Using Formal Methods and Object-Oriented Analysis to Reverse Engineer Shuttle Software*

Betty H. C. Cheng†
Michigan State University
Department of Computer Science
East Lansing, Michigan 48824-1027
U. S. A.

Brent Auernheimer‡
California State University, Fresno
Department of Computer Science
Fresno, California 93740-0109
U. S. A.

Abstract

Correctness is an important issue in safety-critical software control systems. Unfortunately, failures in critical segments of software for medical equipment, communications, and defense are familiar to the public. Such incidents motivate the use of software development techniques that reduce errors and detect defects. The benefits of applying formal methods in requirements-driven software development (forward engineering) are well-documented; formal notations are precise, verifiable, and facilitate automated processing. This paper describes the application of formal methods and object-oriented modeling to *reverse engineering*, in which formal specifications are developed for existing, or *legacy*, code. In this project, several layers of formal specifications were constructed for a portion of the NASA Space Shuttle Digital Auto Pilot (DAP), a software module that is used to control the position of the spacecraft through appropriate jet firings. The reverse engineering process was facilitated by the *Object Modeling Technique* (OMT), an informal software development approach that uses graphical notations to describe software requirements.

1 Introduction

Correctness is necessary in safety-critical software control systems [1]. Critical software failures in medical equipment, communication networks, and defense systems are familiar to the public. The large number of software malfunctions regularly reported to the software engineering community [2], new statutes concerning liability for such failures, and a recent National Research Council Aeronautics and Space Engineering Board Report [3], additionally motivate the use of software development techniques that reduce errors and detect defects.

The benefits of using formal methods in requirements-driven software development (forward engineering) are well-documented [4]. A formal method is characterized by a formal specification language and a set of rules that govern the manipulation of expressions in that language.

*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. A preliminary version of this paper was presented at the NASA/Goddard Software Engineering Workshop, Greenbelt, Maryland, December, 1993.

†This author is also supported in part by NSF grant CCR-9209873 and CCR-9407318.

‡This author gratefully acknowledges the Software Engineering Institute at Carnegie Mellon University for support as a Visiting Scientist, Spring 1994.

One way to take advantage of the benefits of formal methods in legacy systems is to reverse engineer the existing program code into formal specifications [5, 6]. The resulting formal specifications can then be used as the basis for change requests and the foundation for subsequent verification and validation [7]. Common reverse engineering methods currently used by software maintenance engineers are observation (for example, test case analysis) and examination of source code. These techniques are often tedious and error-prone. Considering the high cost of re-implementation and the need to preserve critical functionality, reverse engineering of code into formal specifications offers an alternative to traditional *ad hoc* approaches to maintaining safety-critical systems.

A highly visible example of a legacy system is the software for the NASA Space Shuttle, which was conceived in the early 1970s and has been operational for over ten years. One component of the Shuttle software is the flight software that provides guidance, navigation, and control for the Space Shuttle while it is in orbit. The navigation function determines where the shuttle is, the guidance function determines where it should go next, and the control function determines *how* to implement the next move.

Presently, the Space Shuttle flight software project has a well-defined process for managing requirements evaluation. This process is responsible for ensuring that requirements generated by an engineer are consistent, implementable, and will solve the problem at hand. However, this process does not include a well-defined set of analytical methods and techniques [7, 8]. When a change is needed, a detailed description of the reasons for the change, known as a *change request* (CR), must be constructed before the system can be re-engineered to include the changes. Next, the requirements analyst performs an in-depth analysis of the CR, guided by a list of generic error categories, followed by an inspection of the CR by several representatives of the software project, including the author of the CR, requirements analyst, developer, verifier, and so on. When all inspections have been conducted for a CR and all issues (potential errors) have been resolved, a CR is ready for implementation. At this point, a baseline for the project, a milestone that describes the current system with the accepted changes, is created and scheduled for implementation.

The analysis step of the CR process involves studying, understanding, and analyzing the contents of a CR. Three major deficiencies in this process have been identified by requirements analysts [8]. First, there is no specific methodology for conducting the analysis of the CR. Second, there are no specific completion criteria to indicate when sufficient information has been obtained for the CR. Third, there is no specific structured mechanism for documenting the results of the analysis process. Moreover, since there is no structured approach for documenting the analysis, the understanding of the CR developed by the requirements analyst is not formally recorded for future use.

This paper describes a project that applies formal methods and object-oriented analysis to a subsystem of the DAP of the Shuttle, known as the `Phase_Plane`. This module determines whether jet firings are needed to achieve and hold an attitude (position relative to a specific frame of reference) specified by the crew. The objective of this project is to provide formal specifications of the requirements and functionality of the system that can be used to facilitate automated verification and validation of future changes and to facilitate re-engineering tasks. 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 optimized source code. We developed several layers of formal specifications that capture the details of the requirements of the `Phase_Plane` module. In order to facilitate the construction of the layers of specifications, we constructed a pictorial description of the subsystem using the *Object Modeling Technique* (OMT) [9], an informal software development approach that uses graphical notations to describe software requirements.

The remainder of the paper is organized as follows. Section 2 describes the Phase Plane project, including sample specifications and a discussion of the object-oriented analysis. Section 3 contains a summary of the process that we used to reverse engineer the Phase Plane subsystem. This section also includes lessons learned from this project and the benefits and the limitations of our approach. Finally, conclusions and future investigations are described in Section 4.

2 Project Description

Due to the criticality and the volume of flight system software, recent flight system projects are incorporating formal methods into the software development process [1, 4]. In order to apply formal methods to legacy flight software, however, reverse engineering is needed. The `Phase_Plane` project is associated with a larger multi-NASA site project to apply formal methods to a portion of the flight control software for the NASA Space Shuttle [7, 8]. The criteria that led to the selection of `Phase_Plane` included finding a module whose requirements were difficult to understand and which will likely be the target of future critical change requests.

The development of the high-level formal specifications was divided into two major tasks. First, we acquired a concise description of the original requirements of the module. Much of this information was obtained from a functional requirements document, consisting largely of wiring diagrams similar to those used for circuit design, the (astronaut) crew training manual, source code, informal design notes, and discussions with Shuttle software personnel. We used the resulting description to develop

an "as-built" (implementation-biased) formal specification, capturing the functionality depicted in the wiring diagrams.

Second, in order to obtain a more abstract specification and eliminate the implementation bias present in the as-built layer, we developed object modeling diagrams (OMT) [9] to represent the integral information from the low-level specifications. These diagrams facilitated the identification of abstractions that we introduced into the higher-level specifications. This process of developing a level of formal specification, followed by the construction of the corresponding OMT diagrams, enabled the identification of the high-level, critical requirements of the `Phase_Plane` module. Sample specifications and OMT diagrams are described below.

2.1 Phase Plane

The *Reaction Control System* (RCS) Digital Auto Pilot system (DAP) achieves desired positions via necessary movements through jet firings. Figure 1 gives a pictorial representation of translation (x, y, and z coordinates of the vehicle) and attitude (rotational position of the vehicle in terms of roll, pitch, and yaw) as they relate to the position of the Shuttle.

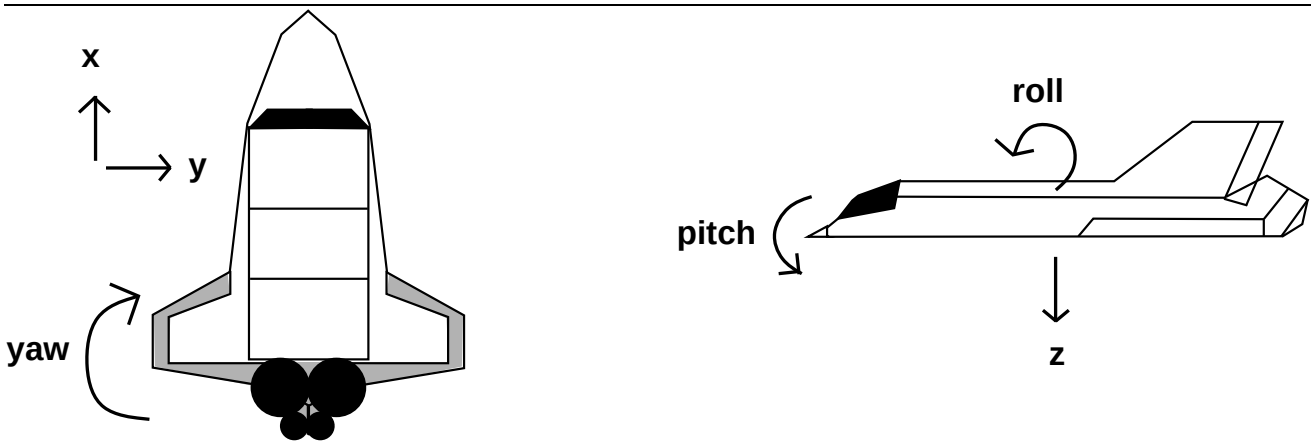

Figure 1: Shuttle Translational and Rotational Axes

In order to maintain the Shuttle at a specific attitude, the crew specifies two values: attitude deadband and rate deadband. *Attitude deadband* refers to how much drift (positive or negative) will be tolerated in any axis before jets are fired to correct the error. *Rate deadband* refers to the allowable rate changes of the attitude (positive or negative) before jet firings are required to null the error. Figure 2 gives a high-level view of the DAP; the `Phase_Plane` component compares information from the *State Estimator* that describes current attitude values, taking into consideration spacecraft dynamics (e.g., fuel usage

and inertia) and the crew supplied values. Depending on the amount of error correction necessary, the `Phase_Plane` component requests jet firings, where the `Jet_Select` component determines which jet(s) to fire (the topic of the larger multi-NASA site project).

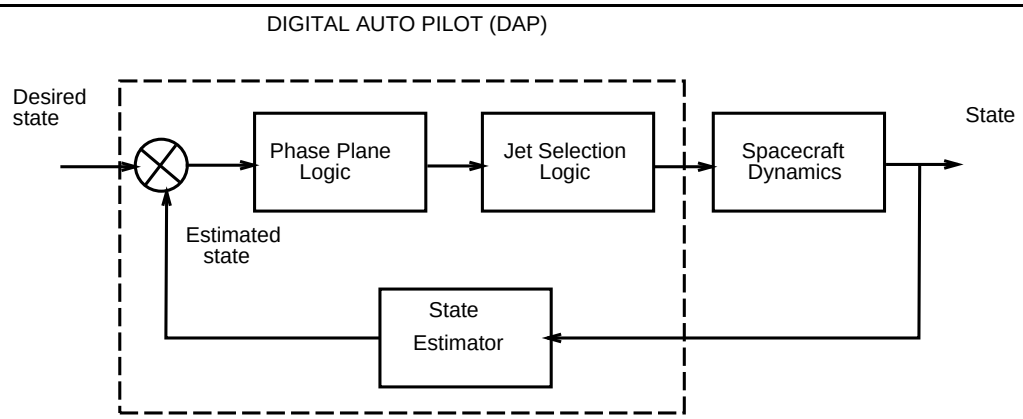

Figure 2: Context for the `Phase_Plane` module

Figure 3 gives a simplified graphical representation of the phase plane. A phase plane is represented as a graph plotting spacecraft rate errors against attitude errors for one rotational axis. In an attitude hold situation, *deadbanding* occurs (indicated by dashed lines), which means that the error plot cycles around the zero error point with jets firing each time the limits of the "box" are exceeded. Each "◯" indicates points that the Shuttle is changing system state with respect to thruster firings. The shaded *coast regions* depict situations where the Shuttle needs no 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, are taken into consideration in order to preclude unnecessary jet firings.

The requirements for the `Phase_Plane` module are described in a functional specification that includes a simplified wiring diagram (see Figure 4), which identifies the input and output values, as well as several tables that contain equations from control theory to calculate the boundaries of the phase plane and its regions. For historical reasons, the functional descriptions use notation commonly used for circuit design, even though the system being described is software-based. The solid lines represent data flows and dashed lines represent control. In Figure 4, the dashed line indicates that the *enable* flag must be set by the crew in order to enable the auto pilot mode.

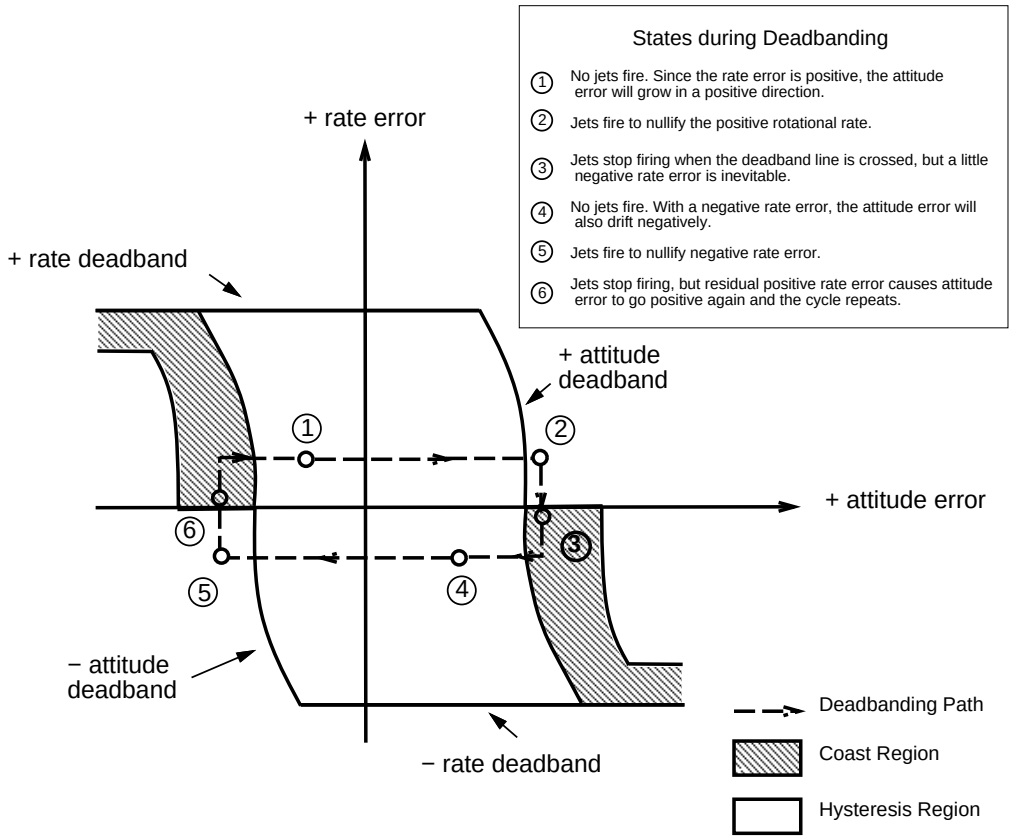

Figure 3: Graphical depiction of the phase plane, with coast and hysteresis regions

2.2 Formal Specifications

One aspect of formal methods for software development is the use of a formal specification language, a rigorous notation to precisely define the functionality and requirements of the system. There exists many types of formal specifications, but we can categorize them into two major types: model-oriented and property-oriented. *Model-oriented* specifications define system's behavior directly by constructing a model of a system in terms of mathematical structures, such as tuples, functions, sets, or sequences. Examples include VDM and Z for sequential systems and CSP and Petri Nets for concurrent and distributed systems [10]. *Property-oriented* specifications define a system's behavior indirectly by stating a set of properties (usually in terms of axioms) that the system must satisfy [10]. Two subcategories are *axiomatic specifications* typically expressed in terms of pre- and postconditions in first-order predicate logic and *algebraic specifications* that use axioms to specify properties, where axioms are in equation format. The *PVS* (Prototype Verification Systems) formal specification tools [11] (e.g. syntax checker and theorem prover) were used for this reverse engineering project. *PVS* is

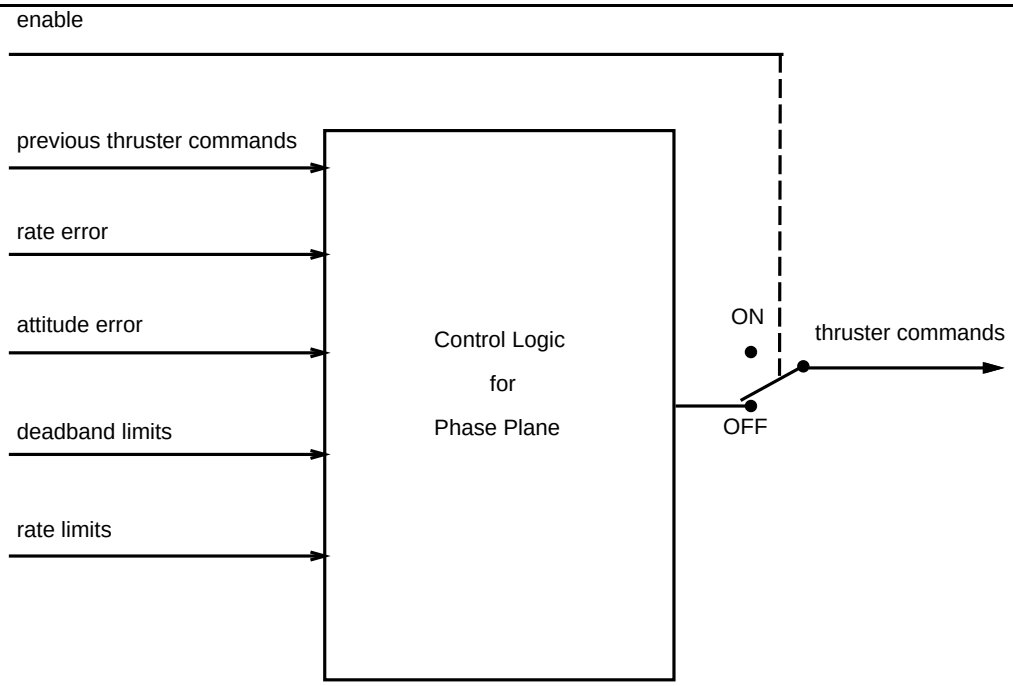

Figure 4: Simplified wiring diagram for the `Phase_Plane` module

a property-oriented specification language, where a specification comprises a collection of *theories*. Each theory consists of a *signature* for the type names and locally declared constants, as well as the axioms, definitions, and theorems associated with the signature. In addition to its property-oriented attributes, which enable the straightforward construction of and reasoning about specifications that describe behavior and desired (required) properties, we chose the *PVS* language for its modularity and the availability of tools, such as syntax- and type-checkers.

In order to obtain a specification of the high-level requirements from the existing documentation and source code, we constructed several layers of *PVS* specifications, where each layer is more abstract than the preceding layer. Specification of a system through increasingly detailed levels of abstraction is a well-established method [10, 12]. Reverse engineering the `Phase_Plane` project involved a mixture of a bottom-up with a top-down approach. We developed the specifications in the following order: low-, high-, and mid-level. High-level natural language descriptions of this portion of the Shuttle DAP were available, as was source code. Given the range of details available from the two types of documentation (prose versus code), we decided to start with the low-level specifications to ensure that we captured an accurate description of the current functionality. Next, we used the high-level descriptions from the crew training manual to construct several OMT diagrams, all of which was used to assist in the identification and specification of high-level requirements. Finally, in order to bridge the information gap between the

low-level, implementation-specific and the high-level specifications, we constructed a set of mid-level specifications. The OMT diagrams introduced abstraction into the low-level specifications, and the high-level specifications identified critical properties applicable to the overall component; the combined information provided the constraints for the mid-level specifications.

2.2.1 Low-level Specifications

We developed the low-level formal specification of `Phase_Plane` from the existing source code, the crew training manual, and the low-level wiring diagrams. This specification mirrored the functionality of the existing system, but did not offer an abstract view of the module's functional requirements. The optimized source code consisted of several calculations for determining the regions within the phase plane. In Figure 3, we have provided a high-level depiction of the regions within the phase plane, where, in actuality, the coast and hysteresis regions each have more fine-grained partitions with a total of five regions as determined by fourteen boundaries (labeled `s1-s14`). The boundary calculations made extensive use of several constants stored in a table, which represent initialization values for a given flight. The code also dictated how control actions were calculated depending on which region the shuttle was located. In order to calculate the control actions, values of variables that serve as the interface between the `Phase_Plane` and other components within the DAP were used. Example values include error rate limits, deadband values, current rate error, current position, and the previous jet firing commands. For brevity, we do not include the complete low-level specifications here, but the specifications may be found in the appendix.

2.2.2 High-Level Specifications

Next, we developed a high-level "black-box" specification, which did not include implementation details. At this level, it was straightforward for us to state abstract properties that any software implementing `Phase_Plane` must possess. The high-level specification describes properties that characterize the Shuttle's position in terms of attitude and rate deadband values: if the Shuttle travels outside the specified regions, then the jets need to be fired to bring the Shuttle back into the phase plane region. We defined a few predicates to describe general properties of the Shuttle, where Boolean predicates are denoted by a "?" suffix, and the types of the predicate arguments are enclosed in square brackets. First, the `is_deadbanded?` predicate determines whether the Shuttle is in a deadbanding state, where there are four arguments to the predicate corresponding to the attitude deadband, rate deadband, current attitude error, and current rate error represented by their respective types.

```
is_deadbanded? : pred[attitude_deadband_type,rate_deadband_type,
                      attitude_error_type,rate_error_type]
```

Next, two predicates are defined to check whether rate and attitude errors are in a region where jets need to be fired to decrease rate error (generate positive rate error).

```
decrease_rate_error? : pred[attitude_deadband_type,rate_deadband_type,
                            attitude_error_type,rate_error_type]

increase_rate_error? : pred[attitude_deadband_type,rate_deadband_type,
                            attitude_deadband_type,rate_deadband_type]
```

Figure 5 contains an abbreviated version of the top-level specifications. In this case, `wiring_phase_plane` refers to the low-level specifications. The referenced states are those depicted in Figure 3.

The following high-level axiom, based on the specification for the six states, relates the attitude to the rate deadbands, as well as the rate and attitude errors. Specifically, the axiom asserts the invariant that if the Shuttle is in the deadband regions, then there is no need to fire jets to increase or decrease the rate error.

```
AXIOM FORALL
(att_db:attitude_deadband_type),(rate_db:rate_deadband_type),
(att_err:attitude_error_type), (rate_err:rate_error_type):
        is_deadbanded?(att_db,rate_db,att_err,rate_err) <=>
           NOT (decrease_rate_error?(att_db,rate_db,att_err,rate_err)  OR
                increase_rate_error?(att_db,rate_db,att_err,rate_err)
               )
```

2.2.3 Mid-Level Specifications

Finally, we outlined a mid-level formal specification that captures critical aspects of functionality and requirements at a level that would be useful to Shuttle requirements analysts when reviewing proposed modifications to the module. Code developed from this specification would implement the "Phase Plane Logic" box of the low-level wiring diagram (Figure 4). The challenge at the mid-level was 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. In constructing the mid-level specifications, we made several assumptions. First, we did not consider external acceleration disturbances. This assumption means that by taking advantage of symmetry, it is sufficient to specify only the upper (nonnegative rate error) half of the `Phase_Plane` diagram, as shown in Figure 6. Second, the hysteresis region is treated as a coast region. Finally, an implementation bias previously imposed in the wiring diagrams to allow the crew to enable the module was removed. We also removed the explicit assertion that the calculations will be done once for each axis (roll, pitch, and yaw).

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%  Module:  High-Level Specifications of Properties for Phase Plane Module
%
%  The following characterize the 6 states of Shuttle when it is deadbanding
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
high_level_phase_plane: THEORY
BEGIN
  USING wiring_phase_plane  % low-level specifications for phase plane
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% No jets fire. Since the rate error is positive, the attitude error will
% grow in a positive direction. (State 1)
%
no_jets_positive_rate?(att_db,rate_db,att_err,rate_err) : bool =
           is_deadbanded?(att_db,rate_db,att_err,rate_err) &
           rate_err > 0      &       att_err  > 0
%
% Jets are firing to correct positive rotational rate (State 2)
%
jets_fire_correct_pos_attitude_error? (att_db,rate_db,att_err,rate_err):bool =
           NOT (is_deadbanded?(att_db,rate_db,att_err,rate_err)) &
           decrease_rate_error?(att_db,rate_db,att_err,rate_err)
%
% Jets stop firing when deadband line is crossed, but a little negative
% rate error is inevitable. (State 3)
%
jets_stop_negative_rate_error?(att_db,rate_db,att_err,rate_err) : bool =
           is_deadbanded?(att_db,rate_db,att_err,rate_err) &
           rate_err < 0
%
% No jets fire. With negative rate error, the attitude error will also
% drift negatively. (State 4)
%
no_jets_negative_rate?(att_db,rate_db,att_err,rate_err) : bool =
           is_deadbanded?(att_db,rate_db,att_err,rate_err) &
           rate_err < 0  &       att_err  < 0
%
% Jets are firing to correct negative attitude error (State 5)
%
jets_fire_correct_neg_attitude_error?(att_db,rate_db,att_err,rate_err): bool =
           NOT (is_deadbanded?(att_db,rate_db,att_err,rate_err)) &
           increase_rate_error?(att_db,rate_db,att_err,rate_err)
%
% Jets stop firing, but residual positive rate error causes attitude
% error to go positive again and cycle starts over (State 6)
%
jets_stop_positive_rate_error?(att_db,rate_db,att_err,rate_err) : bool =
           is_deadbanded?(att_db,rate_db,att_err,rate_err) &
           rate_err > 0

...

end high_level_phase_plane
```

Figure 5: Sample high-level specifications of `Phase_Plane`

Figure 7 defines a few deadbanding functions to take advantage of the symmetry and y represents the vertical axis (absolute value of rate error) and x is the horizontal (attitude error) axis. The symmetry property enables us to generalize the calculations to those in the upper half of the deadband

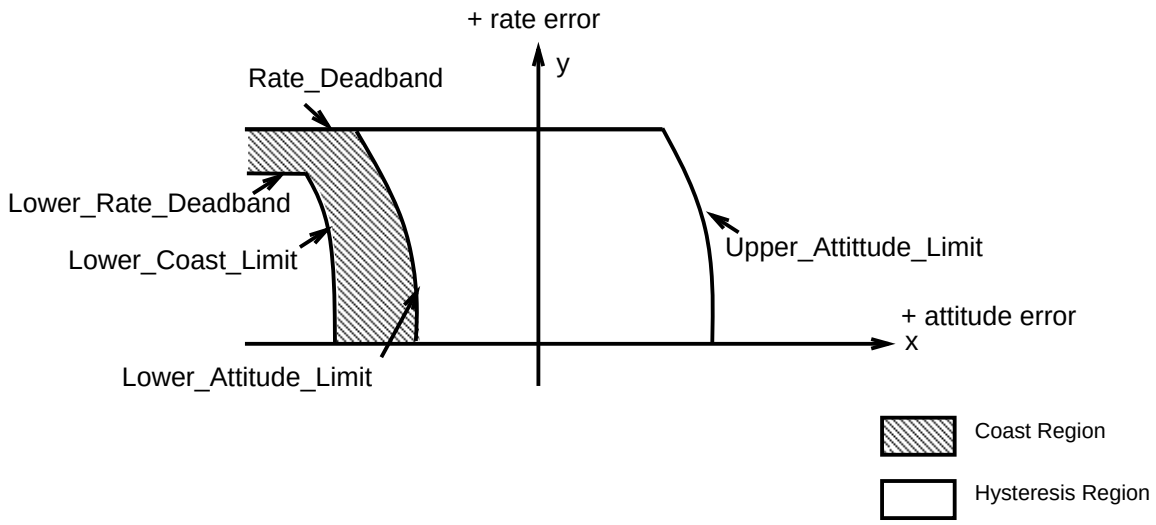

Figure 6: Upper Half of Phase Plane

region. The `adjust_for_symmetry` function accounts for symmetry of the phase plane and returns the new thruster command given the current rate error and thruster command. The calculations for `upper_attitude_limit` and `lower_attitude_limit` are a generalization of a portion of the low-level specifications. These limits determine the bounds of the hysteresis regions, and, as mentioned previously, are a function of the jet firings.

The tail of the coast region is defined by the `rate_deadband` above and the `lower_rate_deadband` below. The following specification gives the `lower_rate_deadband` as a real and asserts that the `lower_rate_deadband` is at most the `rate_deadband`.

```
lower_rate_deadband:        real
rate_deadband_relationship: AXIOM lower_rate_deadband <= rate_deadband
```

The lower (left) boundary of the coast region is defined by the `lower_attitude_limit` (a function declared below) and `attitude_deadband`. The specification asserts only that the `lower_coast_limit` is at most the `lower_attitude_limit`.

```
lower_coast_limit:          real
coast_limit_relationship:   AXIOM lower_coast_limit <= lower_attitude_limit
```

The primary function `control_action` returns a thruster command. Thruster hysteresis can be used to minimize thruster firings due to delays, sensor noise, or movement between state transition boundaries. Figure 8 gives the specification for calculating the thruster commands. First, it must be

```
%
% Calculate coordinates for plotting attitude and rate errors
%
  y: absolute_rate_error_type = abs(rate_error)
  x: real = sign(rate_error)*attitude_error

% Because all calculations are done in the upper half of the deadband
% region, the calculated thruster command may need to be reversed.

  adjust_for_symmetry(t: thruster_command_type,
                           re: rate_error_type) : thruster_command_type =
      IF (t = zero_thrust) OR (sign(re) >= 0)
         THEN t
         % re was negative, so thruster commands must be reversed
         ELSE IF t = positive_thrust
                 THEN negative_thrust
                 % t was negative_thrust
                 ELSE positive_thrust
              ENDIF
      ENDIF
%
% Calculate boundary of hysteresis region based on a function of jet firings
%
  upper_attitude_limit: real = -sqr(y)/(2*thruster_impulse) + attitude_deadband
  lower_attitude_limit: real = -sqr(y)/(2*thruster_impulse) - attitude_deadband
```

Figure 7: Variables and deadbanding functions to adjust for symmetry in phase plane

determined if the spacecraft is outside the deadband area and thrusters should be fired "downward". Second, it must be determined whether the spacecraft is outside the deadband area and thrusters should be fired "upward". Third, if the spacecraft is within the "coast" zone, then do not fire thrusters. If all the above cases do not apply, then incorporate thruster hysteresis.

```
thruster_hysteresis:      thruster_command_type = zero_thrust

control_action: thruster_command_type =
  IF (y > rate_deadband) OR (x > upper_attitude_limit)
     THEN adjust_for_symmetry(negative_thrust, rate_error)
     ELSE IF (y < lower_rate_deadband) AND (x < lower_coast_limit)
             THEN adjust_for_symmetry(positive_thrust, rate_error)
             ELSE IF      (y <= rate_deadband)
                      AND (lower_rate_deadband <= y)
                      AND (x <= lower_attitude_limit)
                  OR      (x <= lower_attitude_limit)
                      AND (lower_coast_limit <= x)
                      AND (lower_rate_deadband <= y)
                      THEN zero_thrust
                      ELSE  thruster_hysteresis
                  ENDIF
          ENDIF
  ENDIF
```

Figure 8: Specification of Function to Calculate Thrust Commands

2.3 Construction of OMT Diagrams

In the early stages of software development, including object-oriented approaches, diagrams are frequently used to describe requirements and guide development. The OMT [9] notation combines three complementary diagramming notations in order to document system requirements: object models, dynamic models, and functional models. An *object model* describes the architecture of an overall system in terms of the elements (objects) of a system and identifies allowable relationships among objects. As a result, the object model constrains the set of possible states that the system may enter. A *dynamic model* describes valid transitions between system states and indicates the conditions under which a state change may occur. Dynamic models are described in terms of state transition diagrams. A *functional model* is a data flow diagram that describes the computations to be performed by the system. In a complementary fashion, these three types of diagrams are used to model the properties of the system, including flow of control, flow of data, patterns of dependency, time sequence, and name-space relationships. The OMT approach is appealing since it offers multiple views of software requirements, and since a single notation is not forced to describe many different perspectives of a given system, the notation for each type of diagram is simple to use and easy to understand.

Since the original `Phase_Plane` software was not object-oriented, we began the OMT analysis with the source code and implementation-specific wiring diagram of the `Phase_Plane` module and constructed two levels of data flow diagrams. 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. 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 Shuttle deadbands. We generated a high-level specification based largely on the dynamic model. The object and the functional models offered one level of abstraction, which directly enabled us to develop of the next layer of formal specifications (mid-level specifications describing data structures and operations on the data structures).

2.3.1 Functional Models

Data flow diagrams (DFD) facilitate a high level understanding of systems and are used in both forward and reverse enginering. Static analysis of program code provides information that accurately describes flow of data in a system. 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 is a *context diagram*, or Level 0 DFD; the Level 0 DFD for the `Phase_Plane` module is shown in Figure 9, where the entire phase plane module is reduced to a process bubble, with the external input and output labeled. The Level 0 DFD closely resembles the structure of the wiring diagram for `Phase_Plane` given in Figure 4.

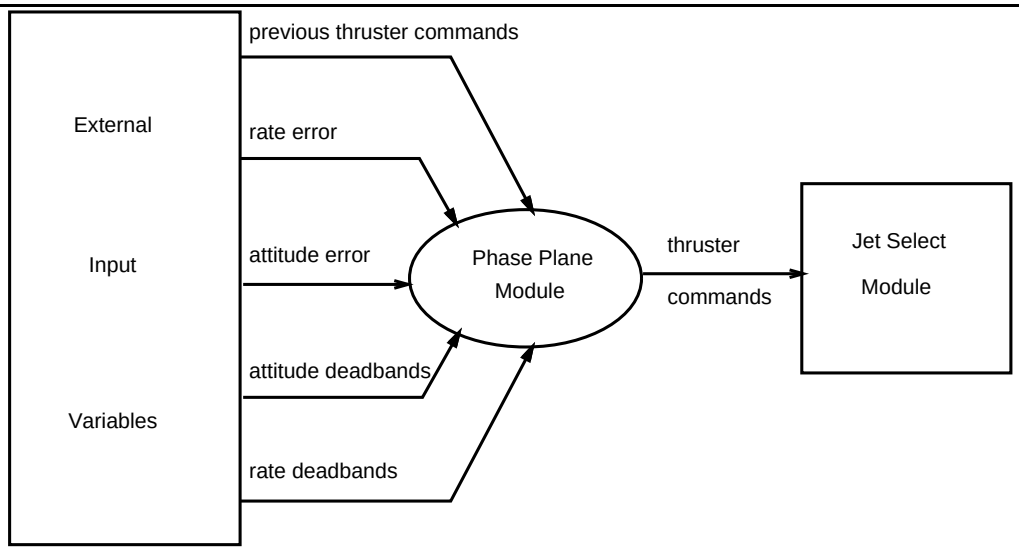

Figure 9: High Level (0) DFD for `Phase_Plane` Module

Figure 10 gives the next level DFD, which shows the different processes that constitute the `Phase_Plane` module. As shown in this figure, the input variables are used to calculate boundaries for the phase plane. The boundaries, the attitude and the rate deadbands, are supplied to the `Phase_Plane` module, which calculates thruster commands (jet firings). The thruster commands are then supplied to the `Jet_Select` module that determines which combination of jets should be used to achieve the desired thruster effect.

2.3.2 Object Models

Figure 11 depicts a high-level object model for the entire DAP, consisting of the State Estimator, Phase Plane, and the Jet Select classes, corresponding to the diagram given in Figure 2. Each class consists of three parts corresponding to the name of the class, list of attributes, and list of operations, respectively. 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. The Phase Plane class uses the class Crew Supplied

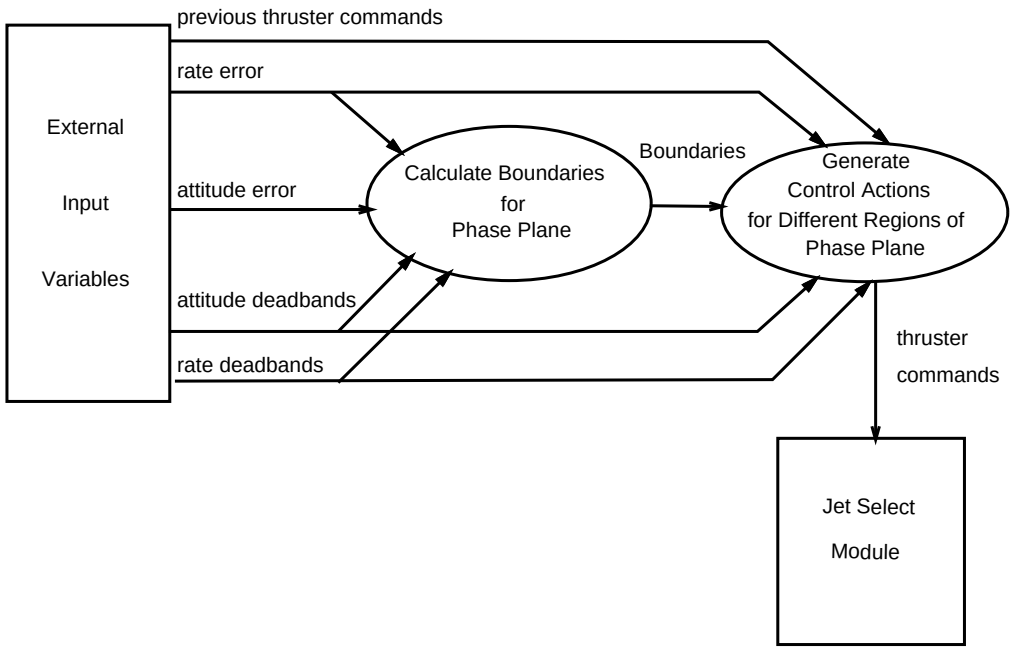

Figure 10: Level 1 DFD for `Phase_Plane` Module

Information, which represents the deadband limits that are used in the calculation of the phase plane boundaries.

Figure 11 also contains the object diagram for the Phase Plane class,with attributes *rate error*, *attitude error*, and *rotation axis*. The operation for this class is *calculate thrust commands*, based on the difference between the current rate and attitude error values and those respective limits supplied by the crew. The filled circle attached to the Phase Plane class, indicates that the DAP contains three phase plane components, one to calculate different thrust commands for each of the specific rotational axes: roll, pitch, and yaw. There are two components for each Phase Plane object, Coast Region and Hysteresis Region. In the coast region, only the values of the attitude and rate errors are used to determine whether the Shuttle is still within the deadband limits. In the hysteresis region, however, additional information, such as fuel usage, sensor noise, and other spacecraft dynamics, is used to calculate thrust commands.

2.3.3 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 position of the Shuttle is given in Figure 3.

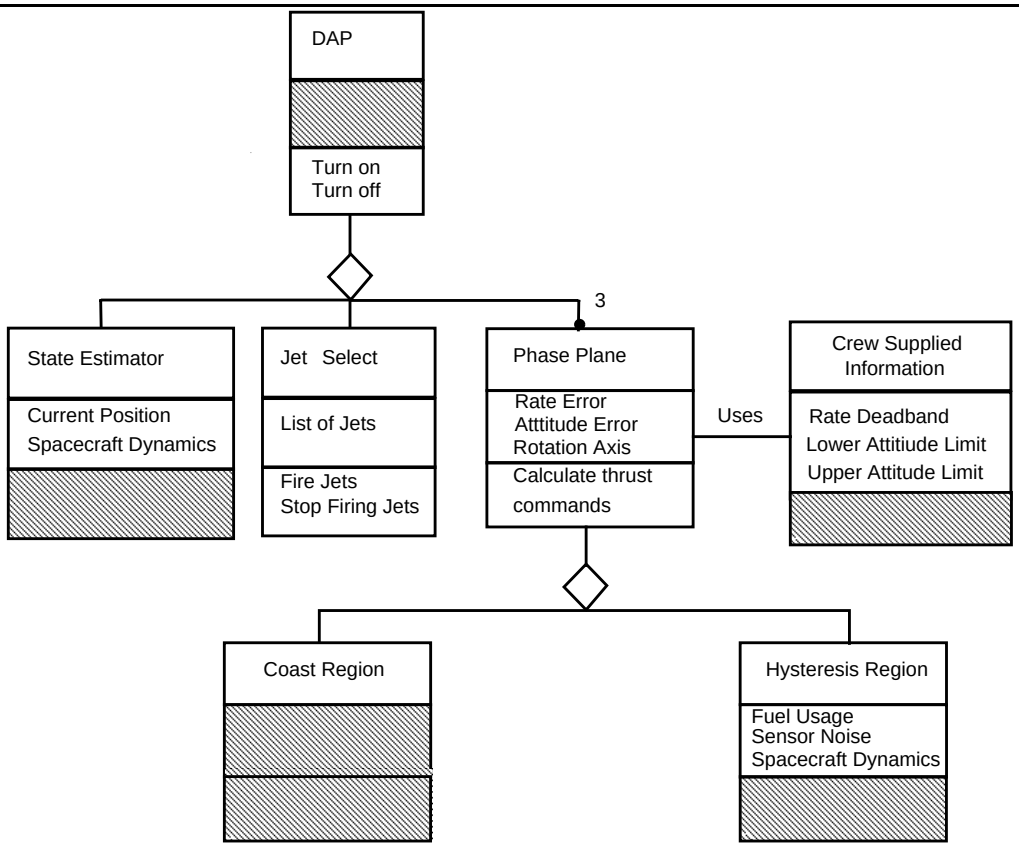

Figure 11: Object model for DAP

Since the `Phase_Plane` module is an event-based system, the state transition diagram is straightforward to construct.

Figure 12 gives a state transition diagram of the states through which the Shuttle transitions while it is deadbanding. The state transitions are in the form of jets terminate (begin) firing and the Shuttle drifting into (out of) the deadband region.

Note that Figure 3 depicts the clockwise traversal of the states in which the Shuttle cycles through the deadband limits. It is also possible for the Shuttle to traverse the cycle in a counterclockwise fashion, in which case, the arrows in Figure 12 would be reversed.

Finally, a very high-level view of the states in which the Shuttle can be is given in Figure 13. Included in the diagram are the actions or conditions that cause the Shuttle to transition from one state to the next: jet firings and drift. The rectangle containing "Phase Plane" and the labeled arrows pointing to the states indicate that the state transitions describe the `Phase_Plane` module.

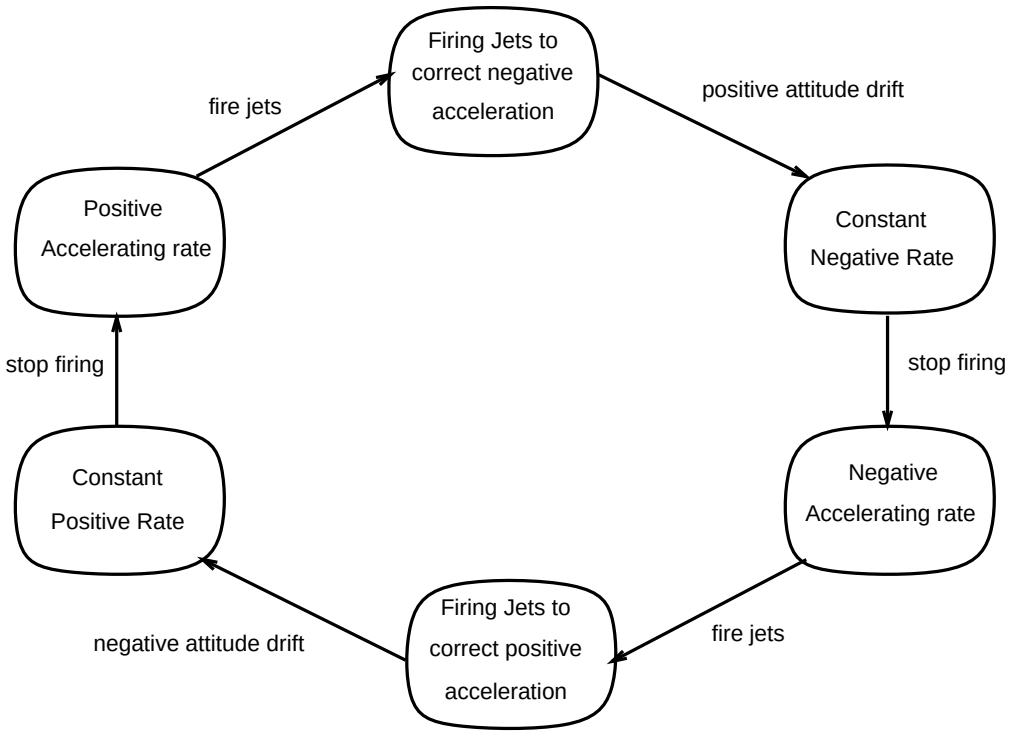

Figure 12: States representing the clockwise deadbanding of the Shuttle

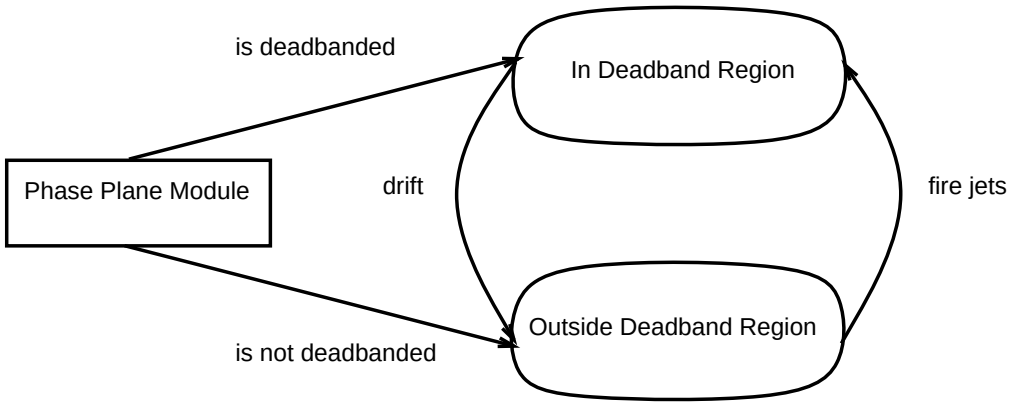

Figure 13: High-level states for Orbiter with respect to the `Phase_Plane` module

3 Summary and Evaluation of Reverse Engineering Process

This section summarizes the reverse engineering process and discusses the lessons learned. The benefits to the overall project gained from the reverse engineering results are described. Finally, the limitations and problem areas of this approach are discussed.

3.1 Process Summary

The `Phase_Plane` module consisted of approximately 200 lines of highly optimized code. In order to reverse engineer `Phase_Plane` and understand its context, we had to analyze approximately five times that amount. While a precise cost analysis was not performed for this project, an estimate of the cost of the analysis and construction of formal specifications and accompanying OMT diagrams was approximately two person months. This cost includes the time needed to learn PVS and how to use its tools, acquire supporting documentation, gain a minimal understanding of control theory, and refine several times the specifications and diagrams. Combining the cost information from the Jet Select reverse engineering project, it does appear that the costs are within reason and are roughly comparable to the current cost of requirements analysis [7]. When considering highly critical subsystems the cost is not prohibitively expensive.

The results from this reverse engineering project have provided several lessons for future reverse engineering projects. First, in order to obtain high-level requirements for existing software, it is difficult to obtain the specifications (formal or informal) in a single step. Instead, several layers of specifications should be developed, starting with the as-built specification. By closely mirroring the programming structure of the existing software, this specification provides traceability through the different levels of specifications.

A summary of the overall process used to reverse engineer `Phase_Plane` is given in Figure 14.

3.2 Benefits to the Software Development Process

Formal specification languages and their corresponding reasoning systems provide a framework for integrating disparate sources of project information to describe a system at many levels of detail. The project information may be documented in a variety of formats, come from different sources (often physically distributed), and subjected to varying levels of formal review. For this particular project, information was obtained from implementation-specific wiring diagrams, definitions and instructions from a crew training manual, source code, informal design notes, and discussions with Shuttle software personnel. We analyzed and distilled the information into specifications and OMT modeling diagrams. These products will increase the capability for future analysis of the `Phase_Plane` component. That is, because the requirements information are now described in a formal notation (annotated with easy to understand diagrams), automated analysis and validation are possible, which will greatly facilitate future approvals of change requests. In addition, the *PVS* proof system provided an automated mechanism for checking the completeness and consistency of the specifications.

- Identify components of software to be analyzed.
- Gather supporting documentation, including functional requirements, source code, design-level documentation, and user-manuals (as available).
- Define what the "hard constraints" of the specification are. What documentation should be used as the source for describing the critical requirements of the system? For reverse engineering projects, typically, the source code and functional requirements document is used to determine critical requirements.
- Create "as-built" layer of specifications. This layer of specification should directly mirror the functionality observed from the source code. This mirroring effect will provide traceability from the final layers of specifications to the source code.
- In order to introduce abstraction, create multiple levels of DFDs and begin the object-oriented (OO) analysis. The OO analysis is used to create an architectural view of the system, which is applicable even if the original system was not developed with object-orientation.
- Using high-level documentation (e.g. user manual) to identify the high-level system requirements, which should then be pictorially represented in terms of the dynamic model (state transition diagram).
- Based on the state transition diagrams, create high-level specifications.
- Refine the object-models of the system using information from the DFDs, code, and high-level documentation.
- Construct the mid-level specifications by developing properties that provide the linkage between the implementation-specific information from the low-level specifications and the required properties described in the high-level specifications.
- After constructing the specifications, use proof tools to check for consistency between specification layers.

Figure 14: General process for reverse engineering using formal methods and object-oriented analysis

Third, the results of this project demonstrate that benefits of object-oriented analysis can be exploited for reverse-engineering as well as forward engineering projects. Specifically, object-oriented analysis assists in the understanding of large, complex systems. Furthermore, an object-oriented perspective facilitates future modifications by providing the requirements analyst and the developer with a high-level, abstract view of system components.

Finally, a process consisting of the construction of a level of formal specifications, followed by a set of corresponding diagrams facilitates the development of several layers of specifications. The diagrams introduce abstractions that can be used to guide the construction of the next level of specifications. Furthermore, the three complementary notations in the OMT approach enable the specifier to represent different components of the system using the best-suited type of diagram.

3.3 Limitations to this Approach

While there are several benefits to using an integrated approach consisting of formal specifications and OMT diagrams, several limitations exist. Currently, in order to perform consistency and completeness checks of the *PVS* specifications for a specific component or subsystem, theories that describe related components may need to be constructed. In our case, we had the advantage that a team had constructed *PVS* specifications for the Jet Select component. Also, the specifications have focused thus far on functional properties. In future investigations, we will study the amenability of *PVS* to non-functional properties.

Second, the specification and diagram construction process is not automated, however, once the specifications are created, they can be analyzed and manipulated using automated tools. This limitation is due largely to the current software development practice. First, system requirements are typically described in documents that may contain ambiguous language. Second, as software ages and development teams change, information concerning specific decisions during the analysis and design processes may become more difficult to find. Third, different conventions may be used by different participating parties to describe software systems. Therefore, it is difficult to develop tools to interpret and integrate information from such disparate and wide ranging information. There exist, however, research projects currently investigating several of these issues with the intention of automating as much of the reverse engineering process as possible [5, 6, 13].

Finally, we found that those projects that involve significant domain-specific information or specialized areas of expertise, such as the use of control theory in the `Phase_Plane` project, require additional effort to capture the special information in the specifications and its corresponding documentation. This effort could be in the form of contacting the original authors, experts in the specialty area, or learning the necessary knowledge from archived sources, such as textbooks. However, once the appropriate information is captured in the requirements specifications, future maintenance tasks will greatly benefit from such documented knowledge.

4 Conclusions and Future Investigations

Using formal specifications and object-oriented analysis to describe the software that implements the `Phase_Plane` module of the Space Shuttle DAP has demonstrated that these complementary analysis and development techniques can be used for existing, industrial applications. The different levels of specifications, with increasing abstraction, supplemented by the OMT diagrams provided a means for integrating different types of information regarding the `Phase_Plane` module from disparate sources.

Having access to the formal specifications and diagrams will facilitate the verification that the original 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 is useful to the overall software development and maintenance processes of legacy (safety-critical) systems [8].

Future investigations will continue to refine the mid-level and high-level specifications and develop theorems to relate the levels of specifications. We will continue to investigate the use of automated techniques to reverse engineer specifications from code using a derivational approach [5]. Technical reports and other papers relevant to this project may be found by browsing the world-wide web site for the Software Engineering Research Group at Michigan State University, `http://web.cps.msu.edu/~chengb/serg.html`.

5 Acknowledgements

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

Reference herein to any specific commercial product, process, or service by trade, name, trademark, manufacturer, or otherwise, does not constitute or imply its endorsement by the United States Government or the Jet Propulsion Laboratory, California Institute of Technology.

References

[1] J. Rushby, "Formal methods and the certification of critical systems," Technical Report SRI-CSL-93-07, SRI International, Computer Science Laboratory, 333 Ravenswood Ave., Menlo Park, CA 94025-3493, November 1993. Available via anonymous ftp from `ftp.csl.sri.com`.

[2] P. G. Neumann and contributors, "Risks to the public," in *Software Engineering Notes*, ACM Special Interest Group on Software Engineering, 1993.

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

[4] S. Gerhart, D. Craigen, and T. Ralston, "Experience with Formal Methods in Critical Systems," *IEEE Software*, vol. 11, January 1994.

[5] G. C. Gannod and B. H. C. Cheng, "Facilitating the maintenance of safety-critical systems," *Int. J. of Software Engineering and Knowledge Engineering*, vol. 4, no. 2, pp. 183–204, 1994.

[6] M. Ward, "Abstracting a specification from code," *Journal of Software Maintenance: Research and Practice*, vol. 5, pp. 101–122, 1993.

[7] J. C. Kelly, R. G. Covington, and D. Hamilton, "Results of a formal methods demonstration project," in *Proc. of WESCON*, (Anaheim, California), pp. 62–66, September 1994.

[8] Jet Propulsion Laboratory, Johnson Space Center, and Langley Research Center, "Formal Methods Demonstration Project for Space Applications: Phase I Case Study: STS Orbit DAP Jet Select." D-11432, Jet Propulsion Laboratory, Pasadena, California, December 1993.

[9] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-Oriented Modeling and Design*. Englewood Cliffs, New Jersey: Prentice Hall, 1991.

[10] J. M. Wing, "A Specifier's Introduction to Formal Methods," *IEEE Computer*, vol. 23, pp. 8–24, September 1990.

[11] N. Shankar, S. Owre, and J. Rushby, "The PVS specification language and tools," technical report, Computer Science Laboratory, 333 Ravenswood Ave., Menlo Park, CA 94025-3493, 1993. Available via anonymous ftp from `ftp.csl.sri.com`.

[12] C. B. Jones, *Systematic Software Development Using VDM*. Prentice Hall International Series in Computer Science, Prentice Hall International (UK) Ltd., second ed., 1990.

[13] P. T. Breuer and K. Lano, "Creating specifications from code: Reverse-engineering techniques," *Journal of Software Maintenance: Research and Practice*, vol. 3, pp. 145–162, 1991.

A Description of Variables

This section contains descriptions of variables referenced in the body of the paper.

Name	Description
Inputs	
attitude_error	Body angle error
bypass	open-loop control, by axis, (1=open loop)
deadband	magnitude of deadbands
deltav_minimp	magnitudes of changes in vehicle angular rate due to 80ms RCS firing command
force_fire	rate damping flag
phase_plane_accel	magnitude of average control acceleration available for each axis, scaled for phase plane use
primary_vernier_sw	flag indicating that primary jets are being used for control
RCS	RCS (Reaction Control System) mode indicator
rot_jet_cmd	rotation command from previous cycle
rate_error	body angular rate error
rate_limit	magnitudes of rate error limits
undesired_accel	total undesired body angular acceleration
Outputs	
omega_e_desired	desired angular rate data
rot_jet_cmd	For primary: command to fire plus or minus jets or no jets. For vernier: command to fire plus or minus jets, no jets, or weighted "preference" for off-axis commands.

B Low Level Formal Specifications

This section contains the "as-built" specifications, where there is a direct correspondence to the DFD diagrams. Also included are the specifications describing the calculation for the boundaries (s1-s14) of the different regions of the phase plane. Figure 15 gives the Level 2 DFD for the `Phase_Plane`, where the control actions for five regions are calculated, the boundary values are explicitly calculated at this level. Notice that Figure 10 has one child diagram for the process labeled "region2." `Region 2` corresponds to the DFD shown in Figure 16. The control logic for `Region 2` is much more complicated and corresponds to a series of nested alternative statements. Region 2 is decomposed into three more regions, where the input values are made up of boundary values determined by $S11$, external input. This module generates a value for the thruster command (*rot_jet_cmd*). Figure 16 further refines `Region 2` to three more detailed regions.

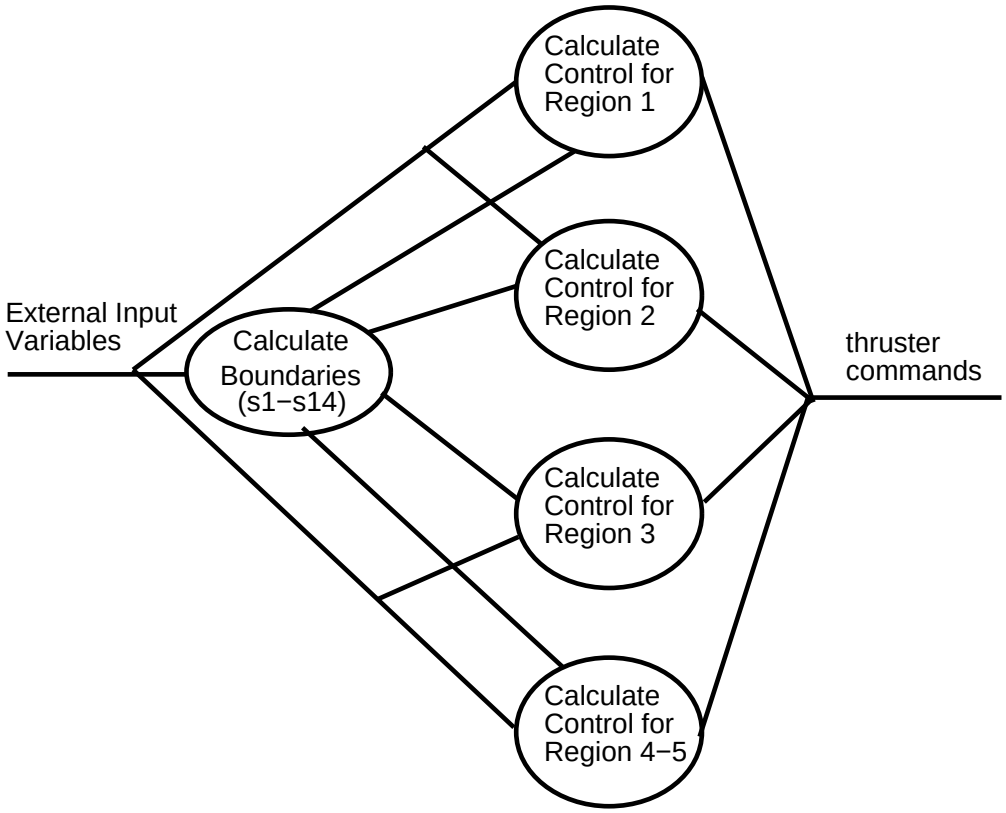

Figure 15: Level 2 DFD for `Phase_Plane`, detailing calculations of control actions for five regions

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                              %
%  Module:                  Phase Plane                                        %
%                                                                              %
%  Author:                  Betty H.C. Cheng                                   %
%                           Brent Auernheimer                                  %
%                                                                              %
%  Created On:              Tue June 22, 1993                                  %
%                                                                              %
%  Last Modified By:        Brent Auernheimer                                  %
%                                                                              %
%  Last Modified On:        Fri July 23, 1993                                  %
%                                                                              %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% This theory is a specification for regions as defined in Table 4.2.2.2.2.1
% in PVS for the Phase Plane component.
%
%
%
%              STS 83-0009D
%               OI-21
%         Dated: February 13, 1991
%
%
types: THEORY EXPORTING ALL BEGIN
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                              %
%           TYPE DECLARATIONS                                                  %
```

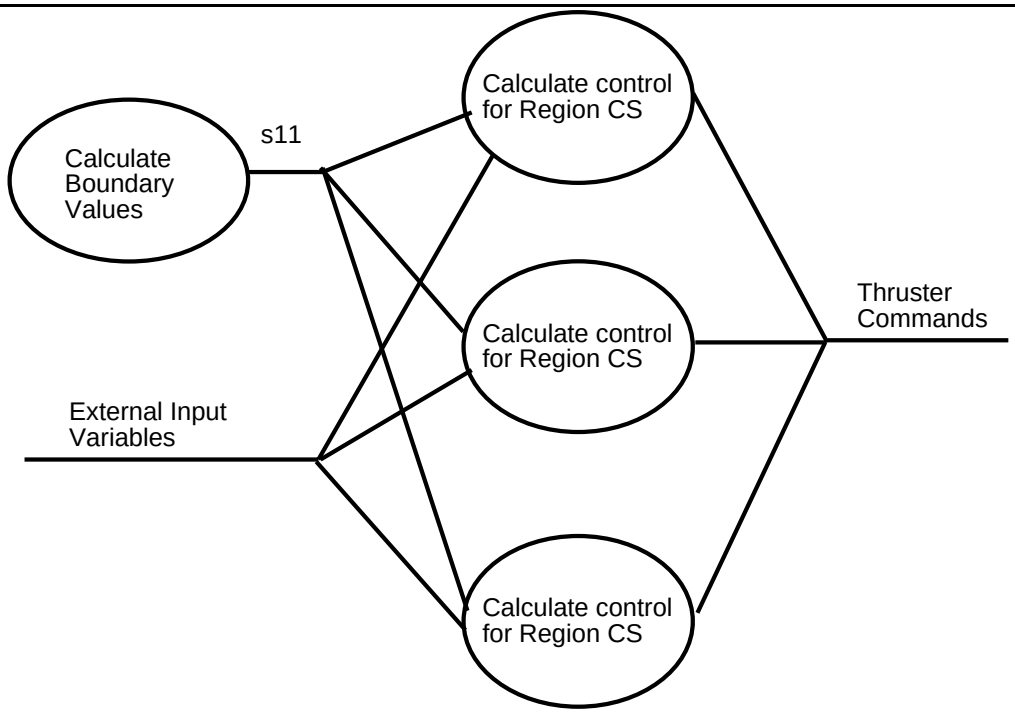

Figure 16: Level 3 DFD for `Phase_Plane`

```
%                                                 %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

  rotation:              TYPE = {roll, pitch, yaw}
  scalar_direction:      TYPE = {x: real | -1 <= x & x <= 1} CONTAINING 0
  scale_factor:          TYPE = {x:real | 0 <= x & x <= 1} CONTAINING 0
  rate_error_type:       TYPE = [rotation -> real] % units are deg/s
  attitude_error_type:   TYPE = [rotation -> real] % units are degrees
  deadband_type:   TYPE = [rotation -> real] % magnitudes of
                                                    % attitude deadbands
  desired_angular_rate:  TYPE = {z: real| -5 <= z & z <= 5} CONTAINING 0
  scalar_rotation_direction:
                         TYPE = [rotation    -> scalar_direction]
  undesired_ang_accel_type:
                         TYPE = [rotation -> real]
  phase_plane_accel_type:TYPE = [rotation -> real]
  deltav_minimp_type:    TYPE = [rotation -> real] % magnitudes of  changes
                                                  % in  vehicle ang. rate  due
                                                  % to  80 ms RCS firing command
  force_fire_type:       TYPE = [rotation -> bool]   % rate damping flag
                                                     %(from Rot_disc)
  rate_error_limit_type: TYPE = [rotation -> posreal]
  tuple_type:           TYPE = [scalar_direction,
                                desired_angular_rate, bool]

END types


i_loads: THEORY EXPORTING ALL BEGIN

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                 %
%        CONSTANT DECLARATIONS                    %
%                                                 %
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

IMPORTING types

% The following are from the I-LOAD Table (4.2.2.2.2-5)



wfrate: scale_factor    % Scale factor for off-axis vernier preference
whigh: real % upper target rate error for TVC  crossfeed
wlow: real % lower target rate error for TVC crossfeed
lrl_tvc: real % TVC lower rate limit from I-load table
kledge:  real            % don't know what this is - see page 4-183

END i_loads

%

nulls_and_undefineds: THEORY EXPORTING ALL BEGIN

% notused, null and undefined values used in the specification

notused: real % used in definition of s4
null:           real % null value
undefined:      real % null value

END nulls_and_undefineds


%

external_inputs: THEORY EXPORTING ALL BEGIN
IMPORTING types

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%    External Inputs:                                %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

        prev_rot_jet_cmd:
                    scalar_rotation_direction
bypass:      bool
u_d:         undesired_ang_accel_type
u_c:         phase_plane_accel_type
db:      deadband_type
force_fire:  force_fire_type
        rcs:         bool
omega_e:     rate_error_type
theta_e:     attitude_error_type
rl:          rate_error_limit_type   % magnitudes of rate error limits
primary_vernier_sw: bool
        omega_min:    deltav_minimp_type     % (local name) mag. of changes in
                                             % vehicle ang. rate due to 80 ms
                                             % RCS firing command

END external_inputs

%

utility_functions: THEORY EXPORTING ALL BUT non_neg_real BEGIN


  sign(x: real):  integer =  IF x >= 0 THEN 1 ELSE -1 ENDIF
  abs(x: real):   real = If x < 0 THEN -x ELSE x ENDIF

  non_neg_real: TYPE = {r: real | r >= 0} CONTAINING 0
  sqrt: [non_neg_real->non_neg_real]
  sqrt: AXIOM FORALL (x, y: non_neg_real): x*x = y IMPLIES x = sqrt(y)

END utility_functions
```

```
%

x_and_y: THEORY EXPORTING ALL BEGIN
IMPORTING types, external_inputs, utility_functions

% x1 and x2 are local variables used in Figure 4.2.2.2.2-2

x1(r: rotation): real = sign(omega_e(r)) * theta_e(r)
x2(r: rotation): real = abs(omega_e(r))

% y1 and y2 are local variables used in Figure 4.2.2.2.2-3

y1(r: rotation): real =  sign(u_d(r)) * theta_e(r)
y2(r: rotation): real =  sign(u_d(r))*omega_e(r)

END x_and_y

%

switching_lines: THEORY EXPORTING ALL BUT se, u_cp, kl, c BEGIN
IMPORTING types, external_inputs, utility_functions, i_loads,
          nulls_and_undefineds, x_and_y

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Specification of switching lines (Table 4.2.2.2.2-3)   %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% se is defined in the note at the bottom of Table 4.2.2.2.2-1
%

se(r: rotation): real = sign(omega_e(r))

% u_cp, kl, and c are defined in the note at the bottom of Table 4.2.2.2.2-3
%

u_cp(r: rotation): real = u_c(r)-sign(omega_e(r))*u_d(r)

kl:real = IF primary_vernier_sw  THEN  kledge ELSE 0 ENDIF

c(r:rotation): real =
                    IF abs(prev_rot_jet_cmd(r)) /= 1 THEN 125/100 ELSE 1 ENDIF

% note that the arguments to s1 need to be either x2 or y2
%
s1(s: [rotation -> real], r: rotation): real  = (s(r)*s(r))/(2*u_cp(r))+db(r)

s2(s: [rotation -> real], r: rotation):  real =
               (c(r)*s(r)*s(r))/(2*u_cp(r)) - (12/10)*db(r) - kl

s3(r: rotation): real  =   rl(r)

s4(r: rotation): real =
                 IF not primary_vernier_sw  THEN 8/10*rl(r) ELSE notused ENDIF

s5(r: rotation): real =
            IF not rcs
              THEN  lrl_tvc
              ELSIF not primary_vernier_sw THEN (6/10)*rl(r)
              ELSIF rl(r)-2*omega_min(r) >= (2/100) THEN rl(r)-2*omega_min(r)
              ELSE (2/100)
            ENDIF

% no s6 in the requirements

% the -1 and +1 are explicit to reflect "K" in the requirements
%
s7(r: rotation): real =
            IF y2(r) >= 0 THEN -1 * (sign(y2(r))*y2(r)*y2(r))/2*u_cp(r) -db(r)
            ELSE  (sign(y2(r))*y2(r)*y2(r))/2*u_cp(r) -db(r)
            ENDIF
```

```
% s8 is the negation of s3
%
s8(r: rotation): real =  -rl(r)


% no s9 in the requirements

s10(r: rotation): real = (c(r)*y2(r)*y2(r))/(2*u_cp(r)) + (12/10)*db(r) + kl

%
% the requirements for s11 imply a two step specification
%
s11_part1(r: rotation): real =
      IF   (((-12/10)*db(r)-kl <= y1(r)) & (y1(r) < -(1/2)*db(r))) OR (not rcs)
         THEN 0
         ELSIF (-(1/2)*db(r) <= y1(r) & (y1(r) <= s10(r))) & rcs
         THEN  -sqrt(2*abs(u_d(r))*(y1(r) + (1/2)*db(r)))+omega_min(r)
         ELSE undefined
      ENDIF

s11(r: rotation):  real =
             IF s11_part1(r)  > 0 THEN 0
               ELSIF s11_part1(r) < -rl(r) + omega_min(r)
               THEN  -rl(r) + omega_min(r)
               ELSE undefined %????
             ENDIF


% no s12 or s13 in the requirements

s14(r: rotation): real =  (y2(r)*y2(r))/(2*u_cp(r))+db(r)

END switching_lines

%

disturbance_hysteresis_logic: THEORY BEGIN
IMPORTING types, utility_functions, x_and_y, switching_lines,
          external_inputs

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                %
% These three function calculates disturbance regions defined in %
% Table 4.2.2.2.2-2 . These values are used to define Region 2   %
% as defined in Table 4.2.2.2.2-1,  and its output are values    %
% for rot_jet_cmd                                                %
%                                                                %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%
% Region CS %
%%%%%%%%%%%%%
region_cs(r: rotation): real =
    IF rcs THEN sign(u_d(r)) * wfrate
                             * ((s11(r) - y2(r))/(rl(r) + s11(r)))
    ELSE 0 ENDIF

%%%%%%%%%%%%%%
% Region HS1 %
%%%%%%%%%%%%%%
region_hs1(r: rotation): real =
    IF prev_rot_jet_cmd(r) = -sign(u_d(r)) THEN null
     ELSIF NOT rcs THEN 0
     ELSIF force_fire(r) THEN -sign(u_d(r))
     ELSE -sign(u_d(r)) * wfrate
                        * ((y1(r) - s11(r))/(rl(r) - s11(r)))
    ENDIF

%%%%%%%%%%%%%%
```

```
% Region HS2 %
%%%%%%%%%%%%%%
region_hs2(r: rotation): real =
    IF prev_rot_jet_cmd(r) = sign(u_d(r)) THEN null
     ELSIF NOT rcs THEN  0
     ELSIF force_fire(r)  THEN sign(u_d(r))
     ELSE  sign(u_d(r)) * wfrate
                       * ((s11(r) - y2(r))/(rl(r) + s11(r)))
    ENDIF



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                  %
% Main control logic for determining disturbance hysteresis regions%
%                                                                  %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
disturbance_hysteresis_p_p_regions(r: rotation): real =
  IF  (  s2(x2,r) <= y1(r)
       & y1(r) <= s7(r)
       & y2(r) >= 0
       & y2(r) <=s3(r))
   OR (  s14(r) <= y1(r)
       & y1(r) <= s10(r)
       & y2(r) >= s8(r)
       & y2(r) <= s11(r))
   THEN region_cs(r)

   ELSIF
      (  s7(r) <= y1(r)
       & y1(r) <= s1(x2,r)
       & y2(r) >= 0
       & y2(r) <= s3(r))
   OR (  s7(r) <= y1(r)
       & y1(r) <= s10(r)
       & y2(r) >= s11(r)
       & y2(r) <= 0)
   THEN region_hs1(r)

   ELSIF
         s7(r) <= y1(r)
       & y1(r) <= s14(r)
       & y2(r) >= s8(r)
       & y2(r) <= s11(r)
   THEN region_hs2(r)
   ELSE null
 ENDIF

END disturbance_hysteresis_logic

%
%
%

control_actions_by_region: THEORY BEGIN
IMPORTING types, external_inputs, switching_lines, disturbance_hysteresis_logic

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                 %
% Main function for control logic for phase plane calculations:  %
% yields three values: (rot_jet_cmd(r), wed(r), force fire(r))   %
%                                                                 %
% This function specifies Table 4.2.2.2.2-1.                     %
%                                                                 %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

control_actions(r: rotation):  tuple_type =

IF bypass THEN (prev_rot_jet_cmd(r), 0, force_fire(r))
%%%%%%%%%%%%
```

```
% Region 1 %
%%%%%%%%%%%%
ELSIF x1(r) > s1(x2,r)  OR x2(r) > s3(r)  THEN
        IF x1(r) <= s3(r)
         THEN (-sign(omega_e(r)), -se(r) * wlow, false)
         ELSE (-sign(omega_e(r)), se(r) * whigh,  false)
        ENDIF
%%%%%%%%%%%%
% Region 2 %
%%%%%%%%%%%%
ELSIF  s2(x2,r) <= x1(r)
     & x1(r) <= s1(x2,r)
     & x2(r) <= s3(r)

% disturbance_hysteresis_logic function specifies rot_jet_cmd value
% wed = 0
       THEN (disturbance_hysteresis_p_p_regions(r), 0,  false)
%%%%%%%%%%%%%%%%%%%
% Regions 3, 4, 5 %
%%%%%%%%%%%%%%%%%%%

ELSIF x1(r) < s2(x2,r)
%%%%%%%%%%%%
% Region 3 %
%%%%%%%%%%%%
      THEN IF x2(r) < s5(r) THEN (sign(omega_e(r)), se(r) * wlow, false)
%%%%%%%%%%%%%%%%
% Regions 4, 5 %
%%%%%%%%%%%%%%%%

           ELSIF s5(r) <= x2(r) THEN
%%%%%%%%%%%%%%%%%%%%%%%%
% Region 4, 5 (case a) %
%%%%%%%%%%%%%%%%%%%%%%%%
              IF x2(r) <= s3(r) THEN
                 IF primary_vernier_sw THEN (0 ,se(r) * wlow, false)
%%%%%%%%%%%%%%%%%%%%
% Region 4 (case b) %
%%%%%%%%%%%%%%%%%%%%
                 ELSIF s4(r) <= x2(r) THEN
                    IF prev_rot_jet_cmd(r) = -se(r)
                      THEN (prev_rot_jet_cmd(r), 0, false)
                      ELSE (se(r) * wfrate * ((((8 / 10) * rl(r))
                                   - x2(r)) / (2 / 10) * rl(r)), 0, false)
                    ENDIF
%%%%%%%%%%%%%%%%%%%%
% Region 5 (case b) %
%%%%%%%%%%%%%%%%%%%%
             ELSE % x2(r) < s4(r)
                IF prev_rot_jet_cmd(r) = se(r)
                   THEN (prev_rot_jet_cmd(r), 0, false)
                   ELSE (se(r) * wfrate * (((8 / 10) * rl(r)
                               - x2(r))/(2 / 10) * rl(r)), 0, false)
                ENDIF
           ENDIF
          ELSE (prev_rot_jet_cmd(r), 0, false)
          ENDIF
       ELSE  (prev_rot_jet_cmd(r), 0, false)
      ENDIF
    ELSE (prev_rot_jet_cmd(r), 0, false)
ENDIF

END control_actions_by_region
```