XMITS: Software Inspections via Formal Verification

Luciana Brasil R. Santos[1], **Camila P. Sales**[1], **Valdivino A. de Santiago Júnior**[2]

[1]Instituto Federal de Educação, Ciência e Tecnologia de São Paulo (IFSP)
Câmpus Caraguatatuba – Caraguatatuba – SP – Brazil

[2]Laboratório Associado de Computação e Matemática Aplicada (LABAC)
Instituto Nacional de Pesquisas Espaciais (INPE)
São José dos Campos – SP – Brazil

lurebelo@ifsp.edu.br,camilapsales27@gmail.com,valdivino.santiago@inpe.br

Abstract. *In this paper, we present version 3.1 of XMITS, a tool developed to support Model Checking-aided inspections improving the design of software. XMITS enables the translation of UML behavioral diagrams representation to Transition Systems and then to the input language of NuSMV model checker. Our tool was applied to real case studies (embedded software) in the space domain. The main objective is to allow the use of Formal Methods (Model Checking, in this case) establishing a solution that can be used in practice.*
Tool video demonstration: *https://youtu.be/dJ0tbJoO3Sg*

1. Introduction

In the context of Software Engineering, Verification and Validation (V&V) discipline is one of the pillars to ensure that software products are of high quality, and this discipline is particularly important if critical systems are considered. V&V activities are usually time-consuming, specially if critical/complex systems are considered. Techniques are developed to reduce and ease the V&V efforts while increasing their coverage. V&V encompasses a large range of activities and techniques, of which one can mention testing [Ammann and Offutt 2016], inspection [IEEE 1990], and Formal Verification [Clarke et al. 1999]. Model Checking is the most popular Formal Verification method and can be defined as an automated technique that, given a finite-state model of a system and a formal property, systematically checks whether this property holds for that model [Baier and Katoen 2008]. This verification is performed using a model checker.

Considering that, we developed a methodology, which we call SOLIMVA 3.0 [dos Santos et al. 2014], aiming at detecting defects within the design of the software product. Our methodology works with three different UML (Unified Modeling Language) [OMG 2015] behavioral diagrams (sequence, behavioral state machine, and activity), translating them into a single Transition System (TS) to support Model Checking of software developed in accordance with UML.

The Formal Verification process is as follows: the analyst collects requirements from software specifications. Such requirements are generally expressed within UML use case models or simply in Natural Language. SOLIMVA 3.0 suggests using Specification Patterns [Dwyer et al. 1999] to direct the formalization of properties. The UML behavioral diagrams should reflect these requirements. So, these diagrams (sequence, behavioral state machine, and activity) are translated into a TS, and then to the notation of the NuSMV model checker [Kessler 2015] by means of a tool developed to support SOLIMVA 3.0: XML Metadata Interchange to Transition System (XMITS). Hence, the properties are verified and it is possible to determine if there are defects with the design of the software product.

XMITS has been continuously developed. In this article, we introduce version 3.1 of this tool where the main differences from previous versions is related to the counterexample, which is detailed in the next section. The counterexample was a limitation in previous versions. This paper is structured as follows. Section 2 shows the main functionality of XMITS, as well as its architecture. Section 3 presents an example of using the tool. Section 4 discusses related tools versus XMITS. Conclusion are in Section 5.

2. XMITS: Main Functionalities and Architecture

XMITS is essential to the application of SOLIMVA 3.0. It facilitates the process of applying Formal Verification during the software development, so that it becomes transparent to the user. Due to the modular nature proposed for the system, the tool is extensible: if one wants to add another UML diagram to the approach, this can be easily implemented in XMITS. Java [Oracle 2011] is the chosen language, as the object-oriented programming paradigm is used.

Figure 1 presents the the architecture of XMITS. The tool interoperates with two other tools: Modelio 3.2 [Modeliosoft 2011], which is the software used to produce the UML artifacts; and the NuSMV model checker, which performs the Formal Verification. XMITS consists of six modules: the Reader, that receive the diagrams in XMI format and transforms them to a list of tags; the Converter, that transforms the list of tags to a single TS; the TUTS (The Unified Transition System), that transforms the single TSs to the unified TS; the Bridge module, that transforms the unified TS to the model checker notation; the Global module, which saves the Transition System output as a txt file; and the Interface module, which is responsible for saving the TS in graphical format and implements the user interface.

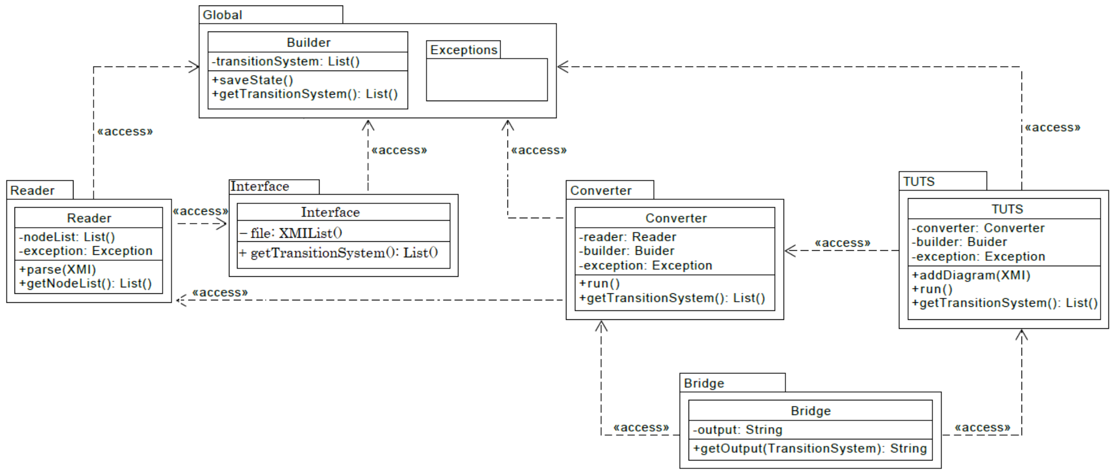
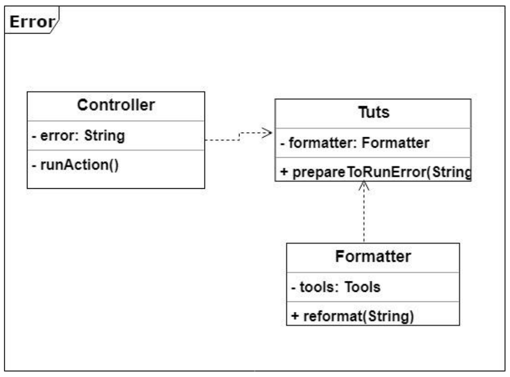


Figure 1. Architecture of XMITS

XMITS has undergone constant updates. The first version encompassed only modules Reader, Converter, and Global. At that time, the translation of the diagrams was performed individually and only a txt file was generated. Then, the second version had, in addition to the previous ones, the modules TUTS and Bridge. Here it was possible to convert the diagrams to a unified Transition System (joining all diagrams) and generate the smv file, the input language of the NuSMV model checker. So, the third version brought the Interface module, where the user no longer needed programming knowledge to use XMITS, an interface was developed. Besides, the final TS could be observed in a graph format, which has substantially improved the validation of the TS model generated.

This work presents version 3.1 of XMITS. The improvement addressed this time refers to the counterexample generated by the model checker, when finding an inconsistency. The model checker shows the state where the requirement (property) was not satisfied within the TS. However, it is necessary to automatically specify the UML diagram(s) where this inconsistency was found and, more than that, the exact point in this diagram(s) where the property was not satisfied. The automated translation of the model checker counterexample back to the UML diagrams is important to identify in which diagram, or diagrams, the detected problem is related. To achieve this goal, three classes were created, as can be seen in Figure 2-A. The interface screen had a minor change: a field to insert the TS state (where the property was not satisfied). Once the user fills this field, the tool points to the original diagram where the problem occurred, showing the state, as well as the position of this state in the diagram. Figure 2-B shows the new field created (highlighted in red).



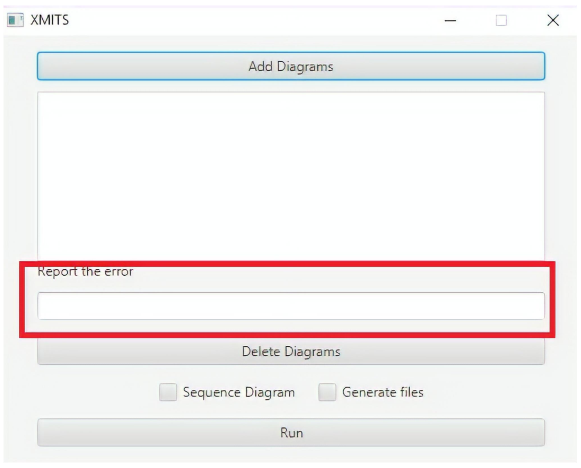


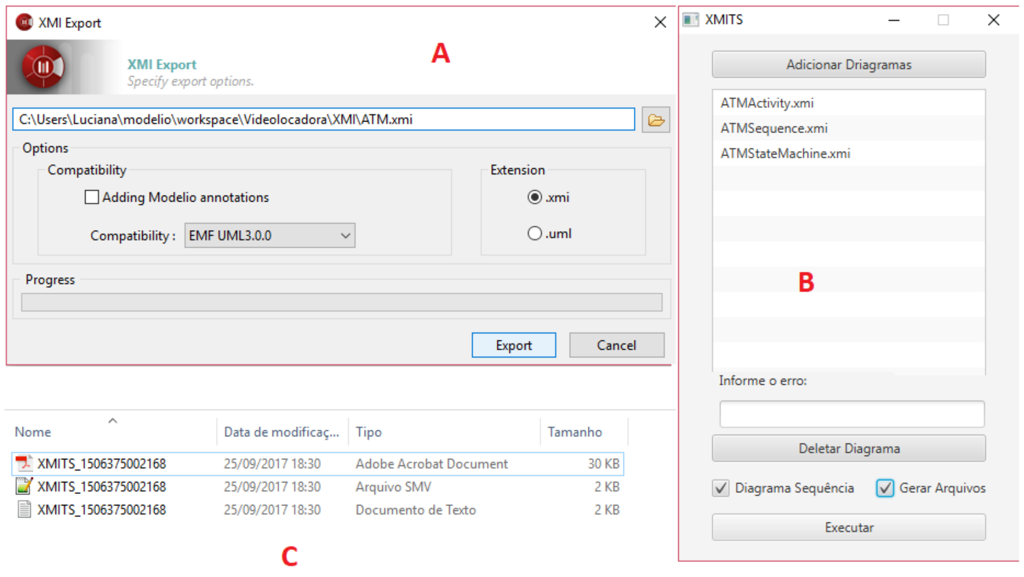
Figure 2-A) XMITS 3.1 classes

Figure 2-B) XMITS 3.1 graphical interface

Figure 2. XMITS 3.1: New classes for identifying the counterexample back to the UML diagrams (A) and graphical interface (B)

3. XMITS in Action: Usage Example

In order to illustrate the use of XMITS, we present the main steps that must be performed by the user. In accordance with SOLIMVA 3.0 methodology, it is necessary to identify a scenario and then select the requirements to be analyzed. Suppose we want to analyze the ATM (Automated Teller Machine) classical example and we choose the following requirement to check: *whenever the specified amount exceeds the level of available funds, it should be possible for the user to request a new cash advance operation if the user wishes to correct the amount*. The ATM Specification shows three behavioral diagrams which are related to this requirement: sequence, activity, and a state machine. Once the diagrams are identified, they must be rewritten using Modelio, so that they can be exported to XMI format, the input of XMITS. Thus, XMITS translates the diagrams into a unified TS and then to a smv file. Figure 3 depicts three screens illustrating some steps of this process: A) Modelio - a file being exported to XMI format; B) XMITS 3.1 - UML diagrams of ATM being translated; and C) Results when running XMITS 3.1. In this case, three files are generated (in the order they appear on the screen): a pdf file showing the TS in graphical mode; an smv file, which is the input to NuSMV; and a txt file showing the TS in text mode.



C:\Users\Luciana\modelio\workspace\Videolocadora\XMI\ATM.xmi

Options

Compatibility

☐ Adding Modelio annotations

Compatibility : EMF UML3.0.0

Extension

● .xmi

○ .uml

Progress

ATMActivity.xmi

ATMSequence.xmi

ATMStateMachine.xmi

Informe o erro:

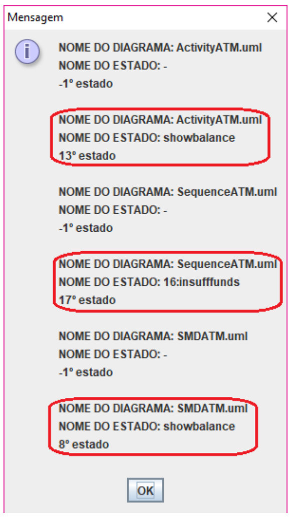
✓ Diagrama Sequência ✓ Gerar Arquivos

Nome	Data de modificaç...	Tipo	Tamanho
XMITS_1506375002168	25/09/2017 18:30	Adobe Acrobat Document	30 KB
XMITS_1506375002168	25/09/2017 18:30	Arquivo SMV	2 KB
XMITS_1506375002168	25/09/2017 18:30	Documento de Texto	2 KB

Figure 3. Screens depicting few steps of SOLIMVA methodology

It is necessary to formalize the property (requirement) to proceed with Model Checking. The property can be formalized using the patterns and scopes proposed by Dwyer [Dwyer et al. 1999]. [1] Once having a formalized property and a model (smv file), it is possible to apply Model Checking. After applying Model Checking, a counterexample is generated, indicating that this property is not satisfied. The state where the property is not satisfied is *_$_$$minus$$coma$__16$dots$insufffunds$coma$__showbalance$_$* because it is not possible, from this state, to reach a state where the customer can request a new cash advance operation. When feeding the error field (highlighted in red in Figure 2-B) with this state, the tool returns the response shown in Figure 4, pointing out the diagram and state in this diagram where the problem occurred, as well as its position within the diagram (highlighted in red in Figure 4). This means that at this point, the diagram does not reflect the requirement.

[1]Santos et al. present a complete explanation of the property formalization [Santos 2015].



```
NOME DO DIAGRAMA: ActivityATM.uml
NOME DO ESTADO: -
-1º estado

NOME DO DIAGRAMA: ActivityATM.uml
NOME DO ESTADO: showbalance
13º estado

NOME DO DIAGRAMA: SequenceATM.uml
NOME DO ESTADO: -
-1º estado

NOME DO DIAGRAMA: SequenceATM.uml
NOME DO ESTADO: 16:insufffunds
17º estado

NOME DO DIAGRAMA: SMDATM.uml
NOME DO ESTADO: -
-1º estado

NOME DO DIAGRAMA: SMDATM.uml
NOME DO ESTADO: showbalance
8º estado
```

Figure 4. The counterexample showed in the UML diagrams

4. Other tools

Coskun et al. present AutoInspect, a tool for semi-automated inspection of design documents [Coskun et al. 2016]. As outputs, a list of defects found during inspection, and a design verification report (in PDF format) are created. Taba and Ow developed a web-based tool, ArSeC, to support their proposed model [Taba and Ow 2016]. It is designed to detect and remove the defects in the first two phases of software development. The result alerts the inspectors about the possible defects and shows the possible causes. Dautovic et al. presented checks software development documents against implemented document quality rules [Dautovic et al. 2011]. The approach suggested by Li and Liu designed methods of deriving functional scenarios and generating inspection tasks by applying consistency properties to each scenario [Li and Liu 2014]. They implemented these specific methods in a support tool.

Some of the tools discussed so far are not available, so the usability aspects cannot be compared. What we could consider is that the approaches, as well as the tools, support particular techniques/methodologies, normally to deal with issues on specific contexts. We are proposing a more broad way to perform requirements checking, which can be applied in different contexts. Our approach allows that an informal language (UML), still quite popular, can continue to be used for creating the design of software systems. XMITS can be applied to any software product that uses UML as the modeling specification language.

5. Conclusion

This paper presented version 3.1 of XMITS, a tool developed to support the translation of UML behavioral diagrams representation to the input language of the NuSMV model checker. The main motivation is to build a solution which can be used in practice. XMITS was applied to two real embedded software in the space domain. Defects were detected within the design of these software systems showing the feasibility of the proposed approach.

This version implements the functionality to catch the feedback from NuSMV and show to the user the model checker counterexample back to the UML diagrams, which was a limitation in older versions of XMITS. Future directions of this research include directives to improve the automation of property formalization. We consider this a very important issue to be addressed, in order to SOLIMVA methodology can be adopted as part of the software development process. The tool, as well as the user manual, are available at https://github.com/abacoresearchgroup/xmits.

References

Ammann, P. and Offutt, J. (2016). *Introduction to software testing*. Cambridge University Press.

Baier, C. and Katoen, J.-P. (2008). *Principles of model checking*. MIT Press.

Clarke, E. M., Grumberg, O., and Peled, D. (1999). *Model checking*. MIT press.

Coskun, M. E., Ceylan, M. M., Yigit¨ozu, K., and Garousi, V. (2016). A tool for automated inspection of software design documents and its empirical evaluation in an aviation industry setting. In *Ninth Proceedings of the IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 287–294.

Dautovic, A., Plösch, R., and Saft, M. (2011). Automated quality defect detection in software development documents. In *First Proceedings of the International Workshop on Model-Driven Software Migration (MDSM 2011)*, page 29.

dos Santos, L. B. R., de Santiago Júnior, V. A., and Vijaykumar, N. L. (2014). Transformation of uml behavioral diagrams to support software model checking. *Electronic Proceedings in Theoretical Computer Science*, 147:133–142.

Dwyer, M. B., Avrunin, G. S., and Corbett, J. C. (1999). Patterns in property specifications for finite-state verification. In *21st Proceedings of the International Conference on Software Engineering*, pages 411–420.

IEEE (1990). Institute of electric and electronic engineers. *Standard glossary of software engineering terminology*, Standard 610.12.

Kessler, F. B. (2015). Nusmv home page. http://nusmv.fbk.eu/.

Li, M. and Liu, S. (2014). Tool support for rigorous formal specification inspection. In *17th Proceedings of the IEEE International Conference on Computational Science and Engineering (CSE)*, pages 729–734.

Modeliosoft (2011). Modelio open source community. https://www.modelio.org.

OMG (2015). Unified modeling language (omg uml). http://www.uml.org/.

Oracle (2011). Javadoc tool home page. http://www.oracle.com/technetwork/java/javase/.

Santos, L. B. R. (2015). *A Methodology to apply formal verification to UML-based software*. PhD thesis, Instituto Nacional de Pesquisas Espaciais (INPE).

Taba, N. and Ow, S. (2016). A new model for software inspection at the requirements analysis and design phases of software development. *The International Arab Journal of Information Technology (IAJIT)*, 13(6):51–57.