Applying Formal Methods and Object-Oricntd Analysis to Existing Flight Software"

Betty 11. C. Ch eng[†]
Michigan State University
Department of Com puter Science
A714 Wells Hall
East Lansing, MI 48824-1029
chengb@cps.msu.edu

Brent Auernheimer
California State University, Fresno
Department of Computer Science
Fresno, CA 93740-0109
brent_auernheimer@CSUFresno.edu

Abstract

Correctness is paramount for safety-critical software control systems. Critical software failures in medical radiation treatment, communications, and defense arc familiar to the public. The significant quantity of software malfunctions regularly reported to the software engineering community, the laws concerning liability, and a recent NRC Aeronautics and Space Engineering Board report additionally motivate the useofcrror-reducing and defect detection software development techniques.

The benefits of formal methods in requirements-driven software development (('forward engineering") is well documented. One advantage of rigorously engineering software is that formal notations arc precise, verifiable, and facilitate automated processing. This paper describes the application of formal methods to reverse engineering, where formal specifications arc developed for a portion of the shuttle on-orbit digital autopilot (DAP). Three objectives of the project were to: demonstrate the use of formal methods on a shu ttle application, facilitate the incorporation and validation of new requirements for the system, and verify the safety-critical properties to be exhibited by the software.

1 Introduction

Correctness is paramount for safety-critical software control systems. Critical software failures in medical radiation treatment [1], communications [2], and defense [3] arc familiar to the public. The significant quantity of software malfunctions regularly reported to the software engineering community [4], the laws concerning liability [5], and a recent NRC Aeronautics and Space Engineering Board report [6] additionally motivate the use of error-rcclucitlg and defect detection software development techniques.

The benefits of formal methods in requirements-driven software development ('(forward engineering") is well documented [7,8, 9, 10, 11, 12]. One advantage to using rigorous approaches software engineering is that formal notations arc precise, verifiable, and facilitate automated processing [13].

We claim that maintenance of critical existing ('(legacy") code also benefits from formal methods. For example, formal specifications can be reverse engineered from existing code. The resulting formal specifications arc then the basis for change requests and the foundation for subsequent verification and validation. Considering re-implemen tation's high cost and) even worse, the failure of critical software, reverse engineering of code into formal specifications provides an alternative or a supplement to traditional approaches for maintaining safety-critical systems.

This paper describes a project that applies formal methods to a portion of the shuttle on-orbit digital autopilot (DAP). Three objectives of the project were to: demonstrate the use of formal methods on a shuttle application, facilitate the incorporation and valid ation of new requirements for the system, and verify the safety-c.ritic,al properties to be exhibited by the software.

In addition to developing formal specifications of a critical module, a graphical depiction of the subsystem was constructed using the *Object Modeling Technique* (OMT) [14] to provide an object-oriented view of the system as it relates to the functional and dynamic views. Lessons learned from this project are described, including discussions of the benefits of constructing and the ability to generate proofs with the formal specifications.

The remainder of the paper is organized as follows. Section 2 gives a brief introduction to formal methods and object-orie nted development techniques. Section 3 gives an overview of the entire project, including a discussion} of the object-oriented analysis and the development of the OMT diagrams. A summary of lessons learned from this project arc discussed in Section 4. Finally, concluding remarks and future investi gati ons arc given in Section 5.

*This work was supported in part by the NASA/ASEE 1,'acuity fellowship program, where the authors' fellowships were spent at the Jet Propulsion Laboratory at the California Institute of Technology.

†This author is also supported in part by NSF grant C(:1(-9209780.

2 1 background Material

This section briefly defines and motivates the usc of formal methods. Also, the benefits of object-oriclltcd analysis and design arc presented.

2 . 1 Formal Methods

Formal methods in software development provide many benefits in the forward engineering aspect of software development [7, 8, 9, 10, 11]. For any specification, there can be any number of implementations that satisfy the specification [15].

Due to the criticality and the volume of much of the software being developed by many agencies involved in flight systems, there arc several projects incorporating formal methods into the software development process [16]. In addition, there have been recent investigations into reverse engineering that focus on the usc of rigorous mathematical methods for extracting formal specifications from existing code [17, 18, 19, 20].

A *formal method* consists of a *formal specification language* and *formally defined inference rules* [7]. The specification language is used to describe the intended system behavior and the inference rules provide a sound method for reasoning about the specifications. Using formal specifications for software design serves several general purp oses. First, it forces the designer to be thorough in the development and the documentation of a system design. Second, the developer is able to obtain precise answers to questions posed about the properties of the system, and therefore be able to rigorously test (Ly developing theorems) the design for the satisfaction of its requirements. Unfortunately, sine.c the requirements arc traditionally expressed informally, there remains a (albeit decreased) potential for errors to remain undetected. Third, the developer is able to reason about the correctness of a system or a safety - critical component of the system with respect to its specification. The latter category of reasoning can be divided into two approaches: *program verification* and *program synthesis*. Program verification is the process of checking the semantics of a program text against its specification. A program whose semantics satisfies its specification is said to be correct. Program synthesis refers to formal techniques for systematically developing a program from a specification such that the correctness of the resulting program (with respect to its specification) is inherent in the development process itself [21, 22, 23, 24].

Formal methods arc typically more difficult to apply than informal approaches and require a great deal more discipline. Furthermore, the state of the current, technology is such that verification and the usc of formal methods is largely done manually, thus requiring a tremendous effort to perform tedious, but necessary tasks. In general, the introduction of formality in software development is a difficult but valuable step in the construction) of reliable and maintainable computer systems. The difficulty is largely due to the quantity of detail required by formalization as well as the tedious process by which the formalisms must be manipulated. However, the detection and correction of design flaws, ability to usc automated tools for manipulation, elimination of ambiguity, precise documentation for maintenance, and improved reusability arc a fcw examples of the overwhelming value, and often necessary benefits, that formal methods brings to the software development process.

2.2 Object-Oriented Techniques

'J'here arc a wide variety of approaches to requirements analysis (see [25, 26] for examples), many of them in the broad category known as *object-oricnkd requirements analysis* (OOA) [27, 28, 29, 30]. An object is a data abstraction, and it is the goal of OOA to construct an abstract, object-based model of the problem domain. The 00A focus on objects is in contrast to the more traditional approach to analysis that focuses on procedures [31]. That is, instead of modeling the problem domain as a system of operations that process data objects, 00A modeling centers on a description of data objects and their interactions.

Most 00A techniques begin by a careful assessment of the natural language problem description. A simple first step in developing an 00A model is to extract the *nouns* from the problem description. Many of these nouns will share common properties and may be more easily described as instances of *types*. For example, *Galileo, Voyager*, and *Magellan* arc all spacecraft, and *Venus, Mars,* and *Mercury* arc all planets. In this context, spacecraft and planet can be considered as types, where the type of an object is called its *class. Some classes,* referred to as *subclasses,* may be specializations of other classes. For example, an interplanetary spacecraft is a specialization of the type spacecraft. As such, 00A organizes types into a class hierarchy based on a *isa (as* in "an X is *a Y")* relationship.

It may be natural to think of an object as being composed of other objects. For example, an interplanetary spacecraft may consist of numerous jets, guidance and navigation control system, and a probe to study a planet's atmosphere. This dependence introduces an additional dimension of relations into the class hierarchy, that is, a *part of* relation. The *parts of* an object arc often called its *attributes*.

The nouns of the problem description can be used to identify candidate objects (and therefore, classes), and accordingly, the verbs in the problem description can provide information on interactions between objects. Some verbs may describe a service for a particular class of objects, such as *fire* in the phrase "fire the jets". Other verbs may describe a possible state of an object, such as *coast* in the phrase "the spacecraft begins to coast." Therefore, verbs help to define the services of a class of objects, usually referred to as the *operations* or *methods* of a class, arid the computational processes of the systemas a whole (the dynamic behavior).

in the early stages of software development, includig object-oriented approaches, diagrams arc frequently used to describe requirements and guide development. For example, data flow diagrams (DFD) [25] have been widely used to visualize functional behavior of processes. Entity-relationship (E-R) diagrams [32] have been used to pictorially describe a wide variety of concepts, foremost among them is the relational data base organization.

In general, a single diagramming notation is not sufficient to capture the complex information needed to build software systems [33]. The *Object Modeling Technique* (OMT) [27] uses DFDs, hybrid E-R diagrams, and statecharts to model software requirements using object-oriented concepts. Collectively, these diagrams address properties that should be modeled, including flow of control, flow of data, patterns of dependency, time sequence, and name-space relationships. The OMT approach is appealing in its multiple views of software requirements and is fairly comprehensive in its (albeit informal) treatment of development issues. Furthermore, OMT is commonly used in industry and in academic settings.

3 Project Overview

A portion of the shuttle software was chosen for a formal methods demonstration project involving NASA'S Jet Propulsion laboratory, Johnson Space Center, and Langley Research Center [34]. This multi-NASA site project was supported as a *Research and Technology Objectives and Plans* (RTOP). A related project of a smaller scale was performed by the authors in conjunction with the larger demonstration project. The Phase. Plane module, the control system for automatic attitude control of the shuttle, was the subsystem selected for the smaller project. The criteria that led to the selection of Phase. Plane included finding a module with difficult to understand requirements and potential for critical change requests. Although the Phase. Plane module has worked correctly in thousands of hours of use (both in terms of simulation and flight), its specific properties remains obscure (at least, to the requirements analyst and software developers [35].

Three tasks were performed in the development of the formal specifications of the module's high-level requirements. First, an understanding of the original requirements was needed. This involved consulting the *Functional Subsystem Software Requirements* (FSSR) document [36] (also known as Level C requirements, consisting largely of "wiring diagrams"), *Guidance and Control Systems Training Manual* [37], source code, informal design notes, and discussions with shuttle software personnel. An "as-built" formal specification capturing the functionality depicted by the FSSR "wiring diagrams" was then developed.

Second, when attempting to derive a more abstract requirements-level formal specification, it was difficult to eliminate the implementation bias present in the as-built layer. A level of OMT diagrams were developed to depict the information from the first level of specifications. These diagrams facilitated the abstraction process and lead to the next higher level of specifications. This iterative process consisting of developing a level of formal specifications, followed by constructing the corresponding OMT diagrams lead to the identification of the high level, critical requirements of the Phase_Plane module. Example specifications and OMT diagrams are described below.

The third task involved outlining proofs between the levels of specifications developed. That is, each specification must be shown to correctly implement the more abstract specification above it. These proofs provide traceability from the implementation details as described by the "wiring diagrams" to the high level requirements.

3.1 Phase Plane

The *Reaction Control System* (RCS) Digital Autopilot system (DAP) works to hold attitude or to achieve an attitude maneuver through an error correction method, involving the control of jet firings. Figure 1 gives a high-level view of the DAP, where the State *Estimator* gives the current attitude, while taking into consideration spacecraft dynamics such as propellant usage and inertia. This information is then supplied to a component that calculates the attitude and rate errors with respect to desired values (specified by the crew).

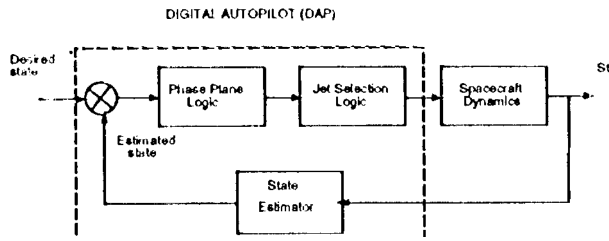

Figure 1: High-level view of DAP, including the Phase. Plane module [38]

This project focused on the Phase-Plane module, where a phase plane may be visualized as a graph plotting spacecraft rate errors against attitude errors for one rotational axis, with a "box" drawn around the center. There is a separate phase plane for each of the vehicle rotation axis (roll, pitch, and yaw). The "box" (with parabolic sides), whose limits are defined by the crew with attitude and rate deadbands, is used to determine when, if, and in what direction rates must be generated to null the errors [38]. If the shuttle is within the specified deadband limits, the rate and attitude errors are represented **by** a point plotted inside the box. If the point travels outside the box, then jets fire to return the point inside the box, thereby reducing the errors and achieving the maneuver request or maintaining the attitude hold as requested by the crew. In an attitude hold situation, the error plot actually cycles around the zero error point with jets turning off and on again each time the limits of the "box" are exceeded. This is known as "limit cycling" or "deadbanding". The phase plane generates positive or negative rate commands on an axis by axis basis, where the jet select component determines which jet(s) to fire (the topic of the RTOP project [34]). Figure 2 gives a graphical representation of the phase plane. The dashed lines outline the "box" that define the deadbanding path. The shaded regions depict the *coast regions* where the orbiter dots not need any corrective action. The remaining regions arc known as *hysteresis regions*, where external factors such as propellant usage, inertia? time lags between firing commands, and sensor noise require the calculation of corrective action to ensure that the Orbiter remains within the deadband limits. The requirements for the Phas e. Plane module arc described in terms of a '(wiring" diagram (scc Figure 3 [36]), indicating the input and output values, and several tables describing the calculation for the boundaries of the phase plane and its different regions.

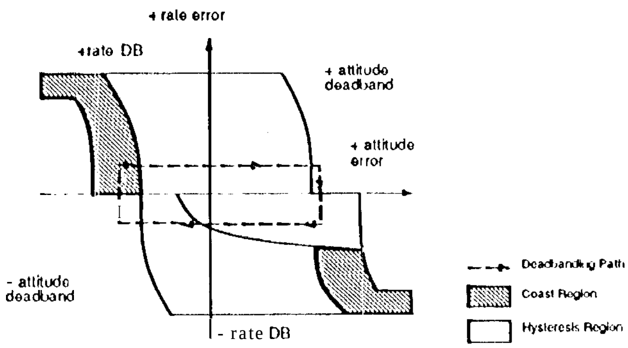

Figure 2: Graphical depiction of the phase plane, with coast and hysteresis regions [36]

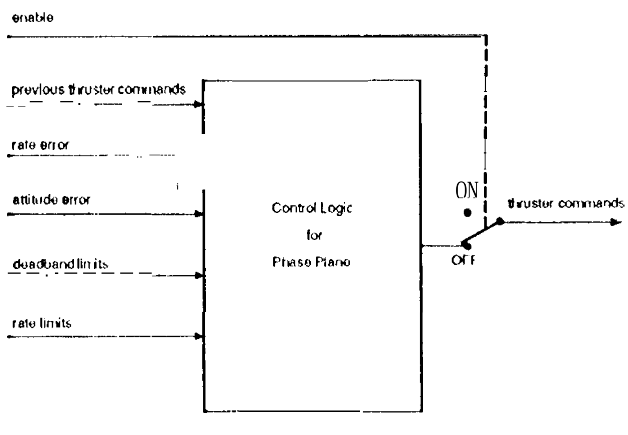

Figure 3: Wiring diagram for the Phase. Plane module [36]

3.2 Formal Specifications

One aspect of formal methods for critical software development is the use of a particular rigorous notation to precisely define the function of the system and requirements that the system software *must* meet. These formal specifications arc syntax- and type-checked using c.ollll)ilcr-like parsers. This project used the *PVS* (Prototype Verification Systems) formal specification tools [39, 40] under development by SRI International. PVS is written) iu Common Lisp but runs on interpreters of other Lisp dialects. A PVS user, however, interacts with a customized Emacs [41] interface and needs no knowledge of Lisp.

Our goal was to specify Phase. Plane's functionality and execution constraints at several levels of abstraction. Specification of a system through increasingly more detailed levels of abstraction is a well-cstal)lisllcd strategy used by specifiers [7, 42, 43]. Although these levels may appear almost disjoint, the proof of correct refinement of a level of specification by the level below assures the specifier the model is correct in addition to providing requirements traceability.

A general rule is that abstract, ul)l)cr-level specifications should establish system inputs, outputs, and basic functionality of the system. Critical correctness requirements that the system must satisfy arc stated at this level and become the criteria by which the specification is judged to be correct. Therefore, uJ)pcr-level specific.atio~[s tend to be black-box models of the system.

Mid-level specifications introduce both data type and functional detail that may constrain the eventual implementation of the system. These levels arc the core of the specification since design decisions and and execution environment issues can be introduced. Change requests for modules will most likely be addressed in these levels.

A 10W-1CVC1 ("as-built") specification is a straight-forward representation ofa particular implementation. It is from this detailed specification that source code can be automatically generated, or verification conditions for programmer-produced code derived.

The nature of Phase_Plane demanded a bottom-up approach instead of the top-down strategy described above. HigII-level English descriptions of this portion of the shuttle DAP were readily available, as was source code that had executed without error in hundreds of hours of use. This project explored the usc of formal specifications to derive requirements that are more detailed aud precise than an English paragraph and less obscure than tightly optimized source code.

A 10W-1CVC1 formal specification was developed from the existing source code, the Crew Training Manual [37], and the low level "wiring diagrams" of data flow and formula tables. This specification mirrored the functionality of the existing system, but did not offer an abstract view of the module's functional requirements.

A llig}l-level black-box specification was then developed corresponding to the level zero DFD (Figure 4). This formal specification did not include implementation details. At this level it was straightforward to state abstract properties that any software implementing Phase-Plane must have.

Finally, a mid-level formal specification was outlined to capture critical aspects of functionality and requirements at a level useful to shuttle "requirements analysts" when reviewing proposed modifications to the module. Due to time constraints, this level is still under development.

The challenge at the mid-level is to omit extraneous implementation details, yet be precise enough to capture necessary properties concerning minimization of fuel usage, thruster firings, and movement about the desired attitude. Included in this challenge is the linkage of the three specification levels by proofs that trace abstract, critical properties from the top-level specification through the mid-level, and to the low "code-level" specification.

It should be noted that since the PVS environment is interactive, it is possible for a user to make a "claim" and attempt a proof of the claim immediately. This feature can be particularly useful when attempting to deduce requirements from a code-level specification. This tactic can also be used to "test" a specification interactively. A current NASA RTOP has documented other advantages of formal methods in general and PVS in particular [34].

3.3 Construction of OMT Diagrams

This section describes the OMT diagrams that have been generated thus far for the Phase_Plane module. Since we started the reverse engineering process with the source code and implementation specific wiring diagram of the Phase. Plane module, we created two levels of data flow diagrams depicting the flow of information into, from, and within the Phase_Plane. These diagrams assisted in the abstraction process to obtain an architectural view of the phase plane as it related to the overall DAP system, thus leading to the construction of the object models. The object and the functional models offered one level of abstraction, thus leading to the development of the next layer of formal specifications (mid-level specifications describing data structure and operations on the data structures). Finally, using the functional and object diagrams in conjunction with the description of the deadbanding states, we created the dynamic model for the Phase_Plane module. The dynamic model depicts the states between jet firings as the orbiter deadbands. A high level of specifications was generated based on the dynamic model.

The remainder of this section describes the OMT diagrams constructed during the reverse engineering and formal specification construction process.

3.3.1 Functional Models

Data flow diagrams (DFD) facilitate a high level understanding of systems, both in terms of forward and reverse enginering. Static analysis of program code provides information that accurately describes flow of data in a system. In general, process bubbles denote procedures or functions of a given system. Arrows represent data flowing from one process to another.

The simplest functional model (DFD) is a *context diagram* or Level O diagram and is shown in Figure 4, where the entire phase plane module is reduced to a process bubble, with the external input and output labeled. This diagram provides the context for the process in question. Note that the Level O DFD closely resembles the structure of the "wiring" diagram for Phase_Pl ane given in Figure 3.

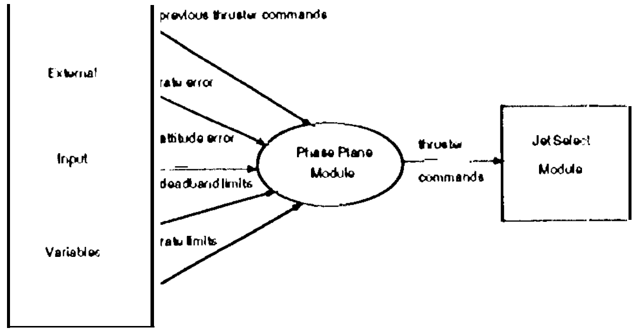

Figure 4: High Level (0) DFD for Phase. Plane Module

The child diagram for Figure 4 gives the next level DFD, which shows the different processes making up the Phase_Plane module and is shown in Figure 5. in this figure, the input variables are used to calculate boundaries for the phase plane. The boundaries and the attitude and rate limits are supplied to the process that calculates the thrust commands (jet firings).

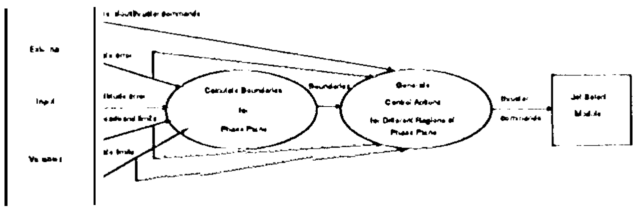

Figure 5: Level 1 DFD for Phase_Plane Module

3.3.2 Object Models

Studying the "as-built" layer of specifications, the different DFDs, and the requirements document for Phase-Plane led to the development of an object model for the Phase_Plane. As mentioned previously, an object is a self-contained module that includes both the data and procedures that act on that data. An object can be considered to be an abstract data type (ADT). A class is a collection of objects that have common use [44].

The object diagram for the Phase. Plane is shown in Figure 6. This diagram is a class entity with attributes *rate error*, *attitude error*, and *rotation axis*. The operation for this class is *calculate thrust commands* based on the rate and attitude errors. Also included in the object diagram are Phase Plane class instances (rounded rectangles) for each of the rotational axes (roll, pitch, and yaw). Each of the class instances will calculate different thrust commands for each of the specific rotational axes. Notice that there are two subclasses for the Phase Plane class, Coast Region and Hysteresis Region. In the coast region, the values of the attitude and rate errors are within acceptable bounds, thus there is no need to calculate new thrust commands. In the hysteresis region, however, the "Wale.ulate new thrust commands" operation is inherited from the Phase Plane class.

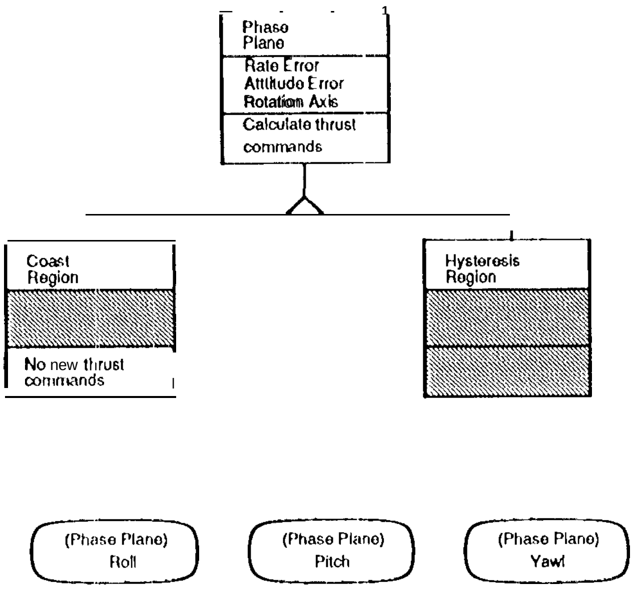

Figure 6: Object Model for Phase Plane Module

Next, we performed more abstraction steps in order to obtain a High-level object model for the DAP, consisting of the *State Estimator*, *Phase Plane*, and the *Jet Select Module*, corresponding to the diagram given in Figure 1. Figure 7 gives the object model for the DAP, where each class consists of three parts corresponding to the name of the class, list of attributes, and list of operations. The diamond symbol denotes aggregation, where the class above the diamond is said to consist of the three classes below the diamond. If either attributes or operations are not known (or do not exist) for a given class, then the corresponding area is shaded.

3.3.3 Dynamic Models

This section gives the dynamic models for the phase plane, which describes the states in which the DAP can be with respect to the Phase_Plane component. Also, included are the transitions that take the DAP from one state to another. A pictorial diagram of the envelope depicting the position of the Orbiter is given in Figure 8. The "O" plots the current vehicle attitude and rate errors with respect to the phase plane. As long as the current position is within the limits imposed by the deadbands (the heavy lines), the deadband constraints are satisfied and no jets will be commanded to fire. Once the Orbiter exceeds the bounds of the '[box]", jets will be commanded to fire in an effort to cancel the errors, thereby reducing the errors and achieving the

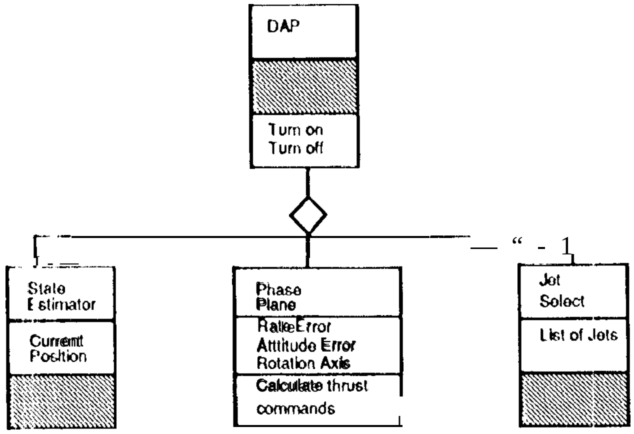

Figure 7: High Level Object diagram for DAP

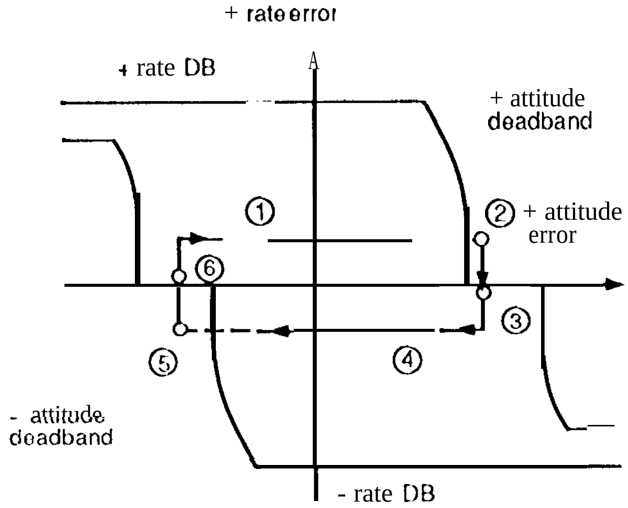

Figure 8: Graphical depiction of the phase plane, with deadbanding cycles [37]

requested maneuver or maintaining the attitude hold, whichever was requested by the crew. Once the Orbiter returns to the deadband area, the jets will stop firing.

Figure 9 gives an explanation of the different states in which the Orbiter can be while it is deadbanding [37]. Figure 10 gives a statechart depiction of the states through which the Orbiter transitions while it is deadbanding. The state transitions are in the form of jets (terminate/begin) firing and the Orbiter drifting (in/out) of the deadband region.

Note that Figure 8 depicts the clockwise traversal of the states in which the Orbiter cycles through the deadband limits. It is also possible for the Orbiter to traverse the cycle in a counterclockwise fashion, in which case, the arrows in Figure 10 would be reversed.

Finally, a very high-level view of the states in which the Orbiter can be is given in Figure 11. Included

1. Ncr jets fire. Since the rate error is positive, the attitude error will grow in a positive direction.
2. Jets fire to nullify the positive rotational rate.
3. Jets stop firing when the deadband line is crossed, but a little negative rate errors is inevitable.
4. No jets fire. With a negative rate error, the attitude error will also drift negatively.
5. Jets fire to nullify negative rate error.
6. Jets stop firing, but residual positive rate error causes attitude error to go positive again and the cycle repeats.

Figure 9: Explanation of deadbanding states [37]

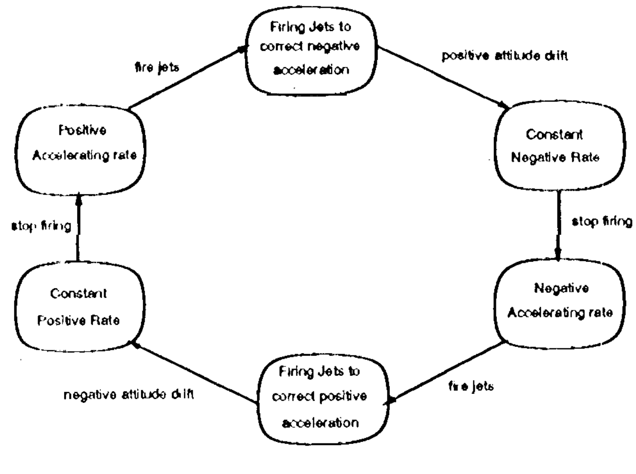

Figure 10: States representing the clockwise deadbanding of the Orbiter

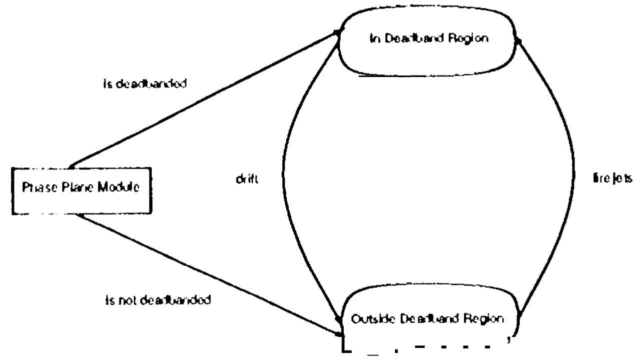

Figure 11: High-level states for Orbiter with respect to the Phase. Plane module

in the diagram are the actions or conditions that cause the Orbiter to transition from one state to the next. The rectangle containing "Phase Plane" and the labeled arrows pointing to the states indicate that the state transitions describe the Phase .Plane module.

4 Lessons Learned

The results from this reverse engineering project have provided several lessons for the overall project as well as for future reverse engineering projects. First, in order to obtain high-level requirements for existing software, it is not feasible to obtain the specifications (formal or informal) in one step. Instead, several layers of specifications must be developed, starting with the '(as-built" specification. The "as-built" specification closely mirrors the programming strut.turc of the misting software in order to provide traceability through the different levels of specifications. After creating the levels of specifications, theorems need to be constructed to demonstrate that critical properties are preserved from one level of specification to the next.

Second, formal specification languages and their corresponding reasoning systems provide a mechanism for bringing together disparate sources of project information into one integrated framework. In particular, the project information may be in a variety of formats, from different sources, and subjected to varying levels of formal review. For this particular project, information was obtained from the *Functional Subsystem Software Requirements* (FSSR) document [36] (also known as Level C requirements, consisting largely of "wiring diagrams"), *Guidance and Control Systems Training Manual* [37], source code, informal design notes [38], and discussions with shuttle software personnel. Accordingly, formal specifications were constructed based on all of the information in order to describe the phase plane operation. The PVS proof system provided a mechanism for checking the completeness and consistency of the specifications, while also supporting the proof construction of the relevant theorems.

Third, the benefits of ol~jcct-oriented analysis and design can be exploited for reverse-engineering as well as forward engineering projects. Specifically, object-oriented analysis and design assists in the understanding and the simplification of the complexity of a large system. Furthermore, having an object-oriented perspective facilitates future modifications by providing the developer with a high-level, abstract view of system components, thus avoiding the difficulties assoc. iated with attempting to understand all of the details of a large, complex system at once.

Finally, an iterative process consisting of the construction of a level of formal specifications, followed by a set of corresponding diagrams is needed to develop several layers of specifications for an existing system. The diagrams introduce al.retractions that can be used to guide the construction of the next level of specifications. Furthermore, the complementary diagrams available in the OMT approach enable the specifier to consider different perspectives of the system with notations best suited for the respective perspective. The major advantage to this diagramming approach is that one notation does not consist of many different symbols in an attempt to capture very different aspects of a system, which would make it too complex to use effectively.

5 Conclusions and Future Investigations

Using formal specifications and object-oriented analysis to describe the software that implements the Phase Plane module of the DAP has demonstrated that this rigorous technology can be used for existing, industrial applications. Constructing the different levels of specifications, with increasing abstraction, supplemented by the OMT diagrams provided a means for integrating information regarding the Phase. Plane module from disparate sources. Having access to this information will facilitate the verification that the original (critical) requirements or properties are not violated by any future changes to the software. In addition to facilitating verification tasks, the formal specifications can be used as the basis for any automated processing of the requirements, including checks for consistency and completeness. Interaction with the requirements analyst and other members of the original development team for the project strongly support the conclusion that the specification construction process, in addition to the actual specifications are useful to the ova-all software development and maintenance processes of existing (safety-critical) systems.

Future investigations will continue to refine the mid-level and high-level specifications and develop more theorems to relate the different levels of specifications. We are also investigating the formalization of the OMT diagramming notation, which will provide a means for using automated techniques for extracting formal specifications from the OMT diagrams in order to facilitate the specification process. Furthermore, extracting the specifications directly from the diagrams will allow us to reason about the completeness and consistency of the diagrammed system, thus greatly facilitating the requirements analysis, design, and maintenance phases of software development.

References

[1] Nancy G. Leveson and Clark S. Turner. An Investigation of the Therac-25 Accidents. *IEEE Computer*, pages 18-41, July 1993.

[2] Bev Littlewood and Lorenzo Strigini. The risks of software. *Scientific American*, pages 62-75, November 1992.

[3] Eric Schmitt. Army is blaming patriot's computer for failure to stop dharan scud. *New York Times*, May 1991.

[4] P. G. Neumann and contributors. Risks to the public. In *Software Engineering Notes*. ACM Special Interest Group on Software Engineering, 1993. Regular column published on a monthly basis.

[5] Victoria Slid Flor. Ruling's Dicta Causes Uproar. *The National Law Journal*, July 1991.

[6] Aeronautics and Space Engineering Board National Research Council. *An Assessment of Space Shuttle Flight Software Development Practices*. National Academy Press, 1993.

[7] Jeannette M. Wing. A Specifier's Introduction to Formal Methods. *IEEE Computer*, 23(9):8 24, September 1990.

[8] Susan L. Gerhart. Applications of formal methods: Developing virtuoso software. *IEEE Software*, 7(5):7-10, September 1990.

[9] Nancy G. Leveson. Formal Methods in Software Engineering. *IEEE Transactions on Software Engineering*, 16(9):929 930, September 1990.

[10] Richard A. Kemmerer. Integrating Formal Methods into the Development Process. *IEEE Software*, pages 37-50, September 1990.

[11] Anthony hall. Seven myths of formal methods. *IEEE Software*, pages 11-19, September 1990.

[12] Susan Gerhart, Dan Craigen, and Ted Ralston. An international study of industrial applications of formal methods. Technical report, NIST, NRL, and Atomic Energy Control, 1992.

[13] Betty H.C. Cheng. Synthesis of Procedural Abstractions from Formal Specifications. In *Proc. of COMPSAC'91*, pages 149 154, September 1991.

[14] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[15] David Grits. *The Science of Programming*. Springer-Verlag, 1981.

[16] John Rushby. Formal methods and the limits of dependability. In *Proceedings of Foundations of Theoretical Computer Science (FTCS 29)*, Toulousse, France, June 1993. Position paper for Panel.

[17] Betty H.C. Cheng and Gerald C. Gannod. Constructing formal specifications from program code. In *Proc. of Third International Conference on Tools in Artificial Intelligence*, pages 125 128, November 1991.

[18] Gerald C. Gannod and Betty H.C. Cheng. A two-phase approach to reverse engineering using formal methods. In *Lecture Notes in Computer Science, Proc. of Formal Methods in Programming and Their Applications Conference*. Springer-Verlag, June 1993.

[19] M. Ward, F.W. Calliss, and M. Munro. The maintainer's assistant. In *Proceedings Conference on Software Maintenance*, pages 307-315, Miami, Florida, October 1989. IEEE.

[20] K. Lano and P.T. Breuer. From Programs to Z Specifications. in J.E. Nicholls, editor, *Z User Workshop*, pages 46 70. Springer-Verlag, Oxford, 1989.

[21] Cliff B. Jones. *Systematic Software Development Using VDM.* Prentice Hall International Series in Computer Science. Prentice hall International (UK) Ltd., second edition, 1990.

[22] Edsger W. Dijkstra. *A Discipline of Programming.* Prentice Hall, Englewood Cliffs, New Jersey, 1976.

[23] D. R. Smith. KIDS: A Semi-automatic Program Development System. *IEEE Transactions on Software Engineering*, 16(9):1 024-1043, September 1990.

[24] Betty Hsiao-Chih Cheng. *Synthesis of Procedural and Data Abstractions.* PhD thesis, University of Illinois at Urbana-Champaign, 1990. Tech Report UIUCDCS-R-90-1631.

[25] Alan M. Davis. *Software Requirements, Analysis and Specification.* Prentice-Hall, Inc., 1990.

[26] John W. Brackett. *Software Requirements : SEI Curriculum Module SEI-CM-19-1. O.* Carnegie Mellon University Software Engineering Institute, December 1988. draft for public. review.

[27] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Sorenson. *Object-Oriented Modeling and Design.* Prentice-Hall, 1991.

[28] Rebecca Wirfs-Brock, Brian Wilkerson, and Lauren Wiener. *Designing Object-Oriented Software.* Prentice hall, 1990.

[29] G. Booch. *Object-Oricuicd Design with Applications.* Benjamin-Cummings, 1991.

[30] Peter Coad and Edward Yourdon. *Object Oriented Analysis.* Prentice-Hall, Inc., 2nd edition, 1991.

[31] Roger s. Pressman. *Software Engineering: A Practitioner's Approach.* McGraw Hill, third edition, 1992.

[32] P. Chen. The entity relationship model: Toward a unifying view of data. *ACM Transactions on Database Systems 1,* pages 9 36, March 1977.

[33] F.P. Brooks. No silver bullet: Essence and accidents of software engineering. *IEEE Computer*, April 1987.

[34] Jet Propulsion laboratory and Johnson Space Center and Langley Research Center. Formal Methods Demonstration Project for Space Applications: Phase I Case Study: STS Orbit DAP Jet Select. Research and Technology Objectives and Plans (RTOP), December 1993.

[35] David Hamilton. Discussion of phase plane requirements. Private Communication, August 1993. Hamilton is a software/knowledge engineer in the Advanced Technology Department of the Federal Sector Division for IBM-Houston working in conjunction with Johnson Space Center on the shuttle project. He works directly with the Requirements Analysts and is a software developer for the shuttle project.

[36] Space Shuttle Orbiter Operational Level C Functional Subsystem Software Requirements: Guidance Navigation and Control - Part C Flight Control Orbit DAP. Technical Report OI-21 edition, Rockwell International, Space Systems Division, February 1991.

[37] Sara Beck. G & C systems training manual: Guidance and Flight Control - Insertion, Onorbit and Deorbit. Technical Report I/O/D G&C2102, Mission Operations Directorate, Training Division, Flight Training Branch, October 1985.

[38] D. Johnson and R. Davison. Phase Plane Logic Design. JSC Memorandum concerning details of the design of the Phase Plane Logic., May 1986.

[39] N. Shankar, S. Owre, and J.M. Rushby. The PVS proof checker. Reference manual, March 1993.

[40] N. Shankar, S. Owre, and J.M. Rushby. The PVS specification language. Technical report, March 1993.

[41] Richard Stallman. *GNU Emacs Manual,* fifth edition, 1986.

[42] John V. Guttag, James J. Horning, K.D. Jones S.J. Garland, A. Modet, and J.M. Wing. *Larch: Languages and tools for formal specification.* Texts and Monographs in Computer Science. Springer-Verlag, 1993.

[43] Dines Bjorner and Cliff B. Jones. *Formal Specification and Software Development.* Prentice Hall, Englewood Cliffs, New Jersey, 1982.

[44] Ann 1, . Winblad, Samuel D. Edwards, and David R. King. *Object- Oriented Software.* Addiso~i-Wesley, Publishing Company Inc., 1990.