Validation and Verification of Formal Specifications in Object-Oriented Software Engineering

Steven A. Thomson

Validation and Verification of Formal Specifications in Object-Oriented Software Engineering

THESIS

Steven A. Thomson, FLTLT, RAAF

AFIT/GE/ENG/00S-01

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GE/ENG/00S-01

Validation and Verification of Formal Specifications in Object-Oriented Software Engineering

THESIS

Steven A. Thomson
FLTLT, RAAF

AFIT/GE/ENG/00S-01

AFIT/GE/ENG/00S-01

Validation and Verification of Formal Specifications in Object-Oriented Software Engineering

THESIS

Presented to the Faculty of the Graduate School of Engineering and Management
of the Air Force Institute of Technology
Air University
In Partial Fulfillment of the
Requirements for the Degree of
Master of Science in Electrical Engineering

Steven A. Thomson, B.Eng, Grad.Dip.Bus Admin
FLTLT, RAAF

September 2000

AFIT/GE/ENG/00S-01

Validation and Verification of Formal Specifications in Object Oriented Software Engineering

Steven A. Thomson, B.Eng, Grad.Dip.Bus Admin

FLTLT, RAAF

Approved:

[illegible] 18 Aug 2000

Maj. Robert P. Graham Jr.
Committee Chair
Date

[illegible] 18 Aug 2000

Dr. Thomas C. Hartrum
Committee Member
Date

Henry B. Potoczny 18 Aug 2000

Dr. Henry B. Potoczny
Committee Member
Date

Table of Contents

Page

List of Figures

Figure Page

List of Tables

AFIT/GE/ENG/00S-01

Abstract

The use of formal specifications allows for a software system to be defined with stringent mathematical semantics and syntax via such tools as propositional calculus and set theory. There are many perceived benefits garnered from formal specifications, such as a thorough and in-depth understanding of the domain and system being specified and a reduction in user requirement ambiguity. Probably the greatest benefit of formal specifications, and that which is least capitalized upon, is that mathematical proof procedures can be used to test and prove internal consistency and syntactic correctness in an effort to ensure comprehensive validation and verification (V&V). The automation of the proof process will make formal methods far more attractive by reducing the time required and the effort involved in the V&V of software systems.

It is commonly perceived that since a formal specification is written using strict mathematical notation, it is a minor task to ensure that the product does in fact meet the original specification and that the specification meets the end user's requirements. This is not the case. The majority of research in formal methods has delved into the development of formal notation and inference rules.

The emphasis of this research is the validation and verification of formal object-oriented (OO) specifications. This research identifies elements and properties of formally specified OO systems that should be proved and why, and implements such proofs using the theorem prover Z/Eves and process modeling tool SPIN. Proofs relating to the functional, dynamic and structural object models are addressed. The OO paradigm used during this research is that of Rumbaugh's Object Modeling Technique (OMT).

Validation and Verification of Formal Specifications in Object-Oriented Software Engineering

I. Introduction

A key element to force multiplication is technology. A technological edge over an opponent in conflict can be the difference between success and failure. As software intensive systems employed by the military continue to become more complex, it is apparent that the ability to formally specify future systems and comprehensively validate and verify them is of growing import to the defense community.

Formal methods are used infrequently today, but their wider application is envisioned "to lead to increased software quality and reliability. Moreover [it is expected that] early verification of specifications [will] increase specification quality, thereby reducing life cycle costs" [Fraser]. Rectification of errors during code maintenance is typically 60 to 100 times more costly than modification of the errant specification [Pressman].

Engineering can be thought of as the application of scientific approaches in order to solve technical problems. In more established engineering disciplines, the use of scientific approaches and formal processes is well established. Not so with software engineering, which is still in its relatively early stages of development. The use of formal methods in software engineering will help transform software engineering from its current state into that of a more structured and scientific approach to problem solution.

The use of formal specifications allows for a system to be defined with stringent mathematical semantics and syntax via such tools as propositional calculus and set theory. There are many perceived benefits garnered from formal specifications, such as a thorough and in-depth understanding of the domain and system being specified and a reduction in user requirement ambiguity. Formal specifications may be applied to systems in any domain, and are not necessarily limited to software specification.

Validation is intended to ensure that a system meets the user's requirements while verification refers to ensuring that a system meets its specification. V&V has in common with other facets of software engineering the fact that it should be carried out over the duration of a project and not just simply upon implemented systems. This is in contrast to most developed validation approaches for Knowledge Based Software Engineering (KBSE) [Meseguer] that perform validation almost entirely post completion.

Probably the greatest benefit of formal specifications, and that which is least capitalized upon, is that mathematical proof procedures can be used to "test (and prove) internal consistency and syntactic correctness" [Fraser] to ensure comprehensive V&V. The automation of the proof process will make formal methods far more attractive by reducing the time required and the effort involved in V&V of software systems.

1.1 Problem

The benefits of formal specifications are well understood in software engineering. How all of these benefits are realized is not as well understood. It is true that simply by attempting to formally specify a system and its domain we gain a more in-depth understanding by having to ensure that what could otherwise be an ambiguously written specification is a properly formed collection of definitions and axioms, but this argument provides only the tip of the iceberg of what benefits formal specifications offer us.

It is commonly perceived that since a formal specification is written using strict mathematical notation, it is a minor task to ensure that the product does in fact meet the original specification and that the specification meets the end user's requirements. This is not the case. The majority of research in formal methods has delved into the "development of formal notation and inference rules" [Fraser].

It is the intent of this research to investigate what elements and properties of a formal specification should be proved and why, and contrast this with what is typically proved in practice. This research will also deal with methods of implementation of such proofs.

The emphasis of this research is the validation and verification of formal object-oriented specifications. Proofs relating to the functional, dynamic and structural object models are addressed. One approach to be followed is the integration of existing tools into the AFIT Wide Spectrum Object Modeling Environment (AWSOME). The Air Force Institute of Technology (AFIT) has developed AWSOME in order to build software based upon formal specifications via semi-automated, correctness preserving transforms. Examples of tools that could be integrated with AWSOME include an object editor [Ashby], theorem prover or, for the dynamic model, a finite state machine analysis tool.

As examples of what could be proved, consider an object-oriented class with its requisite structural, dynamic and functional models as defined by the Object Modeling Technique (OMT) [Rumbaugh]. In the dynamic model we can ensure that states referenced are actually defined, and that there are no states for which the class is incapable of satisfying the guard conditions required for transition.

The functional model defines methods in terms of preconditions and postconditions. Functional model processes describe changes in the state of an object. They can be tested to ensure that invariant conditions are maintained after state transition, and that specified preconditions are satisfied whenever methods are called.

Verification of the structural model could include testing to ensure the concepts of inheritance and aggregation are implemented correctly, for example, ensuring that a subclass does not overload any attribute names used by its superclass(es).

1.1.1 Problem Statement. Propose a set of constraints via which formally specified object-oriented systems may be verified. Provide demonstrative examples of the applicability of the constraints to a formally specified OO system.

1.2 Scope

Research focus is upon formal specifications of object-oriented software emphasizing the investigation of existing techniques and addressing their shortcomings in order to propose new techniques. Illustrative implementation is achieved through verification of an object-oriented system modeled in AWSOME via the addition of a verification methodology to the AWSOME environment.

1.3 Approach

The approach is designed so as to address existing theories and practices in the proof of formal specifications, identify the key concepts of object-oriented analysis that require verification, and provide illustrative examples, preferably via integration of a commercially available, off-the-shelf (COTS) dynamic model verifier and COTS theorem prover.

The OO specification that this research uses was formalized using practices followed by AFIT's Knowledge Based Software Engineering (KBSE) group for the generation of formal specifications of object-oriented models [Hartrum].

Examples of proposed theories are implemented upon AWSOME analysis models as AWSOME caters to the creation of domain models from formal specifications via the use of the AWSOME Wide Spectrum Language (AWL). An AWSOME analysis model is represented and manipulated internally as an abstract syntax tree (AST).

The primary illustrative example used throughout this document is a modified version of a cruise missile system specified by a group of AFIT students. This model is contained at Appendix B.

1.4 Topics Addressed

The fields addressed in this document include object-oriented analysis, formal specifications, validation and verification, Z and an associated theorem prover Z/Eves, and the process modeling language Promela and an associated dynamic model checker Spin.

1.5 Products of This Research

The products of this research include:

- a list of formal constraints to which an object-oriented analysis model must conform in order to satisfy the definitions of correctness, completeness, and consistency defined in Chapter Two,
- an AWSOME analysis model visitor that generates Z specifications from the analysis model with integrated theorem prover commands (for the Z based theorem prover Z/Eves) to simplify the process of verification,
- an AWSOME analysis model visitor that generates Promela syntax process models from the dynamic characteristics specified. The visitor also integrates dynamic model checking commands for the Promela based process model verifier, SPIN,
- assistance in the creation of the AWL parser, and
- creation of a symbol table visitor for AWSOME ASTs that reduces ambiguity in specification component identification and forms the basis for further checking of AWSOME ASTs.

1.6 Document Outline

The organization of this document is as follows:
Chapter Two introduces theory and practices relating to formal specification and object-oriented analysis, Chapter Three proposes a methodology and constraints by

which an analysis model may be verified, Chapter Four details the implementation of the proposed methodology, and Chapter Five evaluates the methodology and implementation, and proposes opportunities for further research.

II. Background

This chapter introduces some fundamental concepts required in order to understand the motivation for, and how to formally specify, object-oriented systems and their subsequent validation and verification (V&V). The first section of this chapter provides an introduction to formal specifications and V&V prior to a discussion of common practices in industrial V&V today.

The second part of this chapter provides background required for an understanding of this research in particular and introduces the Z specification language, Promela modeling language, AFIT's wide spectrum language AWL, Rumbaugh's Object Modeling Technique (OMT) and representation of the OMT models using Z, Promela and AWL.

2.1 Formal Specifications

The purpose of a specification is to define all the characteristics that a system is to embody. Specifications should be defined in such a manner as to make each statement provable. A statement is deemed provable if its fulfillment by the generated product may be proven via formal logic or mathematical argument [Berztiss].

The more comprehensive and thorough that V&V testing is needed to be, the more detailed and precise the specifications are required to be. As such, without formally specifying a system, it is not possible to ensure that ambiguities do not exist, that the requirements are complete and free of errors, or that the test cases created are valuable.

Often a mixture of formal and semi-formal methods is used in the analysis and design of software systems with the amount of effort devoted to specification and verification being dependent upon the criticality of the particular software component. This practice highlights the tradeoff between component criticality and the cost of formal specification and verification.

Potter describes formal specifications of software engineering systems as including: (i) some specification of the input-output behaviour of the system (the relationship between preconditions and postconditions), and (ii) a description of how this behaviour can be effected [Potter]. This perspective is likened by Ghezzi to a "black box" and "glass box view" of the system respectively [Ghezzi]. The black box view of the system deals solely with the external requirements of the system's input/output behaviour whilst the glass box view dictates the internal machinations of the system.

The predominant areas of mathematics that create the formal foundation to OOA are logic, type theory, category theory, and process algebra [Goldsack]. The virtues of formal specifications come at a price. Increased specification complexity leads to increased difficulty and time required to create the specification. It is this balance that sees formal methods typically applied only to components of critical importance or the application of semi-formal methods that are not as robust or extensive [Gulch].

2.2 Validation and Verification

Validation and verification are the terms given to describe the process by which a product is evaluated to ensure that it meets the user's requirements (validation) and the specifications that it was based upon (verification). Verification is often thought of as "building the system right" [O'Keefe] and validation as "building the right system" [Meseguer]. The later that testing occurs in the software process, the greater the risk that errors will result in dramatic increases in cost and time to correct. Progressive V&V throughout the design process ensures that errors are detected as quickly as is practical, hence reducing the risk they offer to project budget and deadline [Brooks].

Formal specifications lend themselves well to V&V techniques. The precise and explicit nature of formal method descriptions may be used to generate comprehensive specifications. The characteristics that formal methods embody are not, however, all beneficial. Formal specifications are quite often cumbersome, time consuming and complex to create [Gulch].

V&V implementations should provide the software engineer with sufficient confidence in a system's correctness, completeness, robustness, precision, safety, and other quality attributes. Presently we find that many of these concerns in OO analysis and design are addressed only partially or by indirect evidence only [Goldsack]. In order to rectify this situation, V&V methods require precise information about the task the system is required to perform [Pressman]. Formal specifications can provide a pivotal advantage in satisfying this need.

2.3 Current Practices in Verification

The most prevalent techniques for V&V in use today can be categorized into one of the following four groups:

1. Inspection,
2. Static Verification,
3. Empirical Testing, and
4. Empirical Evaluation.

Inspection techniques are employed to identify semantic errors and are typically performed by someone who is knowledgeable with respect to the problem domain—such as a domain engineer. A common problem with inspection techniques is that they are usually carried out by the person(s) responsible for the generation of the domain specification in the first place and they may therefore fail to detect errors. This failure to detect errors can usually be attributed to the fact that the possibility of a specific error has remained unconsidered, or that the domain engineers see what they think the specification says rather than what is actually being specified. Hence, it is wise for an independent expert, that is, one who is not involved directly in the code generation, to inspect the system. It must be noted that as the problem becomes more complex, the difficulty that a human expert experiences increases [Pressman].

Static verification searches the domain for anomalies [Meseguer 1992]. An anomaly in the domain relates to an axiom that suggests the possibility of an error within the specification. It is usual for an anomaly to be a contradiction of a general property of the domain such as an inconsistency. Static verification is quite often carried out by automated tools due to the complexity of the errors being checked for. The tools available for static verification vary in the comprehensiveness of their search for anomalies [Verdaguer]. Static verification tools are very much dependent upon the semantics of the specification language used and are therefore infrequently reused in other applications not specified with the same representation.

Opportunities for the application of static verification are frequent within AWSOME analysis models—each class has an invariant proposition that cannot be contradicted by any other proposition of the class. For example, the postcondition of a class method cannot result in a class attribute value being inconsistent with the class invariant, nor may a state invariant be inconsistent with respect to the class invariant.

Empirical testing attempts to check the system via the execution of test sets of data. That is, by injecting known preconditions (the test data) into the system, one is able to predict the correct postconditions and compare these with the actual results witnessed. In order for the testing to be comprehensive, every input that could potentially occur must be tested. This requirement results in an exhaustive set of input conditions to execute—an undertaking of immense proportions for almost any real world system. The lack in practicality of such a test set leads to creation of a finite test set that must carefully be determined—the most common methods of forming such tests sets are with functional and structural testing in mind.

Structural testing has been developed to test as many of the components of a system as possible—examples include the instantiation of as many objects as possible, or the proof of as many axioms as possible. Functional testing takes a more validatory approach by checking specified pre- and postcondition relations with what is observed at execution. The problem of deciding upon what constitutes a comprehensive test case is not the only one to be encountered; in some systems the "correct" behaviour is ill-defined and must be defined prior to test case formulation.

Empirical evaluation is testing that occurs in order to ensure that the system meets the user's requirements regarding specified qualities such as performance, maintainability, documentation, and acceptance. Such acceptance testing is conducted upon an operational system by the users and/or the designers and occurs either on site or under controlled conditions.

The least likely candidate techniques for adoption in this research are inspection testing and empirical evaluation testing. Inspection testing has a high dependence upon the specific domain and application. It is true, however, that although the actual implementation makes for a poor candidate, the underlying methodologies are to a great degree independent of the specification language and as such, are totally relevant to this thesis. Empirical evaluation, however, requires an implemented system to be performed upon, and as such is not as relevant to this thesis, which focuses upon verification of formal specifications.

It is important to note that simply because a proposition (axiom) is logically correct that the specification is still not guaranteed to specify the intended system. It is true that there exists a necessity for all axioms within a specification to be logically correct but this alone is not enough to ensure that the system is in fact verified.

The majority of verification tools can only be applied to systems that have been implemented in direct comparison to systems that have only been formally specified.

Tools that require an implemented system in order to function leave the verification until far too late in the process model, thus leading to increased rectification costs.

2.3.1 Definition and Goals of V&V for this Research.

2.3.1.1 Correctness. Today, much controversy still abounds within the verification community as to what the definition of "correctness" is with respect to formally specified systems and the credibility of such proofs [Berg]. It is difficult to define specification correctness without a specification to provide a context. For the purposes of this research, correctness is defined as preservation of the analysis model's semantics, and that the analysis model is internally consistent and complete.

A useful definition for the correctness of an implementation of a specification is that the implementation be consistent with the specification.

2.3.1.2 Consistent. For the purposes of this research, in order to maintain internal consistency, it must be ensured that contradictory conditions are not capable of being true at the same time. That is, there should be no opportunity for contradictions to exist simultaneously. Consistency also requires postconditions of the analysis model and the specification to be equivalent for a given precondition. It is worth noting that the internal consistency of an analysis model is not related to the specification's consistency with user needs. It is outside the scope of this research to validate informal user needs.

For the purposes of this research, in order to maintain consistency, it must be ensured that conditions of the domain are not contradictory. That is, at no time may two conflicting expressions be satisfiable.

2.3.1.3 Complete. Completeness requires that the analysis model be free from ambiguity and conform to certain semantic properties specified in Chapter Three.

2.3.1.4 Satisfaction of Correctness Requirements. Note that these three objectives also form a subset of those proposed by Collofello : correctness, consistency, necessity, performance, and sufficiency (completeness) [Collofello]. Performance and necessity are both outside the scope of this research.

As a result of the above definitions, for an OO analysis model to satisfy the requirements for correctness, one must show the following:

1. That the constituent structural, dynamic and functional models are consistent and complete—that the specification is provably correct.
2. Execution of the specification (if the specification is executable) is correct if it is conformant (i.e. same output for a given input) with the expected output. This requirement is applicable solely to the dynamic model and is tantamount to an exhaustive state space search by a dynamic model checker coupled with selected use case test sets.

Not all specifications require formal semantics. Nor does a specification need to be entirely formal in order for verification to be applicable. Semi-formal specifications still have associated semantics that may be used as a basis for verification. Quite often, the complexity of the specification and verification of a system component are directly proportional to its perceived criticality. By the same token, the strength of the verification capable of being carried out is proportional to the formality of the specification.

2.3.2 Validation Testing. During post integration testing, the software system is verified to ensure that integration errors are detected and resolved so that a truly integrated system exists. At this point the final phases of testing may proceed. These final phases of testing culminate in the validation of the users' requirements. Pressman defines validation as ensuring the system functions as "reasonably" expected by the customer [Pressman].

This reasonable set of acceptance criteria should be stipulated within the original specification for the system and the validation testing conducted should be based upon the aforementioned criteria. Validation and verification should not necessarily be viewed as two entirely separate entities as the pair are rather interdependent. Some believe that the various levels of verification testing that occur throughout development, together with other testing methods are the only tools a validator requires to obtain maximum evidence as to correctness of the system [Meseguer].

Software validation endeavors to show through a series of tests that the system conforms with the users' needs. The tests conducted are typically black box in nature, that is, they are unconcerned with how the solution is attained, just that it meets the requirements specified [Pressman]. Soon after the specification generation occurs in the software process lifecycle, the types of tests to be conducted and the manner in which they are to be executed should be planned. Specific test cases should be defined that provide for thorough investigation of conformity with requirements. The aim of the test plan produced is to ensure all functional and quality requirements are satisfied.

Two possible results stem from each validation test conducted—either the test results are as expected from the specification, or a discrepancy is determined between the system's behaviour and that specified. Both the user and developers should address the importance of the deficiency and what needs to be done to rectify the situation.

One method by which to ensure our analysis model is conformant with requirements and that it is free from faults, is to execute either the specification or the product of the transformation (in this case an analysis model) [Collofello]. The theorem prover and dynamic model checker used in this thesis are both capable of simulating attribute values and thus meet Collofello's requirements for execution.

One should remain mindful of the fact that in order to mathematically ensure the validation of a system's qualities, those qualities must be formalized. It may be argued that it is the qualities (not just the functionality) of the system that drives its software architecture and that these quality attributes are satisfied via the selection of an adequate software architecture [Williams,Shock]. Assuming the architecture selection/transformation is verifiable, the quality attributes required for validation testing are thus outside the scope of this research.

Other methods of testing that complement verification and lead to system validation but are outside the scope of this research include [Meseguer]:

1. Functional testing—development of black-box input test cases to be compared to expected behavior,
2. Structural testing—path coverage—applicable in module, integration, and regression phases of testing,
3. Error oriented analysis—statistical analysis of erroneous system behaviour focused upon explaining error occurrence,
4. Hybrid testing—an amalgam of other testing techniques suited to the specific problem at hand,
5. Integration testing—type range, top down, bottom up, big bang and threaded,
6. Transaction flow analysis—a structured software design technique that analyzes a system based upon the transactions the system must process,
7. Failure analysis—determination of the exact nature and location of an error in order to correct the error, identify and rectify similar errors, and to initiate action to prevent occurrences of this type of error in the future,
8. Concurrency analysis—testing aimed at evaluating the performance of concurrent systems, and
9. Performance Analysis—dependent upon those qualities that are specified and provided with metrics to assess compliance.

2.3.3 Early Approaches to Verification. Early verification methods included:

1. Hoare Logic—a simple language consisting of assignment, statement sequences, whiles and if-then-elses. Each of these rules is interpreted by a proof rule. The assignment rule is an axiom, the other three are inference rules.
2. Dijkstra's Approach—he extended Hoare's logic by stressing the importance of the postcondition and expressed predicates in terms of the set{P}S{R} where R is the result of applying S to the precondition P. "Note that the [weakest precondition] WP method ensures total correctness. The significant difference between the two approaches is in the way they deal with looping, because the weakest-precondition approach has to ensure loops terminate" [Berztiss].
3. Mill's Functional Correctness—depends upon functions and relations, in contrast with the pre- and post-condition focus of the former methods.

Such approaches remain commonly accepted as being valuable [Meseguer]. Two avenues of verification exist for an algebraic specification—to verify the programs against the specification as it stands using Guttag's method [Guttag], or to transform the specification into a form that can be used with the inductive assertion method [Floyd].

2.4 Transformation Systems

The impetus for transformation systems is the maintenance of system specifications instead of source code. That is, the use of verified correctness-preserving transforms to generate source code from formal specifications results in the software engineer directing the majority of his efforts toward analysis, design and validation without the requirement for code generation as this facet of the software process becomes automatic, or at least semi-automated as depicted in Figure 1 [Balzer].

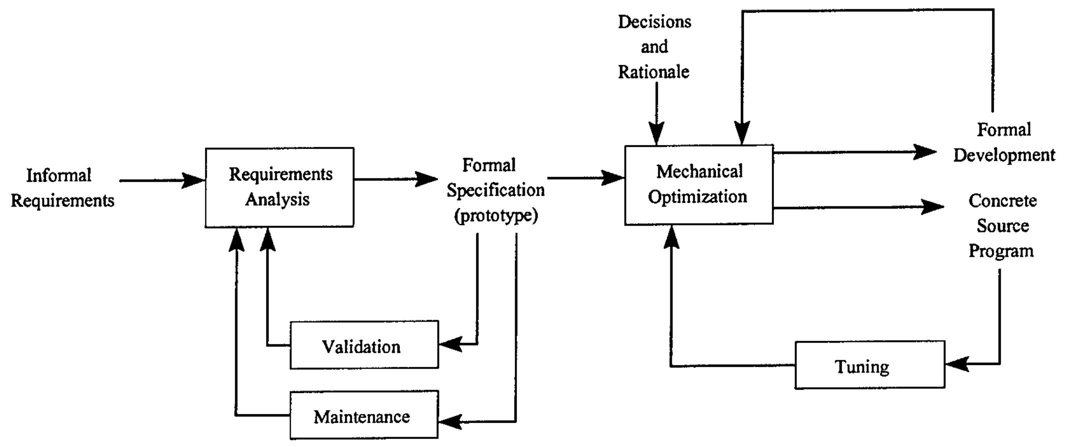


Figure 1. Automated Transformation Paradigm.

It is envisaged that the input elements to a transformation system originate from a repository of domain knowledge that is harvested for pertinent specifications as well as the generation of problem specific analysis that can be added to the repository for later reuse.

2.5 AWSOME

Until recently, proofs of concept relating to transformation system research at AFIT were demonstrated via the proof of concept tool—AFITtool. AFITtool consisted of a domain AST, specification AST and design AST as illustrated in Figure 2 [Hartrum]. The intent of the domain AST was to encapsulate domain knowledge relevant to a specific domain. The problem specification (analysis) AST was generated from the parsing of Z schemas (using Z-LaTeXsyntax) extended with state transition data [Hartrum, Bailor] that formalized the problem into the specification AST along with any harvested domain knowledge from the domain AST [Anderson]. Transformation of the specification AST resulted in the design AST that could then be transformed into executable code [Kissack, Tankersley].

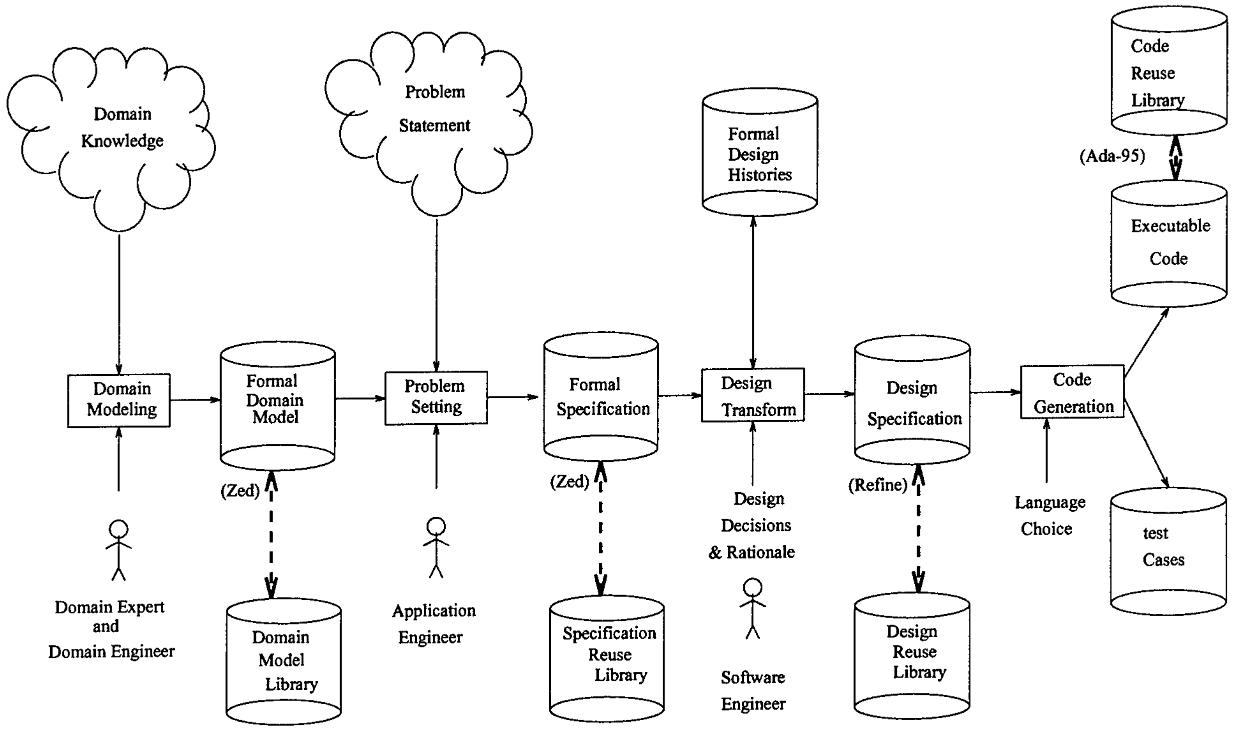
The Z representation of OMT used as input to AFITtool differs only slightly from that introduced in the implementation of this research described in Chapter Four. Differences occur in the representation of structural model associations and dynamic model transitions. Although kept to a minimum, extensions to Z were required to model the entire OMT analysis model as Z's ability to represent the dynamic model is insufficient [Hartrum,Bailor].

AFITtool was implemented in the high level language REFINE [Reasoning]. Noe integrated a commercial GUI object editor, Rational Rose, to the front end of AFITtool that somewhat simplified the formalization effort required to specify a system [Noe]. Rational Rose provides only a semi-formal ability to specify an object-oriented system and thus required augmentation via the addition of the ability to integrate axioms. These axioms conform with Z syntax to allow for straight forward parsing into the analysis AST.

The year 2000 has seen the rebirth of AFITtool as the AFIT Wide-Spectrum Object Modeling Environment (AWSOME). Essentially AWSOME is a redesign of AFITtool, a second generation prototype of a transformation system. The AWSOME tool consists of an analysis AST that contains a representation of the problem being modeled; this model may be semi-automatically transformed via verified transformations into a design model that addresses some details of the problem to a greater level of depth. This design AST forms the foundation of the code to be generated via the use of another set of verified transformations (see Figure 3 [Cornn]). Conceptually, this code may then be validated and any incongruities may be addressed at the specification level.

AWSOME makes use of a language developed within AFIT's KBSE group called the AFIT Wide-spectrum Language (AWL). This language evolved from the work done by Graham that resulted in the wide-spectrum language COIL [Graham]. AWSOME is based upon formal language theory, and as such is capable of the formal specification of object-oriented systems. AWL has been designed as a strict language hence it performs some of the verification effort and reduces specification ambiguity.

Being a wide-spectrum language, AWL is capable of representation of systems at the specification, analysis, design and implementation levels. The lower levels of



Formal Approach to Generation of Correct Domain-Specific Software.

Figure 2. AFITtool Transformation System.

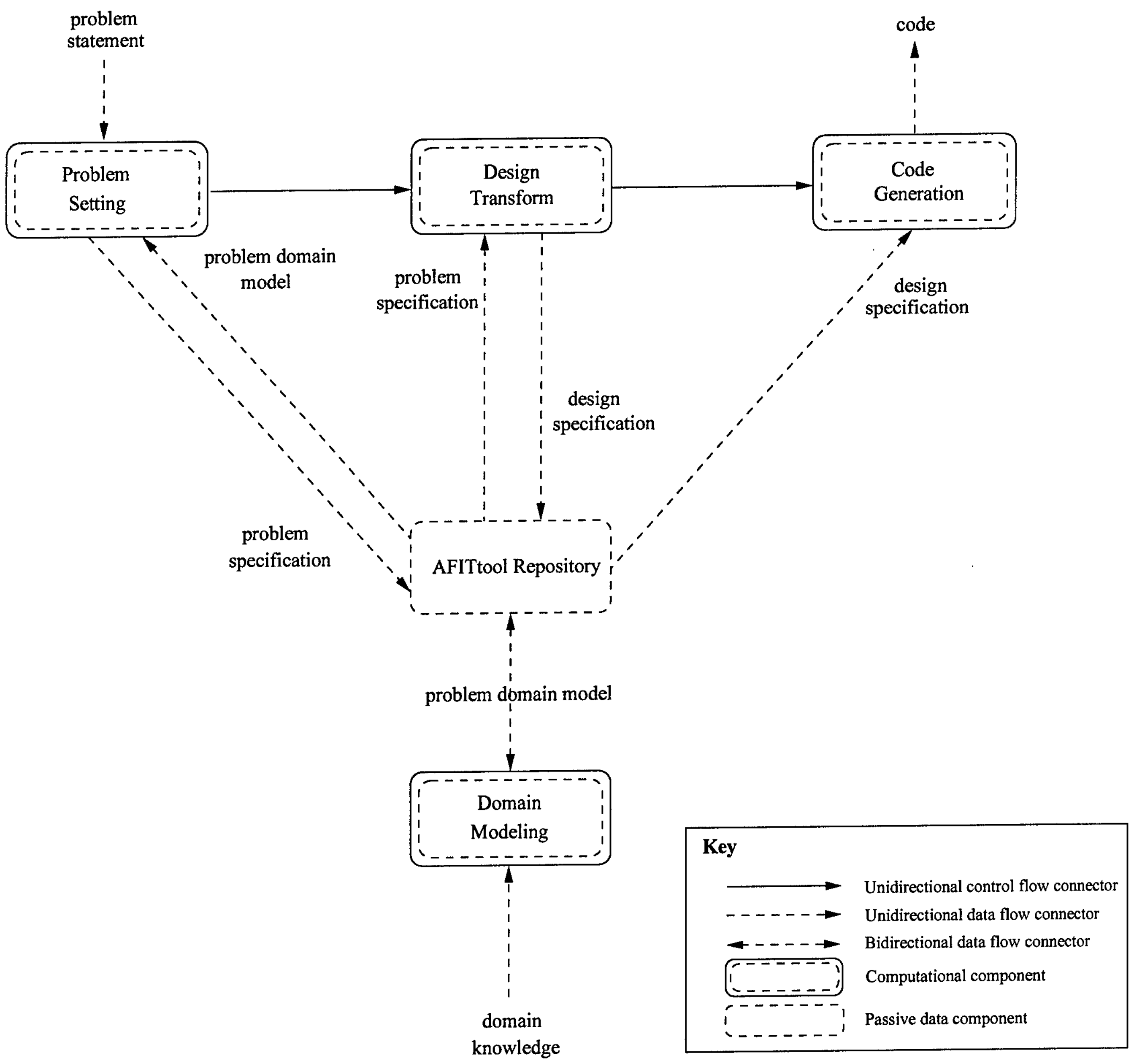


Figure 3. AFIT Wide Spectrum Modeling Environment.

AWL are not dealt with in this research as they are outside the scope of system analysis. AWL is capable of modeling both structural and object-oriented programming styles and has intentionally been kept independent of any other particular software language.

An example of the difference between AWSOME analysis and design ASTs is: at the analysis level, AWSOME represents class methods in terms of pre- and postconditions, therefore free of design decisions relating to any choice of algorithm while at the design level, these pre- and postcondition expressions are transformed into statements that form the body of the method.

The fact that any form of input specification, other than an AWL specification, must be transformed into an analysis AST means that a specification must conform to AWL semantic requirements. That is to say, there are certain productions by which an analysis AST is created using the AWSOME language and these production rules must be enforced by any other input media. This shall be elaborated upon in Chapter Three but suffice it to say here that it meets Berg's requirements for a specification language in that it:

1. is intuitively understandable to specifiers and validators and uses syntax that adheres closely to elements of the OMT,
2. has rigorous mathematical semantics, defined in set theory and propositional calculus,
3. is compatible with the structuring theory and formal methods to be used in this research,
4. engenders wide spectrum applicability and comprehensive expressive power.

2.5.1 Research Conducted by AFIT's KBSE. Research conducted at AFIT and implemented in AWSOME has included:

1. Tool integration, that is, the ability to integrate a number of object oriented domain models based upon their structural attributes into the one AST [Ashby],
2. Generation of relational schemas in the form of Data Description Language (DDL) from an AWSOME specification—affording the capability of persistent relational storage of object-oriented domain models and the transformation of class associations and postconditions to Data Manipulation Language [Buckwalter],
3. Information management in the form of a repository founded in object oriented database technology, giving AWSOME the capability to integrate numerous stand alone software synthesis tools into an integrated environment [Cornn],
4. Generation of executable code via the transformation of dynamic models into structural and functional components and its applicability to agent based systems [Marsh],
5. Semi-automated transformation of relational schemas to AWSOME ASTs [Pearson],
6. The proposition of a taxonomy of software architectures and a methodology for representing software architectures and styles in AWSOME [Williams], and
7. the work conducted as part of this thesis.

2.6 Rumbaugh's Object Modeling Technique

Rumbaugh's Object Modeling Technique (OMT) is the paradigm used to model domains in AWSOME. Using classes as the key foundation, OMT describes their attributes and characteristics via structural, functional, and dynamic models. For a description of AWL and how it represents OMT, refer to Appendix A.

2.6.1 Structural Model. The structural, or object, model represents the static structure of a system via the constituent objects of the system, the associations between those objects and the methods and attributes of each. Of the three models, Rumbaugh considers the structural to be the most important—this is due to the fact that in OOA, object identification is more important than early analysis of functionality [Rumbaugh]. It helps if object classes form intuitive components of the system being modeled, thus object selection is domain dependent. An object class consists of the attributes (data values) and methods (functions and transformations) inherent to the class.

Figure 4 represents the structural model of a cruise missile system while Figure 5 provides sample AWSOME code describing a portion of this model. The cruise missile is an aggregate class consisting of a propulsion system, airframe, warhead, and avionics software. The airframe, propulsion system and avionics software classes are themselves aggregates. The three vectors position, velocity and acceleration are examples of subclasses as they inherit the characteristics of the superclass vector. The cruise missile model is presented in its entirety in Appendix B.

Aggregation is a specific category of association. An aggregate class is one that is comprised of other classes and the aggregation association relates objects of the specific classes.

Inheritance allows one to model the similarities of certain classes whilst maintaining their differences. Take for example, the inheritance relationship between acceleration and vector. Vector is a class with three attributes of type real, namely x, y, and z. Acceleration is a type of vector and inherits the attributes and operations defined for vector but also extends upon those operations by inclusion of other operations specific to the acceleration class.

Conjecture abounds with respect to the semantics of inheritance [Rumbaugh, Booch, Booch et. al., Wegner, TaivalSaari, Alexander]. Zdonick proposes four different categories of inheritance [Zdonick]:

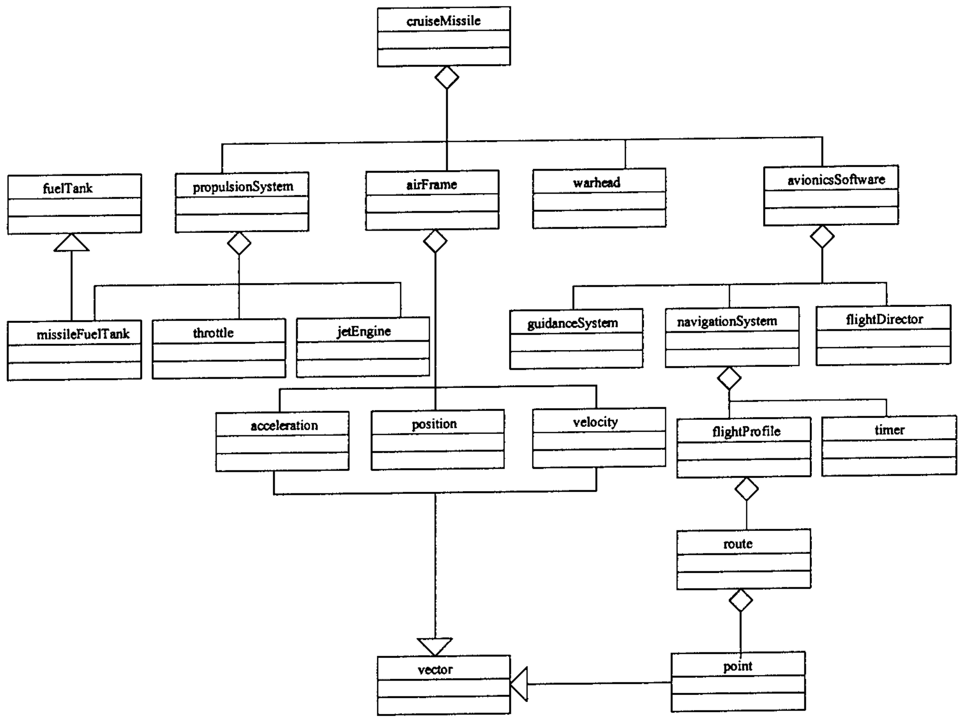


Figure 4. Cruise Missile Structural Model.

1. cancellation—allows for redefinition of class methods or even removal of methods from the subclass,

2. name compatibility—the subclass must preserve the set of names inherited from the superclass but is free to redefine them,

3. signature compatibility—the subclass must embody the syntactic interface of the superclass, and

4. behavior compatibility—the subclass may not modify the characteristics of the superclass. This form of inheritance is termed strict inheritance and ensures that the child class is substitutable for the parent class.

Strict inheritance is the sole form of inheritance referred to within this research.

```
package cruiseMissile is

class fuelTank is
    private fuelLevel : bigReal;
    private outputFlowRate : bigReal;
end class;

class missileFuelTank is fuelTank with
    private fixedWeight : realWeight;
    invariant fixedWeight = tankWeight + (fuelDensity * capacity / 2)
       and inputFlowRate = 0
end class;

class navigationSystem is
    private navState : navStates;
end class;

class flightProfile is
    private timeOnTarget : time;
    private flightPath : route;
end class;

class warhead is
    private weight : mass;
    private munitionType : string;
    private explosiveForce : yield;
    private armed : boolean;
    invariant weight > 0.0 and explosiveForce >= 0.0
end class;

class airframe is
    private pos : position;
    private accl : acceleration;
    private vel : velocity;
    private afState : afStates;
    private heading : real;
    private elevation : real;
    invariant heading <= 2*pi  and heading >= 0.0 and elevation <= pi/2
       and elevation >= -pi/2
end class;
```

Figure 5. Partial AWSOME Language Representation of Cruise Missile Structural Model

```
class propulsionSystem is
    private fuelFeed : throttle;
    private engine : jetEngine;
    private tank : missileFuelTank;
    invariant (tank.fuelLevel = 0.0 => fuelFeed.maximumFlowRate = 0.0)
       and (tank.fuelLevel > 0.0 =>(fuelFeed.maximumFlowRate =
       engine.maximumFuelFlowRate)) and (engine.currentFuelFlowRate =
       fuelFeed.actualFlowRate)
end class;

class cruiseMissile is
    private propulsion : propulsionSystem;
    private frame : airframe;
    private payload : warhead;
    private avionics : avionicsSoftware;
    private cmState : cmStates;
end class;
end package;
```

Figure 5. Partial AWSOME Language Representation of Cruise Missile Structural Model cont.

2.6.2 Dynamic Model. The dynamic model represents the temporal relationships between functional components of the domain model. The dynamic model illustrates what will happen when certain conditions (guards and received events) hold irrespective of how it will happen. The dynamic model also describes the behaviour states by which a class is defined. Figure 6 illustrates the Mealy model representation of the class Airframe while Figure 7 gives the corresponding AWL syntax.

A state may contain an invariant condition but all actions and events occur upon transitions. Transition syntax in AWL is:

```
<IN> currentState <ON> receiveEvent [<IF> guard] [<DO>action]
     [<send> (sendEvent)*]} to nextState
```

With reference to Figure 6, if the airframe is in the state "powered flight" and it receives the event "change course" and the guard condition "true" is satisfied, then the airframe will transition to the state "maneuvering" until it receives a "doManeverComplete" event.

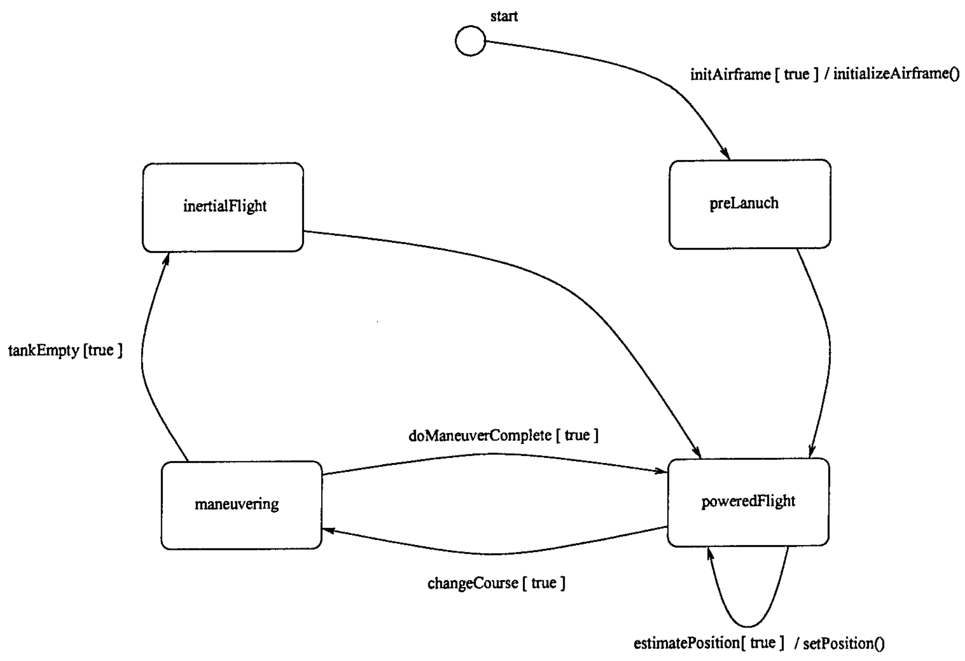

start
initAirframe [true] / initializeAirframe()
inertialFlight
preLanuch
tankEmpty [true]
doManeuverComplete [true]
maneuvering
poweredFlight
changeCourse [true]
estimatePosition[true] / setPosition()

Figure 6. Airframe Dynamic Model.

2.6.3 Functional Model. The computations or transformations of data that occur within classes are represented by OMT's functional model in the form of data flow diagrams. The functional model is hierarchical in structure; that is, each process may be further refined by intermediate levels of detail. At the lowest level of abstraction, the processes of the functional model are termed leaf operations. The functional model does not describe how transformations occur or when they take place, it simply identifies the inputs and outputs of those processes. Figure 8 represents the **calculateAcceleration** leaf operation of the airframe functional model while Figure 9 gives the AWSOME syntax for the missileFuelTank method **changeFlow**.

```
dynamic model is
event initAirframe ();
event doLaunch();
event tankEmpty();
event estimatePosition();
event changeCourse();
event maneuverComplete();
event outOfFuel();

state start invariant afState= startairframe;
state preLaunch invariant afState = preLaunchairframe;
state poweredFlight invariant afState = poweredFlightairframe;
state maneuvering invariant afState = maneuveringairframe;
state inertialFlight invariant afState = inertialFlightairframe;

transition table is
    in initial on initAirframe if true to airframeInit;
    in airframeInit on AUTOMATIC if true send initDone();
       to poweredFlight;
    in poweredFlight on tankEmpty if true to inertialFlight;
    in poweredFlight on getPosition if true send positionCurrent();
       to poweredFlight;
    in poweredFlight on changeCourse if true to maneuvering;
    in maneuvering on doManeuverComplete if true to poweredFlight;
end transition table;
end dynamic model;
```

Figure 7.   Airframe Dynamic Model in AWSOME syntax

2.7 Z

Traditionally, the formal specifications entered into AFITtool took the shape of Z schemas [Hartrum, Bailor]. In 1997, Noe created a set of automated transforms that generated Z specifications from augmented UML diagrams created with the CASE tool Rational Rose. The formal language Z is founded in mathematics such as propositional calculus and set theory. The language is far more extensive than the subset that is capable of being parsed into the AFITtool domain model.

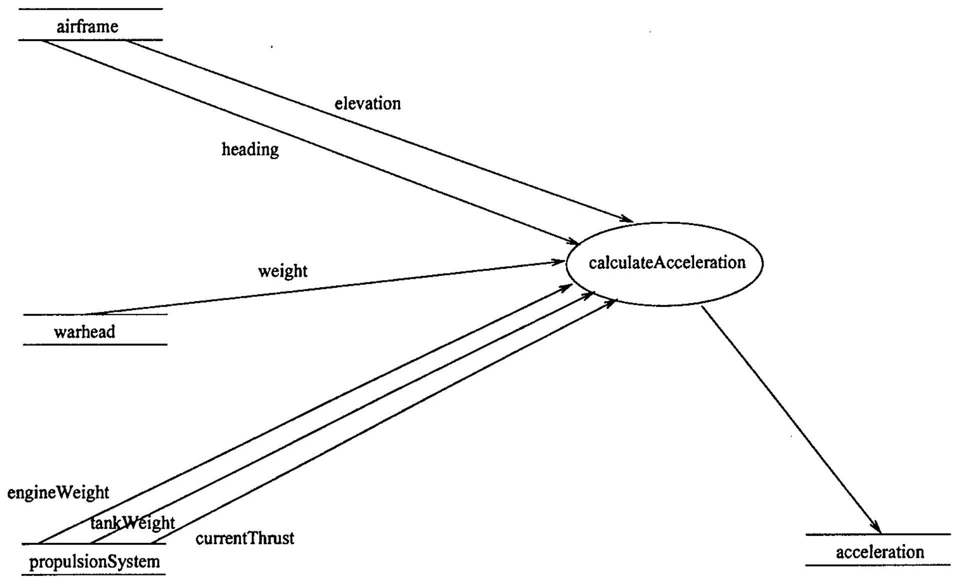


Figure 8. Airframe Functional Model.

```
private procedure changeFlow(actualFlowRate : in bigReal)
guarantees outputFlowRate' = actualFlowRate? and
    fuelLevel' = fuelLevel and capacity' = capacity and
    tankWeight' = tankWeight and fuelDensity' = fuelDensity
```

Figure 9. Missile Fuel Tank Method—changeFlow()

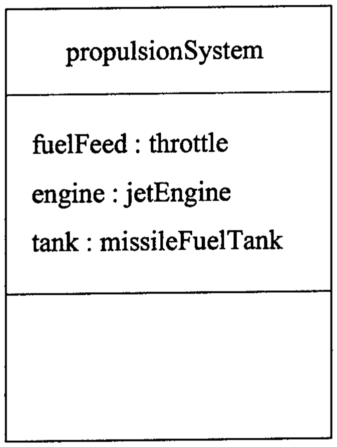


Figure 10. PropulsionSystem Object Model.

propulsionSystem

$fuelFeed : throttle$
$engine : jetEngine$
$tank : missileFuelTank$

$tank.fuelLevel = 0.0 \Rightarrow fuelFeed.maximumFlowRate = 0.0$
$tank.fuelLevel\, 0.0 \Rightarrow fuelFeed.maximumFlowRate =$
$engine.maximumFuelFlowRate$
$engine.currentFuelFlowRate = fuelFeed.actualFlowRate$

Figure 11. PropulsionSystem Schema.

The building block of Z is the schema. This is comprised of a signature and predicate. The signature essentially introduces the attributes or variables of the schema whilst the predicate stipulates the axioms that define those attributes.

Schemas may be used to represent classes, operations, events, and states. Type definitions are detailed via axiomatic definitions, operations by dynamic schemas, and structural components via static schemas. An example structural model for the class cruise missile is located at Figure 10. Note the inability of the OMT model to convey invariant constraints that are represented in the same class when expressed as a Z schema—Figure 11. This is an example of how a semi-formal specification system must be augmented with prose in order to provide the requisite formalization.

Z has been used in a number of object-oriented analysis models but it is noted that Z is not ideal as an OO specification language due to its semantic complexity. In order to achieve a sufficient level of formalism, Z++, ZSPECK, Schuman+Pitt, OOZE, MooZ, and Object-Z have extended Z substantially [Goldsack, Stepney]. None of these versions of Z were adopted for this research due to the fact that by their very nature, their use is not supported by existing Z-based theorem provers and does not conform with Spivey or ISO standard Z[Stepney].

It is the author's opinion that it is better to use multiple analysis representations and verifications tools that are best suited to certain portions of the analysis model than to use an unsupported tool that extends a specification language. If proper integration can be achieved then the user is none the wiser.

2.8 OOA and Verification

During the course of this research, it was discovered that literature dealing with the formal specification of OO systems and their validation and verification (as opposed to either their V&V or formal specification) is rather rare and difficult to come by. By far the majority of OO specification literature is not formally developed to the extent that stringent verification methods could be used against analysis models specified in accordance with such methods.

Extrapolating from Bertziss' research dealing with the verification of abstract data types [Bertziss], it can be proposed that three tasks must be performed in order to verify the correctness of a class:

1. Determination of whether an implementation is going to have certain properties prior to implementation. This comes from knowledge of the method by which the specification is generated and the transformation process. This task is predominantly conducted via inspection by experts but the structure of an analysis AST described by AWSOME's metamodel ensures certain semantic

constraints conforming to this rule such as what may form components of the structural, functional and dynamic models,

2. Demonstration of completeness and consistency of the specification,

3. Implementation should be provably consistent with its specification—or more readily for the AWSOME transformation system—transforms should provably maintain the correctness of the modeled system.

2.9 Software Process Model

Many software engineering process models exist, and until recently, many of those have had little or no avenue for redressing faults until code has been generated as it is the code that is tested to ensure compliance with the specification. Therefore, many of these models attempt to output code quickly and incrementally.

As already pointed out the cost and time required to rectify an error during code maintenance is far greater than modification of a specification. The process model proposed below is a modification of that described by Berg [Berg] such that it facilitates an iterative design, thus allowing for faster detection and rectification of errors and does not make the tenuous assumption that an entire system can successfully be formally specified within a single iteration. It is worth noting that a transformation system such as AWSOME readily facilitates iterative processes due to the fact that as soon as the specification is modified, the product that is to be verified and validated is instantly available.

The modified software process model is as follows:

1. Establishment of user requirements,

2. User requirements are specified in accordance with a formal language resulting in a formal specification that is provably correct as discussed in Chapters Three and Four,

3. The specification is then validated to show that it satisfies the goals of development. This is typically performed by inspection but may also be augmented by automation (specification interpreters). Tools such as these permit input values to be injected into functions in a specification and then return the associated output values that the specification defines, thus allowing for validation via inspection. Note that validation can also include quality attributes that are difficult to formalize and as such are outside the scope of this research.

4. After satisfying the requirements for the properties of being well formed and validated, the specification is used to guide the implementation of the system. In AWSOME, this correlates to driving the requisite transformations to generate code.

5. Berg states that the code is then verified by proofs of correctness that ensure adherence to the validated formal specification. These proofs are typically heavily automated but our previous step that implemented verified transforms makes this a redundant phase of the software development process.

6. Berg then suggests a final testing phase to double-check the proofs. The test strategies are developed to (work well from) the specifications.

7. If any deficiencies are highlighted by validation then repeat the process.

2.10 Relation Between Background and This Research

Many of the concepts introduced in this chapter are used in the methodology proposed as part of this research. Essentially, the methodology put forward is best described as a hybrid—the dynamic, structural and functional models of OMT are better suited to different forms of verification testing than a single technique.

All three OMT models embody characteristics that are suited to static verification—these form the backbone of tests that ensure consistency of the analysis model, such as invariant constraint consistency, and functional pre- and postcondition analysis similar to the work done by Dijkstra.

The dynamic model is well suited to both forms of empirical testing. An exhaustive search of the state space corresponds to structural empirical testing and is relevant to state reachability analysis. The injection of a use case test set into a dynamic model corresponds to empirical functional testing and may be used to validate the dynamic behavior modeled by the specification.

The completeness of an analysis model relies heavily upon the semantics of AWL. The productions rules, type checking and name analysis that occur as part of parsing specifications into an AWSOME analysis AST enforce a great deal of structure that will result in notification of an error to the software engineer if neglected. For example, if a user attempted to define a dynamic model transition without a current state or a next state, the parser would not accept the declaration as it is not complete with respect to the production rule for an AWSOME transition—refer to Appendix A for a list of AWSOME productions.

The next chapter details proposed constraints upon the analysis model that must be enforced as part of the V&V of an object oriented formal specification generated in AWSOME.

III. Methodology

3.1 Introduction

The research conducted focuses upon the aspects of a formal specification that can be checked within the analysis AST itself. Any verification that occurs internal to the analysis AST is independent of the method by which the problem is formally specified. Therefore, if a new method of specification is implemented, the verification methods proposed in this research will continue to be applicable. To put it another way, no matter the specification method—be it Z schemas, a GUI object editor or the AWL—these methods will remain applicable without modification as they function upon the analysis AST itself.

The verification issues addressed by this research apply predominantly to specification consistency, correctness, and completeness as defined in Chapter Two. The problem of verifying correctness, consistency, and completeness has been approached from six different but interrelated avenues. These perspectives spawn from a combination of the granularity of the analysis performed, that is whether the analysis is at the inter or intra-class level, and the focus of the analysis—whether it be the structural, dynamic or functional model of the system. Figure 12 illustrates this breakup as a table, it can be seen that one axis corresponds to the three OMT class models [structural, functional, dynamic] and the other corresponds to the level of abstraction [class, domain]. These delineations were made so as to break the problem into a more manageable size.

By automating some or all of the verification process, the amount of effort required by the specifier for V&V is reduced. This reduction in the effort required to formally verify a system increases the value of formal specifications with respect to other methods of V&V such as inspection. It must be noted, however, that in many cases other than the rules proposed subsequently in this chapter, a test plan is

	Class Level	Domain Level
Structural Model		
Dynamic Model		
Functional Model		

Figure 12. Analysis Model Perspectives.

required that has certain proof goals that provide direction to the verification effort. Verification without a clear plan and direction is far from an optimal solution.

It is the aim of this chapter to identify and explicitly define well-formed constraints to which an object-oriented analysis model must conform. These constraints are introduced according to the six categories detailed above and are first given a textual description and then a formal definition in propositional calculus. These constraints are described without regard for their testability at this stage of the document as it is the constraints themselves that direct object-oriented analysis and thus form the focus of this research and not their automated testability.

3.2 Definition of the OO Analysis Domain Model

To assist in formalization of the analysis model constraints, definition of the domain and its components must be made. Below is a conceptual model of the domain for which AWL provides a concrete surface syntax.

A *domain* consists of the tuple: *Classes, Types, Assocs, AssocObjs, Consts* where Classes is the set of classes present in the domain, Types is the set of data types of the domain, Assocs is the set of associations within the domain, AssocObjs is the set of associative objects, and Consts is the set of global constants.

For the purposes of this document, a *type* is defined as type = *name, inv* where inv is the type invariant. This is an extremely simplistic representation of the capabilities of the AWSOME language to model types, but is sufficient for the rules to be defined.

A *class* is defined as the tuple: *name, inv, Attrs, Ops, Trans, States, Events* where name is the class identifier, inv is the class invariant expression, Attrs is a set of attributes, Ops is a set of operations defining the functional model, Trans is a set of transitions describing the dynamic behavior of the class, and States is a set of states that define the class behavior.

An *expression* is a well-formed boolean or arithmetic expression capable of being proved correct.

Class *attributes* consist of a name and a type, i.e., attribute = *name, type, value* where name is the attribute identifier and type is the data type of the attribute. *Constants* and *parameters* are also represented by the tuple *name, type, value.* Note that AWSOME provides greater depth to the modeling of attributes and parameters such as whether a class attribute is *public* or *private* and whether the mode of a parameter is *in*, *out* or *in and out.*

Data objects are also defined by the tuple *name, type, value.* The function dataSet(expression) returns the set of data objects referred to in an expression. For the sake of dataSet(), there is no difference between a variable's ticked and unticked references. It is worth noting that data object is a generalization of both attribute and class, and as such both inherit the tuple *name, type.*

Operations are defined by the tuple *name, pre, post, Params* where pre and post are the pre- and postcondition expressions of an operation respectively, Params is the sequence of parameters of the operation. Operation calls are invocations of operations and consist of a reference to the operation to be invoked and a sequence of arguments that represent the input parameters of the operation, i.e., operationCall = *name, Args.* For the purposes of this research, at the analysis level, all operations are representative of procedures as opposed to functions.

Transitions are defined by the tuple transition *current, receive, guard, action, Sends, next.* Current and next refer to the current and next states of the transition respectively. The guard condition of a transition is an expression, the action is a call to a class method, receive is the event that triggers the transition, and Sends is the set of events sent as a product of the transition. States are comprised of an identifier and an invariant expression, i.e., state = *name, invariant.* Events consist of a name, a set of arguments (data objects), and an invariant expression, i.e., event = *name, Args, inv.*

An *association* is represented by the tuple: association = end_1, end_2 where end_1 and end_2 refer to the identifiers of the two classes that constitute the association. Although AWSOME is capable of representing associations of a higher order than binary, this research is limited to binary associations due to the greatly increased complexity associated with verifying ternary and higher associations. Associations modeled in AWSOME are more complex than presented here—each end of an association has a role name and a cardinality but these are irrelevant to the constraints proposed in this chapter.

Associative objects are comprised of an association, and operations and attributes particular to the associative object and are defined by the tuple:
assocObj = *name, Attrs, Ops, assoc* where name is the associative object identifier, Attrs is the set of attributes, Ops the set of operations, and assoc is the binary association.

3.3 Class-Level Constraints

The propositions of the following class-level constraints are assumed to relate to an instance of a class **this** of a domain **dom**, i.e., this $\in$ dom.Classes.

3.3.1 Structural Model. Structurally speaking, classes consist of attributes and their invariant predicates that dictate certain characteristics of their behavior.

```
class airframe is
    invariant heading <= (2*pi)  and heading >= 0.0 and elevation <=
        (pi/2) and elevation >= (-pi/2)
    private pos : position;
    private accl : acceleration;
    private vel : velocity;
    private afState : afStates;
    private heading : real;
    private elevation : real;
end class;
```

Figure 13. Elements of the Cruise Missile Structural Model

It is these components that are of concern at the class level of verification as well as ensuring the consistency of any subclasses with their respective superclasses.

Constraint 1 *Attributes Must be Declared Over Defined Types*

The definition of a data type declares the range of meaningful values of that type. Hence it is important to know that each data object is an instance of a data type in order to ensure that the context of any reference to the data object is consistent with its range of values. It is therefore imperative to ensure that each data object is in fact defined over an existing data type.

$$\forall a : attribute \bullet a \in this.Attrs \Rightarrow a.type \in dom.Types \cup dom.Classes$$

As an example, Figure 13 contains the structural portion of the specification for the class **airframe**. This class has the private attributes pos : position, accl : acceleration, vel : velocity, afState : afStates, heading : real, and elevation : real.

The first three attributes refer to other classes, and as such, are aggregate components of an airframe while the remaining attributes are all of types declared within the domain. If any of the attribute types are undefined within the domain, then the model fails to be complete.

Constraint 2 *Any Variables Referenced Within an Object's Invariant Proposition Must be Constants or Attributes of the Object*

In accordance with the object-oriented principle of data-abstraction, data modification or interrogation may only be performed by the class that is responsible for that data's abstraction, or via methods provided by the class. Therefore, a class is only able to reference its own variables and static values of the domain directly.

$\forall n : dataObject \bullet (n \in dataSet(this.inv) \Rightarrow (n \in dom.Consts) \vee (n \in this.Attrs)$

The invariant of the class airframe, represented in Figure 5, refers to heading, elevation, and pi. Heading and elevation are both of type real and attributes of the class while pi is a real constant of the domain.

Constraint 3 *Pre- and Postconditions Must be Consistent With the Class Invariant*

Traditionally, AFITtool has used schema inclusion to imply method pre- and postcondition consistency with the class invariant. This research proposes explicitly ensuring pre- and postconditions do not contradict the class invariant. While logically equivalent to the former method, the latter ensures that the constraint holds rather than simply implying it holds.

$\forall op : operation \bullet op \in this.Ops \Rightarrow (this.inv \wedge op.pre \wedge op.post) \neq false$

Constraint 4 *Invariant Propositions Must be Consistent With Respect to the Types Over Which They Refer*

Data types embody constraints upon the values that a variable of a given data type may have. Consequently, any expression that refers to a class attribute or global constant of such a data type must remain consistent with its constraint. That is, values must remain within the attribute's range and operators must have some associated semantic for the type(s) they are applied to.

This constraint may be separated into two lesser constraints:

1. the invariant of a class must not contradict its attribute type invariants, and

2. invariants must be made up of type compatible operators and operands (elaborated upon as Constraint 9).

The first constraint requires that the class invariant hold over all attributes of the class and is expressed as:

$\forall n : dataObject \bullet a \in this.Attrs \wedge n \in dataSet(this.inv) \wedge n.type.inv \wedge this.inv \neq false$

For example, Figure 14 contains specifications for the enumerated type **flightDirectorStates** and the class **flightDirector**. The state invariants of flightDirector dictate the value of the variable **flightDirectorState**. This constraint states that the state invariants must be consistent with respect to the attribute types to which they refer, and as such, the invariant must not contradict the invariants of the aggregate components. Inspection suggests that there is no conflict between the type and the state invariants as only those values enumerated in the type declaration are ever referenced, that is, the state invariants never conflict with flightDirectorStates' range of allowable values.

This constraint does, however, raise the complex issue of aggregate component visibility within the analysis model. Conceptually, the principles of data abstraction and information hiding mean that classes only have direct access to their own attributes—therefore an aggregate class does not have direct access to its subclass attributes. It is worth noting that all class attributes have been defined private in the cruise missile example specification in order to strictly adhere with the object-oriented software engineering concept of data hiding.

Constraint 5 *Class Invariants Should be Consistent with Other Expressions of the Class*

Both the functional and dynamic models contain expressions that must be consistent with the invariant specified for the class. Obviously, this rule is relevant to all perspectives of the domain analysis model and may seem somewhat repetitive when mentioned elsewhere in the document in slightly different situations.

Let I be the set of expressions of the class—these expressions come from the class state invariants, transition guard conditions and pre- and postconditions of methods and actions.

$$\forall i_1 : expression \bullet (i_1 \in this.I \wedge this.inv \neq i_1) \Rightarrow ((i_1 \wedge this.inv) \neq false)$$

Appendix B contains the entire cruise missile model. The class missileFuelTank has the invariant:

```
invariant fixedWeight = tankWeight + (fuelDensity * capacity / 2)
   and inputFlowRate = 0
```

while the class method changeFuelFlow() is defined by the postcondition:

```
guarantees outputFlowRate' = actualFlowRate and fuelLevel' = fuelLevel
   and capacity' = capacity and tankWeight' = tankWeight
   and fuelDensity' = fuelDensity
```

According to the constraint, the missileFuelTank invariant must be consistent with respect to the postcondition of the class method changeFuelFlow() which appears from inspection to hold true.

Constraint 6 *Propositions of a Subclass Must be Consistent With Those of the Superclass*

Subclasses inherit the methods and attributes of their parent (super) classes. The subclass cannot alter the characteristics of any of its inherited attributes—to do such would mean that the superclass is not in fact a generalization of the subclass. Class methods may be overridden but for the purpose of this research, they must retain the logical equivalence of the inherited propositions.

Let superclass(c_1, c_2) be a function that returns true if c_1 is the superclass of c_2.

$\forall c_1 : class\ \forall c_2 : class \bullet (c_1, c_2 \in dom.Classes \wedge superclass(c_1, c_2)) \Rightarrow (c_2.inv \wedge c_1.inv \neq false)$

As an example, take the class fuelTank and its subclass missileFuelTank in Appendix B. FuelTank has the invariant fuelLevel $\geq$ 0.0. Being a sub class of fuelTank and adhering to the constraint of strict inheritance, missileFuelTank therefore inherits fuelTank's invariant as well as it own invariant of:

```
invariant fixedWeight = tankWeight + (fuelDensity * capacity / 2)
   and inputFlowRate = 0
```

Constraint 7 *Propositions of a Subclass Must be Substitutable For Those of the Superclass*

Strict inheritance requires that a subclass be substitutable for its parent class. Therefore, not only do the invariants need to be consistent, the subclass invariant must also be weaker or equal to the parent class invariant, therefore the subclass invariant must not constrain that of the superclass. Chapter Six discusses the further formalization of strict inheritance.

$\forall c_1 : class\ \forall c_2 : class \bullet (c_1, c_2 \in dom.Classes \wedge superclass(c_1, c_2)) \Rightarrow (c_2.inv \Rightarrow c_1.inv)$

Returning to the example of Constraint 6, strict inheritance dictates that the invariant of missileFuelTank must imply the superclass invariant which it most certainly does.

3.3.2 Functional Model. The functional model represents the operations a class embodies. This model describes the functionality of class operations irrespective of temporal considerations. At the analysis level, these operations are described via pre- and postconditions that define the output expected for a certain input condition. Thus the output (postcondition) is defined as the result of a certain input (precondition) and not as an explicit algorithm.

Constraint 8 *Operation Postconditions Must Maintain The Class Invariant*

The postconditions of class operations must remain internally consistent with respect to the class invariant.

$$\forall\, op : operation \bullet op \in this.Ops \Rightarrow (op.post \wedge this.inv) \neq false$$

Constraint 9 *Mathematical Operators Are for Mathematical Types or Explicitly Defined for the Type*

Operators have certain associated semantics dependent upon the type to which they apply. For example, 2+3 is commonly accepted as equaling 5 in the domain of integers but what does F-16 + F-16 equal? Two F-16s, one F-32?

In general, both operands of a binary operation must have the same type, and the operator is said to have the type of the return value. AWSOME offers a great deal of flexibility in type definitions and the operations they embody. This is elaborated upon in Appendix A.

Let MO be the set of mathematical operators, MATH be the set of predefined mathematical types for Z {integer, natural}, DEFS be the set of user defined types with mathematical operators—DEFS contains any mathematical subtype, set, sequence, or bag declared in the domain.

The following syntax is defined : e_1 o e_2 represents that application of binary operator o to the operands e_1 and e_2 while o(e) represents the application of a unary operator to the operand e.

For binary operations, this constraint is defined as: $\forall\, e_1, e_2 : expression\ \forall\, o : MO \bullet e_1\ o\ e_2 \Rightarrow ((e_1.type = e_2.type \wedge e_1.type = o.type) \wedge (e_1.type \in MATH \cup DEF))$

For unary operations, the constraint is: $\forall\, e : expression\ \forall\, o : MO \bullet o(e) \Rightarrow ((e.type = o.type) \wedge (e.type \in MATH \cup DEF))$

Constraint 10 *Pre- and Postconditions Must Refer Solely to Global Constants, Class Attributes and Parameters*

At the analysis level AWSOME represents class methods by set of formals (parameters), and pre- and postcondition expressions. Class methods must only refer to the parameters passed to them (formals) and the attributes of the class. AWSOME has the capability to model global variables as well as global constants but this research does not address global variables at the analysis level.

$$\begin{aligned}
&\forall a : expression\ \forall b : expression\ \forall o : operation \bullet o \in this.Ops \wedge a = o.pre \wedge \\
&b = o.post \wedge (n \in dataSet(a) \Rightarrow (n \in dom.Consts) \\
&\qquad\qquad\qquad\qquad \vee (n \in this.Attrs) \\
&\qquad\qquad\qquad\qquad \vee (n \in o.Params) \\
&\qquad \vee\ n \in dataSet(b) \Rightarrow (n \in dom.Consts) \\
&\qquad\qquad\qquad\qquad \vee (n \in this.Attrs) \\
&\qquad\qquad\qquad\qquad \vee (np \in o.Params))
\end{aligned}$$

Constraint 11 *Operation Parameters Must be Defined Over Existing Types*

Every parameter referenced via an operation must belong to a data type defined for the problem domain specified.

$$\forall p : parameter\, \forall o : operation \bullet o \in this.Ops \wedge p \in o.Params \Rightarrow p.dataType \in dom.Types$$

3.3.3 Dynamic Model. The dynamic model describes the behavior of a class with respect to how events interact without concern for what functionality the events actually engender. This functionality is described in the functional model.

Constraint 12 *Transitions Must Occur Over States Defined For the Class*

This and the next constraint ensure that references within a dynamic model transition refer to defined elements of the class dynamic model.

$\forall s : state\ \forall t : transition \bullet ((t \in this.Trans) \wedge (s = t.current \vee s = t.next)) \Rightarrow s \in this.States$

Constraint 13 *Transitions May Only Refer to Send and Receive Events Defined For the Class*

$\forall t : transition\ \forall e : event \bullet (t \in this.Trans \wedge (e = t.receive \vee e \in t.Send)) \Rightarrow e \in this.Events$

Constraint 14 *Transitions Must be Deterministic*

If multiple transitions exist from a state then they must be mutually exclusive and an automatic transition (with a guard condition of true) is allowed only if it is the sole transition from the state.

$\forall t_1 : transition\ \forall t_2 : transition \bullet t_1 \in this.Trans \wedge t_2 \in this.Trans \wedge t_1.current = t_2.current \wedge t_1.receive = t_2.receive \wedge t_1 \neq t_2 \Rightarrow (t_1.guard \wedge t_2.guard) = false$

$\forall t_1, t_2 : transition \bullet t_1.guard = true \wedge t_1.receive.name = automatic \wedge t_1.current = t_2.current \wedge t_1.next = t_2.next \Rightarrow t_1 = t_2$

Figure 15 describes in AWL syntax the transition table for the class airframe. An example of a deterministic set of transitions is the two transitions from the current state **poweredFlight**. Although both transitions share the same guard condition (true) determinism is ensured by each responding to a different receive event (tankEmpty and getPosition).

Constraint 15 *States Must be Mutually Exclusive*

The requirement for this rule in a Mealey-based dynamic model is more than questionable as the set of transitions leading to a state fully defines the class behaviour within the state. However, if a Moore-based dynamic model or a hybrid representation is used, then this constraint is an important one. In order that the states of a dynamic model be uniquely identified, the conjunction of the invariant expression of any state with any other state must be false.

$\forall s_1 : state, \forall s_2;\ state \bullet s_1, s_2 \in this.States \wedge s_1 \neq s_2 \Rightarrow (s1.invariant \wedge s2.invariant) = false$

Although a trivial example, Figure 16 defines the state invariants for the class airframe. As can be obviously deducted, the conjunction of any two invariants is false as the enumerated variable afState cannot have two different values simultaneously.

Constraint 16 *State Invariants Must Be Defined Over Attributes of the Class and Global Constants*

This contention is a linking issue that is discussed in Chapter Four. Note also that in a similar fashion to other components of the domain, state invariants should also be type checked and not contradict the class invariant.

$\forall s : State\ \forall n : dataObject \bullet s \in this.States \wedge n \in dataSet(s.inv) \Rightarrow (n \in Attrs) \vee (n \in dom.Consts)$

Constraint 17 *The Transition Guard Must be Defined Over Attributes of the Class, Event Parameters and Global Constants*

$\forall t : transition\ \forall n : dataObject \bullet t \in this.Trans \wedge n \in dataSet(t) \Rightarrow (n \in dom.Consts) \vee (n \in this.Attrs)$

Constraint 18 *The Preconditions of a Transition Must Be Satisfiable For a Transition To Ever Take Place*

This constraint ensures that the guard condition and current state invariant are consistent with the class in order to prove that the conditions leading to a transition are capable of being satisfied.

$\forall t : transition \bullet t \in this.Trans \Rightarrow (t.current.invariant \wedge this.inv \wedge t.guard) \neq false$

Each of the following two dynamic model constraints is based upon an implication that relies upon the preconditions of a transition holding. If the left hand side of an implication is false, the right hand side can be true or false. The aim of this constraint is to ensure that the left hand side of the following two constraints is not false.

Constraint 19 *The Invariant of the Next State Must be Implied by the Transition's Guard and the Postcondition of the Action*

Whereas the previous constraint dictates that the precondition of a transition hold, essentially this constraint states that the next state invariant be a weaker expression than the conjunction of expressions leading to the transition. The aim of this check is to ensure that a cause and effect relationship exists between the current state, the next state and the transition between them.

$\forall t : transition \bullet (t \in this.Trans \wedge t.current.inv \wedge t.action.post \wedge t.guard \wedge this.inv) \Rightarrow t.next.inv$

The structural constraint introduced at Constraint 5 states that all expressions of the class must not contradict the class invariant and as such, the class invariant is not explicitly included in the above proposition.

What must be kept in mind when verifying this constraint is that whichever attributes appear in the postcondition of the action are modified values, and as such, attributes referred to in the invariant of the next state must also be decorated with ticks to ensure that names correctly match and the sequential nature of the transition is maintained.

Constraint 20 *The Invariant of the Send Events of a Transition Must Be Implied By the Transition's Guard and the Postcondition of the Action*

Send events generated by a transition must be consistent with the conjunction of the expressions of the transition that lead to their generation. The aim of this check is to ensure that a cause-and-effect relationship exists between a transition and its send events.

$\forall t : transition\ \forall e : event \bullet t \in this.Trans \wedge e \in t.Send \wedge t.current.inv \wedge t.action.post \wedge t.guard \wedge this.inv \Rightarrow e.inv$

Constraint 21 *The Precondition of an Action Must be Implied by the Conditions of the Transition.*

The precondition of an action must be satisfied by the current state invariant, the class invariant, the guard condition and the receive event invariant for the action to take place.

$\forall t : transition \bullet (t \in this.Trans \wedge this.inv \wedge t.current.inv \wedge t.guard \wedge t.receive.inv) \Rightarrow t.action.pre$

Constraint 22 *Receive Event Parameters Must Match Action Input Parameters*

For a given transition, the signature of the action it invokes must contain the parameters of the transition receive event. Within a class dynamic model, an action refers to a method of the class. The variables a method operates upon stem from attributes of the class, locally defined variables and input parameters that originate from the arguments of the receive event that resulted in the transition being triggered.

For a given action, let inParams be the set of input parameters of an action, i.e., inParams $= \{p \in action.Params \bullet p.in = true\}$.

$\forall t : transition\ \forall rx : event\ \forall act : method \bullet t \in this.Trans \wedge rx = t.receive \wedge act = t.action \Rightarrow rx.Params \subseteq act.inParams$

Constraint 23 *Send Event Parameters Must Match Action Output Parameters*

Similar to the above constraint, for a given transition, action output parameters must form the set parameters of the set of send events of the transition.

For a given action, let outParams be the set of output parameters of an action, i.e., outParams = $\{p \in action.Params \bullet p.out = true\}$.

$\forall t : transition \, \forall tx : event \, \forall act : method \bullet t \in this.Trans \wedge tx \in t.receive \wedge act = t.action \Rightarrow tx.Params \subseteq act.outParams$

3.4 Domain Level Verification

The focus of concern at the domain level is the interaction between classes and their associations. An example of domain level interaction is the relationship between send and receive events of different classes, while an example of an association would be an aggregation.

3.4.1 Structural Model.

Constraint 24 *Associations Must Refer to Classes Defined Within the Domain*

For an association to exist and have meaning in a domain, there must also exist the constituent classes that make up its ends. Below, assocEnd is used to identify an end of an association.

$\forall assoc : association \, \forall class_1, class_2 : class \bullet assoc \in dom.Assocs \wedge class_1 \in dom.Classes \wedge class_2 \in dom.Classes \Rightarrow assoc.end_1 = class_1.name \wedge assoc.end_2 = class_2.name$

This and the following structural model constraints are enforced during the linking phase of the analysis model creation—refer to Chapter Four for a description of the linking process.

Constraint 25 *Associative Objects Must Refer to Classes Within the Domain*

This constraint is merely a specialization of Constraint 24.

Constraint 26 *Aggregation Must Refer to Classes Within the Domain*

Another special case of association, aggregation, is worth discussing in a little more depth due to the fact that aggregation may represented in more than one fashion and hence requires special consideration.

In AWSOME, aggregation may be represented as a special form of association, Figure 17, or somewhat more implicitly, as a class attribute, Figure 18.

In its first form, this constraint of aggregation is formalized by Constraint 1 while in its second form Constraint 24 expresses the required constraint.

3.4.2 Functional Model. Domain level rules dealing with the functional model are predominantly concerned with the invocation of methods—that is, who is capable invoking a method and the consistency of the call made.

Constraint 27 *Operation Calls Must Match Signatures*

The signature of an operation describes the set of formal parameters that declare its input and output characteristics. Each parameter is represented by an identifier, data type, and in/out qualifier. For the purposes of this research, no parameter is allowed to be used for both input and output as this greatly complicates the verification process. AWSOME however, is capable of modeling in/out parameters.

$\forall t : transition \; \forall op : operationCall \; \forall act : action \bullet t \in this.Trans \wedge act \in this.Ops \wedge act.name = op.name \wedge \#t.op.Args = \#act \Rightarrow (\forall i : \mathcal{N} \bullet i \in \operatorname{dom} A \wedge A(i).type = P(i).type$

3.4.3 Dynamic Model. The majority of verifiable rules related to the dynamic model are applied at the class-level. As long as there exists a correlation between events of classes within the domain model, domain-level dynamic modeling is verified.

Constraint 28 *Objects May Only Communicate Via Send and Receive Events*

The interaction of objects within a domain—that is, how they communicate, how aware they are of each others' existence and their ability to invoke each others' operations is a point of contention in the field of object oriented software engineering and as such, this research has adopted the strict constraint that class-level communication is to occur through the sending and receiving of events only.

Constraint 29 *All States Should be Reachable*

In this research, it is assumed each class has an initial state named "start". Start is a magical state in which each class begins and therefore, no transition is needed to it. However, every other state requires not only a transition to it but a transition whose receive event and guard condition are capable of being satisfied. That is, there exists a corresponding send event in another object and the guard must not contradict the class invariant.

This rule does not constrain the analysis model such that all states must be reachable—the intent is to generate a warning to the software engineer that there exist certain states within the domain that are not capable of being reached. Specification reuse may mean that certain class properties are irrelevant to a specific domain; alternatively the warning may bring to light an actual oversight of the model.

The following rule states that for each receive event in the domain, there must be a corresponding send event.

$\forall c_1 : class\ \forall t : transition \bullet t \in c_1.trans \Rightarrow (\exists c_2 : class\ \exists t_2 : transition \bullet t_2 \in c_2.Trans \land t_1.receive \in t_2.Send$

An example of an unreachable state is **maneuvering** in the AWSOME syntax transition table of Figure 15. The absence of any transition to the state is the culprit in this case and as such, a warning to the software engineer should be generated.

3.5 Summary

This chapter introduced a set of formal constraints that an object oriented analysis model must satisfy in order to ensure consistency and completeness. Each constraint is formalized according to the semantics of a domain model introduced early in the chapter. The constraints are grouped depending upon which model of the Object Modeling Technique [Rumbaugh] they are appropriate and whether they are a class- or domain-level issue. The next chapter discusses the implementation of the testing of the constraints proposed within this chapter.

```
type flightDirectorStates is (startflightDirector, idleflightDirector,
     maneuveringflightDirector);
...

class flightDirector is
    private flightDirectorState : flightDirectorStates;

    private procedure initialize()
    guarantees flightDirectorState' = idleflightDirector

    dynamic model is
    event errorSignals();
    event initFlightDirector();
    event maneuverComplete();

    state start invariant flightDirectorState = startflightDirector;
    state idle invariant flightDirectorState = idleflightDirector;
    state maneuvering invariant flightDirectorState = maneuveringflightDirector;

    transition table is
        in start on initFlightDirector if true do
           initializeFlightDirector(); to idle;
        in idle on errorSignals if true send changeCourse(); setElevation();
           setThrottle(); to maneuvering;
        in maneuvering on maneuverComplete if true to idle;
    end transition table;
    end dynamic model;
end class;
```

Figure 14. Declaration of the Type FlightDirectorStates and the Class FlightDirector

```
transition table is
   in initial on initAirframe if true to airframeInit;
   in airframeInit on AUTOMATIC if true send initDone();
      to poweredFlight;
   in poweredFlight on tankEmpty if true to inertialFlight;
   in poweredFlight on getPosition if true send positionCurrent();
      to poweredFlight;
   in maneuvering on doManeuverComplete if true to poweredFlight;
end transition table;
```

Figure 15. AWSOME Syntax Transition Table

```
state start invariant afState = start;
state aiframeInit invariant afState = afInit;
state poweredFlight invariant afState = poweredFlight;
state maneuvering invariant afState = maneuvering;
state inertialFlight invariant afState = inertial;
```

Figure 16. State Invariant for the Class Airframe

```
aggregation propels is
parent missile : cruiseMissile multiplicity One;
child  propulsion : propulsionSystem multiplicity One;
end aggregation;
```

Figure 17. Aggregation Represented Via Association.

```
class cruiseMissile is
   private propulsion : propulsionSystem;
   private frame : airframe;
   private payload : warhead;
   private avionics : avionicsSoftware;
   private cmState : cmStates;
  ...
end class;
```

Figure 18. Aggregation Represented Via Class Attribute.

IV. Implementation

4.1 Introduction

The aim of this chapter is to expand upon each of the constraints introduced in Chapter Three by detailing their implementations. Descriptions of the process by which an AWSOME analysis model is specified and its representations for the theorem prover Z/Eves and dynamic model verifier Spin are also given.

The constraints of Chapter Three are best suited to several forms of verification:

1. Some of the simpler, static, constraints are checked directly on the AST, such as those addressed by name analysis and type checking.
2. Some constraints require logical inference (such as 19, 20 and 21), thus a theorem prover is required. The theorem prover Z/Eves is used to automate verification of these constraints.
3. The dynamic model supports specialized analysis for which tools exist. The dynamic model checker Spin is used to verify constraints such as 14 and 29.

In order to keep the methods independent of the manner in which a system is specified, the majority of verification techniques used focus upon the AWSOME analysis AST. That is, no matter if a system is specified in Z, AWL or via the object editor, so long as it may be stored in the analysis AST these tests can be performed upon it. If the tests had been made input dependent then a new series would be required for each method of input to the analysis AST. However, this has occurred on occasion, where the parser's syntax has defined certain rules that a specification must follow. Other developers must be aware of these factors when designing new methods of analysis AST creation. These grammar-enforced rules will be discussed throughout the chapter.

This chapter is structured in a similar fashion to that of Chapter 3—addressing each constraint presented in the methodology and how its verification was implemented.

4.2 Limitations Placed Upon AWSOME Models in This Research

In order to ensure compatibility with Z/Eves and Spin, the following constraints are placed upon AWL:

1. no underscores are permissible,
2. string values must be introduced as a constant of type string with the value {},
3. no parameters may be of both in and out modes, and
4. dynamic models that use automatic transitions must declare an AUTOMATIC event.

It should be noted that the majority of these limitations are rather easy but time consuming to rectify.

4.3 Creating AWSOME Analysis Models

This section of the chapter details the manner in which a domain model is created and verified in AWSOME:

1. specification generation in AWSOME syntax,
2. parsing the specification into an AWSOME analysis AST,
3. generation of symbol tables that manage name spaces,
4. linking of identifiers with their respective identifier references throughout the analysis AST, and
5. semantic analysis.

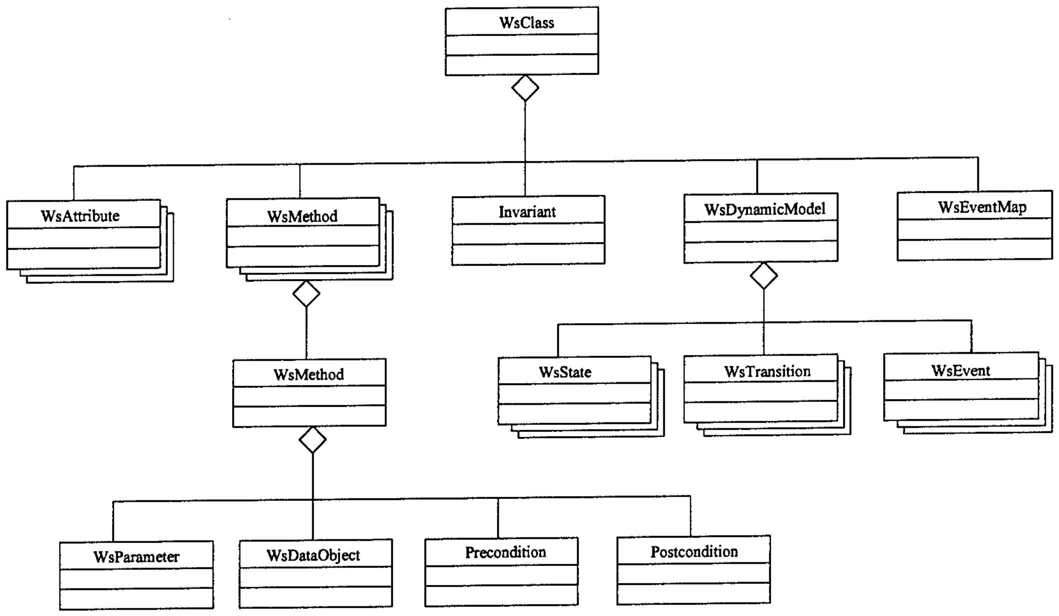


Figure 19. The AWSOME Class Model.

The structure of an AWSOME AST is described by AWSOME's metamodel. The AWSOME metamodel takes the form of an OO inheritance hierarchy—it has approximately 100 classes (termed WsClasses) that are used to model a wide spectrum of object-oriented programming components. Portions of the AWSOME analysis AST especially pertinent to this research include the WsClass and its aggregate components illustrated in Figure 37.

A system may be specified in AWL and, via parsing, be transformed into an AWSOME AST. However, parsing AWL files is not the only method by which specifications may be transformed into AWSOME ASTs and as such there is a need to explicitly define the AWSOME syntax so that tools made subsequently comply with the rules that the parser enforces. If these productions are not enforced, the correct and complete operation of the verification techniques proposed in this research cannot be guaranteed. AWL production rules are included at Appendix A.

The following subsections detail the process followed to generate an AWL specification and the specification's subsequent verification and validation.

4.3.1 Problem Domain Specification. Presently, a user may specify a problem domain via creation of an AWL file or the use of a GUI object editor. It is envisaged that, via an elictor harvester, the software engineer will harvest applicable classes from the existing knowledge base and then specify any deficiencies in AWL syntax. This specification is saved as an AWL file and may then be read by the parser.

4.3.2 Parsing AWL. The AWSOME parser produced in conjunction with AFIT faculty is responsible for verifying the syntactic rules of a specification generated in AWL. The product of the syntactically correct specification is an AST based upon WsClasses. It must noted, however, that the parser requires identifiers to be placed in AWL syntactically correct positions, but at no point does it ensure that the AST is semantically correct.

4.3.3 AWSOME AST Scoping. The first stage in ensuring the correctness of an AWSOME specification is the generation of a symbol table that maintains a list of the declarations visible at any point in the AST.

Certain WsClasses within an AST make declarations that should be only visible to certain other components of the AST. That is, references should be made solely to those components declared within the list of open scopes. Take for example class attributes —the object-oriented concept of data hiding requires that a class keep its data objects hidden from the outside environment, and as such requires the creation of a scope within which these attributes are declared and visible.

The symbol table affords the capability to ensure that declarations of the same name and category are not allowed within the list of open scopes and as such, reduces the possibility for name ambiguity in the specification and errors in object referencing.

4.3.4 AWSOME AST Linking. Once it has been ensured that no components of the same category exist within a mutually open scope, it is possible to link references to declarations with the declared object and raise critical errors or warnings to notify the formalist of incompatible types.

4.3.5 Semantic Analysis. From the analysis AST is generated both a Z/LaTeXfile and a Promela file. The Z/LaTeXfile may then be inspected with Z/Eves in order to prove properties of the specification, and the Promela file may be executed in Spin to highlight any further dynamic model concerns.

These two tools do not address other semantic concerns of the analysis model such as standard compiler-like checking of method signatures and return type consistency. It is envisaged that such checks will be applied to the analysis AST directly by another visitor and do not provide any value to the research interests of this work. The checks implemented as part of this research are more complex. They require deeper levels of analysis such as theorem proving, exhaustive enumeration of state spaces and simulation.

Table 1 summarizes the responsibility, implementation status and automatability of each of the proposed constraints with respect to the components discussed in this section.

4.4 SPIN and Z/Eves

While numerous extensions to Z exist to cater for object-oriented analysis, methods of this research do not modify Z in any fashion—elements of the dynamic model that are difficult to express in Z syntax are specified in the process modeling language Promela. This ensures that the theorem prover behaves as expected and does not result in a less applicable, more esoteric strain of the Z virus. The use of an interactive theorem prover such as Z/Eves allows for modification of the Z model

Constraint	Linker	Z/Eves	SPIN	Semantic	Implemented	Automated
1	x				x	x
2	x				x	x
3		x			x	x
4		x			x	x
5		x			x	x
6		x				
7		x				
8		x			x	x
9		x			x	x
10	x				x	x
11	x				x	x
12	x				x	x
13	x				x	x
14			x		x	
15					NA	NA
16	x				x	x
17	x				x	x
18		x			x	x
19		x			x	
20		x			x	
21		x			x	
22				x		
23				x		
24	x				x	x
25	x				x	x
26	x				x	x
27				x		
28	x	x			x	x
29			x		x	

Table 1. Constraint Verification Responsibility.

prior to modification of the specification to test various hypotheses without having to modify the specification repeatedly to catch a single inconsistency.

SPIN's applicability to this research was essentially limited to state reachability analysis performed via the use of an exhaustive state space search. It became evident, however, that its use is inappropriate for the verification of OMT dynamic models. SPIN's inability to completely represent OMT dynamic models stems from its limited range of data types (numerical types of different sizes) and lack of expressive power in representation of propositions such as invariants and pre- and postconditions.

This does not mean that SPIN is a poor tool—its use is extensive around the world in dynamic model verification.

4.5 Z and Promela Generation

Both the Z and Promela specifications are generated from the analysis AST via the use of the design pattern termed the visitor [Gamma]. A visitor allows for the addition of functionality to certain object-oriented systems without the requirement to modify the classes of the structure, therefore maintaining AWSOME's conceptual integrity.

Neither the Z nor Promela transformations provide a complete representation of the entire OOA model. Only a sufficient amount of the model is transformed to allow for the generation of theorems that express the constraints established in Chapter Three.

The Z specification visitor outputs a specification in the formal language Z, complete with theorems and proof commands aimed at verifying the constraints proposed. The Promela specification visitor outputs a Promela file for execution in Spin.

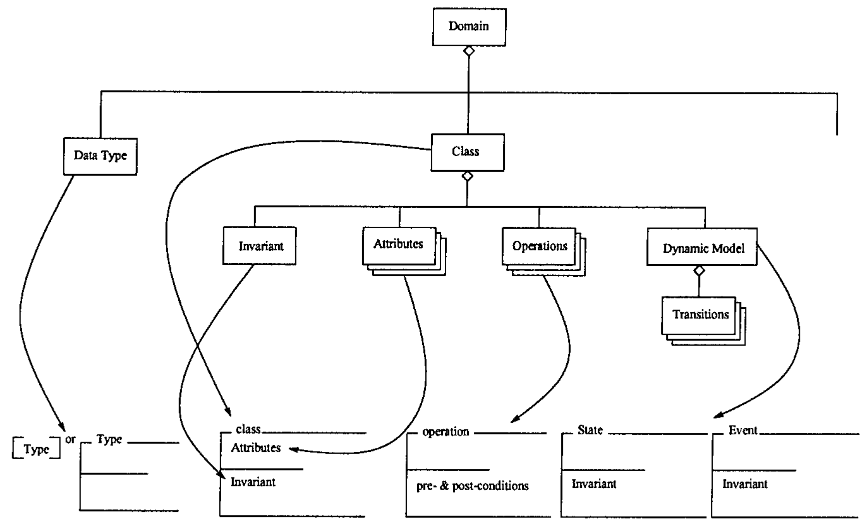


Figure 20. From AWSOME Analysis model to Z schemas.

4.6 Modeling a Domain in Z

The representation of object-oriented systems with the formal language Z is not by any means a new concept. Familiarity with Z while studying at AFIT and the availability of a free theorem prover with satisfactory support were key determinants in the decision to make use of Z as an analysis model representation for the purposes of verification in this research.

4.6.1 Structural Model in Z. The following subsections detail how certain OOA concepts are represented by the Z specification generator produced as part of this research. Figure 20 illustrates the mapping from an OMT structural model to its representation in Z.

4.6.1.1 Types. Abstract types are represented as identified sets. An example in the cruise missile model is the type character. This will be represented by [char] in the Z analysis model. Note that enumerated types are fully defined, e.g., AWSOME syntax for the states of the class airframe is:

type afStates is (start, afInit, onCourse, maneuvering, inertial);

which is transformed into the Z syntax below:

afStates ::= start | afInit | onCourse | maneuvering | inertial

The only types inherent to Z are the integers $\mathcal{Z}$ and the set of naturals $\mathcal{N}$. Therefore, a fixed point approximation was required for real numbers. The chosen approximation is to multiply by an order of magnitude equal to the decimal resolution required, e.g., for a real with a range from 1.0 to 10.0 with a delta of 0.001, the type was represented by an integer of range 1×10^3 to 10×10^3. Obviously, this raises inaccuracies in scale when dealing with operations such as division and multiplication and therefore, checking of real types must take this into account. Floating point types are represented by integers in a similar fashion to that of fixed point real numbers. For most purposes, knowing that the type is a form of number is enough to allow model verification to continue.

Integer types—subtypes of $\mathcal{Z}$—are represented by axiomatic definitions that declare the type's identifier and its range restrictions.

Z has no intrinsic representation character or string types or literals. Without defining character in Z as an enumerated type that contains the entire character set, literal strings (sequences of characters) must all be defined as global constants equal to the empty sequence. To provide a literal string with any value would not make sense as character is defined as an abstract type.

4.6.1.2 Classes. The structural model of a class is represented by a static schema. The name of the schema corresponds to that of the class, the signature of the schema describes the attributes of the class whilst the predicate describes the class invariant. An example class is illustrated in Figure 21.

4.6.1.3 Associations. Associations are generally handled as static schemas. The name of the schema is that of the association, the signature of the schema defines the end roles of the association and the predicate describes multiplicities of the end roles. It must be noted that the special form of association, *cruiseMissile*

$propulsion : propulsionSystem$
$frame : airframe$
$payload : warhead$
$avionics : avionicsSoftware$
$cmState : cmStates$

Figure 21. Aggregation Represented Via Aggregate Class Schema.

propels

$a : propulsionSystem \leftrightarrow cruiseMissile$

$\forall x \in \mathrm{dom}\, a \# (x \lhd a) \leq 1$
$\forall y \in \mathrm{ran}\, a \# (y \rhd a) \geq 0$

Figure 22. Aggregation Represented Via Association Schema.

aggregation, may also be represented via the addition of the aggregate component as an attribute of the aggregate class.

AWSOME facilitates both methods of describing an aggregation as does the Z-schema generator created as part of this research. The two alternate representations of the cruiseMissile/propulsionSystem aggregation are illustrated in Figure 21 and Figure 22.

4.6.2 Functional Model in Z. The methods constituting a class' functional model are represented by dynamic schemas. Dynamic schemas reference the class by delta inclusion (represented by Δ className in the schema signature), thus identifying the schema as a method that may result in a change in class attributes. Figure 23 illustrates the mapping of an AWL class operation to its representation in Z. An example of a method specified in AWL and expressed as a Z schema is given in Figures 24 and 25, respectively.

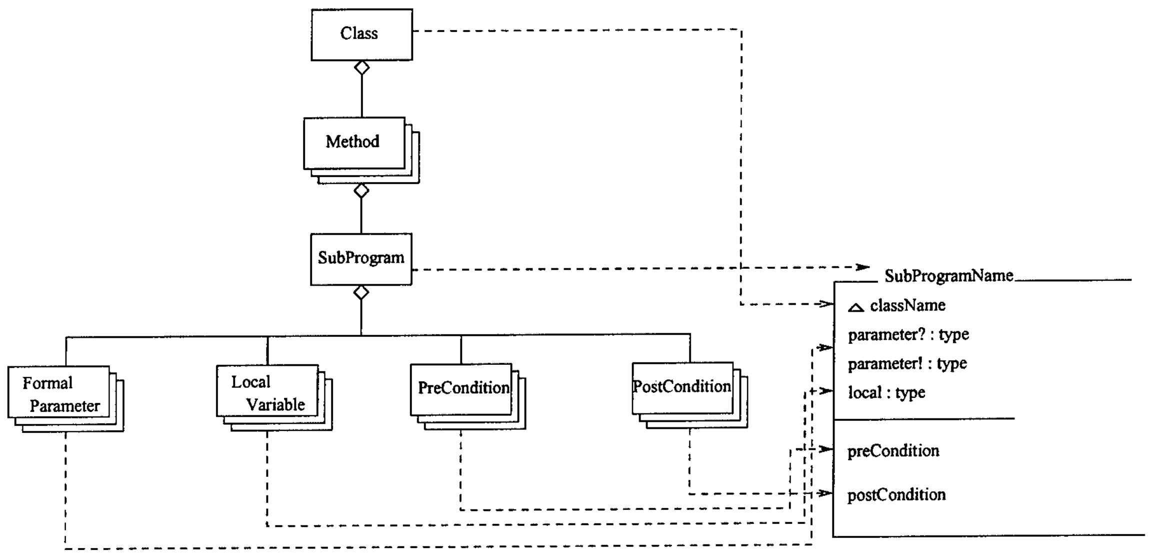


Figure 23. From Class Method to Dynamic Schema.

```
private procedure changeFlow(actualFlowRate : in bigReal)
guarantees outputFlowRate' = actualFlowRate? and
    fuelLevel' = fuelLevel and capacity' = capacity and
    tankWeight' = tankWeight and fuelDensity' = fuelDensity
```

Figure 24. AWSOME Syntax For the Method ChangeFlow.

At the analysis level, AWSOME defines class methods via pre- and postcondition expressions (identified by the keywords "assumes" and "guarantees" respectively). It is these expressions that form the predicate of the method's dynamic schema.

Note, it is considered that if a variable is not explicitly changed with a tick (') then it remains unchanged as opposed to being considered neglected and therefore an error in the specification.

4.6.3 Dynamic Model in Z. Dynamic models in AWSOME are comprised of a set of states, a set of events, and a set of transitions. The set of transitions define the relation between elements of the two former sets.

changeFlow

$\Delta missileFuelTank$
$actualFlowRate? : bigReal$

$outputFlowRate' = actualFlowRate? \wedge fuelLevel' = fuelLevel$
$\wedge\ capacity' = capacity \wedge tankWeight' = tankWeight \wedge$
$fuelDensity' = fuelDensity$

Figure 25. Dynamic Schema For the Method ChangeFlow.

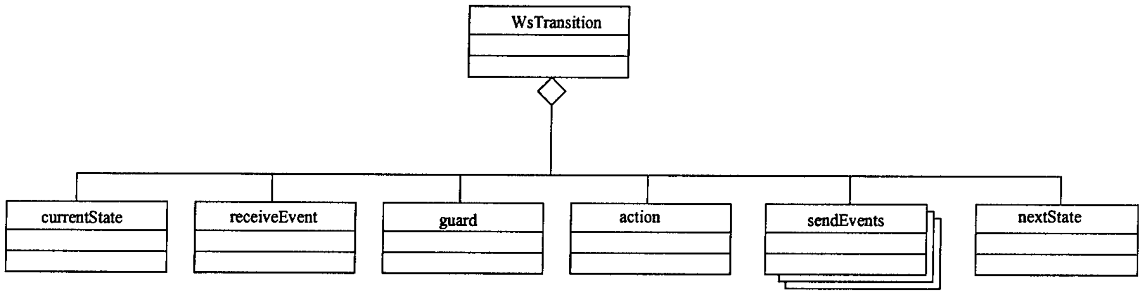


Figure 26. WsTransition and its Aggregate Components.

4.6.3.1 States. A static schema is created for each state of the analysis model and the state invariant, if any, is added to the schema's predicate.

4.6.3.2 Transitions. Transitions in AWSOME consist of a currentState, receiveEvent, guard, action, sendEvent(s), and a nextState (see Figure 26). Transitions are capable of containing a great deal of propositional calculus—both the current and next states contain invariants, as does each send event and the receive event, the guard condition is a Boolean expression, and the action has pre- and postconditions. From this information it can be established that the guards are consistent and complete, transitions are deterministic, and that all states are uniquely identifiable.

No Z construct is capable of representing an entire OOA transition. A number of Z/Eves theorems are generated for each transition in order to test its consistency and completeness with respect to the constraints proposed in Chapter Three (refer to Constraints 18, 19, and 20 for elaboration).

4.7 Verifying Components and Domain Checking With Z/Eves

Relevant analysis model information is harvested from the analysis AST in accordance with the methodology of Section 4.5 and used in the creation of a Z/Eves compatible representation. This representation, in the form of a .zed file, may then be loaded into the Z/Eves environment and the verification goals of this research tested.

This section of the chapter introduces the concept of domain checking as performed by Z/Eves. A more thorough explanation is given in the Z/Eves Reference Manual [reference manual 1.5]. Domain checking automatically occurs when a paragraph[1] is entered in the Z/Eves interactive mode or if the command **check** is executed upon a batch input styled Z-section.

Z syntax allows for the specification of expressions whose semantics are nonsensical [reference manual 1.5]. Two ways that semantic correctness can be compromised are:

1. application of a function outside its domain such as $max\ \mathcal{N}$ or 1 div 0,
2. a proposition is not meaningful if there does not exist a single value for a term such that the predicate holds. For example, $\forall n : \mathcal{N} \bullet n > 5 \land n < 5$.

Z/Eves may be used to check each paragraph of a Z section specification to ensure that function applications are meaningful and that all propositions are semantically correct.

Domain checking provides the backbone of all structural, functional, and the majority of dynamic model semantic analysis performed as part of this research. The

[1]Z paragraphs include abstract types, schemas, axiomatic definitions and theorems.

Symbol	**Grammatical type**
e	expression
P,Q	predicate
ST	schema-text
SE	schema-exp
D	decl-part
n	name

Table 2. Domain Check Notation.

constraints whose implementation are mentioned in this chapter are predominantly performed during domain checking of the analysis specification.

The remainder of this section details the domain checking applied to each form of Z paragraph. A brief description is given prior to the specification of each domain check as described in the Z/Eves Reference Manual. In order to simplify the expressions that follow, the symbols in Table 2 are introduced.

4.7.1 Abstract and Enumerated Types. Abstract and enumerated types are the easiest elements of a specification to verify as they are simply names and are always considered to be true for the purposes of domain checking, i.e., DC([n,...]) = true, where DC represents a domain check and n is the name of the abstract type or the range of enumerated values.

Abstract types have no constraints or operations defined for them other than equality [2] while enumerated types have no constraints or operations defined for them other than equality and test of set membership i.e., membership in the type.

4.7.2 Declarations. Named variables are introduced via declarations. The visibility of declarations is dependent upon where the declaration is made. The visibility of a variable in Z/Eves is either global or local. Declarations made within an axiomatic definition have global visibility beginning from the end of the decla-

[2]AWSOME provides other operators for enumerated types such as $<$ and $>$ but these operators are not defined for enumerated types in either of the verification tools used in this research

ration and spanning the remainder of the specification (i.e., variables may not be referenced prior to their declaration) while declarations made within the signature of a schema remain local to the schema signature and predicate. However, schema inclusion—used in this research to represent methods, states, events, aggregation and inheritance—may be used to introduce variables defined within other schemas into the one currently being declared.

A Z/Eves declaration is structured as follows:

declaration	:: basic-decl;...;basic-decl
basic-decl	:: decl-name-list : expression | schema-ref
decl-name-list	:: decl-name,...,decl-name

Domain checking (DC) of a declaration is dependent upon the form of the declaration—the DC of a name with an expression is simply the domain check applied to the expression, the DC of a set of expressions of a schema is the conjunction of the DC of each schema-expression and so on.

$$\begin{aligned} DC(n,\ldots:e) &= DC(e) \\ DC(S[e,\ldots]) &= DC(e)\wedge\ldots \\ DC(D;D') &= DC(D)\wedge DC(D') \end{aligned}$$

4.7.3 Schemas. Schemas are used to represent classes, states, events, and methods of an AWSOME analysis model. A schema consists of a set of declarations (D) located in the schema signature and a set of propositions (P) located in the schema predicate. Domain checking of a schema entails domain checking the set of declarations and checking propositions over the relevant declaration domains. $DC(Schema) = DC(D) \wedge (\forall D \bullet DC(P))$, where D is the set of declarations in the schema signature and P is the set of propositions of the schema predicate.

4.7.4 *Axiomatic Definitions.*

Axiomatic definitions may be used to represent types with constraints, for example, an integer with constrained upper and lower bounds. The type's name and the fact that it is an integer are declared in the axiomatic definition's signature while the bounds form the predicate.

Domain checking of an axiomatic definition checks the declaration and the propositions of the predicate over the domain of the relevant declarations.
$DC(Axiom) = DC(D) \wedge (\forall D \bullet DC(P) \wedge DC(Q \wedge ...)$, where P and Q are propositions of the axiom predicate.

4.7.5 *Schema Expressions.*

The composition of schema expressions in Z/Eves is as follows:

schema-exp	:: $\forall$ schema-text $\bullet$ schema-exp |
	$\exists$ schema-text $\bullet$ schema-exp |
	$\exists_1$ schema-text $\bullet$ schema-exp |
	schema-exp-1

schema-exp-1	:: schema-ref |
	$\neg$ schema-exp-1 |
	pre schema-exp-1 |
	schema-exp-1 $\wedge$ schema-exp-1 |
	schema-exp-1 $\vee$ schema-exp-1 |
	schema-exp-1 $\Rightarrow$ schema-exp-1 |
	schema-exp-1 $\Leftrightarrow$ schema-exp-1 |
	(schema-exp-1)

The domain checking of schema expressions is:
$DC(\forall D \mid P \bullet SE) = DC(D) \wedge (\forall D \bullet DC(P)) \wedge DC(SE)$
$DC(\exists D \mid P \bullet SE) =$ as above

$DC(\exists_1 D \mid P \bullet SE)$ = as above

4.7.6 Schema Texts. Schema expressions are comprised of sets of schema text where each schema text consists of a declaration and an optional set of predicates, i.e.,

schema-exp-1	:: [schema-text]
schema-text	:: declaration [|predicate]

Domain checking schema texts is very similar to domain checking a schema—the declarations are domain checked and predicates are checked over the domain of the declaration, i.e., $DC([D|P]) = DC(D) \wedge (\forall D \bullet DC(P))$

4.7.7 Schema References. Schema references are optionally prefix-decorated schema names. Schema references are used in this research to represent class methods, inheritance and aggregation (methods are either functions or procedures and are distinguished by the prefixes Ξ and Δ respectively). The schema name refers to the name of the class schema that the method belongs to.

Aggregation and inheritance schema references are not decorated as they are used to declare schema inclusion within the scope of the class being declared.

schema-ref	:: [prefix] word
schema-text	:: declaration [|predicate]
prefix	:: $\Delta \mid \Xi$

Domain checking of schema references entails domain checking of the schema referred to (if not already done) and any expression that forms part of the declaration and is defined as: $DC(S[X,Y][x/y,z:=e])=DC(X) \wedge DC(Y) \wedge DC(e)$.

4.7.8 Z/Eves Prove By Reduce. The most frequent Z/Eves command issued in this research is *prove by reduce.* The command prove by reduce instructs Z/Eves to apply simplification, rewriting, and replacement until the theorem can be no further reduced. Typically, the aim of prove by simplification is to establish a theorem as true, false or not equal to false.

Simplification results in Z/Eves performing equality and integer reasoning, propositional reasoning and tautology checking. Rewriting applies simplification and attempts to simplify the theorem by matching propositions to known patterns. Replacement entails replacing any schema references with their respective definitions.

Figure 27 contains a sample of the output generated by Z/Eves. The first portion is the output generated from domain checking of the schema missileFuelTank. The second portion, beginning at "theorem inheritance", is the proof of a theorem that attempts to verify that missileFuelTank's invariant does not conflict with that of fuelTank. Unfortunately, this proof did not work as anticipated, the superclass schema was reduced to true prior to any further analysis—the proposed solution is to include only the superclass invariant instead of the entire schema—refer to Constraint 6 for further details. The final section of the output, identified by "theorem initializeMissileFuelTankisok" attempts to verify that the operation *initializeFuelTank* does not conflict with the class invariant. For the purposes of this example, the postcondition was modified such that it was inconsistent with missileFuelTank's invariant and Z/Eves returned the proof result "false".

4.8 Modeling a Domain in Promela

In addition to Z/Eves, Spin is used for part of the dynamic model's verification process. The selection of Promela and Spin was made due to the ease of use of the language Promela and the concurrent research conducted by Lacey that resulted in the presence of an active knowledge base of the tool's use at AFIT [Lacey]. The Promela code is generated from the analysis model and focuses solely on class

```
schema missileFuelTank
... theorem missileFuelTank\$domainCheck
... axiom missileFuelTank\$declarationPart
Beginning proof of missileFuelTank\$domainCheck ...

               fuelTank \\
         \land fixedWeight \in realWeight \\
         \land tankWeight \in realWeight \\
         \land capacity \in bigReal \\
         \land inputFlowRate \in bigReal \\
\implies (fuelDensity * capacity, 2) \in \dom (\_ \div \_)
theorem inheritance
... theorem inheritance
Beginning proof of inheritance ...
\exists missileFuelTank @ fuelTank
Which simplifies
forward chaining using KnownMember\$declarationPart, knownMember,
fuelTank\$declarationPart, missileFuelTank\$declarationPart, '[internal items]'
with the assumptions '[internal items]' to ...
\exists missileFuelTank @ true
Proving gives ...
\exists missileFuelTank @ true
schema initializeMissileFuelTank
... schema \Delta missileFuelTank
... axiom Delta\$missileFuelTank\$declarationPart
... axiom initializeMissileFuelTank\$declarationPart
theorem initializeMissileFuelTankisok
... theorem initializeMissileFuelTankisok
Beginning proof of initializeMissileFuelTankisok ...
\exists missileFuelTank @ initializeMissileFuelTank
Which simplifies
with invocation of fuelTank, missileFuelTank, \Delta missileFuelTank,
initializeMissileFuelTank
when rewriting with notEqRule
forward chaining using Delta\$missileFuelTank\$declarationPart,
initializeMissileFuelTank\$declarationPart, KnownMember\$declarationPart,
```

Figure 27. Example Z/Eves Output

```
knownMember, fuelTank\$declarationPart, missileFuelTank\$declarationPart,
'[internal items]'
with the assumptions '&neq$declaration', select\_2\_1, select\_2\_2,
realWeight\$declaration, bigReal\$declaration, '[internal items]' to ...
false
Proving gives ...
false
```

Figure 27. Example Z/Eves Output Cont.

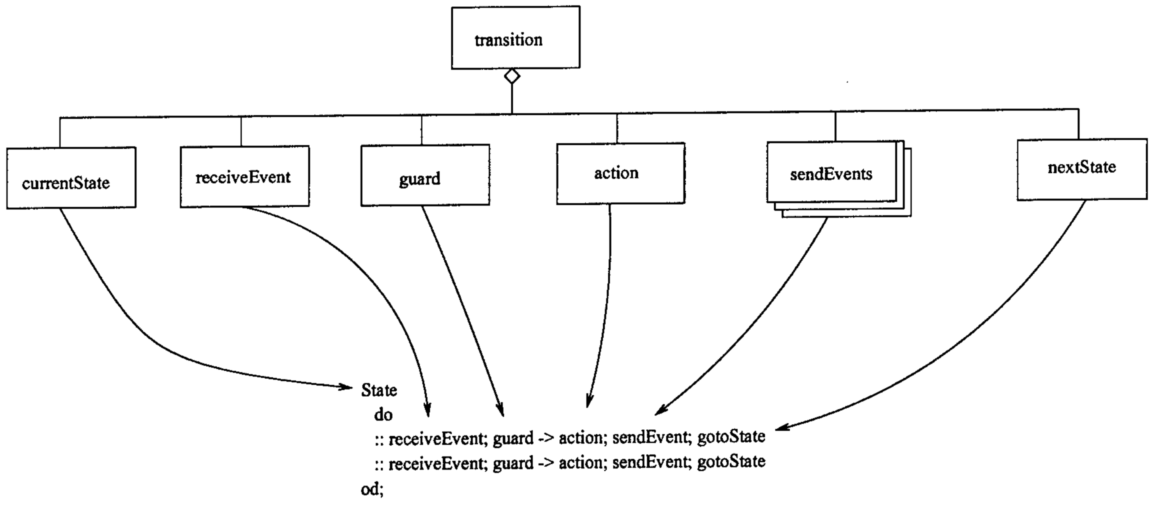

transition
currentState
receiveEvent
guard
action
sendEvents
nextState
State
do
:: receiveEvent; guard -> action; sendEvent; gotoState
:: receiveEvent; guard -> action; sendEvent; gotoState
od;

Figure 28. From WsTransition to Promela.

dynamic models within the domain. Figure 28 illustrates the mapping of an AWL dynamic model to its representation in Promela.

4.8.1 Send and Receive Events. Classes communicate via events and these events are defined as messages in Promela. The first step in the generation of a Promela file is to harvest these events from the analysis model and add their identifiers to the enumerated set "mtype". As an example, the class airframe would result in the events initAirframe, tankEmpty, getPosition, changeCourse, and doManeuverComplete being added to the mtype resulting in a declaration of the form:

mtype = {initAirframe, tankEmpty, getPosition, changeCourse, doManeuverComplete};

4.8.2 Event Maps. The send and receive event names of different classes may well not be the same even though they correspond to the same event. This is due to the fact that possibly not all classes in the domain model were created according to the same standard, either because they were developed by different parties or never actually considered for integration into the same domain model.

The possibility of incompatible dynamic models is what drives the requirement for domain event maps. An event map declares an association between a pair of events in differing classes so as to make integration into the same analysis model a simpler task than requiring the re-specification of the offending dynamic models. AWSOME facilitates domain mapping. However, at this point in time, neither the AWSOME parser nor language provides support for event maps.

It is assumed for the purposes of this research that event names match and as such, there is no need for event maps. If there were, however, a need for event maps, they would be defined as "channels" in Promela. Only a single global channel is required upon which all events are broadcast. Its syntax is:

$$\text{chan global} = [0] \text{ of } \{\text{mtype}\}$$

The above declares a channel "global" of buffer size 0 that carries messages of type mtype. Spin's ability to model temporal systems has resulted in channel definitions stipulating a buffer size. For a non-temporal, object-oriented analysis this buffer size is set to 0—that is, a broadcast event ceases to exist as soon as the next event is broadcast.

4.8.3 Class Dynamic Models. Classes are modeled in Promela as process types (proctypes). Within the proctype are defined the states and their respective transitions as illustrated in Figure 29. It can be seen that the airframe dynamic model is declared as proctypecruiseMissile, the states by their name catenated with State, and the transitions within the current state's do..od loop.

```
proctypeairframe()
{
startState:
   do
   :: (map31?initAirframe; true) -> initializeAirframe;  goto preLaunch
   od;

preLaunchState:
   do
   :: (map12?doLaunch; true) ->  goto poweredFlight
   od;

poweredFlightState:
   do
   :: (map14?outOfFuel; true) ->  goto inertialFlight
   :: (map13?estimatePosition; true) -> setPosition;  goto poweredFlight
   :: (map6?changeCourse; true) ->  goto maneuvering
   od;

maneuveringState:
   do
   :: (map7?doManeuverComplete; true) ->  goto poweredFlight
   :: (map14?outOfFuel; true) ->  goto inertialFlight
   od;

inertialFlightState:
   do
   od;
}
```

Figure 29. Example Promela Proctype.

Each transition is of the form

```
(receiveEvent;guardCondition) -> action; sendEvents; goto nextState
```

The parentheses around the receive event and guard condition are required because of the fact that Spin will not ensure that both are satisfied prior to beginning a transition. That is, if the receive event holds, Spin will select a corresponding transition without considering whether the guard condition also holds and will then wait until that guard condition is satisfied. Therefore without the parentheses, if multiple transitions are triggered by the same receive event but different guard conditions, Spin will arbitrarily pick a transition and lock. The addition of the parentheses ensures the set of transitions is deterministic (if they are indeed deterministic) and that only the transition that satisfies both the receive event and the guard condition is selected for execution.

For situations where transitions are automatic or where transitions have no guard conditions, two constants have been added to the model. All analysis models created for this research use the receive event "automatic" to denote an automatic transition and the guard "true" for transitions with no guard. Both automatic and true are set to the Boolean value true and are therefore always executable to Spin.

4.8.4 Verifying Components With Spin. Spin facilitates verification of the reachability of states. Whereas Z/Eves is used to ensure that sets of transitions are deterministic and that states are uniquely identifiable, Spin is used to simulate the dynamic model and ensure that transitions may be satisfied and all states visited.

Evaluation of state reachability is instigated via prepending the keyword "progress" to the state name. Spin then ensures that the states so annotated are visited during execution or it returns an error message acknowledging the failure to reach such states.

4.9 *The Semantic Analyzer*

The verification performed using Z/Eves and Spin is a form of semantic analysis but it is by no means exhaustive. Many facets of an AWSOME AST that require analysis are outside the scope of this research and as such, should be dealt with by a separate semantic analysis visitor that is capable of traversing the analysis AST in a similar fashion to other visitors implemented in this research. The semantic analyzer would be responsible for semantic analysis not addressed by either Z/Eves or Spin.

Some of the areas not analyzed by Z/Eves and Spin that are pertinent to an analysis model include:

- method signature and method call signature consistency (required for verification of Constraint 27),
- type compatibility and type equivalence (required for complete verification of Constraint 9),
- resolution of return types (required for complete verification of Constraint 9),
- other semantic analysis considerations include facets that are outside the scope of an analysis model—occurring in the AWSOME design AST such as: records, arrays, and method body statements.

4.10 *Class-Level Structural Verification*

Constraint 1 *Attributes Must be Declared Over Defined Types*

Two methods are immediately available to ensure that each data object belongs to a defined type. The first is via the use of the linking visitor—identifierRefs are matched by name and category to identifier symbols present in the symbol table generated by the symbol table visitor. If no defined type exists within the set of open scopes then the linking visitor returns a warning informing the software engineer of the specification's incompleteness.

The other option is to use Z/Eves to perform a type checking run over the Z section. The command **check type** checks the entire Z section and is far more economical and expedient than checking the model declaration by declaration. Both options were used to successfully check that attribute references conformed with this constraint.

Constraint 2 *Any Variable Referenced Within an Object's Invariant Proposition Must be Constants or Attributes of the Object*

The linking process will highlight if an identifierRef refers to a declaration that is not within the set of open scopes but it is not presently capable of fully enforcing such a rule as this. There are three other methods of verifying that the specification adheres to this rule. The first is via domain checking in the Z/Eves environment, the second is via use of manual inspection and is most easily performed upon the class Z-schema. The third option would be the use of a static semantic analysis visitor such as the one that is still in the conceptual phase of development at this time.

Both the linker and Z/Eves domain check were used successfully to check for conformity with this constraint.

Constraint 3 *Pre- and Postconditions Must be Consistent With the Class Invariant*

This constraint is checked via the use of a Z/Eves theorem. The intent of the theorem is to prove that an instance of the class may exist for which the pre- and postconditions of the operation are consistent with the class invariant. The theorem to check the consistency of the operation *initializeMissileFuelTank* with its class *missileFuelTank* is given in Figure 30.

Constraint 4 *Invariant Propositions Must be Consistent With Respect to the Attributes Types Over Which They Refer*

```
\begin{theorem}{initializeMissileFuelTankIsOk}
   \exists missileFuelTank \spot initializeMissileFuelTank
\end{theorem}

prove by reduce;
```

Figure 30. Theorem to Check Operation Expression Consistency With Class Invariant

```
\begin{theorem}{initializeMissileFuelTankIsOk}
   \exists missileFuelTank \spot true
\end{theorem}

prove by reduce;
```

Figure 31. Theorem to Instantiate a Class

Verification of the consistency of a class invariant with respect to the attribute types it refers to is achievable with the Z/Eves theorem prover. A class invariant is represented in the predicate of the corresponding class schema and any proposition associated with a type is present in the type's schema.

By proving that an instance of the class may exist, it follows that the class invariant is consistent with the attribute types to which it refers. The required theorem to verify this constraint is illustrated in Figure 31.

Constraint 5 *Class Invariants Should be Consistent With Other Expressions of the Class*

Expressions capable of occurring within a class:

1. method pre- and postconditions,
2. transition guards,
3. state invariants,
4. event preconditions, and
5. the data types referenced.

It must be ensured that these expressions are consistent with the class invariant. The relation between invariant and data type expressions and verification of this rule was discussed in Constraint 4.

Constraint 3 details the consistency check applied to operation pre- and post-conditions while the consistency of transition guards with respect to the class invariant is addressed in Section 4.12.

State and event invariant consistency is checked in the same fashion as operations are checked in Constraint 3, that is, a theorem is used in an attempt to invoke an instance of the class for which the state or event invariant holds does not cause an inconsistency.

Constraint 6 *Propositions of a Subclass Must be Consistent With Those of the Superclass*

A theorem that instantiates an object of the subclass will return an error if the subclass invariant is inconsistent with respect to superclass invariant. The output of the Z/Eves visitor attempts to instantiate an object of every class as illustrated in Figure 31. This instantiation fails for a subclass invariant that is inconsistent with that of its superclass as the superclass invariant is implicitly included via schema inclusion.

Note that the functional and dynamic models are not further verified with respect to inheritance. These issues are addressed in the Future Work section of Chapter Six.

Constraint 7 *Propositions of a Sublass Must be Substitutable For Those of the Superclass*

This constraint is not checked in the current visitor.

```
Checking schema initializeMissileFuelTank
Error FunctionArgType (line 186) [Type checker]: in application of
\Global (\_ \cup \_), argument 1 has the wrong type.
Error FunctionArgType (line 186) [Type checker]: in application of
\Global (\_ \cup \_), argument 2 has the wrong type.
Error TypesNotSame (line 186) [Type checker]: types of \Local
outputFlowRate'
and \Local capacity \cup \Local fuelLevel are not the same.
Error NoType (line 186) [Type checker]: can't infer type of rel-chain
operand
\Local capacity \cup \Local fuelLevel.
```

Figure 32. Z/Eves Error Message For Type Incompatibility

4.11 Class-Level Functional Verification

Constraint 8 *Operation Postconditions Must Maintain The Class Invariant*

Verification of model conformity with this constraint is handled by the same theorem as presented in Figure 31.

Constraint 9 *Mathematical Operators Are for Mathematical Types or Explicitly Defined for the Type*

It is envisaged that the semantic analysis visitor will be capable of determining the correctness of mathematical expressions with respect to type compatibility, return type determination, and operator semantics. Z/Eves will return an error message during domain checking if an operator is applied to an incompatible type.

Figure 32 illustrates what happens when an operator is applied over incompatible operands—in this case a pair of integers is being conjuncted.

Constraint 10 *Pre- and Postconditions Must Refer Solely to Global Constants, Class Attributes and Parameters*

This constraint is enforced by both the linker and Z/Eves domain checking. That is, the list of open scopes available to the method is comprised of the method scope, class scope, and package scope. This means that the only declarations available to the method are local variables and formals, class attributes, and global constants.

Constraint 11 *Operation Parameters Must be Defined Over Existing Types*

Similar to some of the other rules, two solutions exist to this problem. Linking will ensure that the parameter is of a declared type while Z/Eves ensures that the method pre- and postconditions are consistent with respect to parameter types by domain checking that is automatically done when the schema is declared for the current proof.

This constraint is verified using the same theorem as appears in Figure 30.

4.12 Class-Level Dynamic Verification

Constraint 12 *Transitions Must Occur Over States Defined For the Class*

Linking ensures that referenced states exist within the scope of the dynamic model and notifies the software engineer of any deficiency. This constraint is also addressed by Spin where an error message will be generated for any state that is attempted to be transitioned that does not exist.

The error message below was generated when missileFuelTank referred to the state *noSuchState*. The error message is even kind enough to inform one of the line number where the errant reference may be located.

```
spin: line  46 "cruiseMissile.prm", Error: undefined label noSuchState
```

Constraint 13 *Transitions May Only Refer to Send and Receive Events Defined For the Class*

As discussed in constraint 12, both the linker and Spin provide error messages when this constraint fails to hold.

Spin generated the following error message when the undeclared receive event *noSuchReceiveEvent* was encountered in the dynamic model of missileFuelTank.

```
spin: line  46 "cruiseMissile.prm", Error: undeclared variable:
      noSuchReceiveEvent
```

Constraint 14 *Transitions Must be Deterministic*

In order for the set of transitions to be deterministic, no two transitions may share the same combination of guard condition and receive event. Spin is incapable of detecting non-deterministic transitions, its execution simply selects the first combination of guard and receive event that is satisfied and progresses with that transition.

Spin is capable of identifying non-deterministic transitions. When Spin executes an exhaustive state space simulation, invariably for the same set of preconditions, Spin will select the same transition. Spin returns an error message identifying any transition that is not taken during this simulation. It is then up to the engineer, however, to determine if the cause of this is a non-deterministic set of transitions.

A sample Spin simulation output is illustrated in Figure 35.

Constraint 15 *States Must be Mutually Exclusive*

In order to verify this rule, the schema corresponding to the states of the class must be declared for the current proof in all possible permutations. The state invariant that forms the predicate of each state schema must not be capable of being true if any other invariant is already true for the current proof.

This constraint was removed from the final version of the verification suite as its utility is questionable when compared to the reduction in flexibility it causes to modeling the dynamic behavior of a system—this is elaborated upon in Chapter Five.

Constraint 16 *State Invariants Must Be Defined Over Attributes of the Class and Global Constants*

This is another linking issue—the set of open scopes for a state is the class scope, the dynamic model scope, and the global declarations. It is the responsibility of the linker to warn the software engineer of failure to comply with this rule via an error message.

Constraint 17 *The Transition Guard Must be Defined Over Attributes of the Class, Event Parameters and Global Constants*

This constraint is verified by linking the analysis AST.

Constraint 18 *The Preconditions of a Transition Must Be Satisfiable For a Transition To Ever Take Place*

This constraint is checked by generation of a theorem that instantiates a class and tests to see that the conjunction of the current state invariant, guard condition and operation precondition do not result in an inconsistency.

```
\begin{theorem}{transitionPreconditionsok}
   \exists className \spot t.current.invariant \land t.guard
        \land t.actionPrecondition
\end{theorem}

prove by reduce;
```

Constraint 19 *The Invariant of the Next State Must be Implied by the Transition's Guard and the Postcondition of the Action*

This rule is implemented by declaration of a theorem that conjuncts the transition guard and action postcondition and implies the next state invariant. The syntax for a Z/Eves theorem to ensure this constraint is given below.

```
\begin{theorem}{transitionImpliesNextStateIsOk}
   \exists className \spot t.guard \land t.action.post
        \implies t.next.inv
```

```
\end{theorem}


prove by reduce;
```

Constraint 20 *The Invariant of the Send Events of a Transition Must Be Implied By the Transition's Guard and the Postcondition of the Action*

The following theorem is generated to test the constraint that send event invariants of a transition are implied by the guard condition and action postcondition.

```
\begin{theorem}{transitionImpliesSendEventsIsOk}
   \exists className \spot t.guard \land t.action.post
       \implies send_1.inv \land send_2.inv \land ...
\end{theorem}


prove by reduce;
```

Constraint 21 *The Precondition of an Action Must be Implied by the Conditions of the Transition.*

```
\begin{theorem}{transitionImpliesSendEventsIsOk}
   \exists className \spot (t.guard \land t.current.inv
      \land t.receive.inv) \implies t.action.pre
\end{theorem}


prove by reduce;
```

Constraint 22 *Receive Event Parameters Must Match Action Input Parameters*

This rule would be best enforced by a semantic analysis visitor and is not verified by the products of this research.

Constraint 23 *Send Event Parameters Must Match Action OutputParameters*

As with Constraint 22, this constraint is best enforced by a semantic analysis visitor and is therefore not verified by the products of this research.

4.13 Domain Level Structural Verification

Constraint 24 *Associations Must Refer to Classes Defined Within the Domain*

The only verification relating to associations, associative objects and explicitly defined aggregations (as opposed to those declared as variables of the aggregate class) is provided by the linking visitor.

This research has, however, led to the generation of association schemas for the Z specification model even though they are not further referenced in any proofs of this research. The multiplicity of each end role is a declared integer type. The multiplicities proposed by Buckwalter for the generation of the associative schema are:

1. Optional—representing the cardinality zero or one,
2. ZeroOrMore—self explanatory, and
3. OneOrMore.

For example, there exists an association named flies between an instance of type pilot and an instance of type aircraft. The AWL specification and its corresponding Z representation are depicted in Figure 33.

Note that this method of transformation of associations is unsuitable for specifying associations that are of a higher degree than binary.

Constraint 25 *Associative Objects Must Refer to Classes Within the Domain*

As with Constraint 24, this check is performed by both Z/Eves and the linker.

Constraint 26 *Aggregation Must Refer to Classes Within the Domain*

```
association flies is
    role aviator : pilot multiplicity Optional;
    role ride : aircraft multiplicity ZeroOrMore;
end association;
```

$[pilot]$

$[aircraft]$

Pilots $= P$ pilot

AIRCRAFT $= P$ aircraft

flies

$a : pilot \leftrightarrow aircraft$

$\forall x \in \operatorname{dom} a \#(x \lhd a) \leq 1$
$\forall y \in \operatorname{ran} a \#(y \rhd a) \geq 0$

Figure 33. AWL and Z Representation of the Association *Flies*

A specific form of association, aggregation is verified via the linker and by Z/Eves.

4.14 Domain Level Functional Verification

Both Z/Eves and the semantic analysis visitor are used to determine errors in the functional portion of the analysis model. Dynamic schemas identify the class they modify in the schema signature. The Z specification visitor adds the class identifier to the schema based upon which class the method is declared over and as such, the method is identified as being an operation of that particular class and is only capable of modifying the class' constituent attributes.

Constraint 27 *Operation Calls Must Match Signatures*

Operation signature verification is outside the scope of the current Z specification visitor. The compatibility of operation signatures and operation calls could be verified by the semantic analysis visitor.

4.15 Domain Level Dynamic Verification

Constraint 28 *Objects May Only Communicate Via Send and Receive Events*

The linker enforces this rule. The linker does not allow for classes to directly invoke operations of other classes nor does it allow transitions to be dependent upon events not declared within the class.

Constraint 29 *All States Should be Reachable*

By prepending the keyword **progress** to the name of a state in the Promela file, Spin will monitor the state during execution and provide notice of failure to transition to it if the state is never visited. Spin has two modes of operation—in the first it performs random simulations while the second is an exhaustive verification of the entire state space. It is this second mode that must be used to verify this rule. Spin's exhaustive search method is effective for approximately 100,000 states [Spin] and should therefore remain applicable for the majority of systems being modeled.

Figure 34 provides an example of the output generated from running a random simulation of the system. The command line for such an execution is: *spin339 -c -a cruiseMissile.prm.* The arguments *-c* and *-a* tell Spin how to configure its output and to create an analysis model in the programming language C. The first portion of the output in Figure 34 identifies the process number of each class in the dynamic model. The second portion shows the sending and receiving of events between the classes. The final portion of the output identifies the final state of each class at the end of execution.

It should be noted that simulation of the entire state space requires the compilation and execution of the generated C analysis model. Output of Spin's evaluation of the entire state space is shown in Figure 35.

Figure 35 contains the output of an exhaustive analysis of an erroneous version of the cruise missile model. The output identifies the unreachable states of the model and messages not sent or received. The command line instruction to compile the analysis model is *cc -DBITSTATE -o run pan.c* where -DBITSTATE is a directive for the compiler to compiler the code such that it maximizes memory efficiency during execution. The command line instruction to execute the exhaustive state space analysis is *run -c > out.txt* where -c is the output format and out.txt is the file for the resultant output to be piped to.

4.16 Verifying the Dynamic Model With Spin

After generation of the Promela model, the only thing to be added to the file is a statement that enables Spin to run the specification. The statement instructs Spin to run each of the defined dynamic models and has the following syntax:

```
init
{  atomic
   {
      run missileFuelTank();
      run navigationSystem();
      run guidanceSystem();
      run flightDirector();
      run airframe();
      run cruiseMissile()
    }
}
```

Execution of the Promela model will then verify constraints regarding state reachability and identify states and events not executed during the simulation.

```
proc 0 = :init:
proc 1 = missileFuelTank
proc 2 = navigationSystem
proc 3 = flightProfile
proc 4 = guidanceSystem
proc 5 = flightDirector
proc 6 = avionicsSoftware
proc 7 = warhead
proc 8 = airframe
proc 9 = throttle
proc 10 = jetEngine
proc 11 = propulsionSystem
proc 12 = cruiseMissile
q\p   0   1   2   3   4   5   6   7   8   9  10  11  12
  7   .   .   .   .   .   .   .   .   .   .   .   .   map25!initPropulsionSystem
  7   .   .   .   .   .   .   .   .   .   .   .   map25?initPropulsionSystem
  8   .   .   .   .   .   .   .   .   .   .   .   map23!initThrottle
  8   .   .   .   .   .   .   .   .   .   map23?initThrottle
  4   .   .   .   .   .   .   .   .   .   .   .   map24!initEngine
  4   .   .   .   .   .   .   .   .   .   .   map24?initEngine
  2   .   .   .   .   .   .   .   .   .   .   .   .   map21!initAirframe
  2   .   .   .   .   .   .   .   .   map21?initAirframe
                           timeout
-------------
final state:
-------------
#processes: 13
 24:    proc 12 (cruiseMissile) line 260 "cruiseMissile.prm" (state 6)
 24:    proc 11 (propulsionSystem) line 245 "cruiseMissile.prm" (state 6)
 24:    proc 10 (jetEngine) line 235 "cruiseMissile.prm" (state 10)
 24:    proc  9 (throttle) line 221 "cruiseMissile.prm" (state 11)
 24:    proc  8 (airframe) line 189 "cruiseMissile.prm" (state 11)
 24:    proc  7 (warhead) line 165 "cruiseMissile.prm" (state 6)
 24:    proc  6 (avionicsSoftware) line 150 "cruiseMissile.prm" (state 9)
 24:    proc  5 (flightDirector) line 130 "cruiseMissile.prm" (state 6)
 24:    proc  4 (guidanceSystem) line 103 "cruiseMissile.prm" (state 6)
 24:    proc  3 (flightProfile) line  88 "cruiseMissile.prm" (state 6)
 24:    proc  2 (navigationSystem) line  69 "cruiseMissile.prm" (state 6)
 24:    proc  1 (missileFuelTank) line  44 "cruiseMissile.prm" (state 6)
 24:    proc  0 (:init:) line 289 "cruiseMissile.prm" (state 14) <valid endstate>
13 processes created
```

Figure 34. SPIN Test Run Using: spin399 -s -c -a cruiseMissile.prm

```
pan: invalid endstate (at depth 25)
(Spin Version 3.3.9 -- 31 January 2000)
    + Partial Order Reduction

Full statespace search for:
    never-claim             - (none specified)
    assertion violations    +
    acceptance   cycles     - (not selected)
    invalid endstates   +

State-vector 208 byte, depth reached 26, errors: 1
      17 states, stored
       2 states, matched
      19 transitions (= stored+matched)
      11 atomic steps
hash conflicts: 0 (resolved)
(max size 2^18 states)

1.493   memory usage (Mbyte)

unreached in proctype missileFuelTank
    line 46, state 3, "(1)"
    line 50, state 11, "(1)"
    line 56, state 15, "(1)"
    line 55, state 18, "map1?changeTankFlow"
    line 61, state 22, "(1)"
    line 62, state 26, "(1)"
    line 62, state 28, "map3!outOfFuel"
    line 60, state 30, "map1?changeTankFlow"
    line 60, state 30, "map2?tankEmpty"
    (8 of 33 states)
unreached in proctype navigationSystem
    line 71, state 3, "(1)"
    line 76, state 10, "(1)"
    line 75, state 13, "map5?updatePosition"
    line 81, state 17, "(1)"
    line 81, state 19, "map6!estimatePosition"
    line 80, state 21, "(1)"
    (6 of 24 states)
```

Figure 35. Results of Exhaustive State Space Reachability Test

```
unreached in proctype flightProfile
    line 90, state 3, "(1)"
    line 95, state 10, "(1)"
    line 96, state 14, "(1)"
    line 94, state 17, "map9?addWayPoint"
    line 94, state 17, "map10?removeFirstWayPoint"
    (4 of 20 states)
unreached in proctype guidanceSystem
    line 105, state 3, "(1)"
    line 110, state 10, "(1)"
    line 110, state 12, "map5!updatePosition"
    line 111, state 15, "(1)"
    line 112, state 19, "(1)"
    line 112, state 21, "map5!updatePosition"
    line 109, state 23, "map12?doLaunch"
    line 109, state 23, "map6?estimatePosition"
    line 109, state 23, "map4?outOfFuel"
    line 117, state 28, "map10!removeFirstWayPoint"
    line 117, state 29, "map13!armMissile"
    line 116, state 34, "(1)"
    line 116, state 34, "(1)"
    line 122, state 38, "goto :b12"
    line 125, state 41, "-end-"
    (12 of 41 states)
unreached in proctype flightDirector
    line 132, state 3, "(1)"
    line 137, state 10, "(1)"
    line 137, state 12, "map17!changeCourse"
    line 137, state 13, "map18!setElevation"
    line 137, state 14, "map19!setThrottle"
    line 136, state 16, "map15?errorSignals"
    line 143, state 20, "(1)"
    line 142, state 23, "map16?maneuverComplete"
    (8 of 26 states)
unreached in proctype avionicsSoftware
    line 152, state 3, "(1)"
    line 152, state 5, "map4!initNavigationSystem"
    line 152, state 6, "map11!initGuidanceSystem"
    line 153, state 7, "map15!initFlightDirector"
    line 157, state 13, "goto :b17"
    line 160, state 16, "-end-"
    (6 of 16 states)
```

Figure 35. Results of Exhaustive State Space Reachability Test Cont.

```
unreached in proctype warhead
    line 167, state 3, "(1)"
    line 172, state 10, "(1)"
    line 171, state 13, "map13?armMissile"
    line 176, state 17, "goto :b20"
    line 179, state 20, "-end-"
    (5 of 20 states)
unreached in proctype airframe
    line 191, state 9, "(1)"
    line 196, state 15, "(1)"
    line 197, state 18, "(1)"
    line 198, state 21, "(1)"
    line 195, state 23, "map3?outOfFuel"
    line 195, state 23, "map6?estimatePosition"
    line 195, state 23, "map17?changeCourse"
    line 203, state 27, "(1)"
    line 204, state 30, "(1)"
    line 202, state 32, "map22?doManeuverComplete"
    line 202, state 32, "map3?outOfFuel"
    line 208, state 36, "goto :b25"
    line 211, state 39, "-end-"
    (10 of 39 states)
unreached in proctype throttle
    line 223, state 9, "map28!changeTankFlow"
    (1 of 14 states)
unreached in proctype jetEngine
    (0 of 13 states)
unreached in proctype propulsionSystem
    line 251, state 12, "goto :b31"
    line 254, state 15, "-end-"
    (2 of 15 states)
unreached in proctype cruiseMissile
    line 262, state 7, "map28!initAvionicsSoftware"
    line 267, state 13, "(1)"
    line 267, state 14, "map12!doLaunch"
    line 266, state 16, "map27?launch"
    line 271, state 20, "goto :b34"
    line 274, state 23, "-end-"
    (6 of 23 states)
unreached in proctype :init:
    (0 of 14 states)
```

Figure 35. Results of Exhaustive State Space Reachability Test Cont.

It should be noted that Promela's lack of expressive power in discrete arithmetic made the transformation of some expressions impossible and thus resulted in a lack of representation of many propositions in the simulation model. This effectively means that the reachability analysis performed, although insightful, is inadequate for testing the state reachability of OMT dynamic models.

4.17 Summary

This chapter details the implementation of the constraints proposed in Chapter Three—how they are specified and how they are verified. The analysis model visitors designed to produce Z/Eves and Promela models of the specification are described as is how these theorem proving/dynamic model verifying tools ensure the consistency, completeness, and correctness of a domain model.

The final chapter of this document evaluates the constraints and implementation of this research before drawing conclusions and proposing directions for future, related work.

V. Results

5.1 Introduction

This chapter details the results of testing and execution of the methodology proposed and implemented in Chapters Three and Four. Testing of the practicality, testability, and effectiveness of the constraints proposed was evaluated by their application to an object-oriented formal specification and analysis of the outcomes of the verification process.

The initial analysis model to which these constraints were applied was developed as part of this research effort and is presented at Appendix B. The majority of faults in the analysis model were introduced with the intent purpose of ensuring the implementation of the system while others were simply accidental errors of the specification.

5.2 Implementation Coverage

Table 2 summarizes the state of each constraint, i.e., whether or not verification of the constraint is implemented and whether that implementation is automated. The verification of the majority of the constraints implemented is automatic (once the tools are in execution).

Feedback provided to the user takes the form of the output provided by the tools Z/Eves and Spin and as such, the value of the feedback is limited by the user's experience with the tools.

Automation of the remaining constraints and the provision of feedback in a manner that is specific to OO but not the tools in particular would provide the ability to apply these tools without the requisite knowledge of their internal execution.

5.3 Evaluation of the Constraints

Chapter Three introduced constraints for the three models of OMT. These constraints help to provide and verify formal semantics to the semi-formal foundation of OMT. The list of constraints is by no means complete—Section 5.6 proposes future work in the formalization of aggregation and inheritance alone. They do, however, provide a solid foundation to the concepts required to formalize and verify OOA models and prove that verification is in fact suited to semi-automation.

Other than the identified weaknesses in formalism of the inheritance and aggregation constraints, the question that begs to be answered is "Are the constraints proposed complete?". Unfortunately, the answer is not as simple as the question and should be subdivided into the following sections:

1. Is the list of constraints exhaustive?
2. Is the list of constraints fully implemented?
3. Is each constraint complete?

5.3.1 Is the List of Constraints Exhaustive? The constraints were identified via analysis of the six perspectives of an OMT analysis model as described in Chapter Three. For each perspective, it was attempted to identify the key constituents of the model and ensure that they were formally constrained to assist in verification of the correctness of the entire model.

It has already been acknowledged that this list of constraints is incomplete. Just how to go about proving that a list of constraints is complete is an extremely complex if not fruitless task.

5.3.2 Is the List of Constraints Fully Implemented? Not all of the constraints proposed are checked. The constraints not implemented are identified as such in Table 2. The failure to implement and/or automate those constraints is due to:

1. Z/Eves representing schema invariants as ticked variables only (instead of both ticked and unticked). This leads to the inability of proofs to identify all instances of non-conformity with Constraints 3, 5, 18, 19, 20, and 21. The simplest rectification appears to be the explicit inclusion of the unticked state invariant into each proposition as required.

2. Promela's inadequate variety of data types and subsequent lack of expressive power when dealing with discrete arithmetic. Promela's list of data types is limited to bit, byte, short and int. These data types represent signed and unsigned integer values of differing ranges. The lack of support for sets, sequences and propositions is sorely felt. This meant that verification using Spin was little more than verification of state reachability based solely upon class communication with no regard for guard conditions or class attribute manipulation.

3. Deferral of implementation to a semantic analyzer. Constraints 22 and 23 deal with operation parameter matching—something more suited to a semantic analyzer than a theorem prover.

4. Difficulty in expressing the required theorems. Constraints 6 and 7 deal with the relationship between a subclass class' invariant and its parent's invariant. The theorem proposed simply substituted true for the parent's invariant because Z/Eves believed that as a schema declaration, the schema predicate must be true and thus simplified it as such. A better theorem would extract the superclass invariant and conduct the proof based solely upon it rather than the superclass schema.

Although the aforementioned constraints are not fully functional, these problems could possibly be addressed by other verification tools as they are not all that more complex than any of the other constraints proposed. In fact, all but one of these constraints (Constraint 29) could possibly be addressed by the theorem prover (requiring more complex transforms) and a semantic analyzer. It should also be mentioned that the most surprising error detection was that what was thought to be a robust dynamic model did, in fact, have numerous unreachable states.

5.3.3 Is Each Constraint Complete? The theorems implemented in Z/Eves are based closely upon the formalisms provided in Chapter Three and are therefore likely to be complete with respect to the proposed constraints. Verification of the dynamic model is trivialized somewhat and as such, is obviously incomplete.

5.4 Research Findings

The majority of constraints proposed as part of this research were capable of being expressed and tested without the requirement for user interaction. The automation of much of the formal specification process and subsequent verification simplifies the somewhat overwhelming task and increases the value of formal methods in software engineering.

The results of this research are promising. The constraints determined in Chapter Three identify model attributes that must be verified in order to determine system correctness. The constraints also assist in the definition of OOA semantics for the purposes of automated verification. The implementation proves that the model may be transformed so as to provide the input to commercially available verification tools.

5.5 Conclusion

Automated validation and verification of object-oriented analysis models provides the software engineer with an effective and efficient manner in which to remove a great deal of the effort involved in the use of formal methods. The importance of V&V to correctness-preserving transformations systems cannot be stressed enough. This research provides a framework of constraints that when applied to an OOA system, help verify model correctness at the dynamic, functional, and structural levels.

This research suggests that the initial cost to construct an automated verification suite is worth the effort required due to the increased effectiveness and efficiency it offers to V&V of OOA models. Keeping in mind that the majority of automated tools are only effective for the system they are designed to be used in conjunction with, this research also indicates that implementation and use of such a verification suite is practical and valuable. For the test cases of this research, the theorems generated were relatively easy to prove with a currently available theorem prover—thus providing evidence of the applicability of the proposed verification techniques.

However, it must be stated that the decision as to whether a purpose built V&V suite should be used or a COTS system employed is of critical importance. Therefore, a needs analysis that identifies the constraints to be verified should be conducted prior to the selection of any verification tool.

5.6 Future Directions

A number of avenues for future work based upon this research exist

5.6.1 Further Formalism of Inheritance Towards Strict Inheritance.

Constraints 6 and 7 deal with subclass consistency and substitutability for the superclass invariant. Strict inheritance requires a subclass to maintain the structural, functional and dynamic properties of the superclass so that the child class is substitutable for the parent. This notion is also termed *extension*—the "two systems are indistinguishable if we cannot tell them apart without pulling them apart" [Milner].

The requirements of substitutability exceed the constraints implemented in this research. Constraints 6 and 7 only ensure correctness of the subclass invariant. The expressions of a subclass, be they pre- or postconditions, event or state invariants, or guard conditions must not further constrain the attributes inherited from the superclass. The functionality of operations must not be changed. Newly introduced operators are free to form any expression that does not violate any of the Constraints proposed in Chapter Three.

5.6.2 Further Verification of Aggregate Dynamic Models. System level dynamic model verification should include checking the domain dynamic model for the possibilities of starvation, deadlock, unreachable states, and correct termination. Constraint 29, all states must be reachable, is the only complex formal constraint of domain level dynamic models in this research.

Deadlock occurs when two (or more) classes are waiting for each other to send an event. Although complex to detect, the comprehensive output generated by Spin identifies which classes are waiting upon what events when simulation execution halts—thus identifying where the breakdown in model correctness occurs. Starvation, where a class cannot change state due to a lack of a certain resource, may also be determined in this fashion.

Specific behavior of a domain may be simulated by the injection of messages into the **init** portion of the Promela specification to ascertain if certain initial conditions lead to unreachable states, deadlock, or starvation. The creation of use case test sets to ensure desired behavior of the specified system would assist in validation of system behavior. It is possible in SPIN to introduce a set of events into and execute a dynamic model—the result of such could be compared to the expected behavior of the system.

5.6.3 Event Mapping in Promela. Work was started in the modeling of event maps. The prototype Promela generation visitor maps events according to their name via channel declarations. These mappings are therefore solely based upon event name matching. A more robust form of event mapping could be implemented by declaring a channel for each event map. This would allow for the passing of events to specific classes rather than their broadcast to all classes that have the matching receive event in their dynamic models.

5.6.4 Representation of Reals and Literal Strings in Z. The representation of string values was less than adequate in this research. One possible solution would be the use of an enumerated type (such as char) that declared the permissible values that an element of a string could take and the subsequent declaration of the type string such that it was a power set of sequences of char. This concept is illustrated in figure 36

$char ::= a \mid b \mid c \mid d \;..\; A \mid B \mid C \;..\; 1 \mid 2 \;..\; \mid 0 \;...$

$\mid \; string : P(\mathrm{seq}\, char)$

Figure 36. Declaration of a Literal String Type

The modeling of real numbers in Z is far more difficult to achieve than the modeling of literal strings. The constraints proposed in Chapter Three do not require differentiation between real types and integer types. Thus, the Z specification represents fixed and floating point real numbers as integers.

One limitation of the approach taken in this research is that the literal value is simply cast as an integer and therefore loses a great deal of its value. Multiplication of the value by its decimal place resolution (as done with the bounds of the type) would result in a more accurate representation.

More work is required in the specification of what it is that should be enforced when dealing with real types and these rules would become part of the semantic analyzer toolkit.

VI. Bibliography

R.T. Alexander, J.M. Bieman, J. Viega "Coping with Java Programming Stress," IEEE Computer, pp 30-38, April 2000

S.J. Andriole (Ed) "Software Validation Verification Testing and Documentation," Petrocelli Books, 1986

R. Balzer, T.E. Cheatham, C.C. Green, "Software Technology in the 1990's: Using a New Paradigm," IEEE Computer, vol 16, pp 39-45, November 1983

H.K. Berg, W.E. Boebert, W.R. Franta, T.G. Moher, "Formal Methods of Program Verification and Specification," Prentice Hall, 1982

F.P. Brooks, "The Mythical Man Month" 10th ed, Addison Wesley, 1998

C.N. Fischer, R.J. LeBlanc, Jr "Crafting a Compiler," Benjamin Cummings, 1988

R.W. Floyd "Assigning Meanings to Programs," Proc. Symposium on Applied Mathematics, American Mathematical Society, Vol. 19, 1967

M.D. Fraser, K. Kumar, and V.K. Vaishnavi, "Strategies for incorporating formal specifications in software development," Communications ACM, vol 37, pp 74-86, October 1994

E. Gamma, R. Helm, R. Johnson, J. Vlissides, "Design Patterns. Elements of Reusable Object-Oriented Software," Addison-Wesley, 1995

Gates, Gisselquist, Landry, "CSCE 594 Term Project,"Graduate School of Engineering, Air Force Institute of Technology (AU), Aug 1993

C. Ghezzi, M, Jazayeri, D. Mandrioli "Fundamentals of Software Engineering," Prentice Hall, 1991

A. Ginsberg, "Knowledge-Base Reduction: A New Approach to Checking Knowledge Bases for Inconsistency & Redundancy", Proc. 7th National Conference on Artificial Intelligence (AAAI 88), volume 2, pp 585-589.

S.J. Goldsack, S.J.H. Kent (Eds), "Formal Methods and Object Technology," Springer 1996.

R.P. Graham, JR, T.C. Hartrum, "The AFIT Wide Spectrum Object Modeling Environment : An AWSOME Beginning," to be presented at NAECON, Oct 2000

D.P. Gulch, C.B. Weinstock, "Model-Based Verification: A Technology for Dependable System Upgrade," Carnegie-Mellon University, Pittsburgh, Pa, 1998.

J.V. Guttag, E. Horrowitz, D.R. Musser "Abstract Data types and Software Validation," Comm. Of the ACM, Vol 21, No 12, 1978.

T.C. Hartrum, "An Object Oriented Formal Transformation System for Primitive Object Classes," Class Notes, Air Force Institute of Technology, Wright Patterson AFB, OH, Mar 1999

T.C. Hartrum, P.D, Bailor, "A Formal Extension to Object Oriented Analysis Using Z," Tech Report AFIT/EN/TR-94007, Air Force Institute of Technology, Wright Patterson AFB, OH, Oct 1994

T.C. Hartrum, P.D. Bailor, "Teaching Formal Extensions of Informal-Based Object Oriented Analysis Methodologies," in *Proceedings Software Engineering Education (7th SEI CSEE Conference),* (San Antonio, TX), pp 389-409, Jan 1994

C. Heitmeyer, J. Kirby, B. Labaw "Tools for Formal Specification, Verification and Validation Requirements" IEEE COMPASS, June 1997

R.S. Pressman, "Software Engineering,"fourth Ed, McGraw Hill, 1997

P. Meseguer, A.D. Preece, "Verification and Validation of Knowledge-Based Systems with Formal Specifications," http://www.csd.abdn.ac.uk/~ apreece/Pubs/ KER95.htm

A. Mili, "An Introduction to Formal Program Verification," Van Norstrand Reinhold Company, 1985

R. Milner, "A Calculus of Communicating Systems," Springer-Verlag, 1980

I. Meisels, Software Manual for Windows Z/Eves Version 2.0. Technical Report TR-97-5505-04f, ORA Canada, October 1999

I. Meisels, and Marak Saaltink. The Z/Eves Reference Manual (for Version 1.5). technical Report TR-97-5493-03d, ORA Canada, Septemver 1997

I. Meisels, Software Manual for Unix Z/Eves Version 1.5. Technical Report TR-97-6028-01c, ORA Canada, September 1997

P.A. Noe, "A Structured Approach to Software Tool Integration," Ms Thesis, AFIT/GCS/ENG/99M-14, Graduate School of Engineering, Air Force Institute of Technology (AU), Mar 1999. DTIC No. ADA361674

R.M. O'Keefe, O. Balci, E.P. Smith, "Validating Expert System Performance," IEEE Expert, 2(4):81-90, 1987

W. Polack "Compiler Specification and Verification," Springer-Verlag, 1981

B. Potter, J. Sinclair, D. Till "An Introduction to Formal Specification and Z" Prentice Hall, 1991

A.D. Preece, C. Grossner, T. Radhakrishnan, "Validating Dynamic Properties of Rule-Based Systems" http://www.csd.abdn.ac.uk/˜ apreece/Pubs

T.W. Pratt, "Programming Languages Design and Implementation," second Ed, Prentice Hall, 1984

"Refine User's Guide," Reasoning Systems, Palo Alto, California

J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen, "Object Oriented Modeling and Design", Prentice Hall, 1991

M. Saaltink, The Z/Eves User's Guide. Technical Report TR-97-5493-06. ORA Canada, October 1999

P.N. Scharbach (Ed) "Formal Methods: Theory and Practice,"CRC Press, 1989

M. Shaw, D. Garlan, "Software Architecture," Prentice Hall, 1996

R.C. Shock "Software Architecture Course Notes," Department of Computer Science and Engineering, Wright State University, 1999

S. Stepney, R.Barden, D. Cooper, "Object Orientation in Z", Springer-Verlag, 1992

M. Suwa, A.C. Scott, E.H. Shortliffe, "An Approach to Verifying Completeness and Consistency in a Rule-Based Expert System", AI Magazine 3(4):16-21, 1982

A. Verdaguer, P. Meseguer "Verification of Multi-Level Rule-Based Expert Systems: Theory and Practice", International Journal of Expert Systems: Research and Applications, 6(2):163-192, 1993

J.B. Wordsworth "Software Development with Z", Addison Wesley, 1992

Z/Eves Irwin Meisels and Mark Saaltink. The Z/Eves Reference Manual. Technical Report TR-99-5493-03d, ORA Canada, September 1997

Appendix A. AFIT Wide Spectrum Object Modeling Environment and Language

The aim of this appendix is to provide a brief introduction to the history and development of the AFIT Wide Spectrum Object Modeling Environment (AWSOME), a description of the AWSOME metamodel, and finally, to provide the production rules upon which the language is developed. The productions are provided in order to explicitly define the required structure of any AWSOME analysis model created without making use of the language and parser but wishing to use the verification tools provided as part of this research.

A.1 Introduction

AWSOME is the culmination of the extension and redesign of AFITtool, a correctness-preserving formal software synthesis system developed by students and faculty at the United States Air Force Institute of Technology (AFIT). The aim of AFITtool was to provide for proof of concept of much of the software engineering related research being conducted at AFIT. This tool was implemented in Reasoning's high level language Refine over a period of six years.

Being a wide-spectrum environment, AWSOME is capable of modeling object-oriented systems at various stages of the software development cycle. AWSOME is capable of specification level modeling where much of the system is specified in terms of formal expressions, for example, methods are specified in terms of pre- and postconditions, while classes, their states and events are defined over invariant expressions.

Elements specified in AWSOME may be reused extensively via the facilitation of an object-oriented repository that allows for the harvesting of pre-defined classes to be added to the problem domain currently being specified/analyzed.

Via the use of correctness-preserving semi-autonomous transforms, the analysis-level model may be transformed into a design-level representation. Transformations exist to take the analysis model to the design level as follows:

1. associations are replaced with single and bi-directional access types,
2. associative objects are transformed into to classes,
3. dynamic models are replaced class methods,
4. class invariants become statements of class methods, and
5. method pre- and postconditions are expressed as statements within the method body.

Another series of transformations leads from the design model to executable code. AFITtool is capable of generating Ada code, however, in order to make AWSOME more widely applicable, not only was it implemented in Java, but the output language of choice was also selected to be Java. Some work has been accomplished dealing with the generation of C++ from AWSOME models as well, however, this capability is not as extensive as its Java counterpart.

A.2 The AWSOME Metamodel

This section describes the metamodel that forms the template for analysis models represented by AWSOME ASTs. Following the lines of object-oriented analysis, the AWSOME metamodel structure will be described from the perspective of inheritance and aggregation. Approximately 100 classes exist in the AWSOME metamodel.

A.2.1 AWSOME AST Class Naming. WsIdentifier facilitates the naming of Ws classes. WsIdentifierRef provides the capability to refer to instances of WsIdentifier. The linking phase of AWSOME AST generation provides each instance of WsIdentifierRef with a pointer to its corresponding WsIdentifier.

A.2.2 The Inheritance Model. The root of AWSOME's inheritance hierarchy is the abstract class WsObject. The immediate children of WsObject that are of importance to analysis models and interesting to address further are WsDeclaration, WsExpression, WsDynamicModel, WsAttribute, and WsMethod.

A.2.2.1 WsDeclaration. From WsDeclaration spawn the subclasses for representing data types, variables, constants, states, events, sets, sequences, bags, functions, procedures, and associations. Essentially, from WsDeclaration is inherited attributes and methods related to AST node identification.

A.2.2.2 WsExpression. It is the subclasses of WsExpression that define the structure of expressions contained within an analysis model abstract syntax tree (AST). The WsExpression family of objects deals with the representation of identification references (via WsName and its subclasses), proposition quantification, and binary and unary expressions. It is expected that the binary and unary expression portion of the tree will be extended in order to provide a greater degree of expression functionality to sets and encompass sequence and bag operators.

A.2.2.3 WsAttribute. WsAttribute defines the AWSOME model for representation of class attributes. It is a direct subclass of WsObject and extends it by the addition of attributes describing certain characteristics of a class attribute such as whether it is private or public and by providing database query information [Buckwalter]. A WsAttribute has a WsDataObject as an aggregate component—it this data object that the characteristics of WsAttribute define.

A.2.2.4 WsMethod. Similar to WsAttribute, WsMethod provides attributes to another Ws class. WsMethod defines the visibility of a class method—a private method is only visible within the scope of the class while a public method may be invoked by other classes of the domain. The method pre- and postconditions are described by an aggregate component of WsMethod— WsSubprogram. It should

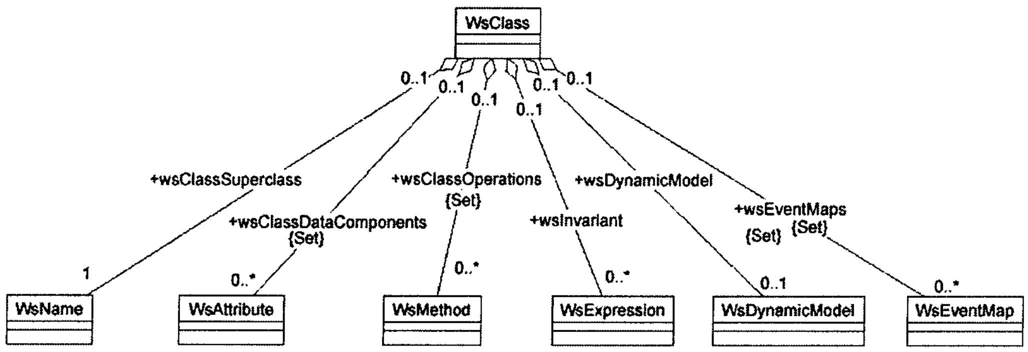


Figure 37. WsClass.

be noted that although public methods may be invoked by other classes, object communication within this research is conducted solely by event passing.

A.2.3 The Aggregation Model. The root node of any AWSOME analysis model is the artifact (don't ask me why—I don't really know). An artifact comprises of one or more packages that form containers for related domain elements. Each package consists of a set of declarations that may be used to declare classes, associations, associative objects, type definitions, constant declarations, and other packages. Figure 37 illustrates the aggregation diagram of the WsClass. At this point it is worth noting that every class of the AWESOME metamodel is prepended with **Ws** which identifies the class as an AWSOME metamodel component.

A.2.3.1 WsClass. It can be seen in Figure 37 that a WsClass consists of a set of attributes, a set of methods (WsMethod), a dynamic model (WsDynamicModel), an invariant expression (WsExpression), an event map set (WsEvent), and a pointer (WsIdentifierRef) to the class' parent class (if, in fact, there is one).

A.2.3.2 WsDynamicModel. AWSOME dynamic models are constructed from the aggregation of a set each of events (WsEvent), transitions (WsTransition) and states (WsState). The set of states exhaustively defines the state space

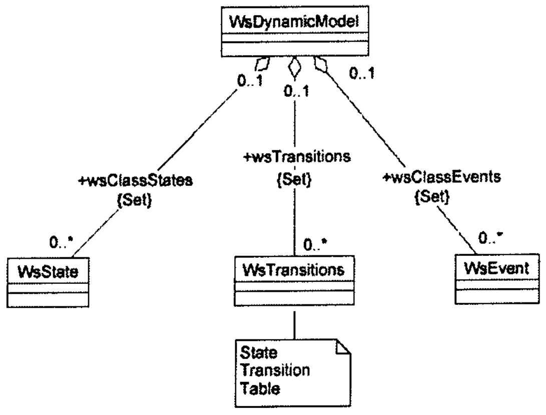


Figure 38. WsDynamicModel.

of a class, the set of events exhaustively defines the send and receive events by which the class communicates with other classes of the domain. Events may contain parameters that allow classes to pass arguments for use in transition actions. Figure 38 illustrates the WsDynamicModel structural diagram.

A.2.3.3 WsTransition. Each transition of a dynamic model is described by an instance of the aggregate class WsTransition. This class defines the event and guard condition precipitating the transition (WsExpression), the action (method invocation) that takes place and any events sent as a product of the transition between states. It should be noted for a given transition that parameters of the receive event must map to the **in** parameters of the action and that parameters of the send events must map to the **out** parameters of the action. Figure 39 depicts the WsTransition structural model.

A.3 AWL Syntax

This section provides production rules of the grammar that dictate the AWSOME language (AWL) syntax. The AWL parser is implemented with the parser design tool JavaCC that provides the ability to define a grammar and generate a

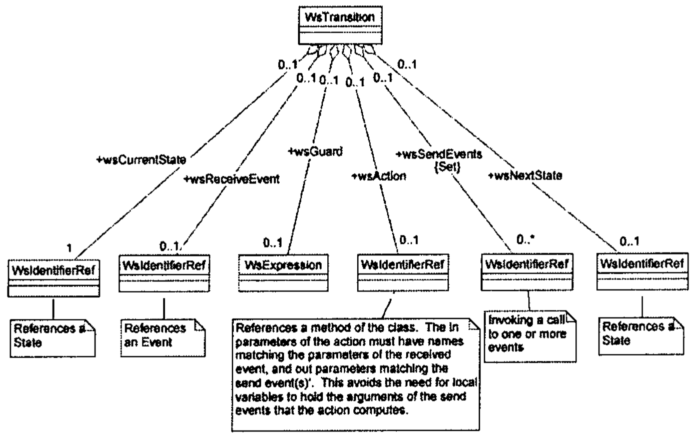

WsTransition
0..1
0..1
0..1
0..1
0..1
0..1
+wsCurrentState
+wsReceiveEvent
+wsGuard
+wsAction
+wsSendEvents {Set}
+wsNextState
1
0..1
0..1
0..1
0..*
0..1
WsIdentifierRef
WsIdentifierRef
WsExpression
WsIdentifierRef
WsIdentifierRef
WsIdentifierRef
References a State
References an Event
References a method of the class. The in parameters of the action must have names matching the parameters of the received event, and out parameters matching the send event(s)'. This avoids the need for local variables to hold the arguments of the send events that the action computes.
Invoking a call to one or more events
References a State

Figure 39. WsTransition.

compilable Java implementation of the parser. To provide the level of functionality required within the AWL parser, augmentation of the JavaCC file with Java code was required. Except for <ID>, which represents an identification string, terms embedded within <> are keywords.

```
compilation unit ::   Package <EOF>

package :: <PACKAGE> Identifier <IS> ( Declaration | Package)*
           <END> <PACKAGE> ";"

Declaration :: TypeDeclaration | Subprogram | DataObject | Association

Identifier ::  t = <ID>

TypeDeclaration :: <TYPE> Identifier <IS> TypeDefinition  | Class

TypeDefinition :: AbstractTypeDefinition
|   ArrayTypeDefinition
|   AccessTypeDefinition
|   BagTypeDefinition
|   DerivedTypeDefinition
|   EnumerationTypeDefinition
|   IntegerTypeDefinition
|   RealTypeDefinition
|   RecordTypeDefinition
|   SequenceTypeDefinition
|   SetTypeDefinition
|   UnionTypeDefinition

AbstractTypeDefinition :: <ABSTRACT> ";"
```

```
AccessTypeDefinition :: <ACCESS> TypeName ";"

ArrayTypeDefinition :: <ARRAY> "[" TypeName "]" <OF> TypeName ";"

BagTypeDefinition :: <BAG> <OF> TypeName ";"

DerivedTypeDefinition :: IdentifierRef <WHERE> Expression ";"

EnumerationTypeDefinition :: "(" EnumerationValue( "," EnumerationValue)*
                             ")" ";"

EnumerationValue ::  Identifier

IntegerTypeDefinition :: <RANGE> ( "*" | Expression) ".."
                         ( "*" | Expression) ";"

RealTypeDefinition :: (<DELTA> Expression | <DIGITS> Expression
        [ <BASE> Expression]) <RANGE> ( "*" | Expression) ".."
        ( "*" | Expression) ";"

RecordTypeDefinition :: <RECORD> ( UninitializedVariable )+ <END> <RECORD> ";"

SequenceTypeDefinition :: <SEQUENCE> <OF> TypeName ";"

SetTypeDefinition :: <SET> <OF> TypeName ";"

UnionTypeDefinition :: <UNION> ( UninitializedVariable)+ <END> <UNION> ";"

Class :: <CLASS> Identifier <IS>
        [<ABSTRACT>]
        [TypeName <WITH>]
        (visibilityPrefix ( Attribute | Method))*
            [ <INVARIANT> Expression]
            [ DynamicModel ]
            <END> <CLASS> ";"

visibilityPrefix :: <PRIVATE> |  <PUBLIC>

Attribute :: DataObject

Method :: [ <ABSTRACT>][ <CLASS>]

DynamicModel :: <DYNAMIC> <MODEL> <IS>
            ( Event)*
        ( State)*
        <TRANSITION> <TABLE> <IS>
        ( Transition)*
            <END> <TRANSITION> <TABLE> ";"
        <END> <DYNAMIC> <MODEL> ";"
```

```
Transition :: <IN> IdentifierRef
        <ON> IdentifierRef
        [ <IF> Expression]
        [ <DO> SubprogramName]
            [ <SEND> IdentifierRef ( "," IdentifierRef)* ]
            <TO> IdentifierRef";"

State :: <STATE> Identifier [ <INVARIANT> Expression ] ";"

Event :: <EVENT> Identifier "("[ Parameter ( "," Parameter)* ] ")"
        [ <ASSUMES> Expression ]";"

Subprogram :: Procedure | Function

Procedure :: <PROCEDURE> Identifier "("[Parameter ( "," Parameter) * ] ")"
        [ <ASSUMES> Expression]
        [ <GUARANTEES> Expression]
        [ <IS> ( DataObject)* <BEGIN>  (Statement)* <END> ";" ]

Function :: <FUNCTION> Identifier "("[ Parameter ( "," Parameter ) * ] ")"
        ":" TypeName
        [ <ASSUMES> Expression]
        [ <GUARANTEES> Expression]
        [ <IS> ( DataObject)* <BEGIN> ( Statement)* <END> ";" ]

Association :: RegularAssociation | Aggregation | AssociativeObject

RegularAssociation : : <ASSOCIATION> Identifier <IS>AssociationEnd
        ( AssociationEnd )+
        [ <INVARIANT> Expression ]
            <END> <ASSOCIATION> ";"

AssociationEnd :: <ROLE> Identifier ":" TypeName <MULTIPLICITY> TypeName
            ( "," TypeName )* ";"

Aggregation : : <AGGREGATION> Identifier <IS>
        ParentEnd
        ChildEnd
        [ <INVARIANT> Expression ] <END> <AGGREGATION> ";"

ParentEnd :: <PARENT> Identifier ":" TypeName <MULTIPLICITY> TypeName
            ( "," TypeName)* ";"

ChildEnd :: <CHILD> Identifier ":" TypeName <MULTIPLICITY> TypeName
            ( "," TypeName)* ";"

AssociativeObject :: <ASSOCIATIVEOBJECT> Identifier <IS>
        AssociationObjEnd (AssociationObjEnd)+
        ( visibilityPrefix ( Attribute | Method))*
        [ <INVARIANT> Expression ]
        <END> <ASSOCIATIVEOBJECT> ";"
```

```
AssociationObjEnd :: <ROLE> Identifier ":" TypeName <MULTIPLICITY> TypeName
            ( "," TypeName)* [ <QUALIFIED> <BY> IdentifierRef]";"

Expression :: OrExpression ( "=>" OrExpression )*

ExpList :: [ Expression ( "," Expression)* ]

OrExpression :: AndExpression ( <OR> AndExpression)*

AndExpression :: RelationalExpression ( <AND> RelationalExpression )*

RelationalExpression :: AddExpression ["="  AddExpression
        | "/=" AddExpression
        | "<"  AddExpression
        | "<=" AddExpression
        | ">"  AddExpression
        | ">=" AddExpression
        |  <IN>       AddExpression
        | <SUBSET>   AddExpression
        | <SUBSETEQ> AddExpression ]

AddExpression :: MultiplyExpression ("+" MultiplyExpression
        | "-" MultiplyExpression)*

MultiplyExpression :: ExponentExpression
            ("*"   ExponentExpression
            | "/"   ExponentExpression
        | <MOD> ExponentExpression
        | <INTERSECT> ExponentExpression
        |  <UNION> ExponentExpression)*

ExponentExpression :: UnaryExpression ("**" ExponentExpression)*

UnaryExpression :: PrimaryExpression
        |   <NOT> UnaryExpression
        |   "-"   UnaryExpression
        |   "+"   UnaryExpression

PrimaryExpression ::    AccessExpression
        |   Allocator
        |   LOOKAHEAD ( ("{*" | "[" | "{" ) Expression "|") ContainerFormer
        |   LOOKAHEAD (SubprogramName "(") FunctionCall
        |   LiteralConstant
        |   LOOKAHEAD (TypeName "'" "(") TypeConversion
        |   Name
        |   QuantifiedExpression
        |   "(" Expression ")"

AccessExpression :: "&" Name
```

```
Allocator :: <NEW> TypeName

ContainerFormer :: BagFormer
        | SequenceFormer
        | SetFormer

BagFormer :: "{*" Expression "|" LogicalVarList Expression "*}"

SequenceFormer :: "[" Expression "|" LogicalVarList Expression "]"

SetFormer :: "{" Expression "|" LogicalVarList Expression "}"

LogicalVariable :: Identifier ":" TypeName

LogicalVarList :: "(" [ LogicalVariable ( "," LogicalVariable )* ] ")"

FunctionCall :: SubprogramName "(" ExpList ")"

SubprogramName :: SimpleName
        ( ("[" Expression "]"
        |   "^" #Dereference
        |   "'" #Tick )*
        "." IdentifierRef)*

LiteralConstant ::
            LiteralCharacter
        | LiteralContainer
        | LiteralInteger
        | LiteralNull
        | LiteralReal
        | LiteralString

LiteralCharacter :: t = <CHARACTER_LITERAL>

LiteralContainer :: LiteralBag
        | LiteralSequence
        | LiteralSet

LiteralBag ::   "{*" ExpList(listLen) "*}"

LiteralSequence :: "[" ExpList "]"

LiteralSet :: "{" ExpList(listLen) "}"

LiteralInteger :: t = <INTEGER_LITERAL>

LiteralNull :: <NULL>

LiteralReal :: t = <REAL_LITERAL>

LiteralString :: t = <STRING_LITERAL>
```

```
 Name :: SimpleName ( NameSuffix )*

 SimpleName :: IdentifierRef | This

 IdentifierRef :: t = <ID>

 This :: <THIS>

 NameSuffix :: "." IdentifierRef
         |   "[" Expression "]"
         |   "^"
         |   "'"

 SelectedComponent :: SimpleName ( ("[" Expression "]"
               |   "^"
               |   "'")*
            "." IdentifierRef)+

 TypeName :: IdentifierRef ( "." IdentifierRef )*

  QuantifiedExpression :
 ExistentialExpression
         |   UniqueExpression
         |   UniversalExpression

 ExistentialExpression :: <EXISTS> LogicalVarList "(" Expression ")"

 UniqueExpression :: <UNIQUE> LogicalVarList "(" Expression ")"

 UniversalExpression :: <FORALL> LogicalVarList "(" Expression ")"

 TypeConversion :: TypeName "'" "(" Expression ")"

 Statement :: LabeledStatement
         | BasicStatement

 LabeledStatement :: ( Label )+ BasicStatement

 BasicStatement :: Name ":=" Assignment
         | Iteration
         | Jump
         | ProcedureCall
         | Selection

 Label :: "<<" Identifier ">>"

 Assignment :: Name ":=" Expression ";"

 Jump :: <GOTO> IdentifierRef ";"
```

```
Iteration :: <WHILE> Expression <DO> ( Statement )* <END> <DO> ";"

ProcedureCall :: SubprogramName "(" ExpList(listLen) ")" ";"

Selection :: <IF> Expression <THEN> ( Statement)*
        [ <ELSE> ( Statement)* ]
        <END> <IF> ";"

DataObject :: Identifier ":" DataObjectTail

DataObjectTail :: <CONSTANT> TypeName [":=" Expression] ";"
        | TypeName [":=" Expression] ";"

UninitializedVariable :: Identifier ":" TypeName ";"

Constant :: Identifier ":" TypeName [ ":=" Expression ] ";"

Parameter :: Identifier ":" (( <IN> [ <OUT> ] ) | <OUT> {out = true;} )
             TypeName
```

Appendix B. The Cruise Missile Problem Domain

B.1 Introduction

This appendix contains the test case used throughout this research. Contained within this appendix is the AWL file that forms the object-oriented analysis model, UML and Z representations of the analysis model, and samples of the generated Z/Eves and Promela output files.

B.2 The UML Analysis Model

Although the specification is implemented in AWL, initial analysis made use of UML, and as an aide to clarity, the UML version of the specification has been reproduced here. It will become apparent that without augmentation with prose, the UML model is not capable of fully specifying the analysis model as specified with AWL.

B.3 The AWL Analysis Model

It will be noted that enumerated type values seem somewhat unwieldly—this is due tothe fact that Z does not allow different enumerated types to have elements with equal values, thus the class name has been appended to each state name.

```
package cruiseMissile is

/*
 * This is a modified version of the cruise missile system modelled by
 * Gates, Giselquist, and Landry in August of 1993.   This code forms
 * the test case used as part of the research conducted for the Masters
 * thesis "Validation and Verification of Formal Specifications in
 * Object-Oriented Software Engineering.
 *
 * References to the simulation system and substates of the airframe
 * have been removed
 *
 * - Thomson
 *
 */

type char is abstract;
type time is abstract;
type timer is abstract;
type string is sequence of char;
type heading is delta 0.01 range 0.0 .. 360.0;
// type position is delta 0.001 range 0.0 .. 1000.0;
type speed is delta 0.1 range 0.0 .. 2000.0;
type mass is delta 0.1 range 0.0 .. 9000.0;
type yield is delta 0.1 range 0.0 .. 20000.0;
type real is delta 0.0001 range 0.0 .. 13000.0;
type bigReal is delta 0.0001 range 0.0 .. 27000.0;
type coordinate is delta 0.001 range - 4000 .. 4000;
type cmStates is (startcruiseMissile, preLaunchcruiseMissile,
     flyingcruiseMissile);
type afStates is (startairframe, preLaunchairframe,
     poweredFlightairframe, maneuveringairframe, inertialFlightairframe);
type navStates is (startnav, waitnav, readSensorsnav);
type flightDirectorStates is (startflightDirector, idleflightDirector,
     maneuveringflightDirector);
type guidanceStates is (startguidanceSystem,idleguidanceSystem,
     processingguidanceSystem,terminalguidanceSystem);
type throttleStates is (startthrottle,idlethrottle);
type warheadArmed is (t,f);
type engineStates is (startengine, idleengine);
type realWeight is delta 0.01 range 0.0 .. 100.0;
pi : constant bigReal := 3.1414159;

class vector is
    private x : coordinate;
    private y : coordinate;
    private z : coordinate;
    private magnitude : bigReal;
end class;

class point is vector with
end class;

type flightPath is sequence of point;

class position is vector with
end class;

type route is sequence of position;

class velocity is vector with
end class;

class acceleration is vector with
end class;

class fuelTank is
    private fuelLevel : bigReal;
    private outputFlowRate : bigReal;
    private fuelDensity : bigReal;
 end class;

 class missileFuelTank is fuelTank with
    private fixedWeight : realWeight;
    private tankWeight : realWeight;
    private capacity : bigReal;
    private inputFlowRate : bigReal;

    invariant fixedWeight = tankWeight + (fuelDensity * capacity / 2)
       and inputFlowRate = 0

    private procedure initializeMissileFuelTank()
    guarantees fuelLevel' = capacity and outputFlowRate' = 0

    private procedure changeFlow(actualFlowRate : in bigReal)
    guarantees outputFlowRate' = actualFlowRate? and
        fuelLevel' = fuelLevel and capacity' = capacity and
        tankWeight' = tankWeight and fuelDensity' = fuelDensity

    dynamic model is
    event initMissileTank();
    event changeTankFlow();
    event tankEmpty();

    state start;
    state empty; //error - never invoked
    state full invariant fuelLevel = capacity and outputFlowRate = 0;
    state using invariant fuelLevel >=  0 and fuelLevel
            <= capacity and outputFlowRate > 0;

    transition table is
        in start on initMissileTank if true do initializeMissileFuelTank();
           to full;
        in full on changeTankFlow if true do changeFlow(); to using;
        in using on changeTankFlow if true do changeflow(); to using;
        in using on tankEmpty if true do changeFlow(); send outOfFuel();
           to empty;
    end transition table;
    end dynamic model;
end class;

class navigationSystem is
    private navState : navStates;

    dynamic model is
    event initAirframe ();
    event tankEmpty();
    event getPosition();
    event changeCourse();
    event doManeuverComplete();
    state start;
    state wait;
    state readSensors;
    transition table is
        in start on initNavigationSystem if true to wait;
        in wait on updatePosition if true to readSensors;
        in readSensors on AUTOMATIC if true send estimatePosition();
           to wait;
    end transition table;
    end dynamic model;
end class;

class flightProfile is
    private timeOnTarget : time;
    private wayPoints : flightPath;

    /* private procedure addPointToRoute(p : in point)
    guarantees wayPoints' = cat(wayPoints,p)

    private procedure removePointFromRoute()
    guarantees wayPoints' = rest(wayPoints)

    dynamic model is
    event initFlightProfile();
    event addWayPoint();
    event removeFirstWayPoint();

    state start;
    state idle;
    transition table is
       in start on initFlightProfile if true to idle;
       in idle on addWayPoint if true do addPointToRoute(); to idle;
       in idle on removeFirstWayPoint if true do
          removePointFromRoute(); to idle;
    end transition table;
    end dynamic model;
end class;

class guidanceSystem is
    private profile : flightProfile;
    private wayPoints : flightPath;
    private guidanceState : guidanceStates;
    private chronometer : timer;

    private procedure initializeGuidanceSystem()

    private procedure output(relTime : in time, message : out string)
    guarantees true /* message = "the cruise missile should reach the
        target at " + char(relTime) */
    private procedure RemoveFirstRoutePoint()
    guarantees wayPoints' = rest(wayPoints)

   dynamic model is
    event initGuidanceSystem();
    event doLaunch();
    event estimatePosition();
    event outOfFuel();
    event armMissile();
    event updatePosition();

    state start;
    state idle;
    state processing;
    state terminal;

    transition table is
        in start on initGuidanceSystem if true to initializeGuidanceSystem;
        in idle on doLaunch if true send updatePosition(); to idle;
        in idle on estimatePosition if true to processing;
        in idle on outOfFuel if true send updatePosition(); to terminal;
        in processing on AUTOMATIC if profile.route~head =
           profile.route~tail do removeFirstRoutePoint(); send armMissile();
           to terminal;
        in processing on AUTOMATIC if not(profile.route~head =
           profile.route~tail) to terminal;
    end transition table;
    end dynamic model;
end class;

class flightDirector is
    private flightDirectorState : flightDirectorStates;

    private procedure initialize()
    guarantees flightDirectorState' = idleflightDirector

    dynamic model is
    event errorSignals();
    event initFlightDirector();
    event maneuverComplete();

    state start invariant flightDirectorState = startflightDirector;
    state idle invariant flightDirectorState = idleflightDirector;
    state maneuvering invariant flightDirectorState =
          maneuveringflightDirector;

    transition table is
        in start on initFlightDirector if true do
           initializeFlightDirector(); to idle;
        in idle on errorSignals if true send changeCourse(); setElevation();
           setThrottle(); to maneuvering;
        in maneuvering on maneuverComplete if true to idle;
    end transition table;
    end dynamic model;
end class;

class avionicsSoftware is
    private navSys : navigationSystem;
    private guidSys : guidanceSystem;
    private director : flightDirector;

    private procedure initializeAvionicsSoftware()

    dynamic model is
    event initNavigationSystem();
    event initGuidanceSystem();
    event initFlightDirector();
    event initAvionicsSoftware();

    state start;
    state avionicsSoftwareInitialized;

    transition table is
       in start on initAvionicsSoftware if true do initializeAvionicsSoftware();
          send initNavigationSystem(); to avionicsSoftwareInitialized; //send
          initGuidanceSystem send initFlightdirector
    end transition table;
    end dynamic model;
end class;

class warhead is
    private weight : mass;
    private munitionType : string;
    private explosiveForce : yield;
    private armed : warheadArmed;

    invariant weight > 0.0 and explosiveForce >= 0.0

    private procedure initializeWarhead()
    guarantees armed' = f

    private procedure armWarhead()
    guarantees armed' = t

    dynamic model is
    event initWarhead();
    event armMissile();

    state start;
    state unarmed invariant armed = f;
    state armed invariant armed = t;
    transition table is
        in start on AUTOMATIC if true do initializeWarhead(); to unarmed;
        in unarmed on armMissile if true do armWarhead(); to armed;
    end transition table;
    end dynamic model;
end class;

class airframe is
    private pos : position;
    private accl : acceleration;
    private vel : velocity;
    private afState : afStates;
    private heading : real;
    private elevation : real;

    invariant heading <= (2*pi)  and heading >= 0.0 and elevation <= (pi/2)
        and elevation >= (-pi/2)

    private procedure initializeAirframe()
    guarantees elevation' = (pi/2) and heading' = 0.0 and pos'.x = 0.0
        and pos'.y = 0.0 and pos'.z = 0.0 and vel'.x = 0.0
        and vel'.y = 0.0 and vel'.z = 0.0 and accl'.x = 0.0
        and accl'.y = 0.0 and accl'.z = 0.0

    private procedure setPosition() is

    dynamic model is
    event initAirframe ();
    event doLaunch();
    event tankEmpty();
    event estimatePosition();
    event changeCourse();
    event maneuverComplete();
    event outOfFuel();

    state start invariant afState= startairframe;
    state preLaunch invariant afState = preLaunchairframe;
    state poweredFlight invariant afState = poweredFlightairframe;
    state maneuvering invariant afState = maneuveringairframe;
    state inertialFlight invariant afState = inertialFlightairframe;

    transition table is
        in initial on initAirframe if true to airframeInit;
        in airframeInit on AUTOMATIC if true send initDone();
           to poweredFlight;
        in poweredFlight on tankEmpty if true to inertialFlight;
        in poweredFlight on getPosition if true send positionCurrent();
           to poweredFlight;
        in poweredFlight on changeCourse if true to maneuvering;
        in maneuvering on doManeuverComplete if true to poweredFlight;
    end transition table;
```

```
      end dynamic model;
   end class;

   class throttle is
       private fuelFlow : real;
       private throttleState : throttleStates;
       private maximumFlowRate : real;
       private actualFlowRate : real;

       private procedure changeFuelFlow()

       dynamic model is
       event initThrottle();
       event changeTankFlow();

       state start invariant throttleState = startthrottle;
       state idle invariant throttleState = idlethrottle;
       transition table is
           in start on initThrottle if true to idle;
           in idle on setThrottle do changeFuelFlow(); send changeTankFlow();
              to idle;
       end transition table;
       end dynamic model;
  end class;

  class jetEngine is
      private thrust :  real;
      private maximumFuelFlowRate : real;
      private currentFuelFlowRate : real;
      private engineState : engineStates;

      dynamic model is
      event initEngine();
      event changeTankFlow();

      state start invariant engineState = startengine;
      state idle invariant engineState = idleengine;
      transition table is
          in start on initEngine if true to idle;
          in idle on changeTankFlow to idle;
      end transition table;
      end dynamic model;
end class;

class propulsionSystem is
    private fuelFeed : throttle;
    private engine : jetEngine;
    private tank : missileFuelTank;

    invariant (tank.fuelLevel = 0.0 => fuelFeed.maximumFlowRate = 0.0)
        and (tank.fuelLevel > 0.0 =>(fuelFeed.maximumFlowRate =
        engine.maximumFuelFlowRate)) and
        (engine.currentFuelFlowRate = fuelFeed.actualFlowRate)

    dynamic model is
    event initPropulsionSystem();
    event initThrottle();
    event initEngine();
    event initMissileFuelTank();

    state start;
    state idle;

    transition table is
        in start on initPropulsionSystem if true send initThrottle();
           to idle; //send initEngine() send missileFuelTank();
        in preLaunch on launch if true send doLaunch(); to flight;
    end transition table;
    end dynamic model;
end class;

class cruiseMissile is
    private propulsion : propulsionSystem;
    private frame : airframe;
    private payload : warhead;
    private avionics : avionicsSoftware;
    private cmState : cmStates;

/* functional model */
    private procedure initializeCruiseMissile()
    guarantees cmState' = preLaunchcruiseMissile

    dynamic model is
    event launch ();
    event outOfFuel();
    event courseUpdate();
    event maneuverComplete();
    event doInit();
    event initDone();
    event intiAirframe();

    state start invariant cmState = startcruiseMissile;
    state preLaunch invariant cmState = preLaunchcruiseMissile;
    state flying invariant cmState = flyingcruiseMissile;

    transition table is
        in start on AUTOMATIC if true do initializeCruiseMissile(); send
           initPropulsionSystem(); to preLaunch; //send initAirframe()
           send initWarhead() send initAvionicsSoftware();
        in preLaunch on launch if true send doLaunch(); to flight;
    end transition table;
    end dynamic model;
end class;
end package;
```

Appendix C. Z/Eves Specification of Cruise Missile

$[char]$

$[time]$

$[timer]$

$string : \mathbb{P}(\operatorname{seq} char)$

$heading : \mathbb{P}\,\mathbb{Z}$

$\forall x : heading \bullet x \geq 0$

$\forall x : heading \bullet x \leq 360$

$speed : \mathbb{P}\,\mathbb{Z}$

$\forall x : speed \bullet x \geq 0$

$\forall x : speed \bullet x \leq 2000$

$mass : \mathbb{P}\,\mathbb{Z}$

$\forall x : mass \bullet x \geq 0$

$\forall x : mass \bullet x \leq 9000$

$$\begin{array}{|l}
yield : \mathbb{P}\,\mathbb{Z} \\
\hline
\forall x : yield \bullet x \geq 0 \\
\forall x : yield \bullet x \leq 20000
\end{array}$$

$$\begin{array}{|l}
real : \mathbb{P}\,\mathbb{Z} \\
\hline
\forall x : real \bullet x \geq 0 \\
\forall x : real \bullet x \leq 13000
\end{array}$$

$$\begin{array}{|l}
bigReal : \mathbb{P}\,\mathbb{Z} \\
\hline
\forall x : bigReal \bullet x \geq 0 \\
\forall x : bigReal \bullet x \leq 27000
\end{array}$$

$$\begin{array}{|l}
coordinate : \mathbb{P}\,\mathbb{Z} \\
\hline
\forall x : coordinate \bullet x \geq -4000 \\
\forall x : coordinate \bullet x \leq 4000
\end{array}$$

$cmStates ::= startcruiseMissile \mid preLaunchcruiseMissile \mid flyingcruiseMissile$

$afStates ::= startairframe \mid preLaunchairframe \mid poweredFlightairframe \mid$
$maneuveringairframe \mid inertialFlightairframe$

$navStates ::= startnav \mid waitnav \mid readSensorsnav$

$flightDirectorStates ::= startflightDirector \mid idleflightDirector \mid maneuveringflightDirector$

$guidanceStates ::= startguidanceSystem \mid idleguidanceSystem \mid processingguidanceSystem \mid terminalguidanceSystem$

$throttleStates ::= startthrottle \mid idlethrottle$

$warheadArmed ::= t \mid f$

$engineStates ::= startengine \mid idleengine$

$realWeight : \mathbb{P}\, \mathbb{Z}$

$\forall x : realWeight \bullet x \geq 0$

$\forall x : realWeight \bullet x \leq 100$

$pi : bigReal$

$pi = 3$

vector

$x : coordinate$

$y : coordinate$

$z : coordinate$

$magnitude : bigReal$

$true$

point

$vector$

$true$

$flightPath : P(\mathrm{seq}\, point)$

position

$vector$

$true$

$route : P(\mathrm{seq}\, position)$

velocity

$vector$

$true$

acceleration

$vector$

$true$

---

fuelTank

$fuelLevel : bigReal$

$outputFlowRate : bigReal$

$fuelDensity : bigReal$

$true$

---

missileFuelTank

$fuelTank$

$fixedWeight : realWeight$

$tankWeight : realWeight$

$capacity : bigReal$

$inputFlowRate : bigReal$

$((fixedWeight = (tankWeight + ((fuelDensity * capacity) \operatorname{div} 2))) \wedge$

$(inputFlowRate = 0))$

---

initializeMissileFuelTank

$\Delta missileFuelTank$

$((fuelLevel' = capacity)$

$\wedge (outputFlowRate' = 0))$

Theorem 1 *initializeMissileFuelTankisok*
$\exists missileFuelTank \bullet initializeMissileFuelTank$

prove by reduce; undo;

changeFlow

$\Delta missileFuelTank$

$actualFlowRate? : bigReal$

$(((((outputFlowRate' = actualFlowRate?) \land (fuelLevel' = fuelLevel))$

$\land (capacity' = capacity)) \land (tankWeight' = tankWeight)) \land$

$(fuelDensity' = fuelDensity))$

Theorem 2 *changeFlowisok*
$\exists missileFuelTank \bullet changeFlow$

prove by reduce; undo;

missileFuelTankinitMissileTank

$\Xi missileFuelTank$

$(fuelLevel > 0)$

Theorem 3 *missileFuelTankinitMissileTankisok*
$\exists missileFuelTank \bullet missileFuelTankinitMissileTank$

prove by reduce; undo;

missileFuelTankchangeTankFlow

$\Xi missileFuelTank$

$true$

Theorem 4 *missileFuelTankchangeTankFlowisok*
$\exists missileFuelTank \bullet missileFuelTankchangeTankFlow$

prove by reduce; undo;

$missileFuelTanktankEmpty$

$\Xi missileFuelTank$

$true$

Theorem 5 *missileFuelTanktankEmptyisok*
$\exists missileFuelTank \bullet missileFuelTanktankEmpty$

prove by reduce; undo;

$missileFuelTankoutOfFuel$

$\Xi missileFuelTank$

$(fuelLevel = 0)$

Theorem 6 *missileFuelTankoutOfFuelisok*
$\exists missileFuelTank \bullet missileFuelTankoutOfFuel$

prove by reduce; undo;

$missileFuelTankstart$

$missileFuelTank$

$true$

Theorem 7 *missileFuelTankstartisok*
$\exists missileFuelTank \bullet missileFuelTankstart$

prove by reduce; undo;

$missileFuelTankempty$

$missileFuelTank$

$true$

Theorem 8 *missileFuelTankemptyisok*
$\exists missileFuelTank \bullet missileFuelTankempty$

prove by reduce; undo;

$missileFuelTankfull$

$missileFuelTank$

$((fuelLevel = capacity) \land (outputFlowRate = 0))$

Theorem 9 *missileFuelTankfullisok*
$\exists missileFuelTank \bullet missileFuelTankfull$

prove by reduce; undo;

$missileFuelTankusing$

$missileFuelTank$

$(((fuelLevel \geq 0) \land (fuelLevel \leq capacity)) \land (outputFlowRate > 0))$

Theorem 10 *missileFuelTankusingisok*
$\exists missileFuelTank \bullet missileFuelTankusing$

prove by reduce; undo;

Theorem 11 *missileFuelTankstartTofullPreconditionHolds*
$\exists missileFuelTank \bullet true \land true \land initializeMissileFuelTank$

prove by reduce; undo;

Theorem 12 *missileFuelTankstartTofullImpliesNextInvariant*
$\exists missileFuelTank \bullet true \land true \land initializeMissileFuelTank$
$\Rightarrow ((fuelLevel' = capacity') \land (outputFlowRate' = 0))$

Theorem 13 *missileFuelTankfullTousingPreconditionHolds*
$\exists missileFuelTank \bullet ((fuelLevel = capacity) \land (outputFlowRate = 0)) \land$
$true \land changeFlow$

prove by reduce; undo;

Theorem 14 *missileFuelTankfullTousingImpliesNextInvariant*
$\exists missileFuelTank \bullet ((fuelLevel = capacity) \land (outputFlowRate = 0)) \land$
$true \land changeFlow \Rightarrow (((fuelLevel' \geq 0) \land (fuelLevel' \leq capacity')) \land (outputFlowRate' >$
$0))$

Theorem 15 *missileFuelTankusingTousingPreconditionHolds*
$\exists\, missileFuelTank \bullet (((fuelLevel \geq 0) \wedge (fuelLevel \leq capacity)) \wedge (outputFlowRate > 0))$
$\wedge\ true \wedge changeFlow$

prove by reduce; undo;

Theorem 16 *missileFuelTankusingTousingImpliesNextInvariant*
$\exists\, missileFuelTank \bullet (((fuelLevel \geq 0) \wedge (fuelLevel \leq capacity)) \wedge (outputFlowRate > 0))$
$\wedge\ true \wedge changeFlow \Rightarrow (((fuelLevel' \geq 0) \wedge (fuelLevel' \leq capacity')) \wedge (outputFlowRate' > 0))$

prove by reduce; undo;

Theorem 17 *missileFuelTankusingToemptyPreconditionHolds*
$\exists\, missileFuelTank \bullet (((fuelLevel \geq 0) \wedge (fuelLevel \leq capacity)) \wedge (outputFlowRate > 0)) \wedge true \wedge changeFlow$

prove by reduce; undo;

Theorem 18 *missileFuelTankusingToemptyImpliesNextInvariant*
$\exists\, missileFuelTank \bullet (((fuelLevel \geq 0) \wedge (fuelLevel \leq capacity)) \wedge (outputFlowRate > 0)) \wedge true \wedge changeFlow \Rightarrow true$

prove by reduce; undo;

Theorem 19 *missileFuelTankusingToemptyImpliesSendInvariants*
$\exists\, missileFuelTank \bullet (((fuelLevel \geq 0) \wedge (fuelLevel \leq capacity)) \wedge (outputFlowRate > 0)) \wedge true \wedge changeFlow \Rightarrow (fuelLevel = 0)$

prove by reduce; undo;

navigationSystem

$navState : navStates$

$true$

navigationSysteminitNavigationSystem

$\Xi navigationSystem$

$true$

Theorem 20 *navigationSysteminitNavigationSystemisok*
$\exists\, navigationSystem \bullet navigationSysteminitNavigationSystem$

prove by reduce; undo;

navigationSystemupdatePosition

$\Xi navigationSystem$

$true$

Theorem 21 *navigationSystemupdatePositionisok*
$\exists\, navigationSystem \bullet navigationSystemupdatePosition$

prove by reduce; undo;

navigationSystemestimatePosition

$\Xi navigationSystem$

$true$

Theorem 22 *navigationSystemestimatePositionisok*
$\exists\, navigationSystem \bullet navigationSystemestimatePosition$

prove by reduce; undo;

navigationSystemAUTOMATIC

$\Xi navigationSystem$

$true$

Theorem 23 *navigationSystemAUTOMATICisok*
$\exists\, navigationSystem \bullet navigationSystemAUTOMATIC$

prove by reduce; undo;

navigationSystemstart

$navigationSystem$

$true$

Theorem 24 *navigationSystemstartisok*
$\exists navigationSystem \bullet navigationSystemstart$

prove by reduce; undo;

```
┌─ navigationSystemwait ──────────────────────────
│ navigationSystem
├──────────
│ true
└──────────────────────────────────────────────────
```

Theorem 25 *navigationSystemwaitisok*
$\exists navigationSystem \bullet navigationSystemwait$

prove by reduce; undo;

```
┌─ navigationSystemreadSensors ───────────────────
│ navigationSystem
├──────────
│ true
└──────────────────────────────────────────────────
```

Theorem 26 *navigationSystemreadSensorsisok*
$\exists navigationSystem \bullet navigationSystemreadSensors$

prove by reduce; undo;

Theorem 27 *navigationSystemstartTowaitPreconditionHolds*
$\exists navigationSystem \bullet true \land true$

prove by reduce; undo;

Theorem 28 *navigationSystemstartTowaitImpliesNextInvariant*
$\exists navigationSystem \bullet true \land true \Rightarrow true$

prove by reduce; undo;

Theorem 29 *navigationSystemwaitToreadSensorsPreconditionHolds*
$\exists navigationSystem \bullet true \land true$

prove by reduce; undo;

Theorem 30 *navigationSystemwaitToreadSensorsImpliesNextInvariant*
$\exists navigationSystem \bullet true \land true \Rightarrow true$

prove by reduce; undo;

Theorem 31 *navigationSystemreadSensorsTowaitPreconditionHolds*
$\exists\, navigationSystem \bullet true \land true$

prove by reduce; undo;

Theorem 32 *navigationSystemreadSensorsTowaitImpliesNextInvariant*
$\exists\, navigationSystem \bullet true \land true \Rightarrow true$

prove by reduce; undo;

Theorem 33 *navigationSystemreadSensorsTowaitImpliesSendInvariants*
$\exists\, navigationSystem \bullet true \land true \Rightarrow true$

prove by reduce; undo;

flightProfile

$timeOnTarget : time$

$wayPoints : flightPath$

$true$

addPointToRoute

$\Delta flightProfile$

$p? : point$

$(wayPoints' = wayPoints \frown p?)$

Theorem 34 *addPointToRouteisok*
$\exists\, flightProfile \bullet addPointToRoute$

prove by reduce; undo;

removePointFromRoute

$\Delta flightProfile$

$(wayPoints' = wayPoints)$

Theorem 35 *removePointFromRouteisok*
$\exists\, flightProfile \bullet removePointFromRoute$

prove by reduce; undo;

```
┌─ flightProfileinitFlightProfile ──────────────
│ ΞflightProfile
├──────────
│ true
└───────────────────────────────────────────────
```

Theorem 36 *flightProfileinitFlightProfileisok*
$\exists\, flightProfile \bullet flightProfileinitFlightProfile$

prove by reduce; undo;

```
┌─ flightProfileaddWayPoint ────────────────────
│ ΞflightProfile
├──────────
│ true
└───────────────────────────────────────────────
```

Theorem 37 *flightProfileaddWayPointisok*
$\exists\, flightProfile \bullet flightProfileaddWayPoint$

prove by reduce; undo;

```
┌─ flightProfileremoveFirstWayPoint ────────────
│ ΞflightProfile
├──────────
│ true
└───────────────────────────────────────────────
```

Theorem 38 *flightProfileremoveFirstWayPointisok*
$\exists\, flightProfile \bullet flightProfileremoveFirstWayPoint$

prove by reduce; undo;

```
┌─ flightProfilestart ──────────────────────────
│ flightProfile
├──────────
│ true
└───────────────────────────────────────────────
```

Theorem 39 *flightProfilestartisok*
$\exists\, flightProfile \bullet flightProfilestart$

prove by reduce; undo;

$$\begin{array}{|l}
\multicolumn{1}{l}{\rule{0pt}{0pt}\_\, flightProfileidle\ \rule[0.5ex]{20em}{0.4pt}} \\
flightProfile \\
\hline
true \\
\hline
\end{array}$$

Theorem 40 *flightProfileidleisok*
$\exists flightProfile \bullet flightProfileidle$

prove by reduce; undo;

Theorem 41 *flightProfilestartToidlePreconditionHolds*
$\exists flightProfile \bullet true \wedge true \wedge addPointToRoute$

prove by reduce; undo;

Theorem 42 *flightProfilestartToidleImpliesNextInvariant*
$\exists flightProfile \bullet true \wedge true \wedge addPointToRoute \Rightarrow true$

prove by reduce; undo;

Theorem 43 *flightProfileidleToidlePreconditionHolds*
$\exists flightProfile \bullet true \wedge true \wedge addPointToRoute$

prove by reduce; undo;

Theorem 44 *flightProfileidleToidleImpliesNextInvariant*
$\exists flightProfile \bullet true \wedge true \wedge addPointToRoute \Rightarrow true$

prove by reduce; undo;

Theorem 45 *flightProfileidleToidlePreconditionHolds*
$\exists flightProfile \bullet true \wedge true \wedge removePointFromRoute$

prove by reduce; undo;

Theorem 46 *flightProfileidleToidleImpliesNextInvariant*
$\exists flightProfile \bullet true \wedge true \wedge removePointFromRoute \Rightarrow true$

prove by reduce; undo;

$$\begin{array}{|l}\hline \textit{guidanceSystem} \\ \textit{profile} : \textit{flightProfile} \\ \textit{wayPoints} : \textit{flightPath} \\ \textit{guidanceState} : \textit{guidanceStates} \\ \textit{chronometer} : \textit{timer} \\ \hline \textit{true} \\ \hline \end{array}$$

$$\begin{array}{|l}\hline \textit{initializeGuidanceSystem} \\ \Delta \textit{guidanceSystem} \\ \hline ((\textit{chronometer}' = 0) \wedge (\textit{guidanceState}' = \textit{startGuidanceSystem})) \\ \hline \end{array}$$

Theorem 47 *initializeGuidanceSystemisok*
$\exists\, \textit{guidanceSystem} \bullet \textit{initializeGuidanceSystem}$

prove by reduce; undo;

$$\begin{array}{|l}\hline \textit{guidanceSysteminitGuidanceSystem} \\ \Xi \textit{guidanceSystem} \\ \hline \textit{true} \\ \hline \end{array}$$

Theorem 48 *guidanceSysteminitGuidanceSystemisok*
$\exists\, \textit{guidanceSystem} \bullet \textit{guidanceSysteminitGuidanceSystem}$

prove by reduce; undo;

$$\begin{array}{|l}\hline \textit{guidanceSystemdoLaunch} \\ \Xi \textit{guidanceSystem} \\ \hline \textit{true} \\ \hline \end{array}$$

Theorem 49 *guidanceSystemdoLaunchisok*
$\exists\, \textit{guidanceSystem} \bullet \textit{guidanceSystemdoLaunch}$

prove by reduce; undo;

guidanceSystemestimatePosition

$\Xi guidanceSystem$

---

true

Theorem 50 *guidanceSystemestimatePositionisok*
$\exists\, guidanceSystem \bullet guidanceSystemestimatePosition$

prove by reduce; undo;

guidanceSystemoutOfFuel

$\Xi guidanceSystem$

---

true

Theorem 51 *guidanceSystemoutOfFuelisok*
$\exists\, guidanceSystem \bullet guidanceSystemoutOfFuel$

prove by reduce; undo;

guidanceSystemarmMissile

$\Xi guidanceSystem$

---

true

Theorem 52 *guidanceSystemarmMissileisok*
$\exists\, guidanceSystem \bullet guidanceSystemarmMissile$

prove by reduce; undo;

guidanceSystemupdatePosition

$\Xi guidanceSystem$

---

true

Theorem 53 *guidanceSystemupdatePositionisok*
$\exists\, guidanceSystem \bullet guidanceSystemupdatePosition$

prove by reduce; undo;

$$\begin{array}{|l}\hline \mathit{guidanceSystemremoveFirstWayPoint} \\ \Xi \mathit{guidanceSystem} \\ \hline \mathit{true} \\ \hline \end{array}$$

Theorem 54 *guidanceSystemremoveFirstWayPointisok*
$\exists\, \mathit{guidanceSystem} \bullet \mathit{guidanceSystemremoveFirstWayPoint}$

prove by reduce; undo;

$$\begin{array}{|l}\hline \mathit{guidanceSystemAUTOMATIC} \\ \Xi \mathit{guidanceSystem} \\ \hline \mathit{true} \\ \hline \end{array}$$

Theorem 55 *guidanceSystemAUTOMATICisok*
$\exists\, \mathit{guidanceSystem} \bullet \mathit{guidanceSystemAUTOMATIC}$

prove by reduce; undo;

$$\begin{array}{|l}\hline \mathit{guidanceSystemstart} \\ \mathit{guidanceSystem} \\ \hline \mathit{true} \\ \hline \end{array}$$

Theorem 56 *guidanceSystemstartisok*
$\exists\, \mathit{guidanceSystem} \bullet \mathit{guidanceSystemstart}$

prove by reduce; undo;

$$\begin{array}{|l}\hline \mathit{guidanceSystemidle} \\ \mathit{guidanceSystem} \\ \hline \mathit{true} \\ \hline \end{array}$$

Theorem 57 *guidanceSystemidleisok*
$\exists\, \mathit{guidanceSystem} \bullet \mathit{guidanceSystemidle}$

prove by reduce; undo;

guidanceSystemprocessing

guidanceSystem

---

true

Theorem 58 *guidanceSystemprocessingisok*
$\exists\, guidanceSystem \bullet guidanceSystemprocessing$

prove by reduce; undo;

guidanceSystemterminal

guidanceSystem

---

true

Theorem 59 *guidanceSystemterminalisok*
$\exists\, guidanceSystem \bullet guidanceSystemterminal$

prove by reduce; undo;

Theorem 60 *guidanceSystemstartToidlePreconditionHolds*
$\exists\, guidanceSystem \bullet true \land true \land initializeGuidanceSystem$

prove by reduce; undo;

Theorem 61 *guidanceSystemstartToidleImpliesNextInvariant*
$\exists\, guidanceSystem \bullet true \land true \land initializeGuidanceSystem \Rightarrow true$

prove by reduce; undo;

Theorem 62 *guidanceSystemidleToidlePreconditionHolds*
$\exists\, guidanceSystem \bullet true \land true$

prove by reduce; undo;

Theorem 63 *guidanceSystemidleToidleImpliesNextInvariant*
$\exists\, guidanceSystem \bullet true \land true \Rightarrow true$

prove by reduce; undo;

Theorem 64 *guidanceSystemidleToidleImpliesSendInvariants*
$\exists\, guidanceSystem \bullet true \land true \Rightarrow true$

prove by reduce; undo;

Theorem 65 *guidanceSystemidleToprocessingPreconditionHolds*
$\exists guidanceSystem \bullet true \land true$

prove by reduce; undo;

Theorem 66 *guidanceSystemidleToprocessingImpliesNextInvariant*
$\exists guidanceSystem \bullet true \land true \Rightarrow true$

prove by reduce; undo;

Theorem 67 *guidanceSystemidleToterminalPreconditionHolds*
$\exists guidanceSystem \bullet true \land true$

prove by reduce; undo;

Theorem 68 *guidanceSystemidleToterminalImpliesNextInvariant*
$\exists guidanceSystem \bullet true \land true \Rightarrow true$

prove by reduce; undo;

Theorem 69 *guidanceSystemidleToterminalImpliesSendInvariants*
$\exists guidanceSystem \bullet true \land true \Rightarrow true$

prove by reduce; undo;

Theorem 70 *guidanceSystemprocessingToterminalPreconditionHolds*
$\exists guidanceSystem \bullet true \land (profile.route\ head = profile.route\ tail)$

prove by reduce; undo;

Theorem 71 *guidanceSystemprocessingToterminalImpliesNextInvariant*
$\exists guidanceSystem \bullet true \land (profile.route\ head = profile.route\ tail)$
$\Rightarrow true$

prove by reduce; undo;

Theorem 72 *guidanceSystemprocessingToterminalImpliesSendInvariants*
$\exists guidanceSystem \bullet true \land (profile.route\ head = profile.route\ tail)$
$\Rightarrow true \land true$

prove by reduce; undo;

Theorem 73 *guidanceSystemprocessingToidlePreconditionHolds*
$\exists guidanceSystem \bullet true \land (profile.route\ head = profile.route\ tail)$

prove by reduce; undo;

Theorem 74 *guidanceSystemprocessingToidleImpliesNextInvariant*
$\exists\, guidanceSystem \bullet true \land (profile.route\ head = profile.route\ tail)$
$\Rightarrow true$

prove by reduce; undo;

flightDirector

$flightDirectorState : flightDirectorStates$

$true$

initializeFlightDirector

$\Delta flightDirector$

$(flightDirectorState' = idleflightDirector)$

Theorem 75 *initializeFlightDirectorisok*
$\exists\, flightDirector \bullet initializeFlightDirector$

prove by reduce; undo;

flightDirectorerrorSignals

$\Xi flightDirector$

$true$

Theorem 76 *flightDirectorerrorSignalsisok*
$\exists\, flightDirector \bullet flightDirectorerrorSignals$

prove by reduce; undo;

flightDirectorinitFlightDirector

$\Xi flightDirector$

$true$

Theorem 77 *flightDirectorinitFlightDirectorisok*
$\exists\, flightDirector \bullet flightDirectorinitFlightDirector$

prove by reduce; undo;

$flightDirectormaneuverComplete$

$\Xi flightDirector$

$true$

Theorem 78 *flightDirectormaneuverCompleteisok*
$\exists flightDirector \bullet flightDirectormaneuverComplete$

prove by reduce; undo;

$flightDirectorchangeCourse$

$\Xi flightDirector$

$true$

Theorem 79 *flightDirectorchangeCourseisok*
$\exists flightDirector \bullet flightDirectorchangeCourse$

prove by reduce; undo;

$flightDirectorsetElevation$

$\Xi flightDirector$

$true$

Theorem 80 *flightDirectorsetElevationisok*
$\exists flightDirector \bullet flightDirectorsetElevation$

prove by reduce; undo;

$flightDirectorsetThrottle$

$\Xi flightDirector$

$true$

Theorem 81 *flightDirectorsetThrottleisok*
$\exists flightDirector \bullet flightDirectorsetThrottle$

prove by reduce; undo;

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{flightDirectorstart}} \\
\hline
\textit{flightDirector} \\
\hline
(\textit{flightDirectorState} = \textit{startflightDirector}) \\
\hline
\end{array}$$

Theorem 82 *flightDirectorstartisok*
$\exists\, \textit{flightDirector} \bullet \textit{flightDirectorstart}$

prove by reduce; undo;

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{flightDirectoridle}} \\
\hline
\textit{flightDirector} \\
\hline
(\textit{flightDirectorState} = \textit{idleflightDirector}) \\
\hline
\end{array}$$

Theorem 83 *flightDirectoridleisok*
$\exists\, \textit{flightDirector} \bullet \textit{flightDirectoridle}$

prove by reduce; undo;

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{flightDirectormaneuvering}} \\
\hline
\textit{flightDirector} \\
\hline
(\textit{flightDirectorState} = \textit{maneuveringflightDirector}) \\
\hline
\end{array}$$

Theorem 84 *flightDirectormaneuveringisok*
$\exists\, \textit{flightDirector} \bullet \textit{flightDirectormaneuvering}$

prove by reduce; undo;

Theorem 85 *flightDirectorstartToidlePreconditionHolds*
$\exists\, \textit{flightDirector} \bullet (\textit{flightDirectorState} = \textit{startflightDirector}) \land \textit{true}$
$\land\ \textit{initializeFlightDirector}$

prove by reduce; undo;

Theorem 86 *flightDirectorstartToidleImpliesNextInvariant*
$\exists\, \textit{flightDirector} \bullet (\textit{flightDirectorState} = \textit{startflightDirector}) \land \textit{true} \land$
$\textit{initializeFlightDirector} \Rightarrow (\textit{flightDirectorState}' = \textit{idleflightDirector})$

prove by reduce; undo;

Theorem 87 *flightDirectoridleTomaneuveringPreconditionHolds*
$\exists \, flightDirector \bullet (flightDirectorState = idleflightDirector) \wedge true$

prove by reduce; undo;

Theorem 88 *flightDirectoridleTomaneuveringImpliesNextInvariant*
$\exists \, flightDirector \bullet (flightDirectorState = idleflightDirector) \wedge true$
$\Rightarrow (flightDirectorState' = maneuveringflightDirector)$

prove by reduce; undo;

Theorem 89 *flightDirectoridleTomaneuveringImpliesSendInvariants*
$\exists \, flightDirector \bullet (flightDirectorState = idleflightDirector) \wedge true$
$\Rightarrow true \wedge true \wedge true$

prove by reduce; undo;

Theorem 90 *flightDirectormaneuveringToidlePreconditionHolds*
$\exists \, flightDirector \bullet (flightDirectorState = maneuveringflightDirector) \wedge true$

prove by reduce; undo;

Theorem 91 *flightDirectormaneuveringToidleImpliesNextInvariant*
$\exists \, flightDirector \bullet (flightDirectorState = maneuveringflightDirector) \wedge true$
$\Rightarrow (flightDirectorState' = idleflightDirector)$

prove by reduce; undo;

$$\begin{array}{|l}
\multicolumn{1}{l}{avionicsSoftware} \\
\hline
navSys : navigationSystem \\
guidSys : guidanceSystem \\
director : flightDirector \\
\hline
true \\
\hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{initializeAvionicsSoftware} \\
\hline
\Delta avionicsSoftware \\
\hline
true \\
\hline
\end{array}$$

Theorem 92 *initializeAvionicsSoftwareisok*
$\exists avionicsSoftware \bullet initializeAvionicsSoftware$

prove by reduce; undo;

$avionicsSoftwareinitNavigationSystem$

$\Xi avionicsSoftware$

$true$

Theorem 93 *avionicsSoftwareinitNavigationSystemisok*
$\exists avionicsSoftware \bullet avionicsSoftwareinitNavigationSystem$

prove by reduce; undo;

$avionicsSoftwareinitGuidanceSystem$

$\Xi avionicsSoftware$

$true$

Theorem 94 *avionicsSoftwareinitGuidanceSystemisok*
$\exists avionicsSoftware \bullet avionicsSoftwareinitGuidanceSystem$

prove by reduce; undo;

$avionicsSoftwareinitFlightDirector$

$\Xi avionicsSoftware$

$true$

Theorem 95 *avionicsSoftwareinitFlightDirectorisok*
$\exists avionicsSoftware \bullet avionicsSoftwareinitFlightDirector$

prove by reduce; undo;

$avionicsSoftwarestart$

$avionicsSoftware$

$true$

Theorem 96 *avionicsSoftwarestartisok*
$\exists\, avionicsSoftware \bullet avionicsSoftwarestart$

prove by reduce; undo;

$$\begin{array}{|l}
\hline \textit{avionicsSoftwareavionicsSoftwareInitialized} \\
avionicsSoftware \\
\hline
true \\
\hline
\end{array}$$

Theorem 97 *avionicsSoftwareavionicsSoftwareInitializedisok*
$\exists\, avionicsSoftware \bullet avionicsSoftwareavionicsSoftwareInitialized$

prove by reduce; undo;

Theorem 98 *avionicsSoftwarestartToavionicsSoftwareInitializedPreconditionHolds*
$\exists\, avionicsSoftware \bullet true \land true \land initializeAvionicsSoftware$

prove by reduce; undo;

Theorem 99 *avionicsSoftwarestartToavionicsSoftwareInitializedImpliesNextInvariant*
$\exists\, avionicsSoftware \bullet true \land true \land initializeAvionicsSoftware \Rightarrow true$

prove by reduce; undo;

Theorem 100 *avionicsSoftwarestartToavionicsSoftwareInitializedImpliesSendInvariants*
$\exists\, avionicsSoftware \bullet true \land true \land initializeAvionicsSoftware \Rightarrow true \land true \land$

prove by reduce; undo;

$$\begin{array}{|l}
\hline \textit{warhead} \\
weight : mass \\
munitionType : string \\
explosiveForce : yield \\
armed : warheadArmed \\
\hline
((weight > 0) \land (explosiveForce \geq 0)) \\
\hline
\end{array}$$

$$\begin{array}{|l}\hline \textit{initializeWarhead} \\ \Delta warhead \\ \hline (warheadArmed' = f) \\ \hline \end{array}$$

Theorem 101 *initializeWarheadisok*
$\exists\, warhead \bullet initializeWarhead$

prove by reduce; undo;

$$\begin{array}{|l}\hline \textit{armWarhead} \\ \Delta warhead \\ \hline (warheadArmed' = t) \\ \hline \end{array}$$

Theorem 102 *armWarheadisok*
$\exists\, warhead \bullet armWarhead$

prove by reduce; undo;

$$\begin{array}{|l}\hline \textit{warheadinitWarhead} \\ \Xi warhead \\ \hline true \\ \hline \end{array}$$

Theorem 103 *warheadinitWarheadisok*
$\exists\, warhead \bullet warheadinitWarhead$

prove by reduce; undo;

$$\begin{array}{|l}\hline \textit{warheadarmMissile} \\ \Xi warhead \\ \hline true \\ \hline \end{array}$$

Theorem 104 *warheadarmMissileisok*
$\exists\, warhead \bullet warheadarmMissile$

prove by reduce; undo;

$warheadstart$

$warhead$

$true$

Theorem 105 *warheadstartisok*
$\exists\, warhead \bullet warheadstart$

prove by reduce; undo;

$warheadunarmed$

$warhead$

$(warheadArmed = f)$

Theorem 106 *warheadunarmedisok*
$\exists\, warhead \bullet warheadunarmed$

prove by reduce; undo;

$warheadarmed$

$warhead$

$(warheadArmed = t)$

Theorem 107 *warheadarmedisok*
$\exists\, warhead \bullet warheadarmed$

prove by reduce; undo;

Theorem 108 *warheadstartTounarmedPreconditionHolds*
$\exists\, warhead \bullet true \land true \land initializeWarhead$

prove by reduce; undo;

Theorem 109 *warheadstartTounarmedImpliesNextInvariant*
$\exists\, warhead \bullet true \land true \land initializeWarhead \Rightarrow (warheadArmed' = f)$

prove by reduce; undo;

Theorem 110 *warheadunarmedToarmedPreconditionHolds*
$\exists\, warhead \bullet (warheadArmed = f) \wedge true \wedge armWarhead$

prove by reduce; undo;

Theorem 111 *warheadunarmedToarmedImpliesNextInvariant*
$\exists\, warhead \bullet (warheadArmed = f) \wedge true \wedge armWarhead \Rightarrow (warheadArmed' = t)$

prove by reduce; undo;

airframe

$pos : position$

$accl : acceleration$

$vel : velocity$

$afState : afStates$

$heading : real$

$elevation : real$

$((((heading \leq (2 * pi)) \wedge (heading \geq 0)) \wedge (elevation \leq (pi \operatorname{div} 2))) \wedge (elevation \geq (pi \operatorname{div} 2)))$

setPosition

$\Delta airframe$

$setPosX? : coordinate$

$setPosY? : coordinate$

$setPosZ? : coordinate$

$(((pos.x = setPosX?) \wedge (pos.y = setPosY?)) \wedge (pos.z = setPosZ?))$

Theorem 112 *setPositionisok*
$\exists\, airframe \bullet setPosition$

prove by reduce; undo;

setHeading

$\Delta airframe$

$setHead? : real$

$(heading = setHead?)$

Theorem 113 *setHeadingisok*
$\exists\, airframe \bullet setHeading$

prove by reduce; undo;

setElevation

$\Delta airframe$

$setEle? : real$

$(elevation = setEle?)$

Theorem 114 *setElevationisok*
$\exists\, airframe \bullet setElevation$

prove by reduce; undo;

calculateDistances

$\Delta airframe$

$pos1? : position$

$pos2? : position$

$true$

Theorem 115 *calculateDistancesisok*
$\exists\, airframe \bullet calculateDistances$

prove by reduce; undo;

$$\begin{array}{|l}
\multicolumn{1}{l}{\_\,initializeAirframe\,\rule{20em}{0.4pt}} \\
\Delta airframe \\
\hline
((((((((((elevation' = (pi \operatorname{div} 2)) \land (heading' = 0)) \land (pos.x = 0)) \land \\
(pos.y = 0)) \land (pos.z = 0)) \land (vel.x = 0)) \land (vel.y = 0)) \land \\
(vel.z = 0)) \land (accl.x = 0)) \land (accl.y = 0)) \land (accl.z = 0)) \\
\hline
\end{array}$$

Theorem 116 *initializeAirframeisok*
$\exists\, airframe \bullet initializeAirframe$

prove by reduce; undo;

$$\begin{array}{|l}
\multicolumn{1}{l}{\_\,setPosition\,\rule{20em}{0.4pt}} \\
\Delta airframe \\
\hline
true \\
\hline
\end{array}$$

Theorem 117 *setPositionisok*
$\exists\, airframe \bullet setPosition$

prove by reduce; undo;

$$\begin{array}{|l}
\multicolumn{1}{l}{\_\,airframeinitAirframe\,\rule{20em}{0.4pt}} \\
\Xi airframe \\
\hline
true \\
\hline
\end{array}$$

Theorem 118 *airframeinitAirframeisok*
$\exists\, airframe \bullet airframeinitAirframe$

prove by reduce; undo;

$$\begin{array}{|l}
\multicolumn{1}{l}{\_\,airframedoLaunch\,\rule{20em}{0.4pt}} \\
\Xi airframe \\
\hline
true \\
\hline
\end{array}$$

Theorem 119 *airframedoLaunchisok*
$\exists\, airframe \bullet airframedoLaunch$

prove by reduce; undo;

$$\begin{array}{|l}
\hline
\textit{airframetankEmpty} \\
\Xi airframe \\
\hline
true \\
\hline
\end{array}$$

Theorem 120 *airframetankEmptyisok*
$\exists airframe \bullet airframetankEmpty$

prove by reduce; undo;

$$\begin{array}{|l}
\hline
\textit{airframeestimatePosition} \\
\Xi airframe \\
\hline
true \\
\hline
\end{array}$$

Theorem 121 *airframeestimatePositionisok*
$\exists airframe \bullet airframeestimatePosition$

prove by reduce; undo;

$$\begin{array}{|l}
\hline
\textit{airframechangeCourse} \\
\Xi airframe \\
\hline
true \\
\hline
\end{array}$$

Theorem 122 *airframechangeCourseisok*
$\exists airframe \bullet airframechangeCourse$

prove by reduce; undo;

$$\begin{array}{|l}
\hline
\textit{airframedoManeuverComplete} \\
\Xi airframe \\
\hline
true \\
\hline
\end{array}$$

Theorem 123 *airframedoManeuverCompleteisok*
$\exists airframe \bullet airframedoManeuverComplete$

prove by reduce; undo;

$airframeoutOfFuel$

$\Xi airframe$

$true$

Theorem 124 *airframeoutOfFuelisok*
$\exists\, airframe \bullet airframeoutOfFuel$

prove by reduce; undo;

$airframestart$

$airframe$

$(afState = startairframe)$

Theorem 125 *airframestartisok*
$\exists\, airframe \bullet airframestart$

prove by reduce; undo;

$airframepreLaunch$

$airframe$

$(afState = preLaunchairframe)$

Theorem 126 *airframepreLaunchisok*
$\exists\, airframe \bullet airframepreLaunch$

prove by reduce; undo;

$airframepoweredFlight$

$airframe$

$(afState = poweredFlightairframe)$

Theorem 127 *airframepoweredFlightisok*
$\exists\, airframe \bullet airframepoweredFlight$

prove by reduce; undo;

$$\begin{array}{|l}
\hline
\mathit{airframemaneuvering} \\
\mathit{airframe} \\
\hline
(\mathit{afState} = \mathit{maneuveringairframe}) \\
\hline
\end{array}$$

Theorem 128 *airframemaneuveringisok*
$\exists\, airframe \bullet airframemaneuvering$

prove by reduce; undo;

$$\begin{array}{|l}
\hline
\mathit{airframeinertialFlight} \\
\mathit{airframe} \\
\hline
(\mathit{afState} = \mathit{inertialFlightairframe}) \\
\hline
\end{array}$$

Theorem 129 *airframeinertialFlightisok*
$\exists\, airframe \bullet airframeinertialFlight$

prove by reduce; undo;

Theorem 130 *airframestartTopreLaunchPreconditionHolds*
$\exists\, airframe \bullet (afState = startairframe) \wedge true \wedge initializeAirframe$

prove by reduce; undo;

Theorem 131 *airframestartTopreLaunchImpliesNextInvariant*
$\exists\, airframe \bullet (afState = startairframe) \wedge true \wedge initializeAirframe \Rightarrow$
$(afState' = preLaunchairframe)$

prove by reduce; undo;

Theorem 132 *airframepreLaunchTopoweredFlightPreconditionHolds*
$\exists\, airframe \bullet (afState = preLaunchairframe) \wedge true$

prove by reduce; undo;

Theorem 133 *airframepreLaunchTopoweredFlightImpliesNextInvariant*
$\exists\, airframe \bullet (afState = preLaunchairframe) \wedge true \Rightarrow$
$(afState' = poweredFlightairframe)$

prove by reduce; undo;

Theorem 134 *airframepoweredFlightToinertialFlightPreconditionHolds*
$\exists\, airframe \bullet (afState = poweredFlightairframe) \wedge true$

prove by reduce; undo;

Theorem 135 *airframepoweredFlightToinertialFlightImpliesNextInvariant*
$\exists\, airframe \bullet (afState = poweredFlightairframe) \wedge true \Rightarrow$
$(afState' = inertialFlightairframe)$

prove by reduce; undo;

Theorem 136 *airframepoweredFlightTopoweredFlightPreconditionHolds*
$\exists\, airframe \bullet (afState = poweredFlightairframe) \wedge true \wedge setPosition$

prove by reduce; undo;

Theorem 137 *airframepoweredFlightTopoweredFlightImpliesNextInvariant*
$\exists\, airframe \bullet (afState = poweredFlightairframe) \wedge true \wedge setPosition \Rightarrow$
$(afState' = poweredFlightairframe)$

prove by reduce; undo;

Theorem 138 *airframepoweredFlightTomaneuveringPreconditionHolds*
$\exists\, airframe \bullet (afState = poweredFlightairframe) \wedge true$

prove by reduce; undo;

Theorem 139 *airframepoweredFlightTomaneuveringImpliesNextInvariant*
$\exists\, airframe \bullet (afState = poweredFlightairframe) \wedge true \Rightarrow$
$(afState' = maneuveringairframe)$

prove by reduce; undo;

Theorem 140 *airframemaneuveringTopoweredFlightPreconditionHolds*
$\exists\, airframe \bullet (afState = maneuveringairframe) \wedge true$

prove by reduce; undo;

Theorem 141 *airframemaneuveringTopoweredFlightImpliesNextInvariant*
$\exists\, airframe \bullet (afState = maneuveringairframe) \wedge true \Rightarrow$
$(afState' = poweredFlightairframe)$

prove by reduce; undo;

Theorem 142 *airframemaneuveringToinertialFlightPreconditionHolds*
$\exists\, airframe \bullet (afState = maneuveringairframe) \wedge true$

prove by reduce; undo;

Theorem 143 *airframemaneuveringToinertialFlightImpliesNextInvariant*
$\exists\, airframe \bullet (afState = maneuveringairframe) \land true \Rightarrow$
$(afState' = inertialFlightairframe)$

prove by reduce; undo;

$$\begin{array}{|l}
\hline
\textit{throttle} \\
\hline
fuelFlow : real \\
throttleState : throttleStates \\
maximumFlowRate : real \\
actualFlowRate : real \\
\hline
true \\
\hline
\end{array}$$

$$\begin{array}{|l}
\hline
\textit{changeFuelFlow} \\
\hline
\Delta throttle \\
inFlow? : real \\
\hline
(fuelFlow = inFlow?) \\
\hline
\end{array}$$

Theorem 144 *changeFuelFlowisok*
$\exists\, throttle \bullet changeFuelFlow$

prove by reduce; undo;

$$\begin{array}{|l}
\hline
\textit{throttleinitThrottle} \\
\hline
\Xi throttle \\
\hline
true \\
\hline
\end{array}$$

Theorem 145 *throttleinitThrottleisok*
$\exists\, throttle \bullet throttleinitThrottle$

prove by reduce; undo;

$$
\begin{array}{|l}
\multicolumn{1}{l}{throttlechangeTankFlow} \\
\hline
\Xi throttle \\
\hline
true \\
\hline
\end{array}
$$

Theorem 146 *throttlechangeTankFlowisok*
$\exists throttle \bullet throttlechangeTankFlow$

prove by reduce; undo;

$$
\begin{array}{|l}
\multicolumn{1}{l}{throttlesetThrottle} \\
\hline
\Xi throttle \\
\hline
true \\
\hline
\end{array}
$$

Theorem 147 *throttlesetThrottleisok*
$\exists throttle \bullet throttlesetThrottle$

prove by reduce; undo;

$$
\begin{array}{|l}
\multicolumn{1}{l}{throttlestart} \\
\hline
throttle \\
\hline
(throttleState = startthrottle) \\
\hline
\end{array}
$$

Theorem 148 *throttlestartisok*
$\exists throttle \bullet throttlestart$

prove by reduce; undo;

$$
\begin{array}{|l}
\multicolumn{1}{l}{throttleidle} \\
\hline
throttle \\
\hline
(throttleState = idlethrottle) \\
\hline
\end{array}
$$

Theorem 149 *throttleidleisok*
$\exists throttle \bullet throttleidle$

prove by reduce; undo;

Theorem 150 *throttlestartToidlePreconditionHolds*
$\exists throttle \bullet (throttleState = startthrottle) \wedge true$

prove by reduce; undo;

Theorem 151 *throttlestartToidleImpliesNextInvariant*
$\exists throttle \bullet (throttleState = startthrottle) \wedge true \Rightarrow (throttleState' = idlethrottle)$

prove by reduce; undo;

Theorem 152 *throttleidleToidlePreconditionHolds*
$\exists throttle \bullet (throttleState = idlethrottle) \wedge changeFuelFlow$

prove by reduce; undo;

Theorem 153 *throttleidleToidleImpliesNextInvariant*
$\exists throttle \bullet (throttleState = idlethrottle) \wedge changeFuelFlow \Rightarrow (throttleState' = idlethrottle)$

prove by reduce; undo;

jetEngine

$thrust : real$

$maximumFuelFlowRate : real$

$currentFuelFlowRate : real$

$engineState : engineStates$

$true$

jetEngineinitEngine

$\Xi jetEngine$

$true$

Theorem 154 *jetEngineinitEngineisok*
$\exists jetEngine \bullet jetEngineinitEngine$

prove by reduce; undo;

$$\begin{array}{|l}
\multicolumn{1}{l}{\_\, jetEnginechangeTankFlow \,\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
\Xi jetEngine \\
\hline
true \\
\hline
\end{array}$$

Theorem 155 *jetEnginechangeTankFlowisok*
$\exists jetEngine \bullet jetEnginechangeTankFlow$

prove by reduce; undo;

$$\begin{array}{|l}
\multicolumn{1}{l}{\_\, jetEnginestart \,\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
jetEngine \\
\hline
(engineState = startengine) \\
\hline
\end{array}$$

Theorem 156 *jetEnginestartisok*
$\exists jetEngine \bullet jetEnginestart$

prove by reduce; undo;

$$\begin{array}{|l}
\multicolumn{1}{l}{\_\, jetEngineidle \,\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
jetEngine \\
\hline
(engineState = idleengine) \\
\hline
\end{array}$$

Theorem 157 *jetEngineidleisok*
$\exists jetEngine \bullet jetEngineidle$

prove by reduce; undo;

Theorem 158 *jetEnginestartToidlePreconditionHolds*
$\exists jetEngine \bullet (engineState = startengine) \land true$

prove by reduce; undo;

Theorem 159 *jetEnginestartToidleImpliesNextInvariant*
$\exists jetEngine \bullet (engineState = startengine) \land true \Rightarrow (engineState' = idleengine)$

prove by reduce; undo;

Theorem 160 *jetEngineidleToidlePreconditionHolds*
$\exists jetEngine \bullet (engineState = idleengine)$

prove by reduce; undo;

Theorem 161 *jetEngineidleToidleImpliesNextInvariant*
$\exists\, jetEngine \bullet (engineState = idleengine) \Rightarrow (engineState' = idleengine)$

prove by reduce; undo;

propulsionSystem

$fuelFeed : throttle$

$engine : jetEngine$

$tank : missileFuelTank$

$((((tank.fuelLevel = 0) \Rightarrow (fuelFeed.maximumFlowRate = 0)) \wedge$

$((tank.fuelLevel > 0) \Rightarrow (fuelFeed.maximumFlowRate =$

$engine.maximumFuelFlowRate))) \wedge$

$(engine.currentFuelFlowRate = fuelFeed.actualFlowRate))$

propulsionSysteminitPropulsionSystem

$\Xi propulsionSystem$

$true$

Theorem 162 *propulsionSysteminitPropulsionSystemisok*
$\exists\, propulsionSystem \bullet propulsionSysteminitPropulsionSystem$

prove by reduce; undo;

propulsionSysteminitThrottle

$\Xi propulsionSystem$

$true$

Theorem 163 *propulsionSysteminitThrottleisok*
$\exists\, propulsionSystem \bullet propulsionSysteminitThrottle$

prove by reduce; undo;

propulsionSysteminitEngine

$\Xi propulsionSystem$

true

Theorem 164 *propulsionSysteminitEngineisok*
$\exists propulsionSystem \bullet propulsionSysteminitEngine$

prove by reduce; undo;

propulsionSysteminitMissileFuelTank

$\Xi propulsionSystem$

true

Theorem 165 *propulsionSysteminitMissileFuelTankisok*
$\exists propulsionSystem \bullet propulsionSysteminitMissileFuelTank$

prove by reduce; undo;

propulsionSystemstart

propulsionSystem

true

Theorem 166 *propulsionSystemstartisok*
$\exists propulsionSystem \bullet propulsionSystemstart$

prove by reduce; undo;

propulsionSystemidle

propulsionSystem

true

Theorem 167 *propulsionSystemidleisok*
$\exists propulsionSystem \bullet propulsionSystemidle$

prove by reduce; undo;

Theorem 168 *propulsionSystemstartToidlePreconditionHolds*
$\exists propulsionSystem \bullet true \land true$

prove by reduce; undo;

Theorem 169 *propulsionSystemstartToidleImpliesNextInvariant*
$\exists propulsionSystem \bullet true \land true \Rightarrow true$

prove by reduce; undo;

Theorem 170 *propulsionSystemstartToidleImpliesSendInvariants*
$\exists propulsionSystem \bullet true \land true \Rightarrow true \land true \land true$

prove by reduce; undo;

cruiseMissile

$propulsion : propulsionSystem$

$frame : airframe$

$payload : warhead$

$avionics : avionicsSoftware$

$cmState : cmStates$

$true$

initializeCruiseMissile

$\Delta cruiseMissile$

$(cmState' = preLaunchcruiseMissile)$

Theorem 171 *initializeCruiseMissileisok*
$\exists cruiseMissile \bullet initializeCruiseMissile$

prove by reduce; undo;

cruiseMissiledoLaunch

$\Xi cruiseMissile$

$true$

Theorem 172 *cruiseMissiledoLaunchisok*
$\exists\, cruiseMissile \bullet cruiseMissiledoLaunch$

prove by reduce; undo;

```
┌─ cruiseMissilelaunch ─────────────────────
│ ΞcruiseMissile
├──────────
│ true
└───────────────────────────────────────────
```

Theorem 173 *cruiseMissilelaunchisok*
$\exists\, cruiseMissile \bullet cruiseMissilelaunch$

prove by reduce; undo;

```
┌─ cruiseMissileinitPropulsionSystem ───────
│ ΞcruiseMissile
├──────────
│ true
└───────────────────────────────────────────
```

Theorem 174 *cruiseMissileinitPropulsionSystemisok*
$\exists\, cruiseMissile \bullet cruiseMissileinitPropulsionSystem$

prove by reduce; undo;

```
┌─ cruiseMissileinitAirframe ───────────────
│ ΞcruiseMissile
├──────────
│ true
└───────────────────────────────────────────
```

Theorem 175 *cruiseMissileinitAirframeisok*
$\exists\, cruiseMissile \bullet cruiseMissileinitAirframe$

prove by reduce; undo;

```
┌─ cruiseMissileinitWarhead ────────────────
│ ΞcruiseMissile
├──────────
│ true
└───────────────────────────────────────────
```

Theorem 176 *cruiseMissileinitWarheadisok*
$\exists\, cruiseMissile \bullet cruiseMissileinitWarhead$

prove by reduce; undo;

$$\begin{array}{|l}
\multicolumn{1}{l}{cruiseMissileinitAvionicsSoftware} \\
\hline
\Xi cruiseMissile \\
\hline
true \\
\hline
\end{array}$$

Theorem 177 *cruiseMissileinitAvionicsSoftwareisok*
$\exists\, cruiseMissile \bullet cruiseMissileinitAvionicsSoftware$

prove by reduce; undo;

$$\begin{array}{|l}
\multicolumn{1}{l}{cruiseMissileAUTOMATIC} \\
\hline
\Xi cruiseMissile \\
\hline
true \\
\hline
\end{array}$$

Theorem 178 *cruiseMissileAUTOMATICisok*
$\exists\, cruiseMissile \bullet cruiseMissileAUTOMATIC$

prove by reduce; undo;

$$\begin{array}{|l}
\multicolumn{1}{l}{cruiseMissilestart} \\
\hline
cruiseMissile \\
\hline
(cmState = startcruiseMissile) \\
\hline
\end{array}$$

Theorem 179 *cruiseMissilestartisok*
$\exists\, cruiseMissile \bullet cruiseMissilestart$

prove by reduce; undo;

$$\begin{array}{|l}
\multicolumn{1}{l}{cruiseMissilepreLaunch} \\
\hline
cruiseMissile \\
\hline
(cmState = preLaunchcruiseMissile) \\
\hline
\end{array}$$

Theorem 180 *cruiseMissilepreLaunchisok*
$\exists\, cruiseMissile \bullet cruiseMissilepreLaunch$

prove by reduce; undo;

cruiseMissileflying

$cruiseMissile$

$(cmState = flyingcruiseMissile)$

Theorem 181 *cruiseMissileflyingisok*
$\exists\, cruiseMissile \bullet cruiseMissileflying$

prove by reduce; undo;

Theorem 182 *cruiseMissilestartTopreLaunchPreconditionHolds*
$\exists\, cruiseMissile \bullet (cmState = startcruiseMissile) \land true \land initializeCruiseMissile$

prove by reduce; undo;

Theorem 183 *cruiseMissilestartTopreLaunchImpliesNextInvariant*
$\exists\, cruiseMissile \bullet (cmState = startcruiseMissile) \land true \land initializeCruiseMissile \Rightarrow (cmState' = preLaunchcruiseMissile)$

prove by reduce; undo;

Theorem 184 *cruiseMissilestartTopreLaunchImpliesSendInvariants*
$\exists\, cruiseMissile \bullet (cmState = startcruiseMissile) \land true \land initializeCruiseMissile \Rightarrow true \land true \land true \land true$

prove by reduce; undo;

Theorem 185 *cruiseMissilepreLaunchToflyingPreconditionHolds*
$\exists\, cruiseMissile \bullet (cmState = preLaunchcruiseMissile) \land true$

prove by reduce; undo;

Theorem 186 *cruiseMissilepreLaunchToflyingImpliesNextInvariant*
$\exists\, cruiseMissile \bullet (cmState = preLaunchcruiseMissile) \land true \Rightarrow (cmState' = flyingcruiseMissile)$

prove by reduce; undo;

Theorem 187 *cruiseMissilepreLaunchToflyingImpliesSendInvariants*
$\exists\, cruiseMissile \bullet (cmState = preLaunchcruiseMissile) \land true \Rightarrow true$

prove by reduce; undo;

Appendix D. Promela Specification of Cruise Missile Model

```
#define true 1

#define false 0

#define AUTOMATIC true


mtype = { initMissileTank,  changeTankFlow,  tankEmpty,  outOfFuel,

initNavigationSystem,  updatePosition,  estimatePosition,

initFlightProfile,  addWayPoint,  removeFirstWayPoint,

initGuidanceSystem,  doLaunch,  armMissile,  errorSignals,

initFlightDirector,  maneuverComplete, changeCourse,  setElevation,

setThrottle,  initWarhead,  initAirframe,  doManeuverComplete,

initThrottle,  initEngine,  initPropulsionSystem,  initMissileFuelTank,

launch,  initAvionicsSoftware };


 chan map0 = [0] of {mtype};

 chan map1 = [0] of {mtype};

 chan map2 = [0] of {mtype};

 chan map3 = [0] of {mtype};

 chan map4 = [0] of {mtype};

 chan map5 = [0] of {mtype};

 chan map6 = [0] of {mtype};

 chan map7 = [0] of {mtype};

 chan map8 = [0] of {mtype};

 chan map9 = [0] of {mtype};

 chan map10 = [0] of {mtype};

 chan map11 = [0] of {mtype};

 chan map12 = [0] of {mtype};

 chan map13 = [0] of {mtype};

 chan map14 = [0] of {mtype};
 chan map15 = [0] of {mtype};
 chan map16 = [0] of {mtype};
 chan map17 = [0] of {mtype};
 chan map18 = [0] of {mtype};
 chan map19 = [0] of {mtype};
 chan map20 = [0] of {mtype};
 chan map21 = [0] of {mtype};
 chan map22 = [0] of {mtype};
 chan map23 = [0] of {mtype};
 chan map24 = [0] of {mtype};
 chan map25 = [0] of {mtype};
 chan map26 = [0] of {mtype};
 chan map27 = [0] of {mtype};
 chan map28 = [0] of {mtype};

proctype missileFuelTank()
 {
goto startState;
 startState:
    do
    :: atomic{map26?initMissileTank; true ->} /* initializeMissileFuelTank;*/
       goto fullState
    od;

 emptyState:
    do
    :: true -> break
    od;

 fullState:
   do
   :: atomic{map1?changeTankFlow; true ->} /* changeFlow; */ goto usingState
   od;

usingState:
   do
   :: atomic{map1?changeTankFlow; true ->} /* changeFlow; */ goto usingState
   :: atomic{map2?tankEmpty; true ->} map3!outOfFuel; /* changeFlow; */
      goto emptyState
   od;
}

proctype navigationSystem()
{
goto startState;
startState:
   do
   :: atomic{map4?initNavigationSystem; true ->}  goto waitState
   od;

waitState:
   do
   :: atomic{map5?updatePosition; true ->}  goto readSensorsState
   od;

readSensorsState:
   do
   :: atomic{AUTOMATIC; true ->} map6!estimatePosition;  goto waitState
   od;
}
```

```
 proctype flightProfile()
 {
goto startState;
 startState:
    do
    :: atomic{map15?initFlightProfile; true ->} /* addPointToRoute; */
       goto idleState
    od;

 idleState:
    do
    :: atomic{map9?addWayPoint; true ->} /* addPointToRoute; */
       goto idleState
    :: atomic{map10?removeFirstWayPoint; true ->} /* removePointFromRoute; */
       goto idleState
    od;
 }

 proctype guidanceSystem()
 {
goto startState;
 startState:
    do
    :: atomic{map11?initGuidanceSystem; true ->} /* initializeGuidanceSystem;
       */  goto idleState
    od;

 idleState:
    do
    :: atomic{map12?doLaunch; true ->} map5!updatePosition;
       goto idleState
    :: atomic{map6?estimatePosition; true ->}  goto processingState
    :: atomic{map4?outOfFuel; true ->} map5!updatePosition;
       goto terminalState
    od;

 processingState:
    do
    :: atomic{AUTOMATIC; /* profile.route~head = profile.route~tail -> */}
       map10!removeFirstWayPoint; map13!armMissile;  goto terminalState
    :: atomic{AUTOMATIC;  /* \lnot profile.route~head =
       profile.route~tail -> */}  goto idleState
    od;

 terminalState:
    do
    :: break
    od;
 }

 proctype flightDirector()
 {
goto startState;
 startState:
    do
    :: atomic{map15?initFlightDirector; true ->} /*
       initializeFlightDirector; */  goto idleState
    od;
 idleState:
    do
    :: atomic{map15?errorSignals; true ->} map17!changeCourse;
       map18!setElevation; map19!setThrottle; goto maneuveringState
    od;

 maneuveringState:
    do
    :: atomic{map16?maneuverComplete; true ->}  goto idleState
    od;
 }

 proctype avionicsSoftware()
 {
goto startState;
 startState:
    do
    :: atomic{map28?initAvionicsSoftware; true ->} map4!initNavigationSystem;
       map11!initGuidanceSystem; map15!initFlightDirector;
       /* initializeAvionicsSoftware; */ goto avionicsSoftwareInitializedState
    od;

 avionicsSoftwareInitializedState:
    do
    :: break
    od;
 }

 proctype warhead()
 {
```

```
goto startState;
 startState:
    do
    :: atomic{map21?initWarhead; true ->} /* initializeWarhead; */
       goto unarmedState
    od;

 unarmedState:
    do
    :: atomic{map13?armMissile; true ->} /* armWarhead; */  goto armedState
    od;

 armedState:
    do
    :: break
    od;
 }

 proctype airframe()
 {
goto startState;
 startState:
    do
    :: map21?initAirframe; true -> /* initializeAirframe; */
       goto preLaunchState
    od;

 preLaunchState:
    do
    :: map12?doLaunch; true ->  goto poweredFlightState
```

```
   od;

 poweredFlightState:
   do
   :: map3?outOfFuel; true ->  goto inertialFlightState
   :: map6?estimatePosition; true -> /* setPosition; */
      goto poweredFlightState
   :: map17?changeCourse; true ->  goto maneuveringState
   od;

 maneuveringState:
   do
   :: map22?doManeuverComplete; true ->  goto poweredFlightState
   :: map3?outOfFuel; true ->  goto inertialFlightState
   od;

 inertialFlightState:
   do
   :: break
   od;
 }

 proctype throttle()
 {
goto startState;
 startState:
   do
   :: map23?initThrottle; true ->  goto idleState
   od;
 idleState:
    do
    :: map20?setThrottle -> map28!changeTankFlow; /* changeFuelFlow; */
       goto idleState
    od;
 }

 proctype jetEngine()
 {
goto startState;
 startState:
    do
    :: map24?initEngine; true ->  goto idleState
    od;

 idleState:
    do
    :: map1?changeTankFlow ->  goto idleState
    od;
 }

 proctype propulsionSystem()
 {
goto startState;
 startState:
    do
    :: map25?initPropulsionSystem; true -> map23!initThrottle;
       map24!initEngine; map26!initMissileFuelTank; goto idleState
    od;
 idleState:
    do
    :: break
    od;
 }

 proctype cruiseMissile()
 {
goto startState;
 startState:
    do
    :: AUTOMATIC; true -> map25!initPropulsionSystem; map21!initAirframe;
       map20!initWarhead; map28!initAvionicsSoftware;
       /* initializeCruiseMissile; */ goto preLaunchState
    od;

 preLaunchState:
    do
    :: map27?launch; true -> map12!doLaunch;  goto flyingState
    od;

 flyingState:
    do
    :: break;
    od;
 }

 init
{
 atomic{ run missileFuelTank();
run navigationSystem();
run flightProfile();
run guidanceSystem();
run flightDirector();
run avionicsSoftware();
run warhead();
run airframe();
run throttle();
run jetEngine();
run propulsionSystem();
run cruiseMissile() };
}
```

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

1. REPORT DATE *(DD-MM-YYYY)* SEP 2000	**2. REPORT TYPE** Master's Thesis	**3. DATES COVERED** *(From - To)* SEP 1998 - AUG 2000

4. TITLE AND SUBTITLE Validation and Verification of Formal Specifications in Object-Oriented Software Engineering	**5a. CONTRACT NUMBER**
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) Thomson, Steven, A	**5d. PROJECT NUMBER**
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Air Force Institute of Technology 2750 P Street WPAFB OH 45433-7765	**8. PERFORMING ORGANIZATION REPORT NUMBER** AFIT/GE/ENG/00S-01
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Roy F. Stratton AFRL/IFTD 525 Brooks Rd. Rome, NY 13441-4505 (303) 315-3004	**10. SPONSOR/MONITOR'S ACRONYM(S)**
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

The use of formal specifications allows for a software system to be defined with stringent mathematical semantics and syntax via such tools as propositional calculus and set theory. There are many perceived benefits garnered from formal specifications, such as a thorough and in-depth understanding of the domain and system being specified and a reduction in user requirement ambiguity. Probably the greatest benefit of formal specifications, and that which is least capitalized upon, is that mathematical proof procedures can be used to test and prove internal consistency and syntactic correctness in an effort to ensure comprehensive validation and verification (V\&V). The automation of the proof process will make formal methods far more attractive by reducing the time required and the effort involved in the V\&V of software systems.
It is commonly perceived that since a formal specification is written using strict mathematical notation, it is a minor task to ensure that the product does in fact meet the original specification and that the specification meets the end user's requirements. This is not the case. The majority of research in formal methods has delved into the development of formal notation and inference rules. The emphasis of this research is the validation and verification of formal object-oriented (OO) specifications.

15. SUBJECT TERMS

Formal Methods, Software Engineering, Validation, Verification, Object-oriented

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			Dr. Thomas Hartrum
U	U	U	UU	206	**19b. TELEPHONE NUMBER** *(Include area code)* (937) 255-3636