The Role of Model-Oriented Software Architecture in Safety Engineering

Hassan Reza and Emanuel S. Grant
School of Aerospace Science
University of North Dakota
Grand Forks, ND 58202
{reza, grante}@cs.und.edu

Abstract

The gap between the safety developments for critical systems and safety validations is very significant. In this paper, we propose to study the development of a safety framework that bridges the gap between safety developments and safety validations using a set of different but related models. Our proposed framework, among other things, includes a collection of models capturing both the behavioral and safety requirements of critical systems together with a set of transformation and consistency preserving rules.

Keywords

Hierarchical Predicate Transitions Nets, Software architecture, Architectural Description Languages, Fault Tree Analysis, Model Oriented Software Architecture, Unified Modeling Languages

1. Introduction

Critical systems are software systems that demand ultra safety, because failures in these kinds of systems may result in loss of lives or cost a great deal of money. Examples of critical systems include software that is responsible for command and control over hazardous operations, software that monitors hospital and critical patient care units.

A central characteristic of critical systems is dependability, which is a collective property that combines different but related system level properties, such as availability, reliability, safety, and security [6]. The safety and reliability properties of dependable systems leave no provision for any kind of errors, because even a minor error may result in a major problem [7].

The traditional approaches to the development of safety critical systems places heavy emphasize on verification, validation, and certification to show the functional correctness of a system as dictated by requirement specifications, or the adherence to certain standards and guidelines as dictated by either federal or commercial organizations [8].

The main stream verification techniques include dynamic testing and static testing. The key ingredient of dynamic testing is to run the software product using a set of test cases based on the implementation (i.e., source code), or the specification of the software product. The source code-based testing method is based on the graph theory to select test cases. The specification-based testing (also known as conformance testing) is based on the specification of the software, and it is orthogonal to the software implementations.

Static testing is perceived as a complementary approach to dynamic testing. Static testing involves analyzing or inspecting the software product to reason about its correct behaviors. Static testing can be further subdivided into informal techniques [6, 8], such as code walkthroughs, code inspections, and formal techniques (i.e., techniques that employees mathematical and logical techniques to make verification more precise and hence more reliable). Examples of formal techniques include formal program analysis techniques to demonstrate that the program is logically correct [9], and formal specification analysis techniques to prevent the occurrence of faults [6].

Certification is a confirmation process. It simply attests that the software product (or the process by which the product) has been built complies with the certain standards set by some organizations. The product is certified when it

is ready to be marketed, which is normally at the end of software lifecycle.

In general, the development of critical systems is a complicated task, because it includes all the required system functionalities and capabilities, non-functionalities (e.g., performance, reliability, safety, security), and the process by which the system is built. In order to facilitate the development of these systems and increase the productively, it is reasonable to utilize powerful modeling framework such as, the Eclipse open source development platform (for productivity) [18], informal modeling notations, such as UML 2.0 based Model-Driven Development (ease of communications), and formal models (reasoning) to satisfy the dual requirements of functionality and safety associated with critical systems.

In this paper, we discuss a model-oriented software architecture method that improves safety in cost-effective manner. Our goal is to create a method to enhance quality and safety feature of critical systems in a way that is affordable, and verifiable. To achieve this goal, our proposed framework utilizes a set of complementary models that can be used for verification of design, and its implementation. The common denominator for both design and implementation is software architecture [1, 2]. To this end, there will be (at least) four research issues:

1. How do we architect critical software?
2. How do we verify safety requirements of architectural specification?
3. How do we use feedback from the verification process towards reducing the testing effort?
4. How do we generate test cases for the conformance of the implementation with respect to safety and behavioral specifications of software architecture?

1.1 Motivation and Background

In spite of advances in software engineering, which have advocated early feedback on the correct implementation and evaluation of safety requirements, verification and validations are normally performed at the end of the software lifecycle. Only hands full of techniques are available to date:

- To enforce verification activities for both system functionalities and quality at all levels of abstractions;
- Or to integrate safety models via architectural models into the software lifecycle development.

The existing practices in development of critical systems are not adequate, because systems are modeled using a homogenous models. Using a single-based modeling technique we are unable to capture all the required complexities stems from the interactions across subsystems interfaces, and assumptions about the users and/or the external environment, and quality of the system. Theses are serious limitations that may lead to fatal situations. What is needed is a combination of different but yet complimentary models, processes, and tool support to detect and prevent errors (safety and functional) early in the development stage when the cost of fixing the errors is very inexpensive.

Over past decade, architecture-based approach received a lot of attentions from both the academic and industries [3, 6, 7], because they manage the complexly of systems using appropriate constructs at a higher level of abstractions where system level qualities, such as performance, correctness, dependability and security can be investigated or tested [1,2,4,6,7,8].

Software architecture and its architectural description languages (ADLs) [3] have been used for front-end development activities and early feedback. The benefits of early feedbacks for safety purposes using software architecture and its complementary models make it possible to: verify the correct implementation of safety requirements at high level of abstractions, and to evaluate the correct response of software to safety violations and hazardous situations.

2 Approach

The complexity of software used on critical systems such as space mission software means that key criteria for software success (e.g., safety, reliability) cannot be examined by just going through data structure and algorithms. Software architecture of these systems is a multi-dimensional entity, and can be better understood and managed when they are represented by a set of models.

Early examinations of these models allow verifying reliability, safety, and security needed for correct implementation of critical systems. We propose to apply a blend of mature and widely applied methodologies rather than inventing new ones. To this end, we will employ Model Oriented Software Architecture (MOSA) and supporting tools [22]. More specifically, we will utilize models for formal analysis and transformation, because precise models that abstract out unnecessary details provide precise blueprints, automated analysis, efficient simulation and testing, and automatic code generation.

The Unified Modeling Language (UML) [12] (developed by the Object Management Group[1]) is emerging as a *de facto* software design standard based on objected-oriented methodology. The UML is a set of graphical and textual notations for modeling various aspects of software systems, using object-oriented (OO) concepts, and technology. UML is general-purpose modeling language that consists of a set of well-defined syntactically correct diagrams, which can be used to model both the structure and functional behavior of a system. UML has the advantage of being a visual language that supports intergraded design environment and tool (Rational Rose®[2]); the tool not only can be used to specify a system at low-level of design but also can be used to generate source code.

In the later half of the 20th century, we embraced the phenomena of graphical modeling language at the design phase of software development and concurrently, textual and graphical notation for system description at the architectural level of system development. Over the past few years, there have been attempts to consolidate the benefits that have been realized at these various levels of abstraction in software development. This consolidation has been manifested in the form of defining mappings/transformations to move from one level of abstraction to a lower level of abstraction within the software development cycle [21, 22].

In recent years, a lot of research has been focused on bridging the gap between architectural level descriptions of system and the UML design level models. One of the most significant features of the UML is that it facilitates multiple views of the system at the design level, but there is a conceptual and semantic gap between architectural models and the design models, primarily because of the use of different notations at the two levels. Prior works [23, 24] have used earlier versions of the UML (versions 1.1 – 1.5) to model systems at the architectural level. Such research is driven by the need to have homogeneous modeling notation throughout the development cycle.

The latest version of the UML specification (version 2.0) [20] addresses the concern of developing architectural level system models by introducing the metamodel elements of *components* (i.e., locus of computational) and *connectors* for use at the conceptual level of system modeling. In addition to *connector* (i.e., locus of communication), *port* (i.e., interface) is a new metamodeling concept introduced in UML 2.0. These three metamodeling concepts constitute the metamodel elements need to explicitly facilitate the implementation of an architectural description language with the UML. UML offers the following definition of these metamodel constructs:

- Component – "*A component* [is] *a modular part of a system that encapsulates its contents and whose manifestation is replaceable within its environment.*"
- Connector – "*A connector specifies a link that enables communication between two or more instances.*"
- Port – "*A port is a structural feature of a classifier that specifies a distinct interaction point between the classifier and*

[1] *www.omg.org*
[2] *www.ibm.com/rational*

> *its environment or between the (behavior of the) classifier and its internal parts.*"

Ports are connected to properties of classifier by a connector. The metamodel of connectors is illustrated at Figure 1, in which a component is a ***ConnectableElement***.

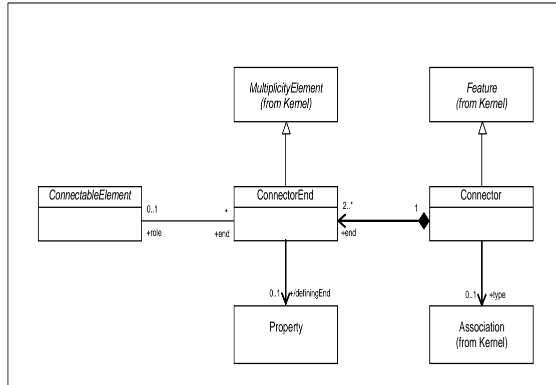

Figure 1: The metamodel of connectors

UML scenarios in form of use-cases are widely used to document system intended behaviors by specifying the set of all possible interactions between the system under construction and its external environment (i.e., a user or another system). Therefore, use-cases together with their instances (i.e., scenarios) are appropriate to generate test cases for testing the functionalities of a system using UML at a higher level of abstractions.

However, one of the main problems with UML/MDA is that it does not have formal semantics. As such, it cannot be used to formally analyze the intended behaviors of a system. Therefore, we need a formal model. For this propose, we are going to use a formally defined dynamic model to demonstrate the behavioral aspect of systems. To this end, we employ Hierarchical Predicate Transition Nets (HPrTNs) [10], which are executable, graphical formalism based on the sound theoretical foundation of Petri Nets [11]. Therefore HPrTNs can be used to formally analyze and verify the behavior of a system.

3. The model oriented software architecture

A straightforward approach for developing a critical system is to begin with the characterization of what constitutes a system's functionality, and what constitute the system's non-functionality (e.g., safety). The next logical step is to select a model representing a system at the highest level of abstraction. Software architecture, among other things, provides a highest level of abstraction in which a system is defined using high level constructs, such as components (i.e., computational elements), connectors (i.e., protocol of the interactions), configurations (i.e., topology), and non-functionality (i.e., performance, safety, reliability, etc.)[13].

Model Oriented Software Architecture (MOSA) consists of a set of models represented by different modeling notations at same (i.e., horizontal) or different level of abstractions (i.e., vertically). More specifically, each model can be used to specify and analyze the structure, behavior, and safety requirements of a system.

The first model, UML, is used to abstract the system by organizing it into architectural constructs (i.e., components, connectors, configurations) using classes, attributes, and methods. The main justifications for using UML as an ADL are that UML is supported by software development paradigm known as model driven architecture (MDA) [22], and the target platform for code generation known as Eclipse [18]. The main reason for using Eclipse and MDA is that they serve as a foundation from which systems can evolve from requirements to specifications to design to code generations. The liability with using UML is that UML does not have an underlying formal theory. As such, UML cannot be used to formally analyze the behavior of a system needed for the safety propose.

The second model, fault tree analysis (FTA), is needed to identify the possible causes forcing the system into the hazardous states and accidents. Fault Tree Analysis (FTA) is a safety engineering method to identify safety and security violations [15].

In brief, a fault tree is a tree with hazard state identified as the root of a tree, a set of primitive events causing an accident, and they are represented as leaves; and a set of intermediate nodes modeled by gates (e.g.,

'AND', 'OR', etc), which combine primitive events that contribute to the accident represented by the root node.

The FTA model is an ideal place to identify hazards; it can also be used for generating system level safety test cases. These test cases constitute what is known as misuse cases modeling safety violation scenarios.

To generate test cases, we will progress from leave nodes documenting the basic processing events in a tree hierarchy. The transitions between different nodes at different levels take place by logical gate operators (e.g., AND or OR). The existence of all possible paths from a set of basic events (a set of initial events) combined by logical gate operators to the root of a tree (final event) constitutes a set of test cases corresponding to the safety violation scenarios.

The third model, HPrTNs [10], are specification methods with well defined semantics and have been applied to describe both the structure (static semantics), and behaviors (dynamic semantics) of a system [10]. Formally, a hierarchical predicate transition net (HPrTN) is a tuple (P, T, F, ρ, *SPEC*, *INSC*) where P is a finite set of Predicates; T is a finite set of Transitions; F is a finite set of flows, ρ is hierarchical function mapping super nodes to HPrTNs; *SPEC* is an algebraic specification, and is defined by a tuple (S, OP, Eq) where: S and OP are called signatures. S is a set of Sorts (i.e., types) and $OP = (S_1,\ldots,S_n)$ is a set of sorted operations for $s_1,\ldots,s_n$, $s \in S$.

The main justifications for using HPrTNs [10] are that they belong to the class of visual formalisms with much wider modeling capabilities and applicableability based upon the sound theoretical foundation of Petri nets [3], and related analysis techniques, such as net executions and simulations, reachability graph, and place/transition invariants. They are useful to model both abstract dynamic and concrete dynamic architecture modeled by UML regardless of the style in which the architecture might be implemented. Therefore, HPrTNs specifications are ideal models to represent formal semantics models that can be subjected to simulation and/or execution using the nets or reachability graphs (RGs).

The UML architectural model is mapped to abstract HPrTNs models via mapping rules to represent the static and dynamic behavior of software architecture initially specified in UML. Abstract models can be used to test software architecture by testing the protocol of communicating and glue (composition). Abstract models can also be refined into concrete models to generate test cases that can be used to test source code.

To determine unacceptable state reachable from a specific state or certain functionalities are ensured, we generate reachability graph (RG) from the net. A sequence of enabled transitions together with all reachable markings from that specific marking to final marking constitutes a set of paths; these paths are used as a set of potential test cases to validate: 1) system level properties (e.g., safety), functional correctness, and the absence of livelock, and/or deadlock.

In order to guarantee a fail-safe system, the HPrTNs [10] models are then tested against those misuse cases generated via FTA. For example, if an HPrTNs model fails every single test case generated by FTA, then we can be confident that the safety aspect of a system represented by HPrtNs models has been ensured.

In general, the design and testing in the MOSA framework can proceed as follows:

1. Define use cases that collectively document the required functionality of the software system;
2. Identify misuse cases via Fault Tree Analysis (FTA) that collectively document the required level of safety of the software system;
3. Specify software architecture in UML;
4. Map a UML model representing the software architecture of a system to the dynamic model of a software in HPrTNs;
5. Test HPrTNs against use cases (scenarios) corresponding to the functionality of a system,;
6. Test HPrTNs against paths corresponding misuse cases in FTA to detect the safety violations;

7. Use the results of HPrTNs testing to debug design errors in UML models and its implementation (i.e., HPrTNs).

Based on the behavioral model of a system in HPrTNs, and safety models in Fault Tree Analysis (FTA), we should be able (at least) to detect:

1. Safety violations and potential hazards;
2. The inconsistency between functional behaviors and safety requirements;
3. Absence of identified safety violations from the integrated models of functionality and safety.

The feedback from HPrTNs can be used to fix UML [20] model. This activity, in turn, makes it possible to implement a safe design by preventing the propagation of errors.

The proper traceability between UML/MDA models and formal models, HPrTNs, will be established by architectural transformation as follows:

- Domain-specific transformations;
- Transformations to artifacts amenable to formal analysis of safety requirements.

The results of the above mentioned activities are needed to answer to the following questions:

- What types of activities are involved in an MOSA approach to developing space critical soft-ware?
- How does MOSA support dependability requirements and safety concerns?
- How to assess the impact of MOSA on developer productivity, software, and software safety?

The results of the above activities will be used to architect a MOSA framework that provides developers with a degree of flexibility that makes it possible to explore alternatives and hence make tradeoffs analysis in cost-effective manner.

4. Conclusions and discussions

The importance of software for safety critical systems has been growing rapidly over the past decade. In this position paper, we have proposed Model Oriented Software Architecture (MOSA), which is composed of a set of models, as an achievable alternative for managing the complexity attributed to critical systems, because a single model is not expressive enough to capture all the required complexities. Therefore, a combination of models is needed to raise the level of abstraction at which software is conceived, designed, analyzed/verified, and finally implemented. This approach, in turn, translated to reduced software development costs (in terms of the number of test cases) and increased safety assurance related to reliability and safety of critical systems.

In this position paper, we discuss an on-going research effort to combine a set of models, such as UML 2.0 [20] for understandability and communicate-ability, HPrTNs for analyzability, and FTA for safety, for developing critical systems.

As discussed earlier in this paper, we are planning to document our designs in UML2.0 and debug and test them at the model level, automatically generate code and test, and debug the system using Eclipse [18]. To achieve this, we are going to assess the feasibility of: automatic translation of HPrTNs from UML; automatic generation of test cases from FTA and HPrTNs; devise a set of UML-to-HPrTNs mapping (or transformation) rules to map UML to HPrTNs; write plug-ins for Eclipse [18] to generate code from HPrTNs via UML models;

A model checker normally detects a safety violation when it finds a sequence of operations that includes one or more operations associated to a safety property, and these operations do not observe a specific order or do not satisfy specific conditions [19]. The interface between SPIN's [17] and HPrTNs is a new and promising research work, because using SPIN's analytical capabilities, we should be able to automatically verify the violation of a safety (or a progress) property in the proposed model. For example, if a safety property violation is

detected, then SPIN generates a counter example.

5. Acknowledgement

This work was financially supported in part by ND-NASA EPSCoR thought NASA grant # NCC5-582.

6. References

[1] D. Perry, and A. Wolf., *Foundations for the study of software architecture,* ACM SIGSOFT Software Engineering Notes, October 1992.

[2] M. Shaw, and D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline,* Prentice Hall, 1996.

[3] N. Medvidovic, and R. Tayler, *Classification and Comparison Framework for Software Architecture Description Languages,* IEEE Transaction on Software Engineering January 2000.

[4] A. Bertolino, P. Inveradi, H. Muccini *Formal Methods in Testing Software Architecture,* LNCS 2804, pp: 122- 147, 2003.

[5] J. Kramer, J. Magee, S. Uchitel, *Software Architecture Modeling & Analysis: A Rigorous Approach,* LNCS 2804, pp: 44-51, 2003.

[6] J. Bowen, V. Stavridou, *Safety-Critical Systems, Formal Methods, and Standards*, IEE/BCS Software Engineering Journal, vol.8, no.4, July 1993.

[7] D. Parnas, A. Von. Schouwen, P. Shu *Evaluation of Safety Critical Software*, Communications of the ACM, vol.33, no.6, June 1990.

[8] C. Ghezzi, M. Jazayeri, D. Mandriolo, *Fundamentals of Software Engineering*, 2nd Edition, Prentice Hall, 2002.

[9] D. Gries, *The Science of Programming,* Springer-Verlag. 1981.

[10] X. He, *A Formal Definition of Hierarchical Predicate Transition Nets*, Proceedings of the 17th International Conference on the Application and the Theory of Petri Nets, Osaka, Japan, June 1996.

[11] T. Murata, *Petri Nets: Properties, Analysis and Application,*. Proceeding of the IEEE, vol.77, no.4, April 1989, pages: 571-580.

[12] Object Management Group, *Unified Modeling Language (UML) Specification*, Version 1.5, OMG Headquarters, Massachusetts, USA, Sept. 2001.

[13] G. Abowd, R. Allen, and D. Garlan, *Formalizing style to understand descriptions of software architecture*, ACM Transactions on Software Engineering and Methodology (TOSEM), vol.4, no.4, 1995.

[14] A. Egyed, *Validating Consistency between Architecture and Design Descriptions*, Proceedings of 1st Workshop on Evaluating Software Architecture Solutions (WESAS), Irvine, CA, May 2000.

[15] G. Helmer, J. Wong, M. Slagell, V. Honavar, L. Miller, Y. Wang, and R. Luz, *Software Fault Tree and Colored Petri Net Based Specification, Design and Implementation of Agent-Based Intrusion Detection Systems,* Requirements Engineering Journal, vol.7, no.4, 2002.

[16] H. Reza and E. Grant, *Using Architectural Modeling for Integration Testing,* International Conference on Software Engineering Research and Practice (SERP'05), Las Vegas, June 2005.

[17] G. Holzmann, *The SPIN MODEL CHECHER; Primer and Reference Manual*, Addison Wesley, 2004.

[18] D. Carson. Eclipse Distilled. Addison Wesley, 2005.

[19] E. Clarke, O Grumberg, and D. Peled. Model Checking. The MIT Press, 1999.

[20] Unified Modeling Language: Version 2.0 Infrastructure. Published by Object Management Group (OMG), 2005.

[21] Jean Bézivin, *From Object Composition to Model Transformation with the MDA*, Proceedings of TOOLS'USA, Volume IEEE TOOLS-39, Santa Barbara, August 2001.

[22] Shane Sendall and Wojtek Kozaczynski, *Model Transformation: The Heart and Soul of Model-Driven Software Development*, IEEE Software, September/October 2003 (Vol. 20, No. 5) ISSN: 0740-7459.

[23] Conrad Bock, *Composition Model*, Journal of Object Technology, vol. 3, No. 10, November – December 2004, pp. 47-73.

[24] M. M. Kandé, V. Crettaz, A. Strohmeier, and S. Sendall, *Bridging the Gap between IEEE 1471, Architecture Description Language and UML*, Journal of Software and Systems Modeling, Springer Berlin / Heidelberg, vol.1, no. 2, pp. 113-12, December 2002.