Verification & Validation of Object-Oriented Functional Design using Formal Specification Techniques

Vanessa Jackson & Emanuel Grant
Department of Computer Science
University of North Dakota
Grand Forks, North Dakota 58202
vanessa.jackson@und.edu

Abstract

UML is now an ISO standard used for graphically representing software systems. It possesses key advantages such as simplicity, intuitiveness and recently it has been considered as a semi-formal specification notation. However, UML falls short in the latter area because it utilizes loose semantics which leads to ambiguity among its models. In some cases ambiguity can be negligible, however in safety critical systems this may lead to detrimental consequences. One technique to eliminate this ambiguity is to transform UML models into an analyzable representation with the use of formal specification techniques. Prior work has been conducted in formalizing UML class diagram attribute constraints and from that research, the work conducted here follows to demonstrate formalizing the operation signatures of the said classes within the UML class diagram. This effort will look at how UML model verification and validation can be done by analyzing pre- and post-conditions of user-defined functions using Z notation – a formal specification language.

Keywords: Model transformation, formal specification technique, UML notation, safety critical system

1 Introduction

Common to every scientific field are the many levels of technical understanding among different stakeholders of any development process. In order to sustain effective communication among distinct levels, information is typically represented through some type of model. This model serves to abstract low-level or context-specific information in a manner that is comprehendible to most (if not all) stakeholders of the development undertaking. Likewise in software engineering, models are used to depict the key functional elements of a software system, and they are typically used to convey the essence of these functional elements to stakeholders of different technical backgrounds. The information they receive from models is used for many different purposes, all of which are important for successful integration of the efforts made by stakeholders in the software development process.

The elements that are within a model illustrate the individual system components and the communication which lies among them. In most instances, a model in itself can highlight how complex a software system is or will be based on the number of components displayed in the model and the coupling among them [1]. Models used in software engineering can be used in either a prescriptive manner – where the model illustrates what the upcoming system ought to be, or descriptively – where the model captures a system that has already been created. Nevertheless, the primary purpose of modeling (to provide effective communication) can be accomplished when used in either approach. The model being analyzed in this paper was used in a descriptive manner.

The most popular modeling language within software engineering used in both academia and industry is the Unified Modeling Language (UML). UML is a standard for system modeling and has been widely recognized and highly approved by industry professionals in most problem domains as a standard for modeling software systems. Now certified as an ISO standard, UML is established in the field as “the model” for representing software systems due to its object-oriented design, general purpose (not domain specific) nature and most importantly its ease of use. UML contains a wide variety of models which captures both the static and dynamic aspects of software systems. It allows the designer of the model to stereotype standard UML notations into his/her own application-specific notations within a particular domain and therefore it provides room for customization [2]. While this may be an advantage to provide flexibility in using UML modeling, one weakness in doing so is that it introduces ambiguity among its models. Hence, the key issue which arises when software engineers consider using UML independently for formal specification is that due to the existence of fragile semantics, it can only be considered as a semi-formal specification language. To compensate for this shortcoming and to make it worthy of formal specification utilization, we demonstrate in this paper the transformation of UML models to an analyzable representation with the use of formal specification techniques. The analyzable representation of the model is then explored and necessary corrections are carried out on the UML system models.

In this research, we will formalize a UML class diagram which has been developed for a safety critical system. These are complex software systems where failure could lead to

loss of life, significant harm to property or to the environment in which it operates [3]. The safety critical system we explore is concerned with the successful implementation of an Unmanned Aerial Vehicles (UAV) monitoring system, which will operate in U.S. airspace. Modeling in a problem domain of this nature and criticality is very significant for primarily communication among software developers and then secondarily, communication from the standpoint of developers to the many non-technical stakeholders involved in the project. As previously stated a UML class diagram has been developed of this system and the Z language will be used to formalize the model into an analyzable representation which will further be used for verification and validation (V&V) of the UAV monitoring system. A methodical approach will be used to demonstrate the transformation technique from UML class diagram function signature pre- and post-conditions to Z specification and an analysis conducted to demonstrate its efficacy within safety critical software systems.

2 Overview of UML and Z

2.1 UML

UML has been favored for decades to graphically represent software systems. “The objective of UML is to provide system architects, software engineers, and software developers with tools for analysis, design, and implementation of software-based systems” [4]. While the developers of the UML highlight its objective to be one concerned with the technical stakeholders of software development cycle, it has been noticed that throughout all industries UML upholds such intuitiveness that non-technical persons of interest of a software system are able to grasp understanding of a software design and its complexity.

The UML consists of a suite of object-oriented models which can be used to depict many aspects of a software system. These aspects include the static and dynamic concerns of the system, its behavior, implementation etc. Structure diagrams of the UML suite are used to model the static attributes of a system. The most commonly used model of this type is the class diagram – we will take look at this UML model in further detail. Other types of UML models include behavior and interaction which are both used to model the dynamic nature of a software system. Many may argue that the class diagram, though it is deemed a structure diagram, also illustrates dynamism in which it includes the function operators on each class in the diagram. Nevertheless, one cannot dispute that there isn’t a model within the UML suite that does not capture effectively some if not all the characteristics of any software system.

2.1.1 UML Class Diagram

Unarguably the most popular model of the UML group is the class diagram. It represents the notion of object-oriented design very well where it defines a system conceptually as a

group of classes and the relationships among them [5]. Each class has some attributes and the functions that manipulate them - altogether depicting the idea of encapsulation and information hiding. A class is identified by a rectangular object divided into three sections where the first and topmost shows the name of the class, the second lists the attributes of that class and the third segment lists the functions (operations) of the class. Class diagrams capture low level detail of a software system implementation in a manner that can still be comprehended by non-technical persons – one of the major factors making it a model of choice in the software engineering industry.

Figure 1 below illustrates an example of two classes that have some relationship between each other depicted by the line connecting them. Each has its own unique attributes (for example, Class A has Attribute_A_0 through to Attribute_A_n) and also some operations listed below the attribute list. Note that operation signatures may contain a list of parameters as well. For the scope of this paper, focus will be made on the operation signatures of classes.

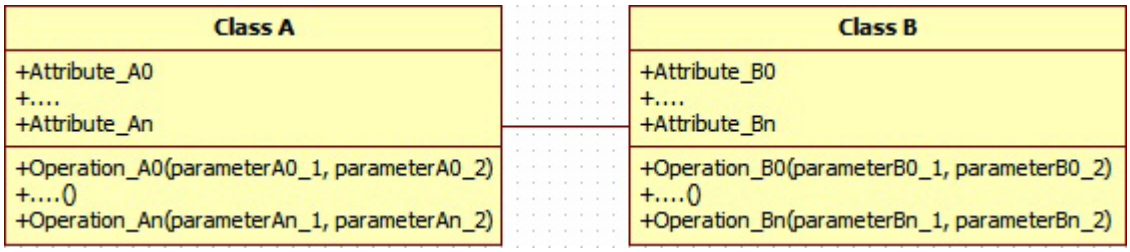

Figure 1 Example of UML Class Diagram

Operation signatures describe the name of the user-defined function in a class, the return type of that operation as well as a list of parameters that may participate in the execution of the respective function. As we go further in this paper we will demonstrate with use of examples, class diagrams that consist of implementation-level detail where operation signatures include the individual data type of each parameter passed to that user-defined operation. The operation signatures of classes implicitly depict the dynamic behavior of a class. In object-oriented design, we are knowledgeable of the fact that our user-defined functions in our classes are the elements that will change the state of that class - and the system consequently.

In an effort to transform our UML class diagram using formal specification techniques, we will uncover the notion of how the operation signatures of our user-defined functions will change the state of the system. When doing so we will identify pre-conditions for safe execution of each operation in each class as well as the post-conditions after successful operation execution. The formal specification language Z effectively captures this concept of operation execution and/or dynamism then precisely specifies the changes in the state of the UML classes, their attributes and parameters that are passed to operations within each class.

2.1.2 Disadvantages of using UML

UML's primary downfall is its very advantage - it is open to user or domain-specific modifiability. The lack of precise semantic rules in using the UML results in ambiguity among models and ambiguity in the interpretation of models. For example, UML allows one to annotate typical UML model objects (such as a class) using natural language. From this many interpretations can be yielded from the annotation and/or model, however for any system to function in a deterministically safe behavior there must be one and only one true interpretation of the functions of the system. As a result UML does not satisfy a key requirement for it to be used for formal specification especially NOT in the area of safety critical systems.

Even though UML contains behavioral and interaction models which both represent dynamism of a system's operations, they do not specify safe execution of the operations they model. As it relates to the class diagram, though it models the static aspect whilst 'implying' dynamic behavior through its operations, it still does not explicitly specify how the state of the system will change and if so will it be done with guaranteed safe execution. This type of specification is left up to the rigorous semantics of a formal specification language. UML class diagrams can be transformed using such a language in which safe execution of operations can be specified as well as explicit definitions of state changes in the software system can be clarified formally.

2.2 Z Notation

Developed at Oxford University, Z is a typed language based on set theory and first order predicate logic. As well as a basic mathematical notation, Z includes a schema notation to aid the structuring of specifications [6]. Z language uses typed mathematical facts including sets, relations, and functions in conjunction with first order predicate logic to build schemas. Schemas define its relevant variables and specify the relationship between them. A schema describes the stored data that a system accesses and alters [7]. A basic type is like a typical data type such as integer, natural number etc. however it is user-defined and problem specific. A schema may include one or several basic types. Fig. 2 and fig. 3 below shows how a schema and basic type is represented respectively.

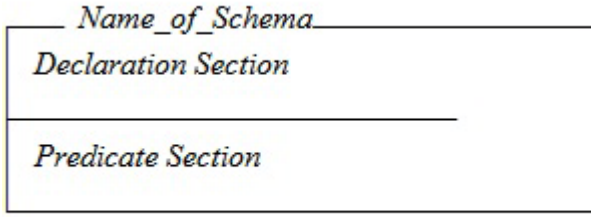

Figure 2 Example of a Z Schema

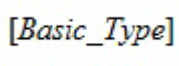

Figure 3 A Basic Type Represented in Z

There are two (2) types of schemas: state schemas and operation schemas. State schemas are used to define the static attributes of a system while operation schemas capture

dynamic aspects [5]. For the purpose of this research where we are concentrating on the verification and validation of user-defined functions by analyzing their pre- and post-conditions, the operation schema is the schema of focus to demonstrate our methodology. We use operation schemas for defining object-oriented user-defined functions in terms of the relationship between the state before function execution and after its completion. The declaration section of an operation schema contains variables representing the before and after states, inputs and outputs of a user-defined function. The predicate part of the operation schema will go further to define the relationship between the operation's before and after states [5].

An important aspect of formal specification on a whole (not just in Z) is that of 'state'. A system can be in one of several different states. Z specifies the concept of state in the way it captures the data that a system stores and how it changes them in its schemas. There are three types of conditions that are associated with operations. These include invariants, pre-conditions, and post-conditions [7]. An invariant defines what is guaranteed *not* to change by the execution of an operation in a schema. With the use of special notations, Z allows us to define a change in the state of individual schemas in a system. Some of the notations used in this research are listed below with their respective purposes.

Notation	***Description***	***Example***
Δ[Schema Name]	Shows that there is change in the state of the schema after the execution of an operation.	ΔAdd_Item
Ξ[Schema Name]	Shows that there is *no* change in the state of the schema after the execution of an operation.	ΞView_Item
Primed Variables	New value of a variable after execution of an operation.	x' (variable with ' after it)
Unprimed Variables	Value of a variable before execution of an operation.	x
Input Variables	Input variable to an operation.	y? (variable with ? after it)
Output Variables	Output variable from an operation.	z! (variable with ! after it)

Table 1 Notations used to Specify Change of State in Operation Schemas

3 The Need for Formal Specification

The high costs during the implementation and early test phases are most times caused by errors in specification and design phases [8]. This is a result of issues such as contradiction, ambiguities, vagueness, incompleteness and mixed levels of abstraction in the specification; these result in errors which are often propagated throughout subsequent stages in development where lesser error checking is done.

It is clear that natural language in itself is not an ultimate solution to specification. The use of UML models along with natural language has somewhat improved specification yet not completely and as precisely as it can absolutely be. In safety critical systems, where there is unconditionally no room for complete system failure, formal specification should be exploited when specifying systems of such superiority. During recent years, entities of safety critical industries have been requiring software developers to engage in exercising the benefits of formal specification to gain industry acceptance- the avionics industry is one key example as it also relates to this research. The standard that enforces this requirement in avionics is the DO-178B: Software Considerations in Airborne Systems and Equipment Certification, circa 1992. It defines strict software verification of safety critical systems with the use of formal specification. In order for this safety critical UAV system to be accepted by the United States Federal Aviation Administration (FAA) and all other entities of interest, the software must conform to the standards of the DO-178B – verification via the use of formal specification.

3.1 Formal Specification Techniques

According to [1], formal specification techniques (FST) incorporate the use of a specification language to describe software models with precision. As noted previously, the specification language utilized for this research is Z. FST also permits the use of proofing tools which identify errors in specifications executed within the proofing tool environment. The employment of FST (in order to accomplish UML model transformation as well as model V&V in this research) will look at checking and analyzing the Z schemas that have been derived from the system's UML class diagram. Z/EVES is a tool which has shown to be effective in detecting syntax and semantic errors of the Z representation of our UML model. Carrying out a series of analysis and error checking using the Z/EVES proofing tool is a significant phase in validating of our system model.

4. Model Transformation

Model transformation works by accepting one or more models then by applying some rules called transformation rules, a target model is attained which is equal to the input model [1]. In an effort to automate model transformation in the future as a byproduct of this research, a set of model transformation rules will be highlighted throughout the methodology to follow. Transformation is currently being conducted manually, however with some established transformation rules the process can be done automatically. Therefore as a result of this research, automatic UML class diagram transformation into their equivalent Z schemas will be a focus in future works. To aid in this potential study, the methodology below aims to highlight a step-by-step set of transformation rules which will be used to accomplish automatic UML class diagram model transformation.

5. Methodology

There has been numerous research conducted in UML class diagram model transformation using FST. One research in particular [1] from which this work is inspired looks at incorporating the works of [5] and [9] and as a result this paper will highlight some of their transformation strategies as well. As with this work, the above researchers/contributors demonstrated how to transform UML class diagrams too; however most of their work is focused on the static aspect the models. This paper goes further into formalizing the dynamic aspects of a system's class diagram and as a result, FST application will conduct verification and validation (V&V) on the UML system models. The derived approach achieved here through the amalgamation of the works of the above researchers will be used in a case study to conduct V&V on the UAV system safety critical system, research currently being conducted at the University of North Dakota (UND). From this we can identify the utility of formal specification in safety critical system domains.

To accommodate the future automation effort of model transformation and V&V processes which have been conducted in this research, our methodology outlines a stepwise approach such that if they are followed correctly, they should yield correct Z schemas from UML class diagrams.

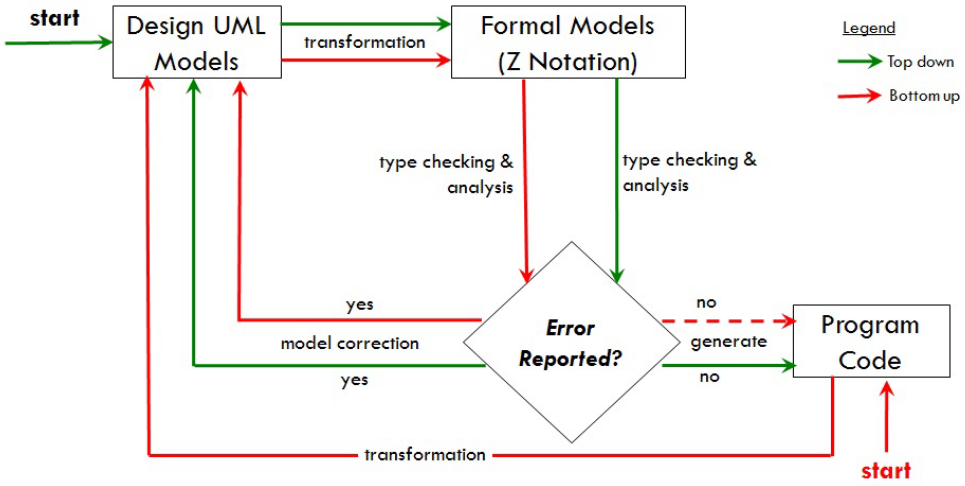

Figure 2 Transformation and V&V Process

Our approach is demonstrated in fig. 4 above, particularly the process flow depicted by the lines in red. Note that the process begins from program code – contrary to the 'textbook' approach of the software development. In theory, model transformation and V&V should ideally begin from the green starting point, however due to constraints on software development such as delivery time, competition, etc. sometimes a working prototype has to be developed. A prototype can be used just for demonstrating how the final product may look and appeal to the customer. Another use of prototyping is to get feedback and refine user requirements (or specifications). The system we are formalizing

here started off as a prototype and is now undergoing refinement and transformation into a feasible end-product. As a result, our model transformation starts from program code.

The code is transformed firstly to a UML class diagram, afterward FST is applied to the class diagram to translate it into formal models written in the Z language. The Z/EVES proofing tool is utilized to check the syntax and semantics of our derived formal models. If errors are found, then corrections are carried out to correct these errors in both our Z and UML models then to the program code, enforcing consistency throughout all design elements. This process is repeated until the entire system model has been transformed to Z, all Z schemas have been checked for errors by Z/EVES and all errors have been documented and corrected in the system model and program code.

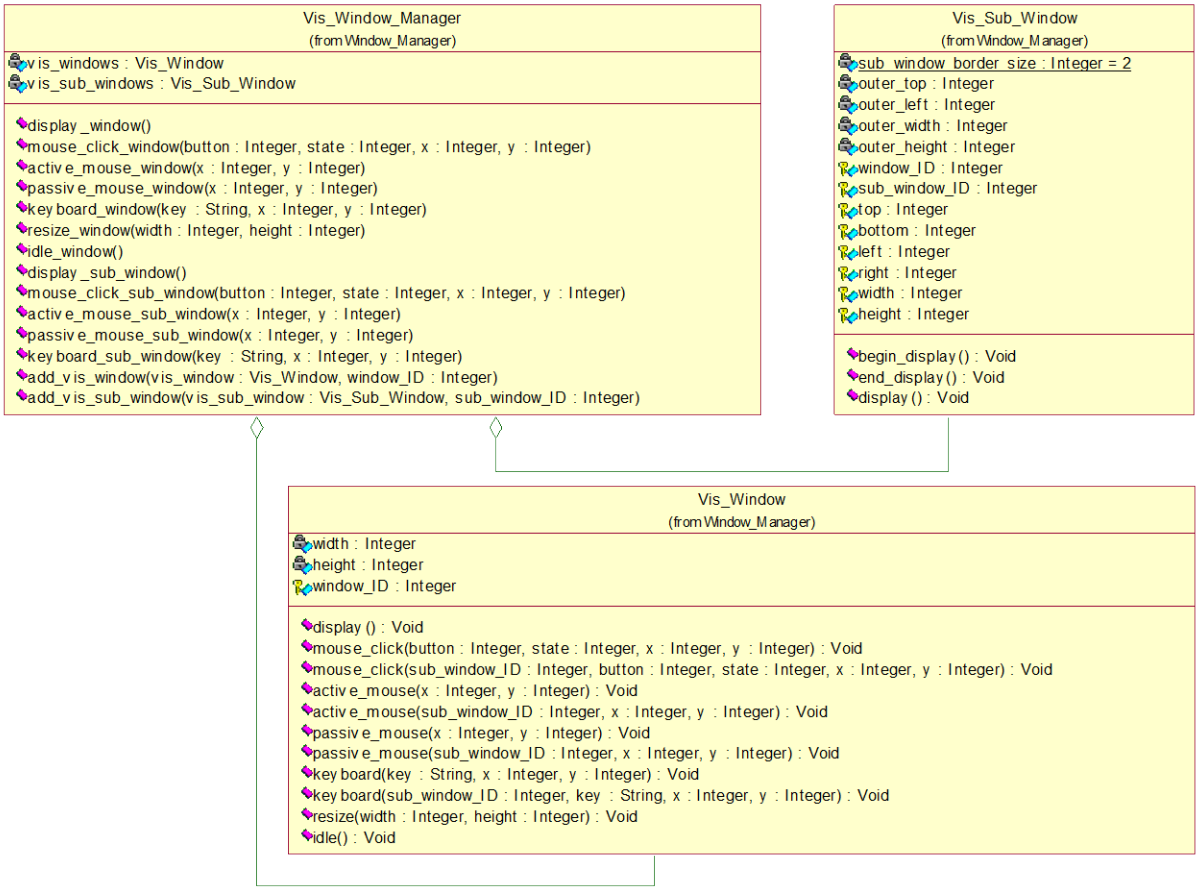

Figure 3 UAS Window Management Class Diagram Subset

The above is a small subset of classes from the system model currently being transformed. We will demonstrate the execution of our transformation rules on these classes. Our transformation rules include:

1) Defining attribute schemas
    - The attributes of a class are defined in the declaration section of a schema. If any constraints exist on any attributes, then these are defined in the predicate section of the schema.

2) Defining class schemas
    - Each class schema is defined by use of schema inclusion. This is done by including the previously defined attribute schema inside the declaration section of the class schema. Any constraints are placed in the predicate section.

3) Defining relationship schemas
    - This schema will define the relationship between each class schema that has been previously defined. It will put constraints on relationships between classes such as multiplicities in the same manner a class diagram would.

4) Defining parameter schemas for each method within a class
    - Each parameter listed in the operation signature of class methods (or user-defined functions) is defined similarly to the definition of attribute schemas.

5) Defining individual operation schemas
    - Again with the use of schema inclusion, an operation schema is defined by including the respective parameter schema of that operation. In addition, any other variable within the operation is declared and if any constraints exist on variables or parameter values, they are defined in the predicate section of the schema. A key notation that is also included in operation schemas is an identifier that indicates if the operation schema causes a change or not to the state of the system.

In every stage of defining the above Z schemas, each schema is checked for syntax and semantic inaccuracies using the Z/EVES proofing tool. Though this methodological effort warrants the need for automation, some constraints placed on the system require human involvement in the formalization process. An example is when defining constraints on variables that are inherent (domain-specific) or system-based (operational). There is no way for a computer application in itself to identify specific variables which should have certain values and constraints on them and then apply these constraints to them. Such an intelligent impression can only be deduced by humans who have adequate knowledge of the problem domain for which the system is designed.

The following is a set of Z schemas that have been developed from the above subset of classes from the system model (fig. 5). The results below were achieved with the use of our transformation rules outlined above.

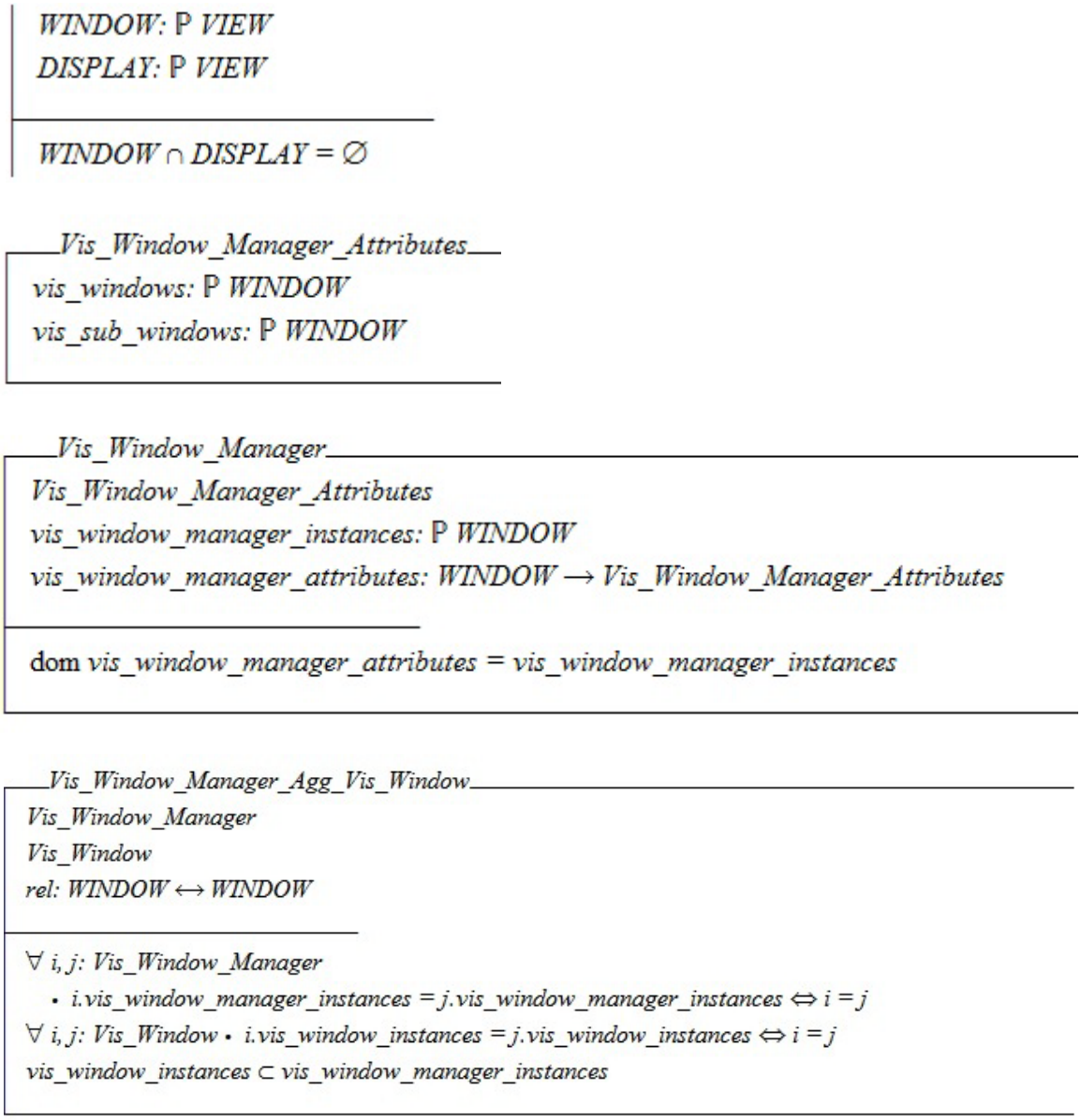

Figure 4 UAS Window Management Z Schema

Vis_Window_Manager_Agg_Vis_Sub_Window

Vis_Window_Manager
Vis_Sub_Window
rel: WINDOW $\leftrightarrow$ *WINDOW*

$\forall$ *i, j: Vis_Window_Manager*
- *i.vis_window_manager_instances = j.vis_window_manager_instances* $\Leftrightarrow$ *i = j*

$\forall$ *i, j: Vis_Sub_Window*
- *i.vis_sub_window_instances = j.vis_sub_window_instances* $\Leftrightarrow$ *i = j*

vis_sub_window_instances $\subset$ *vis_window_manager_instances*

Vis_Window_Manager_mouseClickWindow_Parameters

button: $\mathbb{P}\ \mathbb{Z}$
state: $\mathbb{P}\ \mathbb{Z}$
x: $\mathbb{P}\ \mathbb{Z}$
y: $\mathbb{P}\ \mathbb{Z}$

Vis_Window_Manager_mouseClickWindow

Ξ*Vis_Window_Manager*
Δ*Vis_Window*
Vis_Window_Manager_mouseClickWindow_Parameters
window: $\mathbb{P}\ \mathbb{Z}$
button': $\mathbb{P}\ \mathbb{Z}$
state': $\mathbb{P}\ \mathbb{Z}$
x': $\mathbb{P}\ \mathbb{Z}$
y': $\mathbb{P}\ \mathbb{Z}$

$\forall$ *w: window* • *w* $\geqslant$ 0
button' = *button*
state' = *state*
x' = *x*
y' = *y*

Figure 7 UAS Window Management Z Schema (cnt'd)

6. Conclusion

Formal specification removes ambiguity and adds strictness to software development. It has many benefits and contributes tremendously to software engineering. Though FST has been in existence before UML, its unattractive learning curve is a major trade-off in comparison to the intuitiveness of UML modeling. Due to the technicality in mathematical and predicate logic composition of formal methods, it is unlikely that typical general purpose software developers will utilize it for specification and/or V&V purposes.

In this research, we have exploited UML for communication between technical team members involved in the software development and also to non-technical stakeholders who each play important roles to make this project a reality. FST was used to alleviate the ambiguities inflicted by our UML models while adhering to the requirements of using formal methods for V&V in the safety critical system being developed. Our methodology highlighted a manual model transformation, however we believe the steps undertaken in the process can be used to develop some tool to automatically conduct UML class diagram transformation into Z notation - some human interaction may be necessary. It is evident to us that automation is necessary because the V&V of a large, complex safety critical systems being formalized manually is tedious and very time consuming – which leads to higher cost. If each user-defined function is formalized and its pre-conditions and post-conditions are explicitly specified then the chances of system failure is very unlikely.

Formal specification and proof does not *guarantee* that the software will be reliable in practical use [10]. This is so because the possibility does exist where the specification may not meet the requirements of the user, some of which are ever changing. Nevertheless, formal verification and validation increases confidence in safety critical systems. This is evident as most authoritative entities in safety critical industries are requiring the use of formal specification in order for end-products to be certified. This implies that there is high demand for formal V&V and that this practice is warranted.

Acknowledgement

The Joint Unmanned Aircraft Systems Center of Excellence (JUAS COE) at Creech Air Force Base, Nevada, provides technical administration of this research. DoD Contract Number FA4861-07-R-C003. Funding is granted under the University of North Dakota – UAS Risk Mitigation Strategy Project, Department of Defense Federal Initiative, USAF. Conference presentation of this work has been partially sponsored by the UND Intercollegiate Academic Funding Award No. [43]. Initiation of the research conducted by authors in [1] contributed immensely to this work.

References

[1] Clachar, S. & Grant, E (2010). *A Case Study in Formalizing UML Software Models of Safety Critical Systems*. In Proceedings of the Annual International Conference on Software Engineering. Phuket, Thailand.

[2] Snook, C. & Butler, M. (2006). *UML-B: Formal Modeling and Design aided by UML*. ACM Transactions on Software Engineering and Methodology (TOSEM) Volume 15 Issue 1, New York, NY, USA.

[3] Knight, J.C. (2002). *Safety Critical systems: challenges and directions*, Software Engineering, 2002. ICSE 2002. Proceedings of the 24rd International Conference on Software Engineering, vol., no., pp. 547- 550, doi: 10.1109/ICSE.2002.146305.

[4] OMG Unified Modeling Language (OMG UML), Infrastructure Specification, v2.3. May 2010.

[5] Shroff, M., France, R. B., (1997). *Towards a Formalization of UML Class Structures in Z*. COMPSAC '97 - 21st International Computer Software and Applications Conference. pp.646

[6] Bowen, J., (2003). *Formal Specification and Documentation using Z: A Case Study Approach.* Revised 2003. pp 4-6.

[7] Pressman, R., (2005). *Software Engineering: A Practitioner's Approach*. Boston, Mass.: McGraw-Hill.

[8] Potter, B. & Sinclair J., (1996). *An Introduction to Formal Specification and Z*. 2nd Edition. Prentice Hall.

[9] Evans, A., France, R., Lano, K. & Rumpe, B. (1998). The UML as a Formal Modeling Notation. Computer Standards and Interfaces, Volume 19, Issue 7, 2 November 1998, Pages 325-334.

[10] Sommerville, Ian. (2007). *Software Engineering*. Harlow, England.: Addison-Wesley.

[11] Ledru, Y. (1998). *Identifying Pre-conditions with Z/EVES Theorem Prover*. Proceedings of the 13th International Conference on Automated Software Engineering. IEEE Computer Society Press, Honolulu, Hawaii, pp. 32-41.