Towards a Formal Approach to Validating and Verifying Functional Design for Complex Safety Critical Systems

Emanuel S. Grant, Vanessa K. Jackson, and Sophine A. Clachar

***Abstract*— The quality and reliability of safety critical software systems are highly dependent on proper system validation and verification. In model-driven software development, semi-formal notations are often used in requirements capture. Though semi-formal notations possess advantages, their major disadvantage is their imprecision. A technique to eliminate imprecision is to transform semi-formal models into an analyzable representation using formal specification techniques (FSTs). With this approach to system validation and verification, safety critical systems can be developed more reliably. This work documents early experience of applying FSTs on UML class diagrams as attribute constraints, and pre- post-conditions on procedures. The validation and verification of the requirements of a system to monitor unmanned aerial vehicles in unrestricted airspace is the origin of this work. The challenge is the development of a system with incomplete specifications; multiple conflicting stakeholders' interests; existence of a prototype system; the need for standardized compliance, where validation and verification are paramount, which necessitates forward and reverse engineering activities.**

***Index Terms*— model transformation; formal specification techniques; requirements engineering**

I. Introduction

IN history, the uses of UAS technologies lie at the core of military operations such as surveillance, target identification and designation, mine detection, and reconnaissance [1]. Unmanned Aircraft Systems (UAS) technologies are categorized as safety critical systems. This is due to their being employed in high-risk tasks that require rigorous development methodologies to assure its integrity. A system that is defined as safety critical can have serious ramifications if a fault occurs. These implications include the risk of injury, loss of life, data, and property. Therefore, 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 United States RTCA DO-178B [16] to foster its acceptance by the Federal Aviation Administration (FAA) and other interested parties. The DO-178B focuses on all aspects of round trip software engineering and requirements based testing as key elements of software verification to uncover errors [9].

The University of North Dakota (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. This resulted in the rapid development of a prototype to assist in exploring and developing additional requirements.

The Unified Modeling Language (UML), developed in the early 1990s, is the ISO standard for designing and conceptualizing graphical models of software systems [2]. Graphical software models, such as UML models, possess simplistic designs and promote good software engineering practices. However, they are not without flaw as they are often imprecise and ambiguous. In addition, they are not directly analyzable by type checkers and proof tools. This makes it difficult to evaluate the integrity and correctness of graphical software models.

Formal Specification Techniques have been advocated as a supplementary approach to amend the informality of graphical software models [3] [4]. 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 [8]. Unlike graphical models, formal models can be analyzed directly by a proof tool. Detractors of FSTs claim, they increase the cost of development, require highly trained mathematicians, and are not used in real systems [5]. However, they have been used in case studies which unveiled that, FSTs facilitate a greater understanding of the requirements and their feasibility [6] [7] [13]. Although the use of FSTs is sometimes controversial, their benefits to

Manuscript received February29, 2012. This work was partially funded under the University of North Dakota - UAS Risk Mitigation Strategy Project. Unmanned Aerial System Remote Sense, Department of Defense Federal Initiative, Joint Unmanned Aircraft Systems Center of Excellence at Creech Air Force Base, Nevada, DoD Contract Number FA4861-07-R-C003.

E. S. Grant is an Associate Professor with the Department of Computer Science, University of North Dakota, Grand Forks, North Dakota, ND 58202 USA, phone: 701.777.4133, fax: 701.777.333., email: grante@aero.und.edu.

V. K. Jackson is a graduate student at the Department of Computer Science, University of North Dakota, Grand Forks, North Dakota, ND 58202 USA..

S. A. Clachar is a graduate student at the Department of Computer Science, University of North Dakota, Grand Forks, North Dakota, ND 58202 USA..

DOI: 10.5176_2010-2283_2.1.151

critical systems offset the disadvantages.

In the traditional approach to software engineering, graphical models would precede code generation. However, it is common for a prototype to preexist. In such scenarios, reverse engineering activities are used to derive the graphical models. This approach was undertaken in this research; along with forward engineering tactics, which ensured the derived models were abstract and susceptible to evolution.

II. PROJECT DESCRIPTION

An obstacle to field-testing of UAS is integrating their flight with manned aerial vehicles (MAV) in national air spaces (NAS). In order for this integration to be possible there is the requirement for a system that ensures the possibility of a mid-air collision or near-collision is the same as or better than that which now exists for MAVs operation [1]. Towards this goal the UND – UAS Risk Mitigation Project was started. This project will provide support to UAV experimentation and training, and defense assistance to civilian authorities.

The UND – UAS Risk Mitigation Project architecture is composed of three main components: a radar system, a data computation unit, and a display system. The display system software is the focus of the work presented herein. The project had a strict deadline for a series of deliverables over an initial one-year Phase 1 timeframe, and successful completion of the first phase would result in a subsequent two-year Phase 2 timeframe for the project. At the heart of Phase 1 efforts was the development of the system architecture, and the agents assigned to the architectural components. The development of a prototype display system was stated at the outset, with the intent of identifying the requirements of the proposed project. This report documents the effort and initial results of the system validation and verification effort, and its relationship to the system requirements specification. Work on the system requirements specification, and validation and verification is being conducted by the same project sub-team.

A project sub-team made significant progress with the development of the display system, which was viewed as the core of the project, from the point of demonstrating to the various stakeholders the viability of a system to monitor airspace and facilitate safe operation of both MAVs and UAVs. Consequently the validation and verification team were forced into conducting both forward and reverse engineering activities at an early stage in the project life cycle. The diagram below captures the dual-approach software engineering being carried out on the project.

Fig. 1 outlines the concurrent approaches in use on the project, for formally verifying and validating the software system, specifically for the display component. The solid arrow lines of Fig. 1 depict the forward engineering path of the process. A set of graphical design models (in this case UML class diagram [2]) are developed beginning with the system specification. The graphical models are transformed into a formal specification (in this case the Z notation [8]) representation for analysis. From the formal analysis, the decision is made whether to modify the graphical models or proceed with code generation from the models, based on the presence or absence of identifiable errors. The dotted arrow lines of Fig. 1 depict the reverse engineering part of the process. This begins with reverse engineering of the graphical design model (in this case UML class diagram), from the source code (in this case the display system). Once the models have been recovered from the code, the process follows the path of the forward engineering steps. The exception is that code is not generated, but modified (as it already exists); this is depicted by the dashed arrow line from "Error Reported" to "Program Code". The aspects of the work documented in this report are, the formalization steps defined to transform the graphical models into the formal specification notation, and the early lessons learnt in carrying out the work.

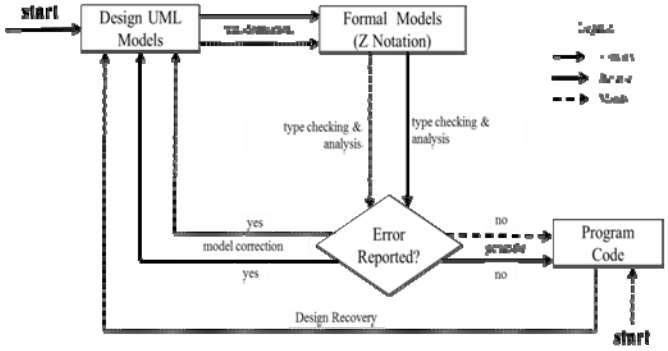

Figure 1. Forward/Reverse Engineering for System V & V

III. BACKGROUND

The focus of Model Driven Engineering (MDE) is to transform, refine, and integrate models into the software development life cycle to support system design, evolution, and maintenance [12]. Models serve many purposes and their use varies from stakeholder to stakeholder. The purpose of modeling, from a developer's standpoint, is to represent the proposed system. Models should be a coherent, cohesive, and abstract means of showing how the proposed system will address the user's needs. 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 [11]; while reverse engineering is the process of recovering design decisions, abstractions, and rationale from a source code [10].

The UML is an object-oriented modeling language for specifying, visualizing, constructing, and documenting the artifacts of software systems [2]. Diagrams in UML are categorized as structure or behavior diagrams. Structure diagrams represent the static framework of the system, whereas behavior diagrams depict the dynamic features of the system. These informal models have an advantage of being expressive – which makes them easily conveyed to both technical and nontechnical stakeholders. However, UML

lacks precise semantics, which results in its models being subject to multiple interpretations. This is exacerbated by the use of natural language annotations – as a means of clarification and explanation of the modeling techniques adopted.

Formal specification has been in existence decades before the inception of UML, and employs mathematical concepts and principles to describe software models with precision through rigorous analysis [3]. Employing FSTs is not a substitute for graphical software models; they are complementary. While formal models reveal inconsistencies and omissions, the informal model is an explicable version of the formal models [13]. The specification language chosen in this work is Z notation. A specification written in Z notation models the proposed system by naming the components of the system and expressing constraints between those components [8]. Its formal basis enables mathematical reasoning, and hence proves that desired properties are consequences of the specification [8]. From these proofs, one can prove the specification is accurate and complete.

System behavior should always be deterministic in the domain of safety critical systems. These software systems encompass numerous highly complex processing components and have high demands for reliability and accuracy. Due to the continuous use of UML in software development, there is a need to resolve the informal semantics of the models it produces [6]. Transforming UML models into Z equivalences also provide formal analysis to accomplish validation and verification of software systems.

IV. TRANSFORMATION METHODOLOGY

A. Deriving Graphical Models

The prototype of the air truth system was designed using C++. There are many software tools, which have the functionality of reverse engineering graphical software models from source code; however, there is a sizable semantic gap between UML and C++ [10]. Therefore, these tools often derive differing abstract representations of the same source code. Consequently, the use of an automated tool to assist in deriving abstract representations of the safety critical system was deemed inappropriate; since many of these tools were closed source and one cannot understand or influence how these tools interprets the prototype. Given that the DO-178B is process oriented, one needs to have an existing methodology, or familiarity with the workings of any intermediary software, to establish compliance.

B. Identifying Classes, Attributes, and Operations

The UML class diagram is a graph of classifier elements connected by their various static relationships [2]. In C++, classes are usually preceded by the keyword "struct" or "class" and followed by a class name and delimiters. The declarations of attributes are preceded by its data type followed by the attribute's name. To define attributes in UML, their respective data types were replaced with data types that were platform independent. An alternative means of identifying attributes is from the collection of accessors and mutators [10]. An accessor is a function that returns a copy of a member variable without modifying its value, whereas a mutator is a function that modifies the value of member variables. Since accessors and mutators are not always present in the declaration of classes, this method was used to reinforce that certain variables are, in fact, members of the class.

Defining operation signatures in UML was synonymous to their declaration in C++. The difference be is that platform independent data types were used and all parameters were stripped of pointer references and array tokens. The source code in Table 1 is a simplified example that will be used to demonstrate the process of deriving a UML class diagram. From this source code, four classes were identified: Aircraft, MAV, UAV, and Coordinate. If a class referenced another class, e.g. the Aircraft class referenced the Coordinate class; this was not represented as an attribute of the class. Both Aircraft and Coordinate had methods and they were declared in their respective classes in the appropriate sections.

TABLE I. AIRCRAFT DATA SOURCE CODE

```
class Aircraft
{
 string call_sign;
 integer roll;
 integer air_speed;
 integer heading;
 Coordinate coordinate;
 void print();
};
class MAV : public
Aircraft
{
 string MAV_ID;
 string MAV_class;
};
```

```
class UAV : public
Aircraft
{
 string UAV_ID;
 string UAV_class;
};
class Coordinate
{
 double longitude;
 double latitude;
 double altitude;
void
resolve_points(double
latitude, double
longitude);
};
```

C. Identifying Binary Class Relationships

A binary relationship is an association that connects exactly two classifiers [2]. In UML, associations can be of three different kinds: 1) ordinary associations, 2) composite aggregate and 3) sharable aggregate [2]. C++ associations are identified as member variables, which reference another UML class [10]. An example of this was represented when the Aircraft class referenced the coordinate class. Distinguishing between general associations and aggregations was difficult. Research has shown that the weak form of aggregation is structurally equivalent to a general association between the two classes [10] [14]. Therefore, domain knowledge was used to determine whether a general association or aggregation is suitable.

D. Identifying Inheritance

UML represents inheritance through the generalization/specialization hierarchy. C++ clearly represents this in the declaration of each class generalizable class – where a sub-class declaration contains the signature of the super-class from which it inherits. In Figure 2, both MAV

and UAV classes are sub-classes of Aircraft.

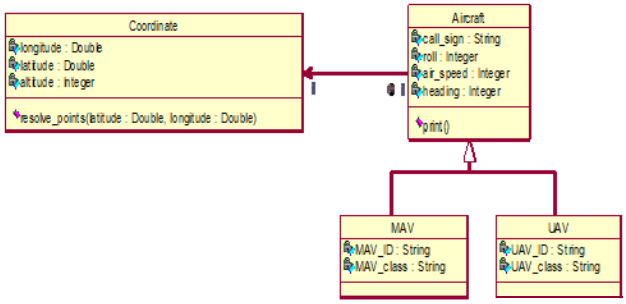

Figure 2. Reengineered UML Class Diagram

E. Identifying Multiplicities

A multiplicity is a specification of integer intervals; where an interval represents a range of integers, in the format: lower-bound..upper-bound [2]. Identifying multiplicities, in the source code, was one of the most challenging aspects of this work. According to [10], only three types of multiplicities can be unambiguously identified from C++ source code: 1) only a single instance of a class is declared, 2) a reference to a single instance is declared, and 3) a fixed size array is declared. Other cases may result in inaccurate upper and lower bounds and requires further knowledge of the problem domain. From domain knowledge, an aircraft can be associated with one or more coordinates throughout its lifetime – since each aircraft will have a path from its origin to its destination. Focus was on the fact that an aircraft, at a given point in time, must have at most one coordinate; and a coordinate is associated with zero or one aircraft. Fig. 2 also shows the multiplicity that was derived from the source code.

F. Transforming Graphical Models to Formal Models

There is a plethora of literature on transforming UML class diagrams using FSTs. However, the disparity between these works does not advocate a strict methodology that is appropriate for automation. This work amalgamates the works of J. Anthony Hall and Robert France to define a strict set of sequential rules that will yield correct formal models. After the UML models were designed, the attributes, operations, and relationships of each class were 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. Some of the set of rules is as follows:

1. Declaration of Basic Types, Composite Types and Global Variables: Data types in Z are often referred to as basic types or given sets of the specification. A feature of the Z notation is that it offers a calculus for building large specifications from smaller components [4] – and basic types facilitate this. Currently, identifying basic types for a Z specification is a manual task. The software engineer must examine the attributes of each UML class to identify data types, which do not have an equivalent representation in the Z mathematical toolkit. To automate this process, an operation defined as a 'Basic Type Parse' can be performed on each UML class. This process will scan the attributes of each class to identify any data types that are not present in the Z Mathematical Toolkit. The data types identified can be declared as basic types. The result of a basic type parse on Figure 2 will return two new basic types – [STRING] and [DOUBLE].
2. Establishing Data Types for the Object Identity of each Z Schema: The creation and manipulation of objects are essential in the object-oriented paradigm. A class along with its respective attributes and operations embodies an object. Its framework may remain the same; however, its state will change. To account for this, each Z schema will have a property called an object identifier. An object's identifier makes it unique and distinguishes it from all other objects within the system [4]. A basic type will be declared to represent the object identifier of each UML class. This step only seeks to establish the basic types. Applying this step to Figure 2 will produce the following: [AIRCRAFT] [COORDINATE] [MAV] [UAV].
3. Define Attribute Schemata: Each UML class may contain zero or more attributes and an arbitrary number of operations. Therefore, two cases arise:

   Case 1:- UML classes with zero attributes and one or more operations. In case 1, the definition of an attribute schema is unnecessary. However since there are operations performed by the class; proceed to the subsequent step – declaration of the class schema. In the event that no attributes or operations are defined in the UML class, a representative Z schema would be invalid and rejected by Z/EVES.

   Case 2:- UML classes with one or more attributes and zero or more operations. The attributes of each class will be declared in its attribute schema. A UML class can be described in terms of its intension and extension. The intension defines attributes and constraints on the class; whereas the extension describes characteristics that instances of the class should have in common [2]. To obtain the attributes and their respective initialized values, each class will be subject to an 'Intension Parse'. This task is performed sequentially on each attribute of the UML class, in two stages, to identify: 1) the attribute's name and data type; and 2) values, which initialize or constrain the attribute. In the initial phase, there needs to be a one-to-one mapping between the attribute and one of the previously defined basic types or a data type present in the Z mathematical toolkit. The second phase will identify attributes and their respective initialized values. This will be declared in the schema's predicate part.

Two categories of constraints were identified in this

work:

- Domain-specific constraints: - these constraints are based on the problem domain e.g., headings are values between 0 and 360 degrees. These constraints cannot and will not change irrespective of the system being designed.
- Operational constraints: - these are constraints that are imposed on the system's operation; e.g., an aircraft's air speed should not exceed 250 knots. Even though these values can be exceeded, in the current system, if that occurs it should be flagged as erroneous.

Domain constraints will be defined in the class schema and system constraints will be defined in the attribute schema.

4. Define Class Schemata: This step focuses on the extension of UML classes. It is not mandatory to separate the definition of the attribute and class schemata; however, combining them may result in a cumbersome schema. In addition, information hiding allows one to focus on what is relevant to the extension and intension of a class. A schema will be created with the name of each UML class, which will comprise of its attribute schema – via schema inclusion. Schema inclusion does not link the schemata; it only permits direct access to the contents of the included schema. Therefore, a variable will be created which binds the attribute schema to its class schema. The schema binding is represented as a partial function – which maps the schema's given set to the attribute schema.

The final step in the definition of the class schema is to establish a relationship between the object's identifier and the attributes of the class. This is represented in the schema's predicate part and states that object identifiers are associated with data items – i.e. the attributes. The naming convention for the class schemata is the UML class name followed by the keyword 'classifier'.

5. Define Identity Schema: The definition of this schema will reinforce that an object identifier cannot change. To capture this, an operation schema will be defined which will indicate that any change in an object's state should not affect its identifier.
6. Define Relationship Schemata: A relationship schema defines the types of relationships that exist between classes. It also depicts the number of object instantiations that are permissible for each class; these are represented by their respective multiplicities. Binary relationships and hierarchies are the focus of this work.
7. Define Parameter Schemata: This step will be conducted only if an operation accepts parameters. When creating these schemata, each data item in the parameter list of an operation is defined in a similar manner as the definition of class attributes. Each parameter will be identified by its name and corresponding data type, mapping each parameter name to a Z data type or a basic type, which was previously defined. The naming convention used for parameter schemata is the name of the class followed by the name of the method and the keyword 'parameters'.
8. Define Operation Schemata: Schema inclusion is exploited when creating operation schemata. It is used in the case where an operation has parameters – the corresponding parameter schema for an operation is included in the operation schema definition. Key notational conventions are used in operation schemata definition. They denote if the execution of a particular operation changes the state of the system. These respective notations will be included in the operation schema declaration part, followed by the name of the class schema which the operation execution has produced some change. In addition, if other variables are defined locally within an operation they will be declared as well. All constraints existing on variables and/or parameter values will be defined in the schema's predicate part. Most importantly, the pre – and post-conditions of operation safe execution is specified in these schemata.
9. Define a configuration schema: This schema will entail all the previously defined relationship schemata, along with the operation schemata.

Table 2 presents a portion of the schemata that were developed from conducting the formalization technique outlined above, on the class diagram of Fig. 2. Aircraft_Classifier, Coordinate_Classifier and Aircraft_print are two classes and one method, respectively, from Fig. 2.

V. I. RESULTS

At the project end, the requirement specification (forward engineering) activities have been through a number of refinement iterations. The reverse engineering activities have resulted in the manual development of a UML class diagram of the display system, as the first component to be reverse engineered. This class diagram is composed of 174 classes, including user-defined types, enumerations, and header file functions. There are over 2,250 attributes across these classes, which are linked by 383 associations (generalizations/specializations, aggregations, compositions, and regular associations). The model includes over 580 operations (methods) that specify 268 parameters.

In this paper, formal methods were applied on a simplified example to demonstrate the transformation process. The methodology was applied to the class diagram of component from the UAS Risk Mitigation 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 schemata, basic types, and axiomatic definitions. Some errors which were

discovered included: improper use of data types to handle data of a certain nature (for example, using strings to store numerical values that may be used in calculations), inconsistent data type declarations of similar variables, improper variable assignments and storage across functions, parameter/attribute conflicts, among other errors that are not detected in typical software testing.

TABLE II. SUBSET OF Z SCHEMATA FOR FIGURE 2.

[*STRING*]
[*AIRCRAFT*]
[*COORDINATE*]

MAV: ℙ *AIRCRAFT*
UAV: ℙ *AIRCRAFT*

MAV ∪ *UAV* ⊆ *AIRCRAFT*

Aircraft_Attributes

call_sign: *STRING*
roll: ℙ ℕ
air_speed: ℙ ℕ
heading: ℙ ℕ

∀ *air_speed*: *air_speed* • *air_speed* ⩽ 250

Aircraft_Classifier

Aircraft_Attributes
aircraft_instances: ℙ *AIRCRAFT*
aircraft_attributes: *AIRCRAFT* ⇸ *Aircraft_Attributes*

---

dom *aircraft_attributes* = *aircraft_instances*
heading = 0 .. 360
roll = 0 .. 360

Aircraft_OID

Δ*Aircraft_Classifier*

---

aircraft_instances' = *aircraft_instances*

Aircraft_print

Ξ*Aircraft_Classifier*
Aircraft_OID

∀ *a*: *air_speed* • *a* ≠ 0
∀ *h*: *heading* • *h* ≠ 0

Coordinate_Classifier

Coordinate_Attributes
coordinate_instances: ℙ *COORDINATE*
coordinate_attributes: *COORDINATE* ⇸ *Coordinate_Attributes*

---

dom *coordinate_attributes* = *coordinate_instances*
longitude = - 180 .. 180
latitude = - 90 .. 90

The application of the steps, outlined in the methodology, enlightened us on pragmatic approaches to applying formal methods in the validation and verification of other components in the project. The work effort, however, was very tedious which resulted in sporadic human errors in the specification. Consequently, there is need for a tool to support and simplify the formalization process. This will alleviate the workload, as well as reduce the possibility of human errors in the specification.

VI. Conclusion

Safety critical systems must adhere to stringent guidelines on validation and verification. As a result, the work documented in this report entails preliminary results and experience in conducting system validation and verification, via a formal specification technique. Due to the popularity and the standardized use of graphical software modeling, UML notation was for this system. The Z notation was selected for formal system representation and analysis because of the experience of the developers with this notation, and the availability of open source support tools.

Experience gained on this project has reassured the importance and benefits of FSTs to software development. The process identified numerous errors in the system, which was not detected during testing. These included design anomalies that were also identified. A deeper understanding of the system which FSTs forced the developers to attain as discussed in [13], drove the discovery of these latent errors.

References

[1] U.S. Dept. of Defense: FY2009-2034: Unmanned Systems Integrated Roadmap, (2009)

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

[3] France, R. B., Evans, A., Lano, K., Rumpe, B.: The UML as a Formal Modeling Notation. In Computer Standards & Interfaces, vol 19, issue 7, 325--334 (1998)

[4] Hall, A.: Using Z as a Specification Calculus for Object-Oriented Systems. In Proceedings of the Third International Symposium of VDM Europe on VDM and Z - Formal Methods in Software Development, 290--318 (1990)

[5] Hall, A.: Seven myths of formal methods, Software, IEEE , vol.7, no.5, 11--19, (1990)

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

[7] Jackson, V.: Verification & Validation of Object-Oriented Functional Design Using Formal Specification Techniques, In Proceedings of the 44th Annual Midwest Instruction and Computing Symposium, Duluth, MN (2011)

[8] ISO/IEC 13568, Information Technology: Z Formal Specification Notation - Syntax, Type System and Semantics. First ed. ISO/IEC (2002)

[9] Brosgol M. B.: Safety and security: Certification issues and Technologies. CrossTalk: The Journal of Defense and Software Engineering vol. 21 (10) 9--14 (2008).

[10] Sutton A., Maletic, J. I.: Recovering UML class models from C++: A Detailed Explanation. Information Software Technology. 49, 3, 212--229 (2007)

[11] Chikofsky, E. J., Cross II., J. H.: Reverse Engineering and Design Recovery: A Taxonomy. IEEE Software. Vol. 7, 1, 13--17 (1990).

[12] Mens, T., Van Gorp, P.: A Taxonomy of Model Transformation, Electronic Notes in Theoretical Computer Science, vol 152, In Proceedings of the International Workshop on Graph and Model Transformation (GraMoT 2005), 125-142, ISSN 1571-0661 (2006)

[13] France, R.B., Bruel, J., Larrondo-Petrie, M.M.: An Integrated Object-Oriented and Formal Modeling Environment. In Proceedings of JOOP. 25--34. (1997)

[14] France, R.: A Problem-Oriented Analysis of Basic UML Static Requirements Modeling Concepts. In Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, Denver, Colorado, United States, 57-69 (1999)

[15] Potter, B., Sinclair J.: An Introduction to Formal Specification and Z. 2nd ed. Prentice Hall (1996)

[16] RTCA, Inc, EUROCAE: DO-178B, Software Considerations in Airborne Systems and Equipment. SC-167 (1992)