Roadmap to a DO-178C Formal Model-Based Software Engineering Methodology

Emanuel S. Grant, *Member, IAENG*, Tanaya Datta

Abstract—**Aircraft software systems are categorized as safety critical systems. This is due to them being employed in high-risk tasks that require rigorous development methodologies to assure their integrity. Designing these systems require: 1) thorough understanding of their requirements, 2) precise and unambiguous specifications, and 3) metrics to verify and validate the quality of software produced. Safety critical aviation systems must adhere to standards such as the RTCA DO-178C in order to be acceptable by regulatory agencies. The DO-178C focuses on all aspects of round trip software engineering. This paper outlines a software engineering methodology that is model-based and incorporates formal specification techniques towards being DO-178C compliant.**

Index Terms— **Formal specification technique, methodology, Z notation, UML, DO-178C**

I. Introduction

Avonic software systems are categorized as safety critical systems. This is due to them being employed in high-risk tasks that require rigorous development methodologies to assure its integrity. Failure of safety critical systems could result in injury, loss of life, data, and property. Safety critical aviation systems must adhere to standards such as the RTCA DO-178C [1] in order to be acceptable by the United States of America (USA) Federal Aviation Administration (FAA) and other interested parties. The DO-178C focuses on all aspects of round trip software engineering and requirements based testing as key elements of software verification to uncover errors.

Model-based software development (MBD) [2] places software models as the primary artifacts of development. Models are abstractions of software implementations and can be used to show a particular view of a system (e.g., the communication between system components or real-time performance aspects). Precise models that abstract out irrelevant details enable clear documentation, automated analysis, efficient simulation, testing, and automated code generation. The complexity of software used on avionic systems means that key criteria for software success (e.g., safety, reliability) cannot be assessed by examining the code alone. Abstractions of the code are needed to verify reliability and safety properties that are necessary for mission success.

The focus of MBD is to transform, refine, and integrate models into the software development life cycle to support system design, evolution, and maintenance [3]. They can be derived through forward or reverse engineering. Forward engineering is the process of moving from high-level abstractions and implementation independent designs to the implementation of a system [4]; while reverse engineering is the process of recovering design decisions, abstractions, and rationale from source code [5].

The Unified Modeling Language (UML) [6] is a set of graphical and textual notations for modeling various views of software systems, using object-oriented (OO) concepts. The UML is a standard modeling notation that was developed in response to the problems arising out of a proliferation of OO modeling notations, and has been accepted as the de facto modeling notation for OO software systems. System validation and verification are fundamental to assuring quality and reliability of safety critical systems. In model-driven software development, informal notations are often used in requirements capture and detail system design. Informal notations possess advantages, but are imprecise.

Formal Specification Techniques have been advocated as a supplementary approach to amend the informality of graphical software models [7] [8]. They promote the design of mathematically tractable systems through critical thinking and scientific reasoning. FSTs use a specification language, such as Z notation, to describe the components of a system and their constraints. Unlike graphical models, formal models can be analyzed directly by proof tools – which checks for errors and inconsistencies. Detractors of FSTs claim, they increase the cost of development, require highly trained experts, and are not used in real systems [9]. However, they have been used in case studies which unveiled that, FSTs facilitate a greater understanding of the requirements and their feasibility [10] [11]. Although the use of FSTs is sometimes controversial, their benefits to critical systems offset the disadvantages.

On a recently ended (but not concluded) UND UAS Risk Mitigation Project [10] [12] software development methodologies that comply with DO-178C objectives were required. The definition and implementation of such software development methodologies is a new, important, and urgent area of research for airborne operation software, and the broader safety critical software system domain. Key areas of learning from the UAS project were:

1. An algorithmic process for transforming the semi-

Manuscript received January 21, 2015. This work was supported in part by the University of North Dakota Faculty Research Seed Money Grant, May 2014. Emanuel S. Grant, Ph.D. is an associate professor with the Department of Computer Science, University of North Dakota, North Dakota, USA phone: 701-777-4133; fax: 701-777-3330; e-mail: grante@cs.und.edu. Tanaya Datta is a graduate research student with the Department of Computer Science, University of North Dakota, North Dakota, USA. tanaya.datta@my.und.edu.

formal software system representation to a formal for analysis and correction feedback was defined, i.e. a repeatable process (see Figure 4). This repeatable process will be compliant with DO-178C specification.

2. Automation or semi-automation tool use has to be a part of FST validation and verification process. Manual definition of the formal specification would result in the introduction of errors and the process must be repeatable.
3. The formal representation of the system will act as specifications for a health and status monitoring system (HSMS) for the process control system. A HSMS acts as an overseer of the process control system in operation, and report normal and abnormal changes in the state of the process control system. This will provide actionable knowledge to the operators in the event of any system failure.

A. *Research Goal*

This on-going work addresses continuing research from the UND UAS project. The continuing research focuses on: (1) definition of an object-oriented model-based software development methodology that features formal specification techniques (FST) for software validation and verification that comply with DO-178C guidance, (2) development of tool support for FST representation transformation, and (3) specification of health and status monitoring system for safety critical system development. Only (1) is reported on in this paper.

The following Section 2 outlines the research areas of the project, while Section 3 presents a UML description of the DO-178C specification. Section 4 presents the defined model-based software development methodology and Section 5 presents the conclusion and future work of the project.

II. RESEARCH COMPONENTS

A. *The UML*

Graphical object-oriented modeling languages are a subset of visual languages and are used for the modeling of problems and solutions within the software development field. Modeling languages are intended to be used for not only specifying models of software systems but to also facilitate documentation of the systems [6]. Use of modeling languages in software development is now focused around the UML (Unified Modeling Language) [6]. The UML as a language is used to communicate among developers about a system by means of "…captured knowledge (semantics) about a subject and expressed knowledge (syntax) regarding the subject" [1].

The UML as a modeling language focuses on the understanding of a system (subject) from the specification of graphical models of the system (subject) and the system's (subject's) related context. In this context the models contains knowledge about the system (subject). This leads to an understanding of visual software modeling languages as being similar to that of visual languages, i.e. comprising a syntax and semantics, as previously defined, but to be used to specify and document what is required and to be realized of a software system.

Diagrams in UML are categorized as structure, behavior, or interaction diagrams. Structure diagrams represent the static composition of the system. Examples of structure diagrams include class, component, object, deployment, and package diagrams. Behavior diagrams illustrate the dynamic features of the system by showing how the system is acted upon during execution. These diagrams include use case, activity, and state diagrams. Interaction diagrams are an extension of behavior diagrams but focuses mainly on the internal elements of the system. Examples of interaction diagrams include sequence and collaboration diagrams. Class diagrams and use case diagrams facilitate communication between nontechnical stakeholders and developers. The more complex UML diagrams such as sequence and state chart diagrams are more technical and suitable for astute stakeholders such as engineers and developers.

B. *Formal Specification Techniques*

Formal specification techniques (FST) involve the use of a specification language to describe software models with precision. It uses mathematical concepts and principles to design models that are sound and tractable. FSTs facilitate analysis of the syntax and semantics of models using proof tools. If errors are found, amendments can be made to the models in an evolutionary manner. The specification language that is used in this work is the Z notation [13], but use of other formal notation can be conducted. Z notation is used to describe software systems based on the mathematical principles of set theory and predicate logic. It was created by Jean-Raymond Abrial in 1977.

To transform UML models into Z notation, a Z schema will be created for each UML model construct in the class diagram. A schema in Z has two parts: a declaration part and a predicate part [13]. The declaration part is synonymous to the list of attributes in a UML class. However, the fundamental difference between the two is that, primitive data types are not utilized in Z schemas. Variable declaration types are expressed as mathematical notations or user defined types. The predicate part imposes constraints on the variables and its schema. These constraints are critical because they prohibit or permit a schema access to its environs. Figure 3 illustrates the structure of a Z schema.

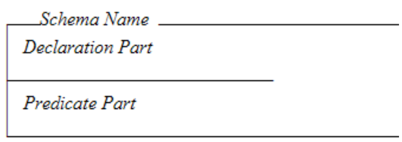

Figure 1. Z schema structure

Once the models have been transformed into the Z notation, they can then be analyzed by tools such as the Z/EVES [14]. Z/EVES is a proof tool that is used to checks the syntax and semantics of Z schemata. This is the process of software validation, by which software models undergo a series of analysis to check for errors and anomalies. It is also used to determine whether the quality of the software produced meets the user requirements and if it performs as

ISBN: 978-988-19253-2-9
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

expected. It is impractical for testing to detect all types of errors, and even the most rigorous testing procedure will, as stated by Edsger Dijkstra, show the presence of bugs but never their absence [15]. FST does not necessarily eliminate the need for software model testing, especially if they are models of a safety critical system.

C. Transforming Models

The level of abstraction provided by models helps developers and stakeholders visualize different aspects of the system while avoiding the details of implementation. This represents two principles of software engineering, namely the abstraction and separation of concern principles [16]. For any given system, a large number of models can exist and it is important to ensure their overall consistency. Model transformation uses a set of rules called transformation rules, which accepts one or more models as input and produce one or more target models as output [17].

Model transformation may be conducted manually or automatically. Manual transformations are conducted when transformation rules are not well defined, and lack an algorithmic description. Automatic transformation applies well-defined transformation rules through a toolkit. It is important, however, that the software engineer have a good understanding of the scope of the project, the syntax, and semantics of the source and target models irrespective of the transformation approach taken. This research defines an automated transformation processes to derive models that are more formal. In order to automate the aforementioned approach, a set of transformation rules are defined and applied to the models. The source models are UML model diagrams and the target model is their equivalent Z schemas.

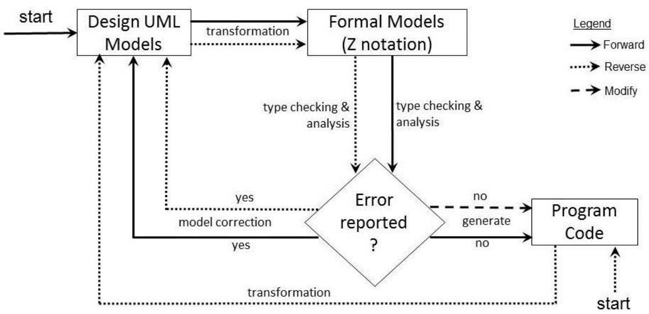

Figure 2. Informal to Formal Transformation Paradigm

At the end of each stage of the model development process, transformation may be conducted to go from an informal (UML) model to a formal (Z notation) representation (model). The purpose for this transformation is to conduct analysis of the formal representation of the system. Errors discovered during the formal analysis are then corrected in the formal models and this transformation-analysis-correction iteration continues until an acceptable level of safety assurance is achieved in the informal (UML) models. The UML models will eventually be transformed into code, once the desired level of detailed in accomplished at the PSM level of representation. Formal specification representations are usually not directly transformable to programming language code. Figure 2 graphically outlines this iterative transformation process for producing code in a model-based approach, as is at the heart of this research effort. Figure 4 captures the duel approaches of forward and reverse engineering, wherein the solid depicts the forward engineering path and the broken line depicts the reverse engineering approach. Eventually, both paths terminate the iteration with the generation of executable code of the safety critical system

III. DO178C IMPLEMENTATION

In order to develop a model-based software development methodology that complies with the DO-178C specification a series of UML models were developed to represent aspects of DO-178C. This approach taken is similar to the approach used in defining the UML specification [6]. Figure 3 depicts a high-level UML package model of DO-178C, which illustrates that the Software Planning Process defines the Software Development Process and the System Integral Process. The Software Integral Process comprises the Software Certification Process, the Software Safety Quality Assessment Process, the Software Verification Process, and the Software Configuration Management Process. Each package is then further refined to provide the detail content of the package.

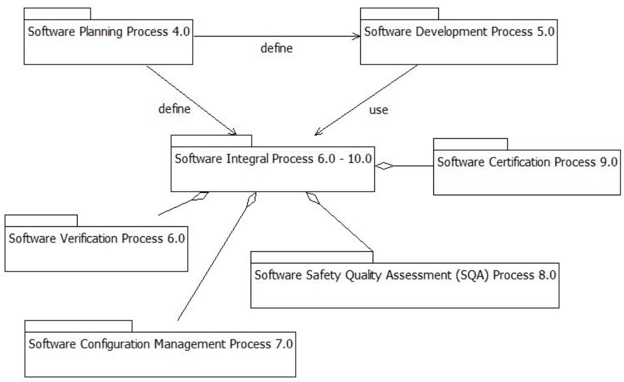

Figure 3. DO-178C high-level UML package diagram

Each of the packages of Figure 3 is decomposed into its components and these components are further decomposed into the low-level constituents of the DO-178C specification. These constituents are made up of processes, data items, and constraints. The goal of this approach is to re-orient the DO178C textual specification into a more understandable hierarchical graphical model that presents an ontological map between the DO178C constituents. Numbers appearing in Figure 3 denotes the section number in the DO-178C specification [1] for the associated item. Figure 4 captures a subset of the high-level DO-178C processes that are necessary in order to be compliant. Similar to the Figure 3, the numbering in Figure 4 references the relevant section of the DO-168C specification.

Figure 5 is an elaboration of the Software Planning Process of Figure 3, which includes the Software System Planning Objective. Figure 5 illustrates that the Software System Planning Activity, which has been stereotyped as <<process>> is of the specialization sup-processes of Develop Software Standard, Plan Software Development, Review Plan & Standard, and Plan Software Integral. In order to accomplish these tasks the data items, which have been stereotyped as <<data item>>, PSAC (Plan for

Software Aspects of Certification), SDP (Software Development Plan), SVP (Software Verification Plan), SCM Plan (Software Configuration Management Plan, and SQA Plan (Software Quality Assessment Plan) are associated (created, and updated).

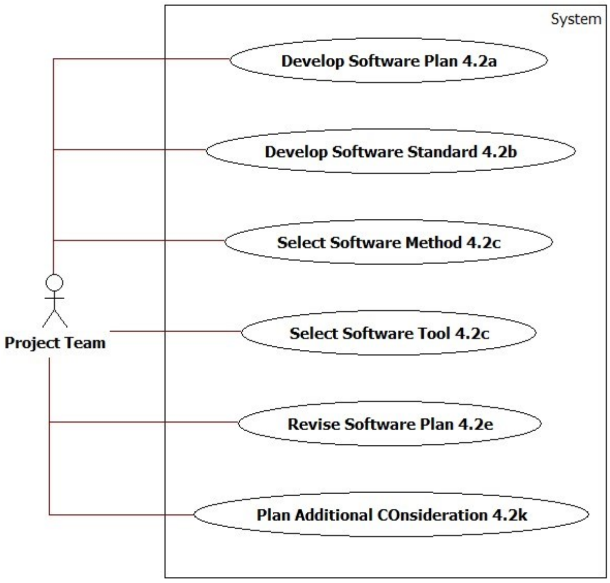

Figure 4. DO-178C Use Case Diagram

Figure 6 is an elaboration of the SDP of Figure 3. In Figure 6 it is shown that the Software Development Plan is composed of three <<data item>>; namely, Software Development Standard Plan, Software Life Cycle Plan, and Software Development Environment Plan. The Software Development Plan is in turn composed of the Software Requirement Standard, the Software Design Standard, and the Software Code Standard. The Software Life Cycle Plan and Software Development Environment Plan are similarly illustrated in terms of their components. The decomposition each component of DO-178C specification continues until the most detailed description is obtained.

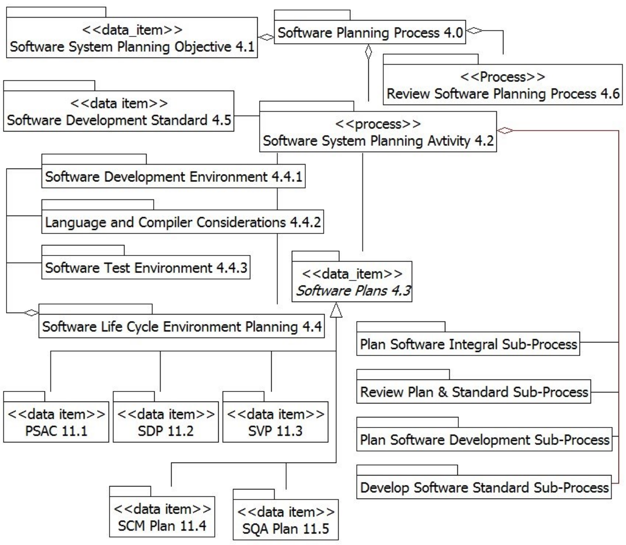

Figure 5. DO-178C Software Planning Process model

Figure7 captures the UML activity diagram description of the DO-178C Software Requirement Process Activity 5.1.2. This model illustrates that the Software Requirement Process consist of four sub-processes, with Acquire Domain Standard/Guideline, Acquire Requirement Document, and Conduct Survey/Interview being done concurrently (as needed) and Conduct Requirement Analysis 5.1.2b being done after the concurrency phase.

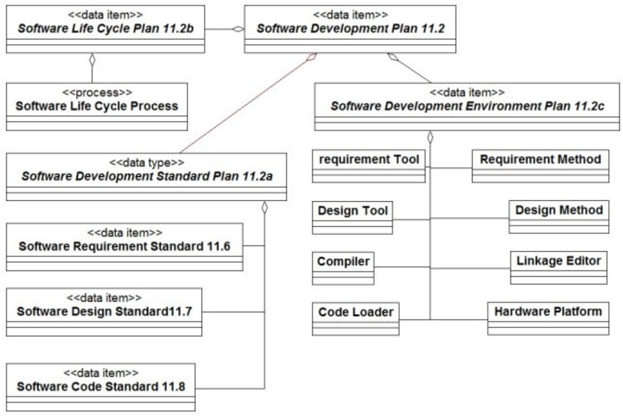

Figure 6. DO-178C Software Development Plan model

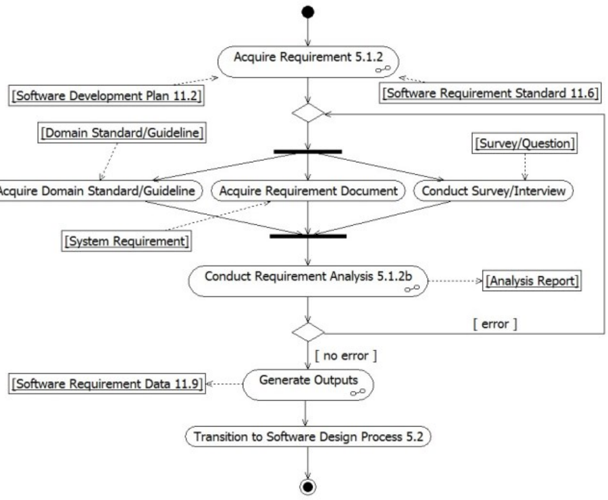

Figure 7. Requirement Process Activity Diagram

IV. MODEL-BASED DO-178C SOFTWARE DEVELOPMENT METHODOLOGY

Once all the UML models of the DO-178C specification have been completed then the model-based development methodology may be finalized. Figure 10 illustrates this methodology as a UML activity diagram.

Figure 8 presents a high-level UML activity diagram of the research model-based software development methodology that is compliant with the RCTA DO-178C specification for airborne software systems. The methodology incorporates tasks as described by the DO-178C for software development and incorporates a set of UML models, which are the bases for the software systems that are produced. The models are produced through a series of iterative, refinement, and transformational processes.

In Figure 8, starting with the input of the Software Requirement Data (11.9) models of UML use case diagram, use case specifications, and requirements-level class diagram, the Conduct High-Level Design sub-activity transforms these models into a series of UML design level class diagram and activity diagrams, and Z schema models as output. These outputs then become inputs to the Verity High Level Design (6.3) sub-activity, where they are transformed and refined into UML collaboration and sequence diagrams and state charts, along with refined Z schema models. The Verify High and Low Level Design sub-activities involve tool analysis of the Z schema models to identify errors. Any such error is reported back to the preceding sub-activity where corrective action is taken. Once errors are corrected, activity then transition in an iterative manner. The required Software Verification Results (11.14) are produced at the end of each sub-activity. Within each sub-activity of Figure 10, the refinement and transformation of the models are conducted in an iterative manner.

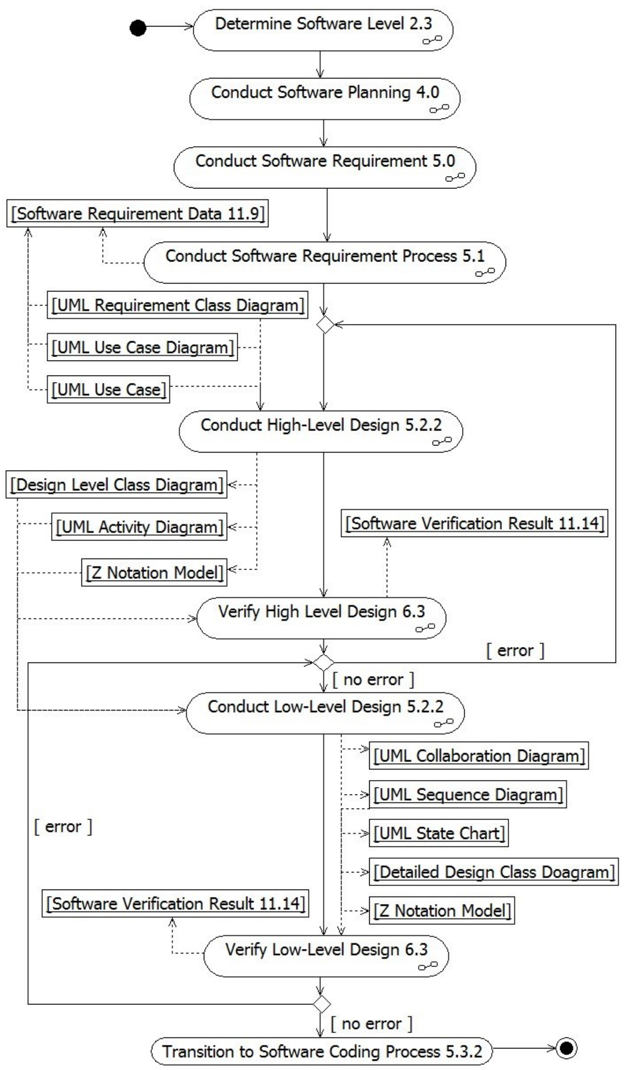

Figure 8. UML model-based methodology

A. Model Transformation Process

After the UML models were designed, the attributes, operations, and relationships of each class are analyzed separately. This analysis highlighted patterns, which appeared standard throughout the manual transformation of the UML models. From these patterns, a set of rules were defined that should yield representative formal models from their graphical counterpart – provided the graphical models are well-formed UML models. The process of transformation a UML model to its formal Z notation representation is captured in a set of rules. These ten rules are:

1. Declaration of Basic Types, Composite Types and Global Variables
2. Establishing Data Types for the Object Identity of each Z Schema
3. Define Attribute Schemata
4. Define Class Schemata
5. Define Identity Schema
6. Define Relationship Schemata
7. Define Parameter Schemata
8. Define Operation Schemata
9. Define configuration schema

An example UML class diagram and a subset of its Z specification are presented in figures 9 and 10 respectively.

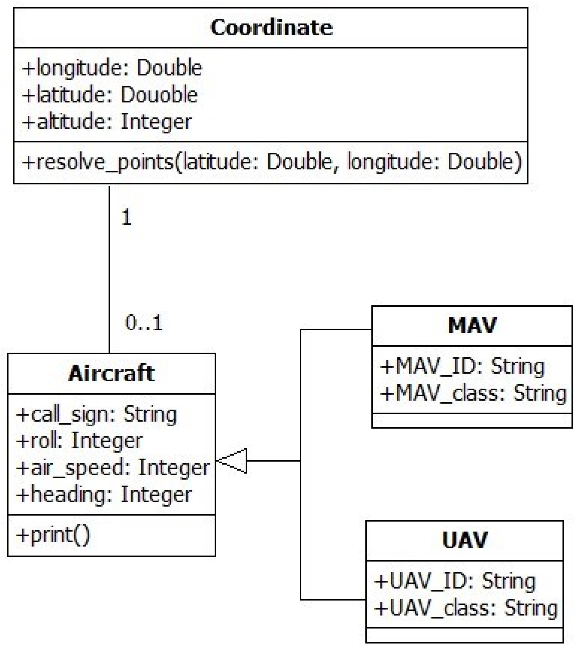

Figure 9. UAS Aircraft Class Diagram

$[STRING]$
$[AIRCRAFT]$
$[COORDINATE]$

$MAV: \mathbb{P}\ AIRCRAFT$
$UAV: \mathbb{P}\ AIRCRAFT$

$MAV \cup UAV \subseteq AIRCRAFT$

*_Aircraft_Attribute*____________
$call\_sign: STRING$
$roll: \mathbb{P}\ \mathbb{N}$
$air\_speed: \mathbb{P}\ \mathbb{N}$
$heading: \mathbb{P}\ \mathbb{N}$

$\forall\ air\_speed: air\_speed \bullet air\_speed \leqslant 250$

Figure 10. Z Schema

The next phase of this project involves implemented it on a large-scale industrial size project. It has already being implemented on large research project at the University of North Dakota. The UND – UAS Risk Mitigation Project was awarded a contract to develop a proof-of-concept air truth system, which monitors the operation of UAVs in the US National Airspace. The project started with minimal requirements; however, the timeframe for delivery was very rigid. This resulted in the rapid development of a prototype to assist in exploring and developing additional requirements.

The methodology was then applied to the class diagram of a component from the UAS Risk Mitigation System – i.e. The UAS Display System. The class diagram for this component contained 9 classes with a combined total of 455 attributes, 16 associations (including hierarchical relationships) and their respective multiplicities. There were a total of 56 operations that were analyzed; as well as the pre and post conditions of their respective 63 local variables and 28 parameters were evaluated. This derived 206 paragraphs in Z/EVES, which included the declaration of schemas, basic types, and axiomatic definitions.

A proposal is currently under review by NASA Aeronautics Research Mission Directorate for funding to conduct this approach to aircraft cockpit flight control systems. With incidents as the Air France, flight 447 crash in 2009, where software failure was a factor in the investigation it is crucial that such software be developed to a standard that is based on rigorous development, analysis, and verification and validation. A second funding proposal has been submitted to a major air cargo corporation for the development of an air cargo flight management system. It is anticipated that the lessons learned from any of these projects will contribute to the growing body of knowledge on model-based software development that incorporates formal specification techniques for verification and validation.

V. Conclusion

In this paper, a model-based software development methodology that complies with the RCTA DO-178C specification was presented. The purpose of this work is to facilitate software development in the domain of safety critical systems, specifically avionic software systems. This research effort is a derivative of work done on a University of North Dakota UAS project. Other related research areas include developing automation of some of the transformation processes defined in this methodology. An example of this is the transformation of UML class diagram graphical models to Z notation schema representation [10, 12]. The validation of this work will be demonstrated on the development of a safety critical system; this is the next phase of the work.

References

[1] RTCA Special Committee 205 (SC-205). "DO-178C - Software Considerations in Airborne Systems and Equipment Certification, RTCA, Washington DC, DO-178C, Dec. 2011.

[2] R. B. France, and B. Rumpe. "Model-driven Development of Complex Software: A Research Roadmap", *Proc. FOSE '07 Future of Software Engineering*, IEEE Computer Society Washington, DC, p. 37-54, 2007.

[3] T. Mens, and P. Van Gorp, "A Taxonomy of Model Transformation", Electronic Notes in Theoretical Computer Science, vol 152, *Proc. of the International Workshop on Graph and Model Transformation* (GraMoT 2005), p. 125-142, March 2005,.

[4] E. J. Chikofskyand J. H. Cross II, "Reverse Engineering and Design Recovery: A Taxonomy". *IEEE Software*. vol. 7, 1, p. 13—17, Jan. 1990.

[5] A. Sutton, and J. I. Maletic. "Recovering UML class models from C++: A Detailed Explanation". *Information Software Technology*. 49, 3, 212—229 2005.

[6] ISO/IEC 19501, Information Technology - Open Distributed Processing, "Unified Modeling Language (UML)" Version 1.4.2, 2005..

[7] R. B. France, A. Evans, K. Lano, and B. Rumpe. "The UML as a Formal Modeling Notation"..*Computer Standards & Interfaces*, vol 19, 7, p. 325—334, Nov. 1998.

[8] A. Hall. "Using Z as a Specification Calculus for Object-Oriented Systems." *Proc. of the Third International Symposium of VDM Europe on VDM and Z - Formal Methods in Software Development*, p. 290—318 April 1990.

[9] A. Hall "Seven myths of formal methods." *Software, IEEE*, vol.7, no.5, p. 11—19 Sept. 1990.

[10] S. Clachar, and E. S. Grant. "A Case Study in Formalizing UML Software Models of Safety Critical Systems." In *Proc. of the Annual International Conference on Software Engineering*. Phuket, Thailand, Apr. 2010.

[11] R. B. France, J. M. Bruel, M. M. Larrondo-Petrie "An Integrated Object-Oriented and Formal Modeling Environment". *Proc. of JOOP*. 25—34, Oct. 1997.

[12] E. S. Grant, V. K. Jackson, S. A. Chachar. "Towards a Formal Approach to Validating and Verifying Functional Design for Complex Safety Critical Systems". *Proc. 2nd Annual International Conference on Software Engineering & Applications (SEA 2011)*, Hotel Fort Canning, Singapore, Singapore. April 2011.

[13] J. M. Spivey "The Z Notation: a Reference Manual." Prentice-Hall, Inc. 1989.

[14] M. Saaltink "The Z/EVES System: The Z Formal Specification Notation." *Proc. of the 10th International Conference of Z Users*, Reading, UK. April 1997.

[15] O. J. Dahl, E. W. Dijkstra, and C. A. Hoare, Eds. Structured Programming. Academic Press Ltd. 1972.

[16] C. Ghezzi, M. Jazayeri, and D.Mandrioli. "Fundamentals of Software Engineering" Prentice Hall, ISBN 0133056996, 2003

[17] S. Sendall, and W. Kozaczynski "Model Transformation: The Heart and Soul of Model-Driven Software Development." *Software, IEEE*, vol.20, no.5, p. 42-45, Sept.-Oct. 2003.

ISBN: 978-988-19253-2-9
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)