N94- 36495

1994031988

S11-61

12693

p- 23

Applying Formal Methods and Object-Oriented Analysis to Existing Flight Software

Betty H. C. Cheng[*]
Michigan State University
Department of Computer Science
A714 Wells Hall
East Lansing, MI 48824-1027
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 are 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 use of error-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 are precise, verifiable, and facilitate automated processing. This paper describes the application of formal methods to reverse engineering, where formal specifications are 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 shuttle 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] are 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-reducing 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]. One advantage to using rigorous approaches to software engineering is that formal notations are precise, verifiable, and facilitate automated processing [11, 12, 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 can then be used as the basis for change requests and the foundation for subsequent verification and validation. Considering re-implementation'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 validation of new requirements for the system, and verify the safety-critical 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 specifications and the ability to generate proofs from the formal specifications.

The remainder of the paper is organized as follows. Section 2 gives a brief introduction to formal methods and object-oriented 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 are discussed in Section 4. Finally, concluding remarks and future investigations are given in Section 5.

2 Background Material

This section briefly defines and motivates the use of formal methods. Also, the benefits of object-oriented analysis and design are presented.

[*]This author is also supported in part by NSF grant CCR-9209873.

2.1 Formal Methods

Formal methods in software development provide many benefits in the forward engineering aspect of software development [7, 8, 9, 15]. For any specification, there can be any number of implementations that satisfy the specification [16].

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

A *formal method* consists of a *formal specification language* and *formally defined inference rules* [15]. 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 purposes. 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 (by developing theorems) the design for the satisfaction of its requirements. Unfortunately, since the requirements are 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, 13].

Formal methods are 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 use 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 use automated tools for manipulation, elimination of ambiguity, precise documentation for maintenance, and improved reusability are a few examples of the overwhelming value, and often necessary benefits, that formal methods brings to the software development process.

2.2 Object-Oriented Techniques

There are a wide variety of approaches to requirements analysis, many of them in the broad category known as *object-oriented requirements analysis* (OOA) [14]. 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 OOA focus on objects is in contrast to the more traditional approach to analysis that focuses on procedures [24]. That is, instead of modeling the problem domain as a system of operations that process data objects, OOA modeling centers on a description of data objects and their interactions.

Most OOA techniques begin by a careful assessment of the natural language problem description. A simple first step in developing an OOA 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* are all **spacecrafts**, and *Venus*, *Mars*, and *Mercury* are 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, OOA 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 are 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, and the computational processes of the system as a whole (the dynamic behavior).

In the early stages of software development, includig object-oriented approaches, diagrams are 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 [26] 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 [27]. The *Object Modeling Technique* (OMT) [14] 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 [28]. 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 in simulation and flight, its specific properties remains obscure (at least to the requirements analyst and software developers) [29].

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 [30] (also known as Level C requirements, consisting largely of "wiring diagrams"), *Guidance and Control Systems Training Manual* [31], source code, informal design notes [32], 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 was 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) achieves and maintains attitude 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. This information is then supplied to the **Phase_Plane** component that calculates the attitude and rate errors with respect to desired values specified by the crew.

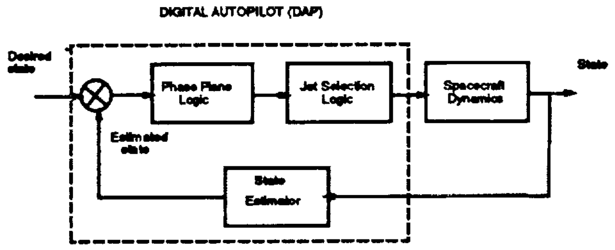


Figure 1: High-level view of DAP, including the **Phase_Plane** module [32]

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 [32]. 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. Figure 2 gives a simplified graphical representation of the phase plane [30]. The shaded regions depict the *coast regions* where the Orbiter does not need any corrective action. The remaining regions are known as *hysteresis regions*, where external factors such as positive (negative) acceleration drift, 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.

In an attitude hold situation, the error plot cycles around the zero error point with jets turning off and on again each time the limits of the "box" are exceeded. This activity 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 [28]). The dashed lines outline the deadbanding path in Figure 2.

The requirements for the **Phase_Plane** module are described in terms of a "wiring" diagram (see Figure 3 [30]), indicating the input and output values, and several tables describing the calculation for the boundaries of the phase plane and its different regions.

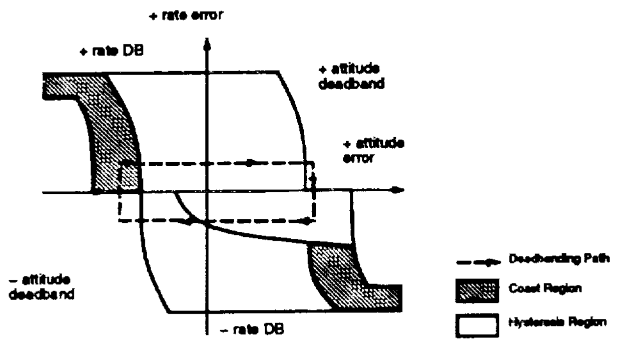


Figure 2: Graphical depiction of the phase plane, with coast and hysteresis regions [30]

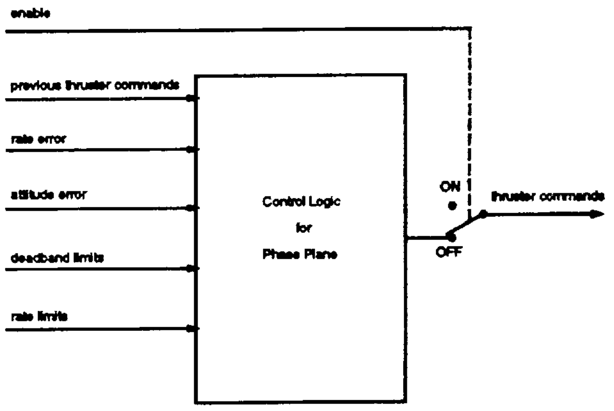


Figure 3: Simplified wiring diagram for the Phase_Plane module [30]

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 are syntax- and type-checked using compiler-like parsers. This project used the *PVS* (Prototype Verification Systems) formal specification tools [33, 34] under development by SRI International. *PVS* is written in Common Lisp but runs on interpreters of other Lisp dialects. A *PVS* user, however, interacts with a customized Emacs [35] 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-established strategy used by specifiers [15, 21]. 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, upper-level specifications should establish system inputs, outputs, and basic functionality of the system. Critical correctness requirements that the system must satisfy are stated at this level and become the criteria by which the specification is judged to be correct. Therefore, upper-level specifications 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 are 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 low-level ("as-built") specification is a straightforward representation of a 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. High-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 use of formal specifications to derive requirements that are more detailed and precise than an English paragraph and less obscure than tightly optimized source code.

A low-level formal specification was developed from the existing source code, the Crew Training Manual [31], 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 high-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 [28].

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. And rectangles represent external entities.

The simplest functional model (DFD) is a *context diagram* or Level 0 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 0 DFD closely resembles the structure of the "wiring" diagram for **Phase_Plane** given in Figure 3.

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

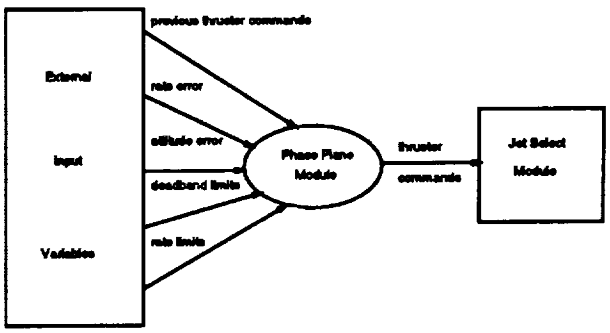


Figure 4: High Level (0) DFD for **Phase_Plane** Module

that calculates the thrust commands (jet firings).

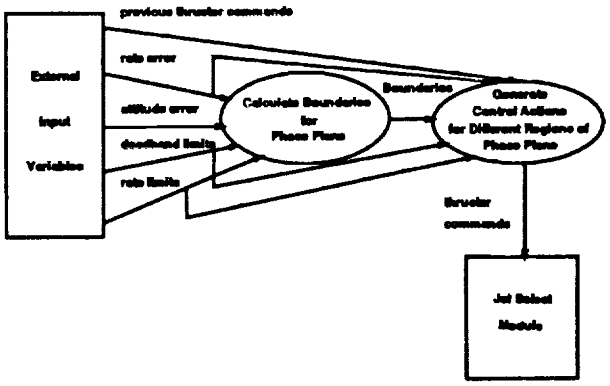


Figure 5: Level 1 DFD for **Phase_Plane** Module

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 [36].

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 "Calculate new thrust commands" operation is inherited from the Phase Plane class.

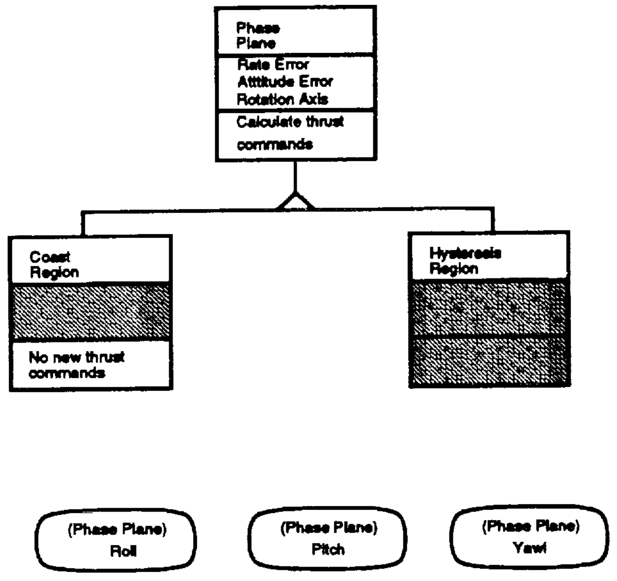


Figure 6: Object Model for Phase Plane Module

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

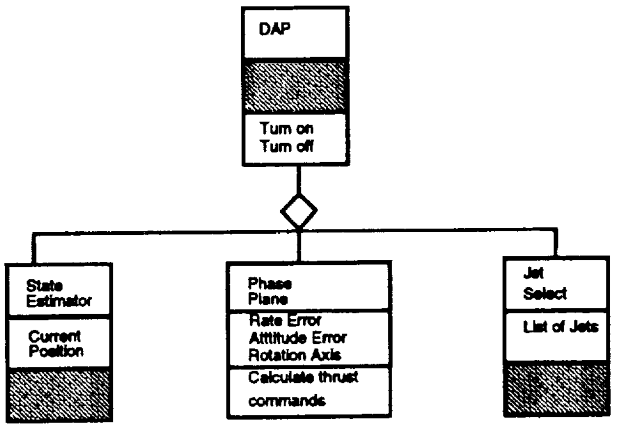


Figure 7: High Level Object diagram for DAP

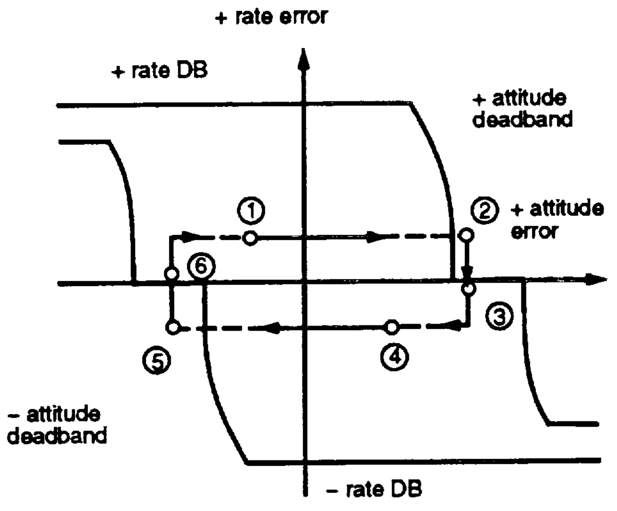


Figure 8: Graphical depiction of the phase plane, with deadbanding cycles [31]

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 [31]. 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

C-4.

1. No 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 [31]

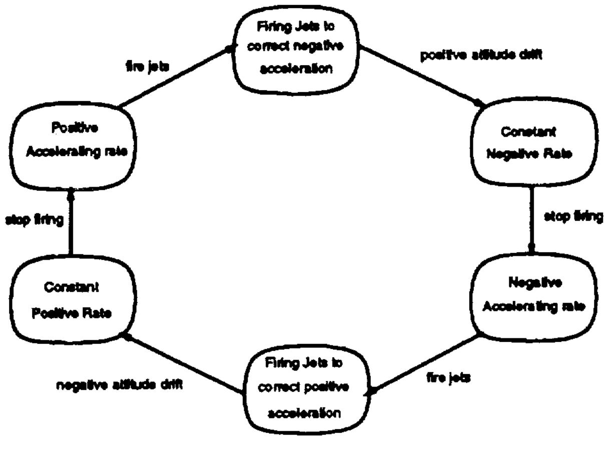

Firing Jets to correct negative acceleration
positive attitude drift
Constant Negative Rate
stop firing
Negative Accelerating rate
fire jets
Firing Jets to correct positive acceleration
negative attitude drift
Constant Positive Rate
stop firing
Positive Accelerating rate
fire jets

Figure 10: States representing the clockwise deadbanding of the Orbiter

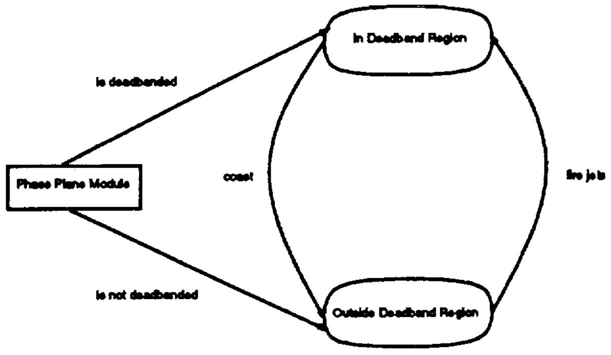


Figure 11: High-level states for Orbiter with respect to the `Phase_Plane` module

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 structure of the existing 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 [30] (also known as Level C requirements, consisting largely of "wiring diagrams"), *Guidance and Control Systems Training Manual* [31], source code, informal design notes [32], 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 object-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 associated 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 abstractions 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 overall 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.

6 Acknowledgements

Several people have provided valuable information and assistance during the course of the project. Specifically, we would like to thank David Hamilton, John Kelly, Rick Covington, and John Rushby.

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

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

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

[10] 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.

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

[12] Jun jang Jeng and Betty H.C. Cheng. Using Automated Reasoning to Determine Software Reuse. *International Journal of Software Engineering and Knowledge Engineering*, 2(4):523–546, December 1992.

[13] Betty H.C. Cheng. Applying formal methods in automated software development. *accepted to appear in Journal of Computer and Software Engineering*, 1993.

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

[16] David Gries. *The Science of Programming*. Springer-Verlag, 1981.

[17] John Rushby. Formal methods and the limits of dependability. In *Proceedings of Foundations of Theoretical Computer Science (FTCS 23)*, Toulousse, France, June 1993. Position paper for Panel.

[18] 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.

[19] 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.

[20] 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.

[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):1024–1043, September 1990.

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

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

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

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

[28] 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.

[29] 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.

[30] 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.

[31] 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.

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

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

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

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

[36] Ann L. Winblad, Samuel D. Edwards, and David R. King. *Object-Oriented Software.* Addison-Wesley, Publishing Company Inc., 1990.

Applying Formal Methods and Object-Oriented Analysis to Existing Flight Software

Betty H.C. Cheng
Department of Computer Science
Michigan State University
East Lansing, Michigan 48824-1027
ph: (517) 355-8344; fax: (517) 336-1061
email: chengb@cps.msu.edu

Brent Auernheimer
Department of Computer Science
California State University
Fresno, California 93740-0109
ph: (209) 278-2573; fax: (209) 278-4197
email: brent_auernheimer@csufresno.edu

18th Annual Software Engineering Workshop, Dec. 1-2, 1993

Background for Project

- Integrate formal methods to portion of shuttle software
- Construct an object-oriented view of system
- Demonstrate the numerous utilities of formal methods in software development
- Facilitate current and future maintenance
  "Due to careful review of changes, it takes an average of 2 years for a new requirement to get implemented, tested, and into the field."
- Facilitate verification of safety-critical properties
- Address one major issue encountered in industry: *reverse engineering* of existing (legacy) system.

Software Engineering Workshop (12/93)-6

Formal Methods

- What is a formal method?
  - Formal languages with well-defined syntax
  - Well-defined semantics
  - Proof systems
- Why use Formal Methods?
  - Improve quality of software systems
  - Reveal ambiguity, incompleteness, and inconsistency in a system
- Important Characteristics:
  - Abstraction
  - Proof obligations
  - Tool support
  - Systematic Process

Software Engineering Workshop (12/93)-1

Object-Oriented Software

- Represent real-world problem domain and maps it into software solution domain
- OO Design interconnects data objects and processing operations
- Modularizes information and processing, not just processing
- Three Main concepts:
  - Abstraction
  - Information hiding
  - Modularity

Software Engineering Workshop (12/93)-2

Object Modeling Technique

- Three diagramming notations give complementary perspectives of system
  - *Object Model* presents the architectural view (traditional object-oriented diagramming notation)
  - *Functional Model* presents a functional view (data flow diagrams)
  - *Dynamic Model* presents the behavioral view (state diagrams)
- More amenable to formalization than other OO diagramming notations
- Widely used in industry and universities, including IBM at JSC.

Software Engineering Workshop (12/93)-3

"What would help me do RA for Orbit DAP"

It is highly unlikely that we'll find a product that will understand shuttle requirements. Some degree of customization will need to be performed in whatever tools we choose to support our formal methods activities.

- From the beginning, shuttle requirements authors were given the freedom to express requirements in whatever form they preferred.
- Consequently, the shuttle requirements are a combination of many formats, styles, conventions, and perspectives.
- It has historically been very difficult to insert new technologies into the shuttle program.
- Any tool that takes steps to the existing shuttle requirements or automatically convert the existing requirements into a format it can understand will be much more likely to succeed.

Software Engineering Workshop (12/93)-5

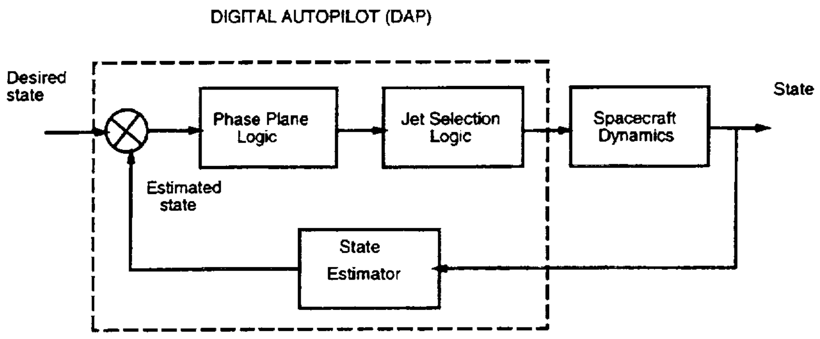
Project Selection Criteria

- Current RTOP is demonstration project:
  - *Jet Select* module for space shuttle
  - Determine which jets should be fired to achieve desired position(s)
  - Select module that is accommodating *Change Requests*
- Faculty Fellowship project complements RTOP project
  - *Phase Plane* module: control system for monitoring angular rotation
  - Determines amount of corrective action needed

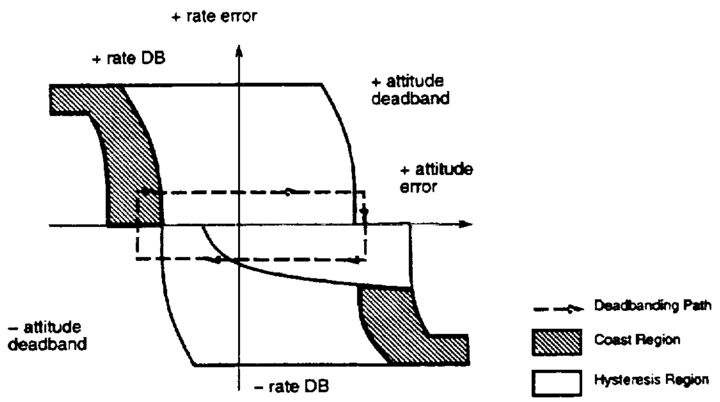
Phase Plane

- JSC expressed keen interest in Phase Plane module
  - JSC had difficulty fully understanding module
  - Difficulty in testing module
  - Will need to make changes in future
  - Results feed directly to Jet Select module
- Phase Plane applicable to other spacecraft
- Main component of control system
  - Uses thrusters to control angular state of spacecraft
  - Monitor state errors
  - Determine when and how corrective control should be applied

Pictorial View of DAP Control Loop



Graphical Representation of Phase Plane

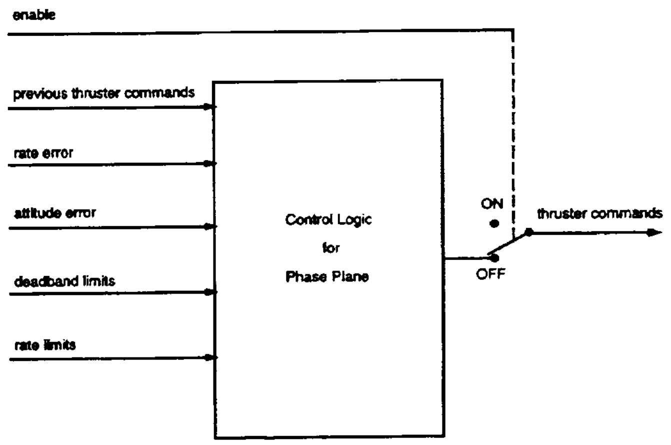


Preliminary Tasks

- Learn new specification language (PVS), including support tools.
- Become familiar with Jet Select and Phase Plane domain
  - *Functional Subsystem Software Requirements* (wiring diagrams)
  - *Crew Systems Training Manual*
  - Informal requirements discussions from JSC, IBM, Draper Labs (software designers), including site visit to IBM at JSC.
  - Informal design notes
- Become familiar with commonly used object-oriented diagramming technique and support tools.

Software Engineering Workshop (12/93)-11

Wiring Diagram for `Phase Plane`



Software Engineering Workshop (12/93)-12

Project Overview

- Apply reverse engineering techniques
- Develop levels of specifications
- Each level is more abstract than previous
- Objective: obtain a high-level specification of requirements
- Identify and prove critical properties that link the levels.
- Develop an OMT hierarchical "roadmap" of module
- Establish linkage between specifications and OMT diagrams.

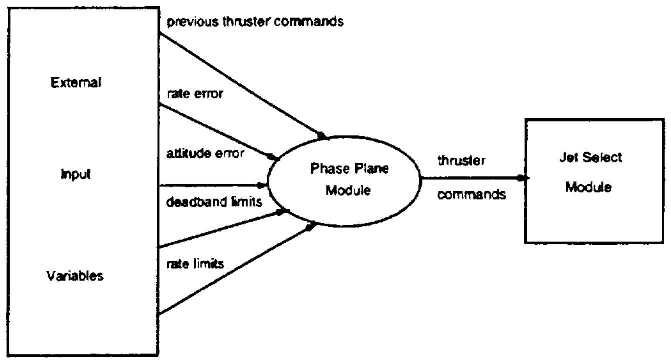
Software Engineering Workshop (12/93)-13

Iterative Process

- Construct low-level specifications correspond to wiring diagrams
- Use code for clarification
- Construct OMT diagrams for wiring diagrams
- Identify properties required for system.
- Construct high-level specifications for properties of Phase Plane
- Construct high-level OMT diagrams that apply to Phase Plane
- Integrate specifications with OMT diagrams.

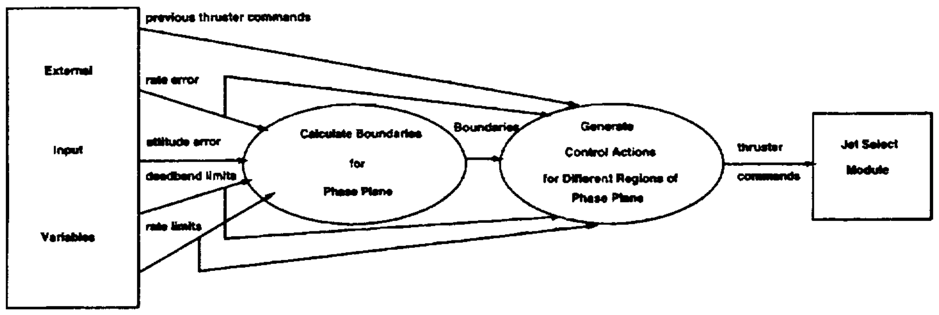
Software Engineering Workshop (12/93)-14

Functional Model: Level 0 (context) DFD



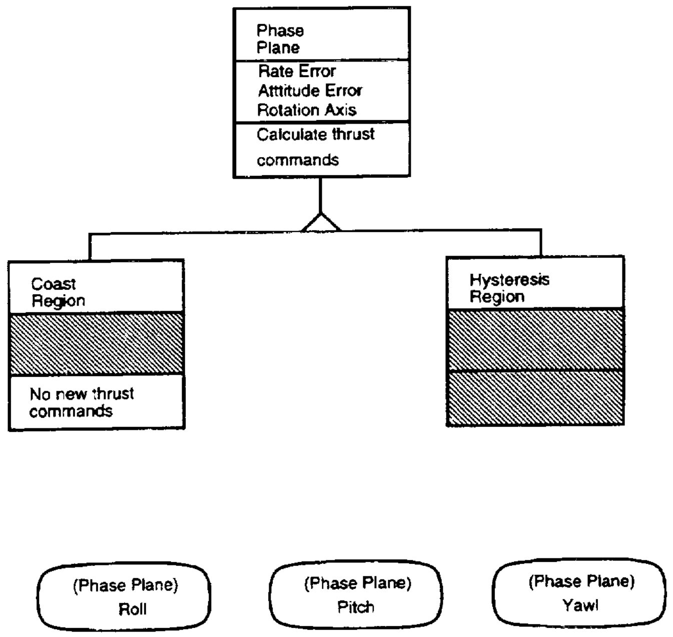
Software Engineering Workshop (12/93)-15

Functional Model: Level 1 DFD


previous thruster commands
External
rate error
attitude error
Input
deadband limits
rate limits
Variables
Calculate Boundaries for Phase Plane
Boundaries
Generate Control Actions for Different Regions of Phase Plane
thruster
commands
Jet Select Module

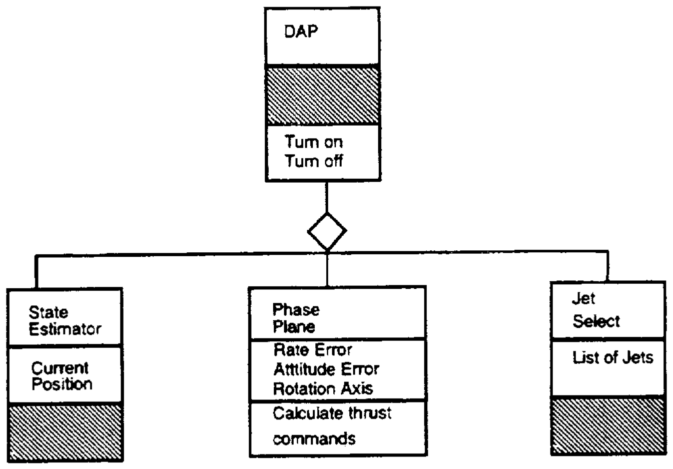
Software Engineering Workshop (12/93)-16

Object Model for Phase Plane



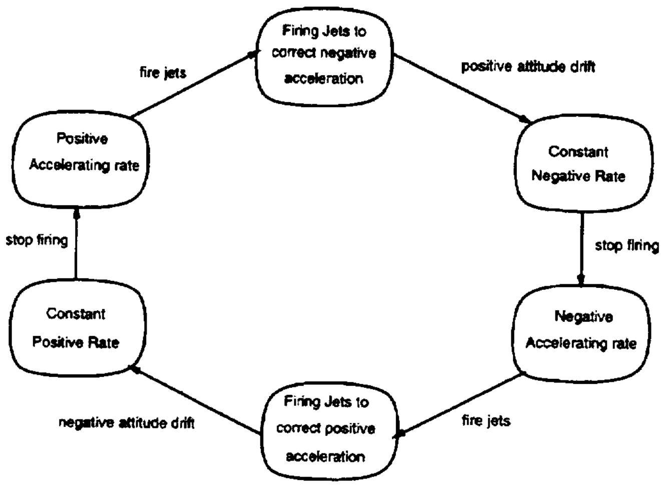
Software Engineering Workshop (12/93)-17

Object Model for DAP



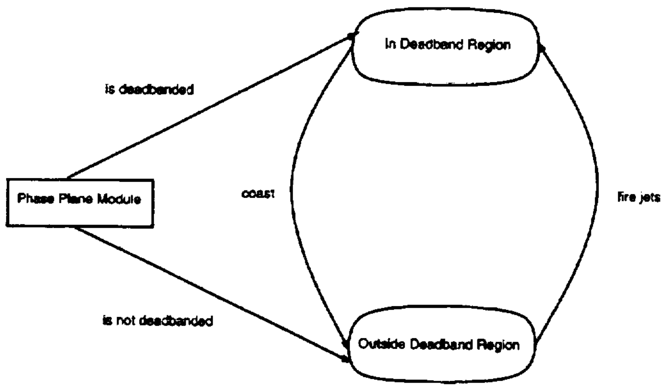
Software Engineering Workshop (12/93)-18

Dynamic Model for Deadbanding


Firing Jets to correct negative acceleration
positive attitude drift
Constant Negative Rate
stop firing
Negative Accelerating rate
fire jets
Firing Jets to correct positive acceleration
negative attitude drift
Constant Positive Rate
stop firing
Positive Accelerating rate
fire jets

Software Engineering Workshop (12/93)-19

More Abstract Dynamic Model for Deadbanding



Software Engineering Workshop (12/93)-20

Lesson I

- More than one step from high-level requirements to existing code.
- Must create several layers of specifications
- "As-built" layer closely mirrors code (traceability)
- Need to construct theorems relating layers of specifications

Software Engineering Workshop (12/93)-21

Lesson II

- Formal methods provide mechanism for integrating disparate sources of project information.
- Project information may be:
  - in a variety of formats,
  - subjected to varying levels of formal reviews
  - located physically apart
- Examples include:
  - Functional Subsystem Software Requirements ("wiring diagrams")
  - Crew Training Manual
  - Design notes
  - Discussions with shuttle software personnel.
- Use formal specifications to integrate information from different sources.

Software Engineering Workshop (12/93)-22

Lesson III

- Object-oriented analysis and design can be exploited for reverse engineering tasks.
- OO introduces abstraction to simplify complexity of system
- OO perspective can facilitate future maintenance tasks

Software Engineering Workshop (12/93)-23

Lesson IV

Reverse engineering process is iterative

- Construct level of formal specifications
- Create a set of diagrams (introduces abstraction)
- Repeat.

Software Engineering Workshop (12/93)-24

Summary

- Incorporate formal methods into existing system
  - Assist maintainers in understanding module
  - Facilitate future changes
  - Facilitate verification of critical properties
- Develop reverse engineering process using FM and OO
- Develop OMT models usable by RTOP project
- Identify obstacles (and solutions) in abstraction (reverse engineering) process usable by RTOP project
- Demonstrate utility of FM and OO on real project.

Software Engineering Workshop (12/93)-25

Current and Potential Future Tasks

- Develop mid-level specifications
- Construct multi-level correctness proofs
- Demonstrate how FM can be used to gain confidence in the correctness of software after modification using critical correctness criteria and proofs.
- Integrate more closely the formal specifications with OMT diagrams.

Software Engineering Workshop (12/93)-26

Acknowledgements

- The work described in this paper was carried out at the Jet Propulsion Laboratory, California Institute of Technology, and was sponsored by the National Aeronautics and Space Administration. Additionally, the authors' work on this project was supported by NASA/ASEE Summer Faculty fellowships.