilian casualties or inadvertent release of weapons (e.g., interceptors targeted at benign objects such as commercial aircraft).
3. The succinctness of some requirements lent themselves well to be expressed in formal specification assertions rather than in UML statecharts. For example, the assertion *TP will never receive ten tracks within less than a minute* is more naturally expressed through a formal specification language than visually through a state diagram.[6]

Based on the BM prototype currently under development at the Naval Postgraduate School, it is estimated that the overall system will consist of up to two hundred assertions.

Consider the following example of requirements for the track processor of Fig. 2 and its associated recovery. The following assertions about the continuity of TP are monitored by a Safety Component (e.g., termed a "safety executive") that monitors the processes in the battle manager:

1. A track must be pulled from Track Data Store within fifteen seconds of its appearance in the Track Data Store.

[6] In this particular example we used temporal logic with counting operators [D2].

2. The first instance of a track file must appear in Battlespace Representation Data Store within thirty seconds of the first appearance of that valid track data in the Track Data Store.
3. If (1) and (2) are not true at the same time, then Safety Component will send reset signal to Track Processing Component. This means that the Track Processing failed to poll data (1) within fifteen seconds of a given set of track data appearing in the Track Data Store and Track Processing failed to discriminate and correlate the track data within thirty seconds of a given set of track data appearing in the Track Data Store. If both assertion violations occur simultaneously, then the Safety Component will reset Track Processing Component.
4. If (1) and (2) are not true at the same time following a Track Processing reset that occurred within the past sixty seconds, then the Safety Component will deem the BM to be inoperable and direct the transfer of control to another battle manager.

In our approach, BM assertions are classified as either *statechart assertions*, of correctness properties pertaining to states in a UML statechart, or *collaboration assertions*, which relate to requirements pertaining to a collaboration diagram.

Collaboration assertions pertaining to Fig. 1 fall into several possible categories, as follows:

1. Input assumptions, such as the requirement: *a single polling iteration delivers at most one track*.
2. Capacity, such as capacity requirement for the discriminator: *a new discrimination request is never received while the previous one is still being processed by the same discriminator object*. Every block in the collaboration diagram has a potential capacity assertion.
3. Causality, such as: *a track is classified as a threat only if Discriminate received an associated track two seconds beforehand*.
4. Liveness, such as: *within two seconds of a track receipt the track must be identified as either a threat or a no-kill threat, but not both*.
5. Consistency, such as: *a track for an object that already induced a track in the past must always be recognized by the discriminator as an existing track*.

BM assertions are also classified into three kinds of assertions (test-time, simulation-time and run-time) according to their intended *context of usage* [DS1]. The most typical set of assertions consists of *test-time* assertions that are intended for testing the correctness of the design and/or implementation. *Simulation-time* assertions use information about the environment not present in run-time. These are used only for and during simulation. For example, the assertion *no missile shall ever hit the Headquarters* is only useful in a simulated environment. Similarly, the following assertion can only be checked in a simulation environment: *if country C initiates a multi-threat missile launch, then BM should assign a weapon within x seconds*.

Run-time, or *deployed* assertions are assertions that can be used during run-time, in the deployed system. Run-time assertions are used to trigger run-time recovery from requirement violations described in the sequel.

The current formal specification language chosen for the BM is linear-time temporal logic with real-time constraint (MTL) and time-series constraints (TLS) described in [Pn, D1]. MTL/TLS was chosen primarily because BM related assertions contain real-time or time series constraints and the language has available commercial monitors and code generators. Future plans for BM-related assertions include the use of non-deterministic statecharts as a specification language [D3].

Specification assertions written in MTL/TLS were simulated under a wide variety of scenarios. Scenarios, which are timed sequences of conditions, vary in the sequencing order of conditions as well as in the absolute timing of conditions. The purpose of simulation is to assure no contradictions exist between three views of a specification, namely (i) the conceptual cognitive requirement in the developer's mind, (ii) the natural language formulation of that requirement, and (iii) the formal language specification of the requirement. Hence, simulation is performed with the following goals:

1. Full examination of each formal assertion under a plurality of scenarios.
2. Comparison of the behavior of an assertion's MTL/TLS formal specification against the behavior of the developer's conceptual requirement.
3. Examine the soundness and completeness of the original conceptual requirement, namely examine how the conceptual requirement behaves under a wide variety of scenarios.

Run-time Execution Monitoring (REM) is a class of methods for tracking the temporal behavior of an underlying application. REM methods range from simple print-statement logging methods to run-time tracking of complex formal requirements (e.g., written in temporal logic) for verification purposes. Recently, NASA used REM for the verification of flight code for the Deep Impact project [DW]. The majority of published REM methods use temporal logic as a specification language [D1, HR].

REM was chosen as the primary verification method for the BM because of its ability to scale, in addition to its support for assertions that include real-time and time-series constraints. In addition, REM is planned to be used: (i) as a component within an execution-based model checker [DH] and (ii) in a closed loop, for run-time execution recovery, as described below.

Run-time Execution Recovery (RER) from violations of formal requirements is a technique that integrates REM and the Monitored System (MS) in a closed loop so that the system, once notified of a formal specification violation, throws an exception and manages this exception in a pre-determined manner. In [D4] Drusinsky describes a RER technique based on the heterogeneous coupling of REM and source code Java exception handling. This method uses a two-layered approach where the REM tool manages specification and monitoring while recovery is performed using Java exception handling.

An alternative technique, currently under development for the BM, uses non-deterministic statechart specifications [D3]. This method is a single-layered approach where specification, monitoring, and recovery are all specified with one coherent language.

An example of an assertion used for recovery is: *a new discrimination request is never received while the previous one is still being processed by the same discriminator object*. The recovery strategy consists of the prioritization of discrimination requests, handling of top-priority requests while storing lower priority requests in a temporary buffer to be processed if discrimination objects become available within a minute.

6. Tools and Environment

UML statecharts are used to model dynamic artifacts of the BM system using the StateRover environment. The StateRover is a Harel Statechart editor, code generator, and visual debug animator integrated with the Temporal Rover run-time monitoring tool [D1, D2]. The StateRover generates a Java class per statechart under design. These control classes integrated with passive classes and stateless component code. Specification assertions are written in MTL and TLS, and are exercised using the DBRover simulator. After simulation, an assertion *A* is associated with a state *S* in a corresponding statechart using one of two interpretations:

1. The MTL assertion *A* must be true starting from the cycle in which *S* was reached, that is, the LTL equivalent of *Always (in S* $\rightarrow$ *A)*, where a cycle is defined as every statechart state evaluation.
2. The assertion MTL *A* must be true for all cycles statechart is in *S*, where a cycle is defined as every statechart state evaluation performed while inside state *S*.

Future tools and techniques to be introduced are:

1. A formal specification tool for non-deterministic statechart specifications [D3]. This tool enables visual, UML-like formal specifications that are as succinct as LTL.
2. REM and RER using non-deterministic statechart specifications.
3. Two methods for white-box Automatic Test Generation (ATG), which when combined with REM yield an executable model checker [DH]. The ATG methods are (i) model-based (i.e., white-box test generation based on observations taken from the UML statechart model) and (ii) specification-based (i.e., white-box test generation based on observations taken from the formal specification, similar to the method used by the ATG-Rover of [D2]).

7. Conclusion

In ballistic missile defense, the warfighter must have confidence that the battle manager will correctly complete critical battle-management functions in its operational environment regardless of the conditions in the operational environment; that is, the battle manager must be a trustworthy system and a dependable system [Mi]. We offer that the use of formal methods can increase the level of dependability and provide evidence to the warfighters to determine the level of trustworthiness of the system for operational use. Rather than discovering system behavior at the end of the development phase through intensive testing prior to fielding the completed product, we believe that developers should apply techniques that support the design and realization of desired system behavior from the earliest phases of concept development and requirements development.

We propose the use of assertions in the development of formal specifications. While formal specification assertions alone will not ensure dependable software, the use of assertions can increase the confidence of the level of dependability and trustworthiness of a system. The use of assertions can significantly reduce the errors introduced in the specifying system behavior. Assertions can considerably increase the level of clarity in the assumptions and responsibilities of system behavior, and reveal errors such as logic omissions and conflicting logic-statements. Assertions can catch common interface faults (e.g., processing out-of-range or illegal inputs) by precisely asserting the legal interface values for variables passed in through an interface.

Software developers should consider verifying the functional specifications with model checking. Model checking is not a proof of correctness; instead, model checking involves creating functional models of a system and analyzing the model against the formal representations of the desired behavior [LC]. For the battle manager, we propose to verify the functional specifications using an automated model-checking tool that can accept either developed specifications or UML statecharts as discussed in [GM], and exercise the temporal-logic assertions over a number of time cycles. Such an approach can support the identification of inconsistencies and breaks in logic through the use of the model-checking tool. From the results of the model checking, developers can correct our specifications and the artifacts from the domain analysis as required.

References

[AJ] M.D. Aagaard, R.B. Jones and C.-J.H. Seger - Combining theorem proving and trajectory evaluation in an industrial environment. In Proc. Design Automation Conf., ACM (San Francisco, Calif., June 1998), pp. 538-541.

[BC] R.W. Butler, J.L. Caldwell, V.A. Carreno, C.M. Holloway, P.S. Miner and B.L. Di-Vito - NASA Langley's research and technology-transfer program in formal methods. In Proc. 10th Annual Conf. on Computer Assurance, IEEE (Gaithersburg, Md., June 1995), pp. 135-149.

[BD] G. Brat, D. Drusinsky, D. Giannakopoulou, A. Goldberg, K. Havelund, M. Lowry, C. Pasareanu, W. Visser and R. Washington - Experimental evaluation of verification and validation tools on Martian Rover software. In J. Formal Methods in System Design 25, 2 (2004).

[Ca] D.S. Caffall - Conceptual Framework Approach for System-of-Systems Software Developments, Master's thesis, Naval Postgraduate School, Monterey, Calif., Mar. 2003.

[CD] J. Crow and B. Di Vito - Formalizing space shuttle software requirements: four case studies. In ACM Trans. Software Engineering and Methodology, 7, 3 (July 1998), pp. 296-332.

[CG] S. Chandra, P. Godefroid and C. Palm - Software model checking in practice: an industrial case study. In Proc. 24th Int. Conf. Software Engineering, IEEE (Orlando, Flor., May 2002), pp. 431-441.

[CP] E. Chang, A. Pnueli and Z. Manna - Compositional verification of real-time systems. In Proc. 9th Symp. Logic in Computer Science, IEEE (Paris, Fr., July 1994), pp. 458-465.

[CT] D. Cook, R. Theron and A. Leishman - Requirements risks can drown software projects. In CrossTalk, 15, 4 (Apr. 2002), pp. 4-8.

[CY1] B. Chen and M. Yamazaki and M. Fujita - Bug identification of a real chip design by symbolic model checking. In Proc. Eur. Conf. on Design Automation, Eur. Test Conf., (Paris, Fr., Feb. 1994), pp. 132-136.

[CY2] H. Choi, M.-K. Yim, J.Y. Lee, B.W. Yun and Y.T. Lee - Formal verification of an industrial system-on-a-chip. In Proc Int. Conf. Computer Design, IEEE (Austin, Tex., Sept. 2000), pp. 453-458.

[D1] D. Drusinsky - Monitoring temporal rules combined with time series. In W. Hunt and F. Somenzi, eds., Lecture Notes in Computer Science No. 2725, Proc. Computer Aided Verification Conf. (Bolder, Colo., July 2003), pp. 114-117.

[D2] D. Drusinsky - The Temporal Rover and the ATG Rover. In K. Havelund, J. Penix, and W. Visser, eds., Lecture Notes in Computer Science No. 1885, Proc. Spin2000 Workshop, Berlin: Springer Verlag, 2000, pp. 323-329.

[D3] D. Drusinsky - Run-time Monitoring and Recovery of Harel Statecharts using Prioritized Non-deterministic Statechart Specifications, submitted for publication.

[D4] D. Drusinsky - Specs Can Handle Exceptions. In Embedded Developers Journal, November 2001, pp. 10-14. (http://i.cmpnet.com/eet/embedsub/2001/nov/Feature1.pdf).

[DH] D. Drusinsky and K. Havelund - Execution-based model checking of interrupt-based systems. In Proc. Workshop on Model-Checking for Dependable Software-Intensive Systems, Int. Conf. Dependable Systems and Networks (San Francisco, Calif., 2003).

[DS1] D. Drusinsky, M. Shing and K. Demir - Test-time, Run-time, and Simulation-time Assertions for RSP. In Proc. 16th IEEE International Workshop on Rapid Systems Prototyping, Montreal, Canada, 8-10 June 2006, pp. 105-110.

[DS2] D. Drusinsky and M. Shing - TLCharts: Armor-plating Harel Statecharts with Temporal Logic Conditions. In Proc. 15th IEEE International Workshop in Rapid Systems Prototyping, Geneva, Switzerland, June 28-30, 2004, pp. 29-36.

[DW] D. Drusinsky and G. Watney - Applying run-time monitoring to the Deep-Impact Fault Protection Engine. In Proc. 28th NASA Goddard Software Engineering Workshop, IEEE (Dec. 2003), pp. 127-133.

[Ga] J.G. Ganssle - Disaster. In Embedded Systems Programming, 11, 5 (May 1998), pp. 113-117.

[GH] P.R Glück and G.J Holzmann - Using SPIN model checking for flight software verification. In Proc. Aerospace Conf, IEEE (Big Sky, Mont., Mar. 2002), pp. 1-105 - 1-113.

[GM] L. Gnesi and M. Massink - Using hybrid automata to support human factors analysis in a critical systems. In Proc. Fourth Int. Symposium on High Assurance Systems Engineering, IEEE (Washington, D.C., Nov. 1999), pp. 46-55.

[HR] K. Havelund and G. Rosu - Monitoring Programs using Rewriting. In Proc. 16th Annual Int. Conf. Automated Software Engineering, IEEE (San Diego, Calif., Nov. 2001), pp. 135-143.

[KG] C. Kern and M. Greenstreet - Formal verification in hardware design: A survey. In ACM Trans. Design Automation of Electronic Systems 4, 2 (Apr. 1999), pp. 123-193.

[LC] G.A. Lewis, S. Comella-Dorda, D.P. Gluch, J. Hudak and C. Weinstock - Model-based verification: Analysis guidelines. Technical Note CMU/SEI-2001-TN-028, Software Engineering Institute, Pittsburgh, Penn., Dec. 2001.

[Lu] R. Lutz - Targeting safety-related errors during software requirements analysis. In Proc. 1st ACM SIGSOFT Symposium on Foundations of Software Engineering, ACM (Redondo Beach, Calif., Dec. 1993), pp. 99-106.

[Mi] J.B. Michael - Gaining the trust of stakeholders in systems-of-systems: A brief look at the Ballistic Missile Defense System. In Proc. Center for National Software Studies Workshop on Trustworthy Software, Technical Report NPS CS-04-006, Naval Postgraduate School, Monterey, Calif., May 2004, pp. 27-29

[Pa] Patriot Missile Software Problem. Report GAO/IMTEC-92-26, U.S. General Accounting Office, Washington, D.C., Feb. 1992.

[Pn] A. Pnueli - The temporal logic of programs. In Proc. 18th Symp. Foundations of Computer Science, IEEE (Providence, R.I., Oct. 1977), pp. 46-57.

[RL] R. Raimi and J. Lear - Analyzing a PowerPC™ 620 microprocessor silicon failure using model checking. In Proc. Int. Test Conf., IEEE (Washington, D.C., Nov. 1997), pp. 964-973.

[TM] A. Tribble, S. Miller and D. Lempai - Software Safety Analysis of a Flight Guidance System, NASA/CR-2004-213005, Mar. 2004.

[WK] D.R. Wallace and D.R. Kuhn - Lessons from 342 medical device failures. In Proc. 4th Int. Symposium High Assurance Systems Engineering, IEEE (Washington, D.C., Nov. 1999), pp. 123-131.

THIS PAGE IS INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1.	Defense Technical Information Center 8725 John J. Kingman Rd., STE 0944 Ft. Belvoir, VA 22060-6218	2
2.	Dudley Knox Library, Code 52 Naval Postgraduate School Monterey, CA 93943-5100	2
3.	Research Office, Code 09 Naval Postgraduate School Monterey, CA 93943-5000	1
4.	Dr. Butch Caffall Missile Defense Agency 7100 Defense Pentagon Washington, DC 20301-7100	1
5.	Dr. James Bret Michael Computer Science Department Naval Postgraduate School Monterey, CA 93943	1
6.	Dr. Doron Drusinsky Computer Science Department Naval Postgraduate School Monterey, CA 93943	1
7.	Dr. Man-Tak Shing Computer Science Department Naval Postgraduate School Monterey, CA 93943	1
8.	LtCol Thomas Cook Computer Science Department Naval Postgraduate School Monterey, CA 93943	1