Security Requirements Verification for Existing Systems with Model Checking Technique and UML

Saeko Matsuura[1], Shinpei Ogata[2] and Yoshitaka Aoki[3]
[1]*Graduate School of Engineering and Science, Shibaura Institute of Technology, Saitma, Japan*
[2]*Graduate School of Science and Technology, Shinsyu University, Nagano, Japan*
[3]*Nihon Unisys Ltd., Tokyo, Japan*

Keywords: UML, Security Requirements, Source Code Verification, Model Checking, Reverse Engineering.

Abstract: In software development, when making migration or specification changes to an existing system, it is important to verify that the new source code meets the original specifications. We propose an effective use of model checking techniques and a supporting tool that allows non-specialized developers to easily verify specification conformance. In this study, we verify security requirements for an ongoing learning management system that has insufficient specification documentation and discuss the applications and challenges for developing the model checking technology.

1 INTRODUCTION

The model checking technique is regarded as an effective method for improving reliability during the early stages of software development. Model checking tools use temporal logic to model a system as a network of automata extended with integer variables, structured data types, user defined functions, and channel synchronization. A system model and query expressions can be defined based on these structures and be used to specify which properties need to be checked. When the specified properties are not satisfied, the tool provides counterexamples to demonstrate how these properties cannot be satisfied. The simulator then helps detect the causes for these defects by tracing the processes in which the counterexamples occur.

Model checking techniques automatically verify a model by exhaustively checking all paths in the model to detect properties that developers often overlook. However, developers typically find it difficult to define an appropriate model and formulas for a given system, because items in the model should be used to define paths and state formulas.

We have been studying a method that allows non-specialized developers to reap the benefits of automatic and exhaustive verification without directly operating a model checking tool.

Figure 1 shows the relationship between the model checking tool, the input data used to derive a system model, and query expressions that are inputted directly. Users directly input a target document and a list of requirements to be satisfied. An input document can be a requirements specification document written in UML (OMG), or the source code of an existing system.

The problems indicated by the red stars in Figure 1 must be solved in order to effectively use the model checking tool without direct operation.

This paper presents a method of verifying security requirements for an existing system based on two different proposed approaches to verification in the early and final stages of software development.

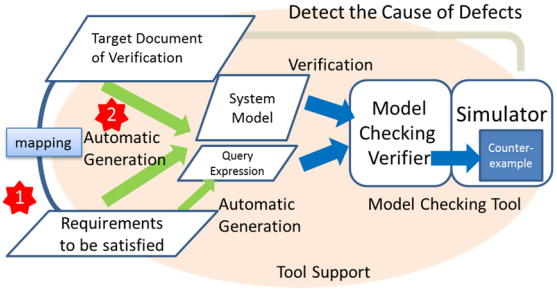

Figure 1: Basic principles of model checking techniques.

The rest of this paper is organized in the following manner. Section 2 briefly summarizes our two approaches and describes a method for solving the two marked problems in Figure 1. Section 3 describes the process of source code verification

Matsuura S., Ogata S. and Aoki Y.
Security Requirements Verification for Existing Systems with Model Checking Technique and UML.
DOI: 10.5220/0006267105290535
In *Proceedings of the 5th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2017)*, pages 529-535
ISBN: 978-989-758-210-3

used for the existing system. Finally, we discuss our results, conclusions, and future research possibilities.

2 VERIFICATION METHOD

2.1 Requirements Specification Verification

Figure 2 presents an overview of the proposed method (Aoki et al., 2013 and September 2014) for verifying Security Function Policies (SFPs) of UML requirements analysis models (RA Model) with the model checking tool UPPAAL (UPPAAL).

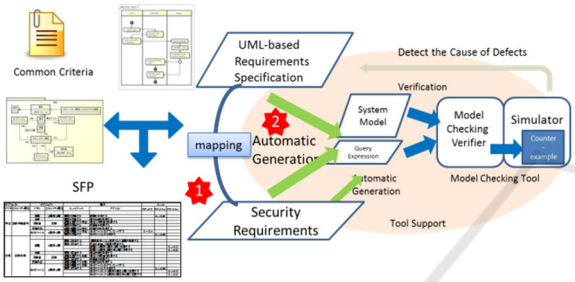

Figure 2: Method for requirements specification.

In this case, the target document is an RA Model using the UML-based requirements specification developed in (Ogata et al., 2008) and based on a use case analysis, which is known to be an effective method for clarifying functional requirements. This model defines a use case in the form of an activity diagram that consists of several action flows as well as object nodes with a class classifier.

Furthermore, the requirements to be satisfied are SFPs. In the proposed security requirements analysis method (Aoki et al., September 2014), we define security requirements as SFPs by relating them to RA Models, according to the Common Criteria (Common Criteria) guide. The Common Criteria for Information Technology Security Evaluation is an international standard (ISO/IEC 15408) for computer security certification.

The first problem can be solved by using a mapping rule between a model for SFPs and RA models, as shown in Figure 3. The mapping rule combines the SFPs with data and actions from the UML RA model, defined through activity diagrams and a class diagram. A rule in SFPs is defined by the relationship between a subject, an operation, and a target object with the subject performing the operation on the target. An RA model consists of actors, use cases (activity diagrams), classes, and actions in the activity diagram. Figure 3 illustrates the mapping rules; a subject corresponds to an actor, an object corresponds to a class, and an operation corresponds to an action in a use case. Some new security attributes need to be defined for both the assets needing protection from malicious users and the subjects who carry out controlled operations. This is because SFPs control action flows through rules based on security attributes. Table 1 shows that SFPs can be defined by actions and data in the target system, according to the relationship between the two. Using these added security attributes from an object, the policy was defined in a state machine diagram.

This correspondence allows the target UML-based requirements specification to define properties that need to be satisfied.

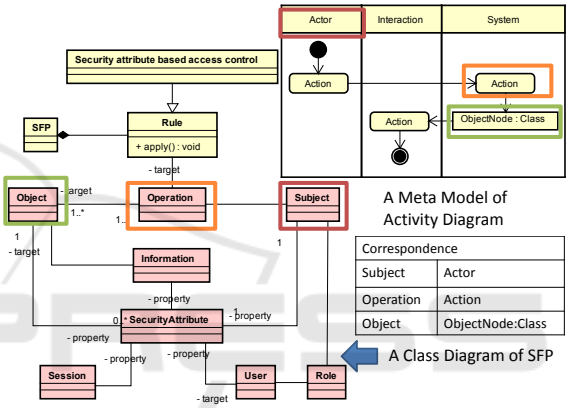

Figure 3: Mapping rule between SFPs and RA model.

To solve the second problem, we developed the automatic translation tool *UML2UPPAAL* (Aoki et al., 2013) to translate from an RA model to the required system model. The details of this tool are described in (Aoki et al., September 2014). The RA models and the added state machine diagrams were translated into a system model that consisted of processes related to each other by the channel synchronization mechanism in UPPAAL. As detailed in Section 3, SFPs can be translated from the logical expressions between attributes of classes into query expressions.

2.2 Source Code Verification

Figure 4 presents an overview of the proposed method (Aoki et al., May 2014, Matsuura, 2014) for verifying defects caused by an infinite loop, or by the user-defined business rules of certain system functions in the source code, with the model checking tool UPPAAL.

In this case, the target document is some source code written in Java. To verify a defect caused by an

infinite loop, the property that must be satisfied is the non-occurrence of a dead lock in the system. In this case, the query expression is independent of the source code, so the first problem discussed does not occur. However, it is somewhat difficult to create a mapping between the user-defined business rules and the source code.

In (Aoki et al., May 2014, Matsuura, 2014), this problem was solved by defining the business rules in a decision table. It is a prerequisite for a tester to extract actions, conditions, and their resulting states from the business rule documents. In this case, the query expression was derived from the decision table.

In the case of source code verification, the second previously discussed problem is a difficult one. This is because a large and complicated source code needs to be well abstracted to avoid an issue known as "state explosion," where the number of states becomes intractably large.

We solved this problem by allowing developers to select appropriate candidate functions based on the previously generated decision table. We also implemented a verification support tool called *Source2UPPAAL* (Aoki et al., May 2014) as an Eclipse plug-in. This tool converts a Java source code control sequence into finite automata in order to detect the cause of defects through the use of UPPAAL.

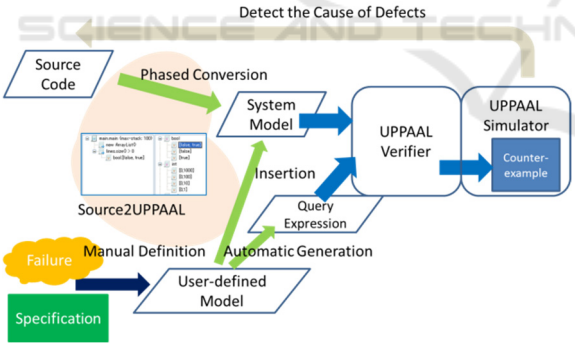

Figure 4: Method for source code verification.

Rather than using the method from the case mentioned in Section 2.1, we use a different technique to more easily and accurately create a mapping. The tester correlates a word in the decision table with a variable or method from the source code and finds an adequate level of abstraction that preserves the related states with the properties to be verified.

Based on the two different approaches proposed for verification at the early and final stages of software development, we present an experiment for verifying security requirements in the source code of the existing system, LUMINOUS.

3 SOURCE CODE VERIFICATION ON AN EXISTING SYSTEM

3.1 Overview of the Existing System

LUMINOUS (LUMINOUS) is a learning management system in our department that enables teachers and students to manage learning materials, reports, questionnaires, etc. We apply our security requirements analysis method to the development of a Bulletin Board System (BBS) for LUMINOUS. There are two kinds of actors, teachers, and students. The BBS has three use cases; a student can post a question, a teacher can answer a question, and both can view topics. Both types of actors can attach a file to a question or answer if necessary and can download an attached file while viewing a topic. A teacher can post anonymous public questions and answers if necessary. Students can read only public questions and answers.

The security requirement for the LUMINOUS BBS is the protection of personal topics, including attached files, without being obstructive to usability.

LUMINOUS is running in our department and has 36,352 steps on the logic side, and 25,973 steps on the user interface side.

3.2 Verification Process

Initially, we create a mapping of RA models and security requirements (outlined in section 2.1) to connect the RA models and source code of the existing system as shown in Figure 5.

Each rule in an SFP is a logical expression relating a use case and some added class attributes. A use case is a function defined by an activity diagram that is typically connected to a function in the source code. In the example of the LUMINOUS BBS, it is also connected to a menu item in the user interface for the system.

All classes can be extracted from the source code as a class diagram. Each class then appears in the activity diagram as an object node.

As a result, we can create a mapping rule between the source code and the properties that need to be satisfied by using identifiers in the source code. To abstract the source code and to avoid state explosion, we focus on a block of statements that

correspond with a use case and update methods of the added attributes in the SFPs.

An SFP rule is expressed through the expected state of its related attributes under a scenario in which several use cases are called in proper order.

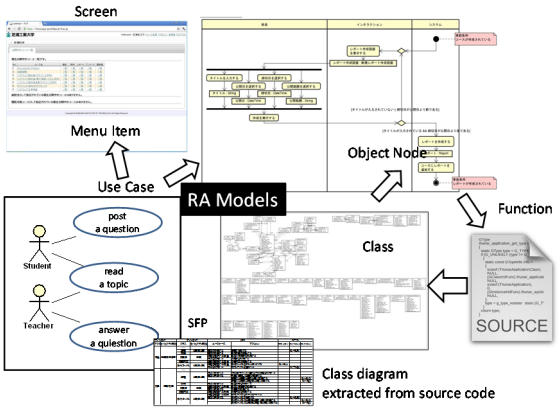

Figure 5: Connection between RA models and source code.

3.3 Reconstructing Requirements Specification

Because there were only a few RA models in LUMINOUS, we reconstructed the RA models in the following manner.

Through direct operation of the system, we created RA models including a use case diagram, activity diagrams for each of the use cases, and a class diagram. Each screen consists of several menu items and links. By focusing on menu items that correspond to target use cases, we can specify a sequence of functions that are available through the links as follows:

- For each menu item on the top screen, define a use case diagram.
- The normal flow of a use case is defined with an activity diagram. Pre and post conditions are then specified and exceptional conditions for every target action are gathered into a guard condition.

To connect these models to the source code, the class name of an object node that appears in the activity diagram translated by referencing the entity class diagram, extracted from the source code, as follows:

- Correlate the object word of an action in the activity diagram with the name of the object node.
- Correlate words in the pre, post, and guard conditions with names in the entity classes.

Furthermore, to analyze the correspondence between a use case and a method in the source code, we define classes corresponding with screens. In this case, each attribute of a class needs to be correlated with an actual label on the screen.

As shown in Figure 5, after these tasks have been completed, we can create RA models that are linked to identifiers in the source code.

3.4 Defining Security Requirements

In LUMINOUS, there are four different user roles: student, main teacher, sub teacher, and teaching assistant. By giving different authority to each role, we can control user access to the functions of LUMINOUS. Some roles have assets that need to be protected from the other roles and from individual students.

Adding certain attributes to the roles or assets allows us to protect against inappropriate access. This security property must be verified.

As mentioned in Section 2.1, we define SFPs against the RA models reconstructed in Section 3.3 based on the Common Criteria as follows:

- Define a scenario that should be verified by combining several use cases according to the pre-conditions of those use cases
  The following is an example scenario that consists of actors and use cases:

  A student posts a question.
  A teacher answers a question.

- According to the mapping rules shown in Figure 3, the SFPs shown in Table 1 were derived from these use cases by extracting all actions and classes from their related activity diagrams.
- Table 1 expresses the SFPs of a BBS that manage questions and answers between a student and a teacher. The class Topic is a data asset that is generated by executing the use case post a question. A Topic is updated by the use case answer a question, which is executed by a teacher. At the time of answering, a teacher can decide whether the Topic is public or private.
- Add security attributes that control a subject and object against data assets. In this case, Topic and Attachment are considered to be assets in the system, and have security attributes.
- Required rules based on the Common Criteria are defined in Table 2.

Table 1: Partial SFPs of LUMINOUS BBS.

Subject		Object		Operation		Rule		
Actor	Securty Attribute	Class	Security Attribute	Use Case	Action	FDP_ACF.1	FMT_MSA.3	FMT_MSA.1
Student	role =Student ID	...						
		Topic	public/private	post a question	create topic		rule B1	
		Date		post a question	get currentDate			
		Contibutor	role	post a question	get contributor			
		Content		read a topi_student	get content			
		Attachment	public/private	read a topi_student	download attachment	rule A		
				post a question	create attachment		rule B2	
Teacher	role =primary Charge	...						
		Topic	public/private	answer a question	select specified topic			
				answer a question	update the topic			
				update topic openness	change attribute of topic(public)			rule C1
				update topic openness	change attribute of topic(private)			rule D1
		Date		answer a question	get currentDate			
		Contributor	role	answer a question	get contributor			
		Content		read atopic_teacher	get content			
		Attachment	public/private	read atopic_teacher	download attachment			
				answer a question	create attachment		rule B3	
				update attachment openness	change attribute of attachment(public)			rule C2
				update attachment openness	change attribute of attachment(privatec)			rule D2

Table 2: Access Control Rules.

rule A	Prior to executing the action, attachment.security_attrubute == public||contributor.role ==student_id
rule B1	After executing the action, topic.security_attribute == private
rule B2	After executing the action, attachment.security_attribute == private
rule B3	After executing the action, (topic.security_attribute == public implies attachment.security_attribute == public|| attachment.security_attribute == private) && (topic/security_attribute == private implies attachment/security_attribute == private)
rule C1	Prior to executing the action, topic.security_attribute == private implies that after executing the action, topic.security_attribute == public
rule C2	Prior to executing the action, attachment.security_attribute == private implies that after executing the action, attachment.security_attribute == public
rule D1	Prior to executing the action, topic.security_attribute == public implies that after executing the action, topic.security_attribute == private
rule D2	Prior to executing the action, attachment.security_attribute == public implies that after executing the action, attachment.security_attribute == private

In order to connect to variables in the source code, the extracted entity class diagram identifies added security attributes.

Furthermore, state transitions of an object related to the security attribute are defined in a state machine diagram.

3.5 Source Code Analysis for Mapping Specification to Source Code

LUMINOUS is an ASP.NET application. Up to this point, we have obtained the entity classes from the source code and defined the UI (User Interface) classes from screen images in the live system.

However, several troublesome tasks must be performed in order to connect this information with the real identifiers in the source code. To extract entity class names and attributes, we need to analyze an *edmx* file that expresses an entity data model in the Entity Framework of C#. We must also analyze a *cs* file to extract UI class names and methods.

We perform static analysis on these files to retrieve the five following pieces of data as a correspondence table between UI components and identifiers in the source code.

1) Entity class name
2) Attribute names of the entity class
3) UI class name
4) Method names of the UI class
5) Identifiers for UI components such as a button or checkbox

A tester manually selects items that will be used to verify a scenario.

Up to this point, the correspondence between items in the activity diagram (use case) and identifiers in the source code is still not clear. We specify the relationship between items in the activity diagram (use case) and identifiers in the source code as follows:

- A user operation typically corresponds with a UI component such as a button, checkbox, or an input tag in the web application. Based on the identity of the component that submits a method call to the transition destination screen, we can derive the name of the called method.
- We define the basic data operations of Create, Read, Update, and Delete in an activity diagram because LUMINOUS uses a database system. We can then create a correspondence between these CRUD actions in the activity diagram and the API (Application Programming Interface) of the database system.

3.6 Generating a System Model and Query Expressions

A system model and query expressions were generated from the previously defined scenario, and the table mentioned in Section 3.4, by using the data mentioned in Section 3.5 with the Source2UPPAAL tool.

As shown in Figure 6, a system model consists of the following processes that are connected to the channel synchronization mechanism in UPPAAL:

- The verification scenario was translated into a UPPAAL model that combines use cases from the scenario in the proper order.
- Each use case model defined in the activity diagram was translated into a UPPAAL model that correctly maps the control flow of actions.
- A state machine diagram that defines a property of a security attribute was translated into a UPPAAL model. This model can connect to use case models via an API model for updating security attributes using the channel synchronization mechanism.

In Figure 6, an arrow denotes channel synchronization between two processes.

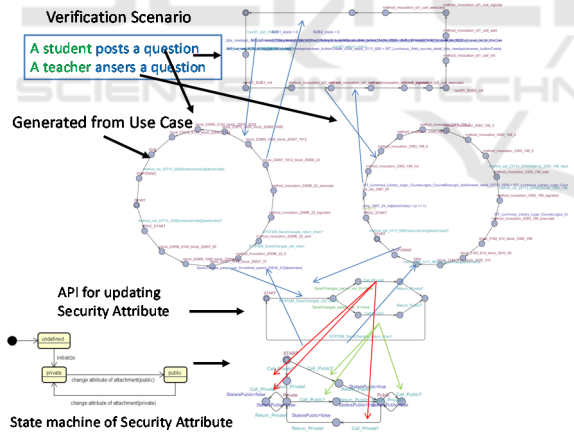

Figure 6: System model generated in UPPAAL.

We generate a query expression from the rules shown in Table 2. Figure 7 shows the query expressions generated for the scenario in Figure 6. It is expressed in the UPPAAL model through variables, location names, etc., that are generated by the support tool *Source2UPPAAL*.

The scenario is as follows:

- A student posts a question. At the time of posting, the Topic is private. Next, a teacher answers the question. At this time, the Topic can be set to public or private at the teacher's discretion.

The query indicates that after answering the question, if a teacher sets the topic *Topic* private, then the *Topic* will have never been given public access.

02	A teacher answers a question.	At the end of answering, if a teacher set Topic private, then Topic is private	A[] not (SYSTEM_SIT_Luminous_Web_course_detail_bbs_ newtopicanswer_buttonCreate_Click(0,0).END && SaveChanges_param_var_0==false && SYSTEM_State_Private_Public.Public)

Figure 7: Query expression generated based on the scenario.

We now discuss improvements that have been made to the verification support tool *Source2UPPAAL*.

Figure 8 shows the interface for editing a scenario. The left screen is a template for a scenario and the right screen expresses the abstract syntax tree of the target source code. A tester selects a component from a use case and applies it to SUB1 or SUB2 on the left screen. To analyze the use case effectively, they then use the UI correspondence table mentioned in Section 3.5. Source2UPPAAL has a function that highlights the key words extracted through this static analysis.

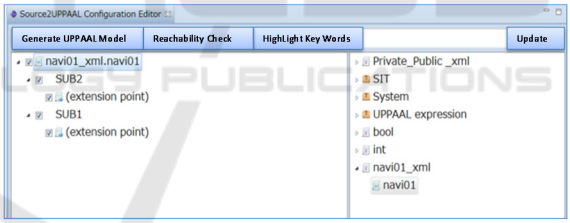

Figure 8: Editing of a scenario.

Figure 9 shows the process of assigning a function to the scenario template. A tester searches for a function by using the UI correspondence table. They then follow the path on the right screen. An actor of the target use case is a student and the function is a function of BBS.

SIT.Luminous.Web.student.detail.bbs::buttonCreate_Click

If the user finds the button Create_Click method, then that component corresponds to a goal function. They can then drag it to the target statement on the left screen as shown in Figure 9. As a result, the target method can be unfolded in the verification scenario.

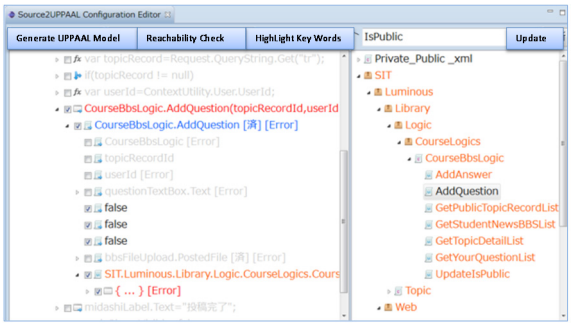

Figure 9: Assigning a function to the scenario.

4 DISCUSSION AND CONCLUSIONS

This paper presented an experiment on the verification of security requirements for the source code of an existing system. The experiment showed that we could verify whether the existing system satisfied the security requirements. Generally, source code verification is difficult and time consuming. There are several different approaches to source code verification (Beyer et al., 2004, Thompson et al., 2008). A tester needs to analyze the details of the target source code and insert assertion statements. In this case, requirements specification implementation will be dependent on the document.

To optimize the approach shown in Figure 1, we solved the following problems through experimentation:

- RA Models are an effective approach for specifying functional requirements. Security requirements are a kind of non-functional requirement related to functional requirements; they can be specified through the systematic method shown in Figure 3. It is important to design and manage concepts such as Cross-Cutting Concerns and Context Awareness modularly so that we can formally verify them. Furthermore, such designs need to be implemented in a clear and consistent manner for the duration of the project.
- In this paper, RA Models were defined manually. We are planning the development of a support tool for these manually defined RA Models to improve their comprehensiveness and accuracy.
- Several problems impede the creation of a correspondence table through static analysis of the source code. Various different technologies such as programming languages, platforms, and application frameworks are used to implement a software system. Because analysis methods depend on the technologies used, an analysis tool can be very expensive to develop. It is also common that developers do not comply with standard coding conventions during the project and team members can vary during the course of the project. This causes inconsistency in definition of operations and fields.

REFERENCES

OMG, "UNIFIED MODELING LANGUAGE", http://www.uml.org/

Y. Aoki and S. Matsuura, Verifying Security Requirements using Model Checking Technique for UML-Based Requirements Specification, Proc. of 1st International Workshop on Requirements Engineering and Testing, pp.18-25, September,2014.

Y. Aoki, S. Ogata, H. Okuda and S. Matsuura, Data Lifecycle Verification Method for Requirements Specifications Using a Model Checking Technique, Proc. of The Eighth International Conference on Software Engineering Advances (ICSEA 2013), pp.194-200, 2013.

UPPAAL, http://www.uppaal.com/, 2016.

S. Ogata and S. Matsuura, "A UML-based Requirements Analysis with Automatic Prototype System Generation," Communication of SIWN, Vol. 3, pp.166-172, 2008.

Common Criteria, "CC/CEM v3.1 Release4", http://www.commoncriteriaportal.org/cc/

Y. Aoki, S. Matsuura, "Verifying Business Rules Using Model-Checking Techniques for Non-specialist in Model-Checking." IEICE TRANSACTIONS on Information and Systems, Vol. E97-D, No. 5, pp.1097-1108, May, 2014.

S. Matsuura, Y. Aoki, and S. Ogata, Practical Behavioral Inconsistency Detection between Source Code and Specification using Model Checking, ISSRE 2014, pp.124-125, 2014.

LUMINOUS, https://lmns.sayo.se.shibaura-it.ac.jp/

D. Beyer, T.A. Henzinger, R. Jhala, and R. Majumdar, An Eclipse Plug-in for Model Checking, Proceedings. 12th IEEE International Workshop on Program Comprehension, pp. 251-255, 2004.

S. Thompson and G. Brat, Verification of C++ Flight Software with the MCP Model Checker, Aerospace Conference 2008 IEEE, pp.1-9, 2008.