Research Proposal : Strategy for Platform Independent Testing

Anders Eriksson

March 27, 2012

Abstract

This work addresses problems associated with software testing in a Model Driven Development (MDD) environment. Today, it is possible to create platform independent models that can be executed and therefore, dynamically tested. However, when developing safety-critical software systems there is a requirement to show that the set of test cases covers the structure of the implementation. Since the structure of the implementation might vary depending on e.g., compiler and target language, this is normally done by transforming the design model to code, which is compiled and executed by tests until full coverage of the code structure is reached. The problem with such approach is that testing becomes platform dependent. Moving the system from one platform to another becomes time-consuming since the test activities to a large extent must start again for the new platform. To meet the goals of MDD, we need methods that allow us to perform structural coverage analysis on platform independent models in a way that covers as much as possible of the the structure of any implementation. Moreover, such method must enable us to trace specific test artifacts between the platform independent model and the generated code. Without such trace a complete analysis must be done at code level and much of the advantage of MDD is lost.

We propose a framework for structural coverage analysis at a platform independent level. The framework includes: (*i*) functionality for generation of test requirements, (*ii*) creation of structural variants with respect to the translation to code, and (*iii*) traceability between test artifacts at different design levels. The proposed framework uses a separate representation for structural constructs involved in coverage criteria for software in safety-critical systems. The representation makes it possible to create variants of structural constructs already at the top design level. These variants represent potential differences in the structure at lower design levels, e.g., target language or executable object code. Test requirements are then generated for all variants, thus covering the structure of different implementations. Test suites created to satisfy these test requirements are therefore, robust to different implementations.

Contents

List of Figures

1 Introduction

Model Driven Development (MDD) is used increasingly in the avionics and automotive industry for specifying software in safety-critical systems. The upcoming revision of the guide-lines for flight-critical software in civil aviation DO-178B (RTCA 1992) has taken this into account by specifying additional guidelines regarding model-based development and formal methods as supplemental documents. This means that technology is required that makes it possible to specify system behavior with an underlying precise semantic. A good starting point is the initiative taken by Object Management Group (OMG) with the Model Driven Architecture (MDA) (OMG 2011*c*), which offers a conceptual framework for defining a set of standards in supporting MDD. The most recognized is the Unified Modeling Language (UML) (OMG 2010*b*) with its Action Sematic package, along with several other technologies related to modeling, like Meta Object Facility (OMG 2011*b*) and XML Metadata Interchange (OMG 2011*d*).

The focus in MDD today is on specifying systems and performing systems verification[1] through model simulation. The verification process in MDD today deals with coverage analysis regarding the functionality rather than the structure. Moreover, MDD is not dealing with model coverage analysis that can be traced all the way down to the executable object code (EOC). This means that the MDD verification process has not yet taken advantage of these new standards that enforce the usage of model transformations of specifications to artifacts, e.g., source code, documents etc., representing different aspects of the system.

Cost and time reduction for verification of safety-critical systems can be achieved by using model coverage analysis at the top design level. This require however, that we establish a trace between artifacts at the top, i.e, the platform independent model of the behavior, to a level with artifacts produced during the transformations, e.g., the source code. Without such a trace a complete analysis must be done at code level and much of the advantage with MDD is lost. If such trace can be established a cost and time reduction can be achieved in the verification process of software in safety-critical systems.

The rest of the proposal is organized as follows. Section 2 describes standards for safety-critical systems and give an overview of the MDA process. Section 3 describes the problem, objectives and initial approach to solve the

[1]The term verification refers to the testing activity and not where a formal proof is used.

problem. Section 4 presents related work in the area of using design models and model coverage analysis combined with standards for safety-critical system.

2 Background

This chapter presents standards for safety-critical systems and the modified condition/decision coverage criterion used by the described standards. This chapter also describes the development process of model driven architecture. At the end of the chapter the relationship between model and code structural coverage are discussed.

2.1 Standards for Safety-Critical Systems

There are several standards and guidelines for software in safety-critical systems. DO-178B is a guideline for avionic systems (RTCA 1992). This document was developed by the commercial avionics industry to establish software guidelines for software developers in avionics. DO-178B describes objectives for software life-cycle processes, activities and design considerations for achieving those objectives, and technics to measure wheter the objectives have been satisfied. One of the main objectives in the guideline is that the functionality of the system under test must be covered by requirements and vice versa. Five safety levels are defined in DO-178B; Level A (highest) to Level E (lowest). The new revision DO-178C aims to clarify areas of misconceptions, while addressing advances in avionics software development and testing. Technical supplemental documents are added to the core of the revised guideline, covering formal methods, model based development, object-oriented technology and tool qualification. Another standard for safety-critical system is ISO 26262 (International Organization for Standardization 2009), which is used in the automotive domain. ISO 26262 is based on IEC 61508 (International Electrotechnical Commission 1999), which is a standard for the development process of safety related components and was originally designed only for hardware. Recently IEC 61508 has been applied to software components as well. Similar to the avionics guideline, the automotive related standard defines automotive safety integrity levels. There are four classes from low safety critical to high safety critical.

The standards and guidelines use coverage criteria to cover the code structure. Different criteria are assigned at different safety critical levels. Both DO-178B and ISO 26262 use a coverage criterion called modified condition-decision coverage (Chilenski & Miller 1994) to cover the code structure in software at the highest safety critical level.

2.2 Modified Condition/Decsison Coverage

Model-based testing can be used in several ways. The authors of the book *Introduction to software testing* (Ammann & Offutt 2008), using models to abstract different kind of coverage criteria. Each coverage criteria are described in a theoretical way and later used in examples to demonstrate their usage. The examples describes how these models can be obtained from various software artifacts. The kind of coverage criteria described in the book are:

1. Graph Coverage
2. Logic Coverage
3. Input Space Partitioning
4. Syntax-Based Testing

MCDC is a logic coverage criterion that covers the structure of predicates. The purpose with MCDC is to complement the requirements-based testing and it is often used as an exit criterion for requirements-based testing. One way MCDC complements requirements-based testing is to provide demonstration of absence of unintended functions. The structural coverage analysis provides a way to confirm that the requirements-based tests exercised the code structure.

The MCDC criterion definition:

- **Condition** - A Boolean expression containing no Boolean operators except for the unary operator.
- **Decision** - A Boolean expression composed of conditions and zero or more Boolean operators. A decision without a Boolean operator is a condition. If a condition appears more than once in a decision, each occurrence is a distinct condition.
- **Modified Condition/Decision Coverage** - Each condition in a Boolean expression used by a decision independently affects that decision's outcome. This can be done by:
  - varying just that condition while holding fixed all other possible conditions (known as unique cause MCDC or RACC (Ammann & Offutt 2008))

- applying principles of Boolean logic to assure that no other condition influences the outcome, even though more than one condition in the decision may change value (known as masking MCDC or CACC (Ammann & Offutt 2008))

To achieve 100% MCDC coverage given a decision with n conditions the minimal number of test cases are n+1 and the maximal number are 2n.

Table 1 shows a truth table for the conditions and decision for the following programming construct. Note a decision need not be a branch point:

`Z := A or (B and C); where A, B, C and Z are boolean variables.`

Entry	A	B	C	(A or (B and C))
0	F	F	F	F
1	F	F	T	F
2	F	T	F	F
3	F	T	T	T
4	T	F	F	T
5	T	F	T	T
6	T	T	F	T
7	T	T	T	T

Table 1: Truth table for decision (A or (B and C)). T and F stands for True and False

2.2.1 Unique Cause MCDC

The entries from Tabel 1 that can show how each condition independently affects the decision are:

For A - {0,4}, {1,5}, {2,6}
For B - {1,3}
For C - {2,3}

Two minimal sets of entries that show that every condition independently affects the decision are {1,2,3,5} or {1,2,3,6}. {0,1,2,3,4} would also work, but of course is not minimal.

2.2.2 Masking MCDC

To show A's independence, *(B and C)* must be false. There are tree ways that *(B and C)* can be false, and any of them can be used to show A's independent affect on the decision outcome. The entries from Tabel 1 that can show how each condition independently affects the decision are:

For A - {0,4},{0,5},{0,6},{1,4},{1,5},{1,6},{2,4},{2,5},{2,6}
For B - {1,3}
For C - {2,3}

The three minimal sets of entries that show that every condition independently affects the decision are {1,2,3,4}, {1,2,3,5} or {1,2,3,6}

2.2.3 Short Circuit Logic and MCDC

When using a standard **and** or **or** operator, both of the operands in the expression are typically evaluated. For some programming languages, the order of evaluation is defined by the language, while for others, it is left as a compiler-dependent decision. Some programming languages also provide short-circuit control forms. In Ada, the short-circuit control forms, **and then** and **or else**, produce the same results as the logical operators **and** and **or** for boolean types, except that the left operand is always evaluated first. The right operand is only evaluated if its value is needed to determine the result of the expression. The && and the || operators in C and C++ are similar.

For Unique Cause MCDC with short-circuit logic, the entries from Tabel 1 that can show how each condition independently affects the decision are:

For A - {0,4}, {1,5}, {2,6}
For B - {1,3}
For C - {2,3}

Two minimal sets of entries that show that every condition independently affects the decision are {1,2,3,5} or {1,2,3,6}.

2.2.4 Expression folding and MCDC

Expression folding eliminates all usage of local variables in expressions. The test sets that achieve coverage over the non-folded expression will not always achieve coverage over the folded expression.

```
Non-folded expression:
Y := B and C; where B, C and Y are boolean variables.
Z := A or Y; where A and Z are boolean variables.
```

One possible test set, *Set1* {(ABC)}: {(TTT),(FFT),(FTT),(TTF)}

```
Folded expression:
Z := A or (B and C); where A, B, C and Z are boolean variables.
```

One possible test set, *Set2* {(ABC)}: {(FFT),(FTF),(FTT),(TFT)}

Both the expressions above have the same outcome for the variable Z. But the test sets to achieve MCDC is not the same. While the test set *Set2* achieves coverage for both the implementations, test set *Set1* only achieves MCDC for the non-folded implementation. There is a need for a *robustness measurement*, so different test sets can be compared and analyzed against different implementations. This issue will be very important in the future where model-based development is used for specifying safety-critical systems.

2.3 Model Driven Architecture

In MDA, a platform-independent model (PIM) is initially expressed in a platform-independent modeling language, such as Executable UML (Mellor & Balcer 2002). The Executable UML is a profile of the UML that defines execution semantics for the selected subset of UML. The subset is *computationally complete*, which means that an Executable UML model can be directly executed. Specifications in Executable UML consist of class diagrams, state diagrams and action semantics to describe the operation behavior. Using a model compiler, the specification is transformed into a platform-specific model (PSM) or any target language appropriate for the target platform. A PSM can be specified in Analysis & Architecture Design Language (AADL) (Feiler, Lewis & Vestal 2006), which was developed to meet special needs for expressing performance-critical aspects in real-time systems such as timing requirements, time and space partitioning. These capabilities make it possible for the system designer to analyze the systems for things like system schedulability, sizing and timing. The results from the analysis can be used for evaluating different architectural tradeoffs and changes.

Figure 1 shows the steps in an MDA process where a PIM is specified in an Executable UML model and the PSM is represented by source code in C++. BridgePoint[2] and iUML[3] are tools suited for the MDA process

[2]http://www.mentor.com/products/sm/model_development/bridgepoint

[3]http://www.kc.com/PRODUCTS/iuml/index.php

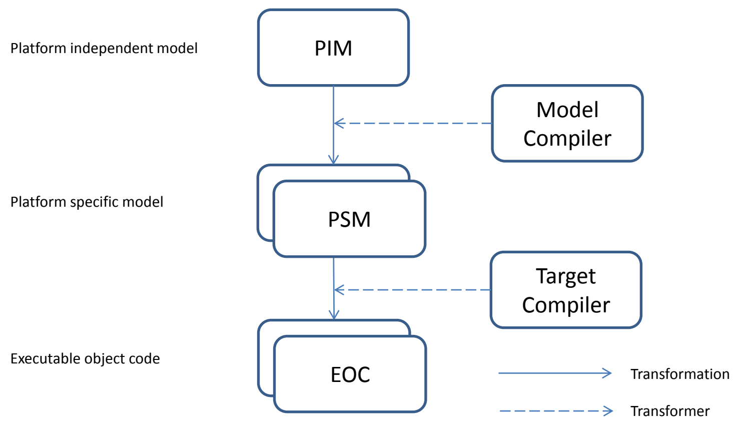

Figure 1: MDA process.

by supporting Executable UML and model transformations. *MDA Distilled* (Mellor, Scott, Uhl & Weise 2004) gives a detailed introduction to MDA and related technologies. The authors argue that for MDA to succeed more standards will be required as MDA matures. The previous mentioned tools BridgePoint and iUML have the disadvantage that their meta-models and action languages are proprietary and not publicly available. Differences in syntax and semantics makes it hard to interchange, for example PIM models between different UML tools.

Two new OMG standards, fUML (OMG 2011*a*) and Alf (OMG 2010*a*), will play a central role in mitigating these differences. This will make it possible to achieve the goal of MDA since models can be transformed and interchanged between different tools supporting these new standards. The standard fUML provides a simplified subset of UML Action Semantics package (abstract syntax) for creating executable UML models, and also defines execution semantics via an Execution Model. The Execution Model itself is modeled in a subset of fUML. The circularity is broken by the separate specification of a base semantics (bUML) for the subset of fUML expressed in axioms of first order logic. The action language Alf is designed to overcome difficulties of creating fUML executable models. The difficulties concerns the UML primitives intended for execution, which are too low level to be useful for large models.

2.4 Model and Code Structural Coverage

Baresel, Conrad, Sadeghipour & Wegener (December 2003) describe experience from using model coverage metrics in the automotive industry when using Simulink/Stateflow. The conclusion of the experiment is that there are comparable model and code coverage measurements but they heavily depend on how the design model was transformed into code. The same conclusion was made by Heimdahl, Whalen, Rajan & Staats (2008) where an empirical study was performed to measure the effect of program and model structures on MCDC test adequacy coverage. The study used six realistic systems from the civil avionics domain. All systems were specified in Simulink. The purpose of the study was to measure structural coverage on two versions of the implementation, with and without expression folding (i.e., inlined and non-inlined). The authors first generated test suites that satisfy MCDC over the non-inlined implementation and then ran the same test suites over the inlined version and measured MCDC coverage. The result was an average reduction of 29.5% in MCDC achieved over the inlined implementation with the generated test suites. The reason for the loss in MCDC coverage is that the non-inlined version does not take the effect of masking into account but the inlined version does. The authors believe there is a serious need for coverage criteria that take masking into consideration irrespective of implementation structure, or a canonical way of structuring code so that condition masking is revealed when measuring coverage using existing coverage criteria.

3 Problem Description

Most of the modeling environments supporting MDA today focus on functional testing, enabling the verification of design models against their specifications. This is normally done via model simulation. When developing safety-critical software systems there is however, as stated in Section 2.1, a requirement to show that the set of test cases covers the structure of the implementation. Normally this is done by transforming the design model to code, which is compiled and executed to collect the data used for structural coverage analysis. If the structural code coverage criteria are not met at the PSM level, additional test cases should be created at the PIM level. However, as coverage is analyzed at the PSM level, iterations between the PIM level and the PSM level are required. These iterations can be very time consuming. Moreover, designing tests at the PIM level to cover the structure at the PSM level requires knowledge about the PIM to PSM transformation to be able to create test cases at the PIM level that will reach specific parts of the PSM. This violates one of the major benefits of MDA, which is the separation of abstraction layers.

Because of this, there are two choices. The first choice is to create additional tests at the PSM level and run them directly. This makes testing time consuming because of the iterations between the PIM and the PSM levels. Any changes to the PIM will be reflected in the PSM, which can lead to manual changes to the additional test cases. This is typically how the problem is handled today. The second choice is to address the structural coverage analysis already at the PIM level in which case testing, to a large extent, can take place at the PIM level without the time consuming iterations between the PIM and the PSM. When the behavior of the model is verified to satisfaction by the structural coverage analysis, the model is translated to a PSM and tested by the same tests again. This approach conforms to MDA and would significantly reduce the time spent on iterations between the PIM and the PSM.

Therefore, we want to raise the level of abstraction for the structural coverage analysis to be performed at the same level as the design model verification, i.e., the PIM level. Tool vendors, for example Mathworks with Simulink/Stateflow[4], have started to apply structural coverage analysis to their domain-specific design models. Simulink/Stateflow is used in control design in domains such as avionics and automotive and is known for its

[4]http://www.mathworks.com/products/simulink

model simulation capability. There is no tool with the same structural coverage analysis functionality for design models specified in Executable UML or fUML. The result from Heimdahl et al. (2008) shows a need for a representation for structural coverage analysis that is independent of the structures used at each design level. With an independent representation it would be possible to create robust PIM test suites that can be used on many different PSMs with the same or almost the same structural coverage.

Problem Statement: To our knowledge, there are currently no methods that can perform structural code coverage analysis on a platform independent design level while managing traceability of the covered structural constructs between the different design levels.

3.1 Research Objectives

To solve the problem, this research must address the following:

- **Generation of test requirements:** Structural code coverage criteria required for safety-critical software should be satisfied on a high design level.

  O1. Select a representation that can capture the properties needed for structural coverage criteria, and adopt this representation to represent an abstract model of structural constructs needed for structural coverage criteria.

  O2. Propose an algorithm that generates test requirements according to selected structural coverage criteria.

- **Robustness with respect to structure:** The method should be independent of model and target language. It should be possible to create variants of the structure within a given design level. Test requirements can then be created for these variants and compared to each other to identify a set that is *robust* with respect to the two variants.

  O3. Select transformations rules that can be applied to the structural constructs to create variants. For example, a rule for expression folding could be to substitute away all local variables and have the expression only depend on formal parameters and global variables (section 2.2.4).

O4. Propose an algorithm that takes two input sets of test requirements and calculates the difference with respect to coverage between these two sets.

- **Traceability of test requirements between artifacts:** Traceability between artifacts makes it possible to trace individual test requirements for artifacts at different design levels and variants, i.e. traceability is a prerequisite for determining the level of coverage preservation.

  O5. Select a representation that can formalize associations between artifacts. These associations should be used to create traceability between structural constructs. Via the associations sets of test requirements for different design levels can be obtained and compared.

The method should have an input format that can capture the requirements mentioned above. The transformation from each of the design levels PIM, PSM and EOC to the input format of the method is not part of the method itself. There are techniques already today to create transformations from the PIM, PSM and EOC.

3.2 Initial Approach

The initial approach is to collect the functionality needed to solve the problem stated above in a framework for structural coverage analysis.

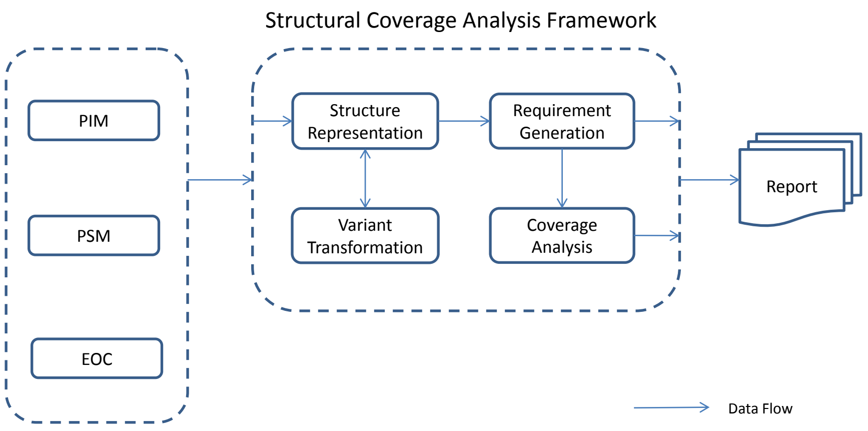

Figure 2: Structural coverage analysis framework.

Figure 2 shows the main components contained in the framework and the information flow between them. The core functionality of the framework is to create test requirements for a specified coverage criteria and be able to compare test requirements from the design levels PIM, PSM and EOC. The creation and comparison of test requirements are done without the need of running any test cases.

The following describes the initial approach **A1-A5** to meet the defined objectives **O1-O5**:

A1. We will start with a literature study about different ways to represent structural constructs and how these can be analyzed formally. A possible representation can be the Ordered Binary Decision Diagram (OBDD) (Bryant 1992) which has the properties of being canonical and extendable. One of the major uses of OBDDs is in formal verification. Another possibility is to use an existing abstract syntax tree (AST) for the C-language and annotate that with extra information if needed.

The expected result is a selected representation that captures structural constructs in a platform independent way and makes it possible to apply further transformations.

A2. We will select criteria from safety-critical standards and generate test requirements corresponding to these criteria using information stored in the selected representation.

The expected result is an algorithm that generates test requirements that satisfies the selected coverage criterion.

A3. We will put requirements on approach **A**2 to make it possible to handle test requirements in sets where set functions like intersection and union can be applied.

The expected result is an algorithm that makes it possible to compare sets of test requirements.

A4. We will start with a literature study in compiler technology about transformations normally done to structural constructs. Also different aspects of target languages should be considered, i.e., short circuit evaluation as described in section 2.2.3. There are several freely available compiler frameworks that are widely used in the industry, like GCC[5]

[5]http://gcc.gnu.org

and LLVM[6], so one of them may be used. Both use Static Single Assignment (SSA) (Cytron, Ferrante, Rosen, Wegman & Zadeck 1991) as their intermediate representation, which is suitable for transforming structural constructs, so they are natural candidates for the study.

The expected result is set of transformation rules that can be automatically applied to the selected representation.

A5. We will start by modeling the traceability functionality as a database schema in Executable UML. After that at least two choices are possible: (i) we use a freely available database like SQLite[7], or (ii) add operational behavior to the model and use a model compiler that generates an application. The first choice is the default for now because the search functionality is already supported.

The expected result is a method that makes it possible to create associations between artifacts in the framework and searching for test requirements within a design level or between design levels.

3.3 Evaluation

The approach for the evaluation is to study the *Software Verification Tools Assessment Study* (SVTAS) (Santhanam, Chilenski, Waldrop, Leavitt & Hayhurst 2007). The aim of the SVTAS was to investigate criteria to improve the process of evaluating structural coverage analysis tools for use on projects intended to comply with the guideline DO-178B. The authors proposed a test suite as an approach to increase objectivity and uniformity in the application of the tool qualification criteria. A prototype test suite was constructed and run on three different structural coverage analysis tools to evaluate the efficacy of the test suite approach. The prototype test suite identified anomalies in each of the three coverage analysis tools, demonstrating the potential for a test suite to help evaluate a tools compatibility with the DO-178B requirements.

By designing a test suite according to the SVTAS from where test procedures are chosen. Our proposed framework for structural coverage analysis can be evaluated by measuring the ability to create necessary test requirements as stated in the SVTAS for the chosen test procedures. The defined objectives **O1-O5** will all be evaluated.

[6]http://llvm.org

[7]http://www.sqlite.org

4 Related Work

Little work has been done in the area of analyzing fUML models. The work done by Lazar, Lazar, Parv, Motogna & Czibula (2009) will make the analysis of fUML models possible, by the defined framework ComDeValCo (Component Definition, Validation, and Composition). The framework includes functionality for specifying components structure and behavior. The behavior is specified by using an action language defined the authors.

Later, the framework was extended by Lazar (2011) to use the action language Alf (OMG 2010*a*). The authors conclude that they can both simulate and test the fUML models without having to generate code. This framework can be combined with our proposed framework to enable structural coverage analysis of fUML models. There are currently no existing tool, to our knowledge, capable of that.

A more restricted way of handling structural coverage analysis compared to our method is described in Kirner (2009). The author focuses on ensuring that the structural code coverage achieved at a higher program representation level is preserved during transformation down to lower program representations. To guarantee preservation of structural coverage, the author defined formal properties that have to be fulfilled during the transformation between different program representations. The defined formal properties has been used in Kirner & Haas (2009) to automatically create coverage profiles that can extend compilers with the feature of preserving any given code coverage criteria by enabling only those code optimizations that preserve it.

Our method work in the opposite by adding test requirements at the highest program representation derived from lower program representations. Both methods share the same possible code transformations, this makes the authors analysis how different code transformations influence the structural coverage preservations important.

Our method has similarities to the approach behind *testability transformation* described by Harman, Hu, Hierons, Wegener, Sthamer, Baresel & Roper (2004), which is a source to source transformation. The transformed program is used by a test-data generator to improve its ability to generate test data for the original program. The source to source transformation serves the same purpose as our creation of variants of the structural constructs. The work done by Baresel et al. (December 2003) is another motivation for our method. The authors analyzed, by an empirical study, the relationship between achieved model coverage and the resulting code coverage. The authors found that the code coverage heavily depends on how the model is transformed into code. This motivates the step in our method where variants can be created to improve the set of test requirements at model level.

References

Ammann, P. & Offutt, J. (2008), *Introduction to software testing*, New York: Cambridge University Press, ISBN 978-0-521-88038-1.

Baresel, A., Conrad, M., Sadeghipour, S. & Wegener, J. (December 2003), The interplay between model coverage and code coverage, *in* 'Proceedings of the 11th european iternational conference on Software Testing, Analysis and Review', EuroSTAR '03.

Bryant, R. E. (1992), 'Symbolic boolean manipulation with ordered binary-decision diagrams', *ACM Comput. Surv.* **24**, 293–318.

Chilenski, J. & Miller, S. (1994), 'Applicability of modified condition/decision coverage to software testing', *Software Engineering Journal* **9**(5), 193 – 200.

Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N. & Zadeck, F. K. (1991), 'Efficiently computing static single assignment form and the control dependence graph', *ACM Trans. Program. Lang. Syst.* **13**, 451–490.

Feiler, P. H., Lewis, B. A. & Vestal, S. (2006), The SAE Architecture Analysis & Design Language (AADL) a standard for engineering performance critical systems, *in* 'Computer Aided Control System Design, 2006 IEEE International Conference on Control Applications, 2006 IEEE International Symposium on Intelligent Control, 2006 IEEE', pp. 1206 –1211.

Harman, M., Hu, L., Hierons, R., Wegener, J., Sthamer, H., Baresel, A. & Roper, M. (2004), 'Testability transformation', *Software Engineering, IEEE Transactions on* **30**(1), 3 – 16.

Heimdahl, M., Whalen, M., Rajan, A. & Staats, M. (2008), On MC/DC and implementation structure: An empirical study, *in* 'Digital Avionics Systems Conference, 2008. DASC 2008. IEEE/AIAA 27th', pp. 5.B.3–1 –5.B.3–13.

International Electrotechnical Commission (1999), IEC 61508: Functional Safety of Electrical/ Electronic/ Programmable Safety-Related Systems.

International Organization for Standardization (2009), ISO 26262: Road vehicles - Functional safety.

Kirner, R. (2009), 'Towards preserving model coverage and structural code coverage', *EURASIP J. Embedded Syst.* **2009**, 6:1–6:16.

Kirner, R. & Haas, W. (2009), 'Automatic Calculation of Coverage Profiles for Coverage-based Testing'. Vienna University of Technology, Institute of Computer Engineering, Vienna, Austria, raimund@vmars.tuwien.ac.at.

Lazar, C.-L. (2011), 'Integrating Alf Editor with Eclipse UML Editiors', *Studia Univ. BabesBolyai, Informatica* **LVI**(3), 27 – 32.

Lazar, C.-L., Lazar, I., Parv, B., Motogna, S. & Czibula, I.-G. (2009), Using a fUML Action Language to Construct UML Models, *in* 'Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), 2009 11th International Symposium on', pp. 93 –101.

Mellor, S. & Balcer, M. (2002), *Executable UML: A Foundation for Model Driven Architecture*, Boston: Addison Wesley, ISBN 0-201-74804-5.

Mellor, S. J., Scott, K., Uhl, A. & Weise, D. (2004), *MDA Distilled: Priciples of Model-Driven Architecture*, Boston: Addison Wesley, ISBN 0-201-78891-8.

OMG (2010*a*), 'Action Language for Foundational UML (ALF), version 1.0 - beta1'. Retrieved September 14, 2011 http://www.omg.org/spec/ALF/1.0/Beta1/.

OMG (2010*b*), 'Unified Modeling Language (UML), Infrastructure, version 2.3'. Retrieved September 14, 2011 from http://www.omg.org/spec/UML/2.3.

OMG (2011*a*), 'Foundational Subset of Executable UML (FUML) version 1.0'. Retrieved September 14, 2011. http://www.omg.org/spec/FUML/1.0/.

OMG (2011*b*), 'Meta Object Facility (MOF) Core Specification Version 2.4'. Retrieved September 14, 2011. http://www.omg.org/spec/MOF/2.4/Beta2/.

OMG (2011*c*), 'Model Driven Architecture (MDA)'. Retrieved September 14, 2011. http://www.omg.org/mda/specs.htm.

OMG (2011*d*), 'XML Metadata Interchange (XMI) version 2.4 - beta2'. Retrieved September 14, 2011. http://www.omg.org/spec/XMI/2.4/Beta2/.

RTCA (1992), RTCA Inc. DO-178B: Software Considerations In Airborne Systems and Equipment Certification. Requirements and Technical Concepts for Aviation.

Santhanam, V., Chilenski, J. J., Waldrop, R., Leavitt, T. & Hayhurst, K. J. (2007), 'Software Verification Tools Assessment Study', Report No. DOT/FAA/AR/AR-06/54. Retrieved September 27, 2011 from http://actlibrary.tc.faa.gov.