

A Rigorous Evaluation of the Benefits of Usability Improvements Within Model Checking-Aided Software Inspections

Luciana Brasil Rebelo dos Santos¹(✉),
Valdivino Alexandre de Santiago Júnior², and Albino Vieira Freitas¹

¹ Instituto Federal de Educação,
Ciência e Tecnologia de São Paulo (IFSP) - Campus Caraguatatuba,
Av. Bahia, 1739, Caraguatatuba, São Paulo, SP, Brazil
lurebelo@ifsp.edu.br, albinofreitas7@gmail.com

² Instituto Nacional de Pesquisas Espaciais (INPE),
Av. dos Astronautas, 1758, São José dos Campos, São Paulo, SP, Brazil
valdivino.santiago@inpe.br

Abstract. In this paper, we show the results of a controlled experiment aiming at assessing the benefits of usability improvements for software inspection methodologies that rely on Model Checking. This work has been carried out in the context of the SOLIMVA 3.0 methodology which uses Model Checking to help in the inspection of software designs. A tool, XMITS, has been developed to support SOLIMVA 3.0. Thus, we compared the benefits in terms of cost mainly related to the Modeling activity of SOLIMVA, by using the new (3.0) and the previous (2.0) versions of XMITS. We considered 20 sets of UML behavioral diagrams from two different space application systems and the ATM system. Results backed by statistical analysis show that XMITS 3.0 was better than XMITS 2.0, helping to decrease the total time spent in the Modeling phase. This fact confirms that true usability improvements in software products can have a significant impact on processes such as inspection.

1 Introduction

The scientific community has increased efforts to carry out more rigorous assessments, such as experimental studies and quasi-experiments. One of the main reasons is that ‘there is an increasing understanding in the software engineering community that empirical studies are needed to develop or improve processes, methods, and tools for software development and maintenance’ [1]. Various surveys with gathered data of an assorted number of papers addressing controlled experiment show the relevance of software engineering experiments to industrial practice and scientific maturity of software engineering research [1–3].

Inspections are used to identify defects in software artifacts. Design documents can be inspected to verify whether software requirements were correctly captured. In this way, inspection methods help to improve software quality, especially when used early in software development. Inspections of software design

can be especially crucial since design defects, such as problems of correctness and completeness with respect to the requirements or internal consistency, can directly affect the quality and effort required for the software development [4]. Nevertheless, the number of companies where software inspection is part of the culture and organizational process is still small compared to the benefits of its use. Therefore, the development of techniques that are applicable in the scenarios normally found in software companies can make the use of inspection effective.

Model Checking [5–7] is the most popular Formal Verification method which has been receiving much attention from the academic community due to its mathematical foundations. It has been adopted in the industry, as well. Model Checking is a formal automatic verification technique that, given a finite-state model of a system (also known as a Transition System (TS)) and a formal property, systematically verifies whether this property is satisfied by that model [5]. However, despite its benefits, in general, practitioners still avoid using Model Checking in their projects due to aspects such as high learning curve and cost, and the lack of commercially supported tools. Thus, efforts to build tools and automate the process for the use of Formal Verification can facilitate and encourage a wide acceptance of Formal Methods in every day software development. In this line, approaches that translate industry non-formal standards such as UML (Unified Modeling Language) [8] to model checkers notation are a great step towards this goal.

In this context, we had started a work to study, analyze and manipulate UML diagrams to make feasible the use of Model Checking addressing inspection in the software industry. For this, a methodology called SOLIMVA [9] was developed aiming to translate several UML behavioral diagrams (sequence, activity, and state machine) into Transition Systems (TSs) to support Model Checking-aided software inspections. XMITS - XML Metadata Interchange to Transition System [10] - is the tool that supports the conversion of the UML diagrams representation to the input language of NuSMV [11] model checker. The diagrams are processed and converted into Transition System (TS), the basis for Model Checking tool entry. So, basically the main goal of XMITS is to transform the UML diagrams into TSs.

We improved our tool with many important features. Some advancement mainly related to usability aspects are addressed in this new version of XMITS (3.0). The major differences relating to the older version (2.0) are: (i) Interface - the user has an interface where he/she can choose the diagrams and generate the TS or the NuSMV input language. In version 2.0, the user should build a Java class to handle XMITS; (ii) Graph format file - the TS output was a txt file in version 2.0. Now, one can also generate another output file, where it is possible to visualize the Transition System in graph format, observing its states and transitions, which substantially facilitates the system model validation process, before applying Model Checking; and (iii) Installer - an installer was generated to create an executable with any necessary dependencies.

In this paper, we present a controlled experiment, a rigorous evaluation, to realize how usability improvements in software products impact on processes,

such as inspections based on Model Checking. This work has been carried out in the context of the SOLIMVA 3.0 methodology which uses Model Checking to help in the inspection of software designs. We compared the benefits in terms of cost, i.e. the amount of time spent, mainly related to the Modeling activity of SOLIMVA, by using the new (3.0) and the previous (2.0) versions of XMITS. Twenty sets of UML behavioral diagrams from two different complex space application systems and the Automated Teller Machine (ATM) case study have been evaluated. Results backed by statistical analysis show that XMITS 3.0 was better than XMITS 2.0, helping to decrease the total time spent in the Modeling phase. We believe that true usability improvements in software products that support Verification & Validation (V&V) processes, such as inspection, can have a significant impact on such efforts.

This paper is structured as follows. Section 2 shows an overview of the SOLIMVA 3.0 methodology and the new version, 3.0, of the XMITS tool. The controlled experiment where we compared the two different versions of our tool in the context of SOLIMVA is in Sect. 3. Section 4 presents related work. In Sect. 5, we state the conclusions and future directions of this research.

2 SOLIMVA and XMITS: Versions 3.0

The verification process established in SOLIMVA 3.0 essentially consists of sequence of scenarios to be checked. Scenarios focus on how the system behaves to implement its functionalities. The analyst gathers requirements from software specifications. In practice, such requirements are generally expressed within UML use case models or simply in Natural Language. SOLIMVA 3.0 suggests using specification patterns [12] to direct the formalization of properties in Computation Tree Logic [5]. The corresponding UML behavioral diagrams that represent the solution to meet the requirements (use cases or pure textual requirements) are taken into account. These UML diagrams (sequence, behavioral state machine, and activity) are input to XMITS. XMITS performs a three-step translation. First, it translates individual types of diagrams (Sequence Diagram, Activity Diagram, Behavioral State Machine Diagram) into a TS in a simple intermediate format. After that, XMITS merges all single TSs into a unified TS. Finally, the tool transforms this unified TS into the notation of the NuSMV model checker. By running NuSMV with the unified TS and the properties in CTL, it is possible to determine if there are defects with the design of the software product. In case the TS does not satisfy a certain property, a counterexample is presented by the model checker. The workflow of SOLIMVA 3.0 is shown in Fig. 1.

XMITS interoperates with two other tools: Modelio 3.2 [13], that is the software used to produce the UML artifacts. The design artifacts are then exported into XMI (XML Metadata Interchange) format, and are inputted to XMITS; and the NuSMV model checker. Currently, XMITS consists of six modules (one more than its older version): the **Reader**, that transforms the XMI file to a list of tags; the **Converter**, that transforms the list of tags to a single TS; the

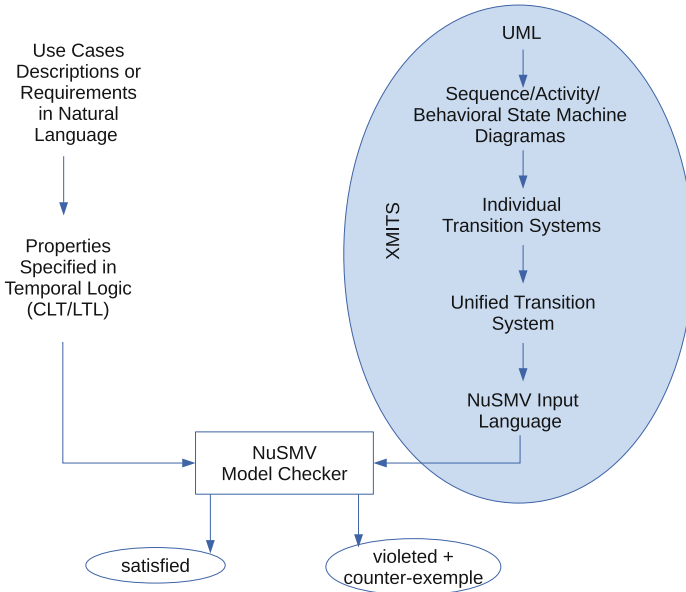


Fig. 1. Schematic view of SOLIMVA 3.0 using XMITS. Adapted from [5].

TUTS (The Unified Transition System), that transforms the single TSs to the unified TS; the **Bridge** module, that transforms the unified TS into the model checker notation; and the **Global** module, which is responsible for several important functions, such as the creation of the TS data structure and the *Printer*, where one can visualize the TS tree. A detailed description of these modules is presented in [10]. And finally, the **Interface** module, which is explained in next subsection.

2.1 The Interface Module

The **Interface** module was developed to support the tool graphical views. This module uses the *JavaFX* API, which in turn, has the purpose to express user interfaces, using a declarative programming style [14]. The module also works with Graphviz, an open source free software for graph visualization [15]. Graphviz uses DOT [16], a language to draw hierarchical or layered directional graphs. In this way, Graphviz can be used for visualization of Transition Systems. The GraphViz Java API is adopted to call Graphviz through the Java application. This module, composed of four classes: the *Main* class, the *Controller* class, the *Graphviz* class, and the *Dot* class.

The **Main** class is responsible for starting the application using *FXML*, a declarative language, XML-based, for constructing a *JavaFX* application user interface; the **Controller** class manages the whole application manipulating the interface and all the tool functionality; the **Graphviz** class is responsible for

generating the pdf file with the graphical image of the TS; and the class **Dot** generates the dot file, used for the Graphviz class to generate the image.

2.2 New Features of XMITS

As stated before, there are three main differences between the two versions, which impacts directly the aspects of usability and cost.

Interface: In the previous version of XMITS the user was required to have a knowledge in programming, as he/she needed to create a project in a Java development environment. The user had to add the Java Archive (JAR) of XMITS and create a class to manually insert the directory of the UML files to be processed. This process influences directly the usability of the application, since it can cause several errors, such as inserting the wrong UML diagram directory, syntax error in code, JAR import errors, among others.

Thus, an interface, which can be seen in Fig. 2, was developed to solve the issues related above. Following, the description of the actions of each numbered button is explained.

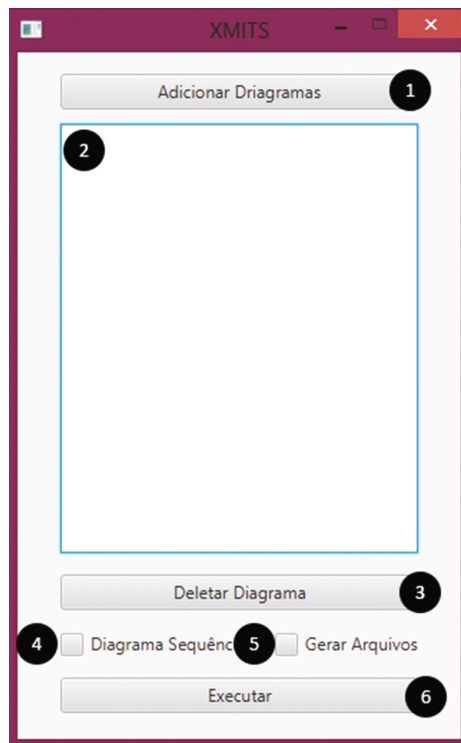


Fig. 2. XMITS graphical interface.

1. (Add Diagrams): This button is used to insert the diagrams that will be processed by the tool. The allowed input formats are XMI, XML and UML. Clicking this button will open a window where diagrams can be selected.
2. (Display List): In this list, the diagrams that are selected will be displayed and processed. The insertion of diagrams with identical names is not allowed.
3. (Delete Diagram): This button allows to delete a diagram from the list, preventing it from being processed. To do this, it is necessary to select a diagram using the mouse click and then press the 3 button or the delete key on the keyboard.
4. (Sequence Diagram): This check box is used to indicate the existence of a sequence diagram and allow the insertion of more diagrams. It was not mentioned, but according to the SOLIMVA methodology, the sequence diagram is mandatory to generate the TS. The description of SOLIMVA is explained in [9].
5. (Generate Files): This check box is used to enable the creation of output files. Three types of files can be created: a text file with the Transition System representation; a file in the pdf extension containing the image of the transition system; and a file in smv format, that is the input for NuSMV.
6. (Execute): This button is used to perform the processing of the input diagrams and generate their respective output. If the check box (Generate Files) is not checked, only the Transition System representation in txt format is displayed on the screen.

Graphical Transition System: In the previous version of XMITS, when the user had to validate the TS model generated, he/she should read the text file, and then generate a graphical representation manually. This action is not a trivial task, since it requires the user to have the knowledge of how the tool generates the output and how it should be read to create the graphical view of the TS. In addition, this task is very time consuming and can lead to several errors, especially when TS has many states. Depending on the number of states, this task can not be performed.

To solve this problem, an automatic generation of the graphical view of the TS was developed. Graphviz is used to perform this action. This software receives as input a file specified in Dot language and then generates an output containing the graphic representation of the specified content, as shown in Fig. 3.

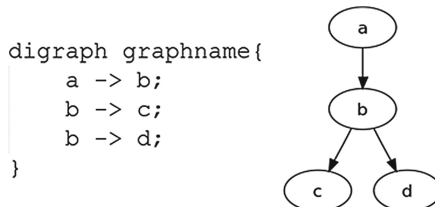


Fig. 3. TS specified in Dot language and its output after being processed by Graphviz.

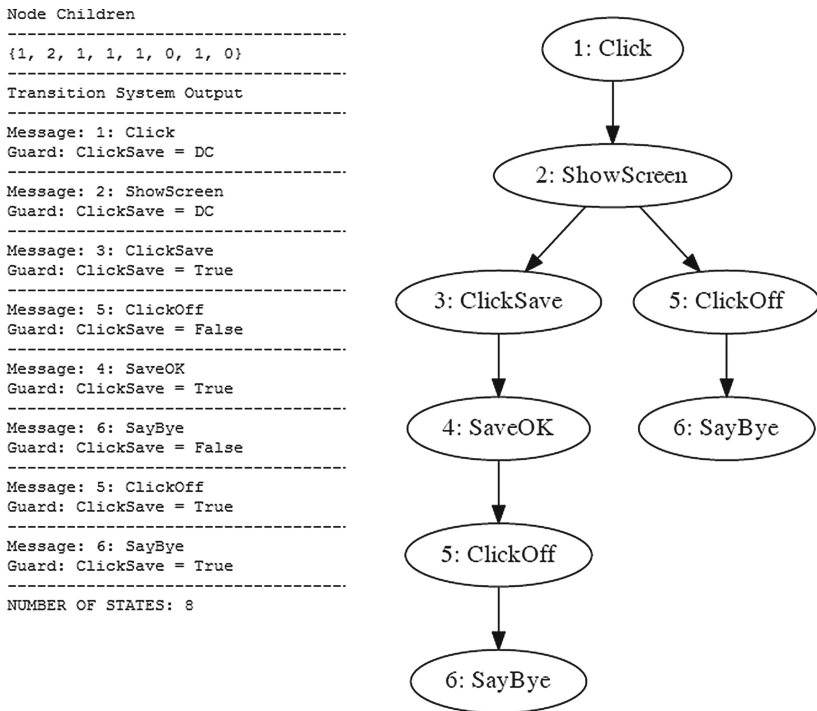


Fig. 4. TS Output of XMITS in both formats.

The Interface module implements the conversion of the TS, which is represented as a linked list in the TUTS module, into a file specified in Dot language. The class *Dot* has a method that traverses the linked list and constructs the TS structure specified in Dot language and saves it to a String. This String is used later to create the “.dot” file that will be processed by Graphviz. Figure 4 shows the TS represented in both output files. Note that building the TS from the text file is not so easy. It is necessary to follow a heuristic and understand how the text file was constructed.

Installer: To use XMITS, an application JAR was generated and should be run in the same directory as the folder where the Graphviz files and dependencies are located. However, to facilitate the execution of the application and to avoid possible errors caused by the use of Graphviz without installation, an installer was generated through the software Inno Setup [17]. Such software allows to create an executable with any necessary dependencies.

Thus, it was possible to join the Graphviz folder to the generated executable file. Therefore, when the XMITS installation is run, a folder is created on the machine’s local disk and the application’s JAR and Graphviz folder are extracted to this directory. The user still has the option to save a shortcut of the application executable to the desktop of the computer.

3 Controlled Experiment

3.1 Definition

The goal of this controlled experiment is to assess the cost of the application of SOLIMVA 3.0 which relies on Model Checking if we consider the new (3.0) and the old (2.0) versions of XMITS. Roughly speaking, the SOLIMVA 3.0 methodology is split in 3 main activities: Properties formalization, Modeling, and Execution of Model Checking. The XMITS tool only relates to the Modeling activity. Thus, we focused only on the Modeling phase which in turn is divided in three sub-activities: Create UML Diagrams in Modelio (Dia), Generate the Unified TS (Gen), and Validate the Unified TS (Val). But, creating UML diagrams is not different in both versions of XMITS: it is basically a manual effort that must be done by the designer. Hence, the cost is the time spent in the sub-activities Gen, Val, and also the Total (Tot) time of the Modeling activity.

3.2 Context

The experiment was conducted by the authors and we have used the directions proposed in [18] to accomplish this evaluation. Three applications, two space software systems developed or under development at INPE (*Instituto Nacional de Pesquisas Espaciais*) in Brazil and one classical example, were considered in our experiment. **SWPDC** (Software for the Payload Data Handling Computer) is a space application product created in a project whose main goal was to outsource the development of software embedded in satellite payload. **SWPDCpM** (Software for the Payload Data Handling Computer - protoMIRAX experiment) is an considerably improvement of SWPDC and it has been adapted to protoMIRAX, a hard X-ray imaging telescope under development at INPE. A balloon will launch this instrument and it will operate between 40 to 42 Km of altitude. The third case study is the classical **ATM** (Automated Teller Machine) where the ATM interacts with a customer via a specific interface and communicates with the bank over an appropriate communication link.

For the three software products, several scenarios have been selected. Hence, a sample is precisely a scenario identified in accordance with the guidelines of SOLIMVA 3.0. In total, 20 scenarios (samples) have been analyzed. Each scenario is defined as a set of UML behavioral diagrams, as presented in Table 1. Moreover, in Table 1 the number means how many diagrams of a certain type exist for a scenario.

3.3 Hypotheses

The following hypotheses were considered in this evaluation:

- **Null Hypothesis H1.0:** There is no difference in cost (time spent) related to the Gen sub-activity between XMITS 3.0 and XMITS 2.0;
- **Alternative Hypothesis H1.1:** There is difference in cost (time spent) related to the Gen sub-activity between XMITS 3.0 and XMITS 2.0;

Table 1. Samples for the controlled experiment: scenarios.

| Scenario | Sequence | Activity | State machine |
|--------------|----------|----------|---------------|
| SWPDC - 1 | 1 | 1 | 1 |
| SWPDC - 2 | 2 | 0 | 0 |
| SWPDC - 3 | 1 | 0 | 1 |
| SWPDC - 4 | 1 | 1 | 1 |
| SWPDC - 5 | 3 | 0 | 1 |
| SWPDC - 6 | 1 | 1 | 1 |
| SWPDC - 7 | 2 | 1 | 1 |
| SWPDCpM - 1 | 1 | 0 | 0 |
| SWPDCpM - 2 | 1 | 0 | 0 |
| SWPDCpM - 3 | 1 | 0 | 0 |
| SWPDCpM - 4 | 1 | 0 | 0 |
| SWPDCpM - 5 | 1 | 0 | 0 |
| SWPDCpM - 6 | 1 | 0 | 0 |
| SWPDCpM - 7 | 1 | 0 | 0 |
| SWPDCpM - 8 | 1 | 0 | 0 |
| SWPDCpM - 9 | 1 | 0 | 0 |
| SWPDCpM - 10 | 1 | 0 | 0 |
| SWPDCpM - 11 | 1 | 0 | 0 |
| SWPDCpM - 12 | 1 | 0 | 0 |
| ATM - 1 | 1 | 1 | 1 |

- **Null Hypothesis H2.0:** There is no difference in cost (time spent) related to the Val sub-activity between XMITS 3.0 and XMITS 2.0;
- **Alternative Hypothesis H2.1:** There is difference in cost (time spent) related to the Val sub-activity between XMITS 3.0 and XMITS 2.0;
- **Null Hypothesis H3.0:** There is no difference in cost (time spent) related to the Total (Tot) effort between XMITS 3.0 and XMITS 2.0;
- **Alternative Hypothesis H3.1:** There is difference in cost (time spent) related to the Total (Tot) effort between XMITS 3.0 and XMITS 2.0.

3.4 Variables and Description of the Experiment

The *independent* variables are those that can be manipulated or controlled during the process of trial and define the causes of the hypotheses [19]. In our case, the independent variables are both versions of XMITS (2.0 and 3.0), the selected scenarios/samples and its UML behavioral diagrams, the ability of the professional to model software designs via UML and to validate the unified TS. In the *dependent* variables, we can observe the result of manipulation of the *independent* ones. The time for generating the unified TS, the time for validating the unified TS, and the total time for the Modeling phase are the dependent variables. All time measures were done in minutes (min).

SOLIMVA 3.0 was executed twice considering the 20 scenarios (see Table 1): 19 scenarios refer to the two space systems, and 1 is from the ATM case study. In the first execution we considered the current version, 3.0, of XMITS, and, in the second execution, the previous version of XMITS, 2.0. For each version of XMITS, we measured and recorded the time spent for generating the unified TS (Gen 2.0, Gen 3.0), validating the unified TS (Val 2.0, Val 3.0), and the Total (Tot 2.0, Tot 3.0) time of the Modeling activity. In the first execution, we recorded the respective times due to XMITS 3.0 (Gen 3.0, Val 3.0, Tot 3.0), and, in the second execution, due to XMITS 2.0 (Gen 2.0, Val 2.0, Tot 2.0).

The cost is simply considered as the amount of time spent for each of the tasks: the lower the time, better the cost. XMITS 3.0 and 2.0 ran in a computer with an Intel Core(TM) i3-4005U CPU @ 1.70 GHz Multicore processor, 4 GB of RAM, running Microsoft Windows 10 Professional 64-bit operating system.

A statistical evaluation for verifying data normality has been done in five steps: (i) by using the Shapiro-Wilk test [20] with a significance level $\alpha = 0.05$; (ii) by using the Anderson-Darling test [21] with a significance level $\alpha = 0.05$; (iii) checking the *skewness* of the frequency distribution; (iv) visually inspecting the Q-Q plot; and (v) visually inspecting the histogram. This 5-step verification gives a greater confidence in the conclusion on data normality compared to an approach that is based only on a normality test (e.g. Shapiro-Wilk, Anderson-Darling), considering the effects of polarization due to the length of the samples [22].

If we concluded that data are from a normally distributed population, then the t-test would be applied with $\alpha = 0.05$. Otherwise, we applied the nonparametric Wilcoxon test (Signed Rank) [23] with $\alpha = 0.05$, too. However, if the samples presented ties, we applied a variation of the Wilcoxon test, the Asymptotic Wilcoxon (Signed Rank) [23], suitable to treat ties with significance level $\alpha = 0.05$.

3.5 Validity

The validity evaluation takes into account the risks that may compromise the validity of the experiment.

Threats to internal validity compromise the confidence in stating that there is a relationship between dependent and independent variables [22]. We can not assert that the samples/scenarios were randomly selected because their choice depends on the software products under evaluation. One factor that could compromise the internal validity of the experiment is the professional skills in the validation of the unified TS, since the same TS must be validated twice, one for each version of the tool (3.0 and 2.0). To minimize the effects of this factor, we first performed the experiment with version 3.0 of XMITS, which favors version 2.0, as the professional already knows the TS at the time of its validation when executing version 2.0.

One category of threat to external validity is the population threat. In this threat, we wish to realize how significant is the sample set (scenarios set in our case) of the population used in the study. In this sense, it is pertinent to examine

Table 2. Cost (time spent in min) due to each of the activities: results and mean value.

| Scenario | Gen 2.0 | Gen 3.0 | Val 2.0 | Val 3.0 | Tot 2.0 | Tot 3.0 |
|--------------|---------|---------|---------|---------|---------|---------|
| SWPDC - 1 | 5 | 0.5 | 7 | 4 | 34 | 26.5 |
| SWPDC - 2 | 5 | 0.4 | 1 | 1 | 22 | 17.4 |
| SWPDC - 3 | 4 | 0.4 | 4 | 8 | 21 | 21.4 |
| SWPDC - 4 | 4 | 0.5 | 11 | 7 | 32 | 24.5 |
| SWPDC - 5 | 5 | 0.5 | 11 | 3 | 47 | 34.5 |
| SWPDC - 6 | 4 | 0.5 | 8 | 6 | 25 | 19.5 |
| SWPDC - 7 | 4 | 0.6 | 10 | 8 | 42 | 36.6 |
| SWPDCpM - 1 | 4 | 0.3 | 4 | 1 | 13 | 6.3 |
| SWPDCpM - 2 | 3 | 0.3 | 3 | 1 | 13 | 8.3 |
| SWPDCpM - 3 | 3 | 0.3 | 5 | 2 | 17 | 11.3 |
| SWPDCpM - 4 | 3 | 0.3 | 4 | 1 | 11 | 5.3 |
| SWPDCpM - 5 | 4 | 0.3 | 3 | 1 | 11 | 5.3 |
| SWPDCpM - 6 | 3 | 0.3 | 3 | 1 | 10 | 5.3 |
| SWPDCpM - 7 | 3 | 0.3 | 3 | 1 | 11 | 6.3 |
| SWPDCpM - 8 | 3 | 0.3 | 5 | 3 | 18 | 13.3 |
| SWPDCpM - 9 | 3 | 0.3 | 3 | 1 | 11 | 6.3 |
| SWPDCpM - 10 | 3 | 0.3 | 2 | 1 | 10 | 6.3 |
| SWPDCpM - 11 | 3 | 0.3 | 2 | 1 | 10 | 6.3 |
| SWPDCpM - 12 | 3 | 0.3 | 2 | 0.5 | 9 | 4.8 |
| ATM - 1 | 3 | 0.5 | 11 | 4 | 30 | 20.5 |
| \bar{x} | 3.6 | 0.375 | 5.1 | 2.775 | 19.85 | 14.3 |

the population used in the experiment. Even though the choice of samples were not random as previously mentioned, altogether the set of UML behavioral diagrams that characterize all scenarios contains most of the important fragments, components, states which define the UML diagrams. Thus, we believe that our choice of the set of samples is considerably satisfactory.

3.6 Results

In this section, we present and discuss the results of our controlled experiment. By using the 5-step approach to check data normality, we concluded that none of the measured data come from a normal distribution. As a matter of record, the greatest *p-value* due to Shapiro-Wilk test was from Val 2.0: 0.004077. And the greatest one due to Anderson-Darling test was from Tot 2.0: 0.002769. We clearly see that these values are below the significance level, α , and hence we reject the null hypothesis that the population is normally distributed. We also checked the *skewness*, Q-Q plot and histograms in accordance with our proposed approach.

Since there are ties, we applied the nonparametric test Asymptotic Wilcoxon (Signed Rank) [23]. In Table 2, we show the time measures (in min) and the

Table 3. Cost (time spent in min) due to each of the activities: Asymptotic Wilcoxon.

| Hypothesis | p-value |
|----------------------|---------------|
| 1: Gen 3.0 ↔ Gen 2.0 | $1.907e - 06$ |
| 2: Val 3.0 ↔ Val 2.0 | 0.000679 |
| 3: Tot 3.0 ↔ Tot 2.0 | $3.815e - 06$ |

mean value (\bar{x}) regarding the six data sets. Thus, for scenario SWPDC - 1, the designer took 5 min to generate the unified TS by using XMITS 2.0 (Gen 2.0) while he/she took 4 min to validate the unified TS via XMITS 3.0 (Val 3.0).

Table 3 shows the p-values due to the Asymptotic Wilcoxon test and the boxplots are in Fig. 5. Based on these results, we conclude that all three null hypotheses (H1.0 to H3.0) were rejected because the p-values are below α . Statistically speaking, there is difference in cost if we apply SOLIMVA 3.0 via the two different versions of the XMITS tool. In a pairwise comparison, i.e. Gen 2.0 × Gen 3.0, Val 2.0 × Val 3.0, Tot 2.0 × Tot 3.0, the mean values (\bar{x}) of XMITS 3.0 are all smaller than the respective mean values of XMITS 2.0. We then conclude that applying SOLIMVA 3.0 with XMITS 3.0 is better than doing the same with XMITS 2.0.

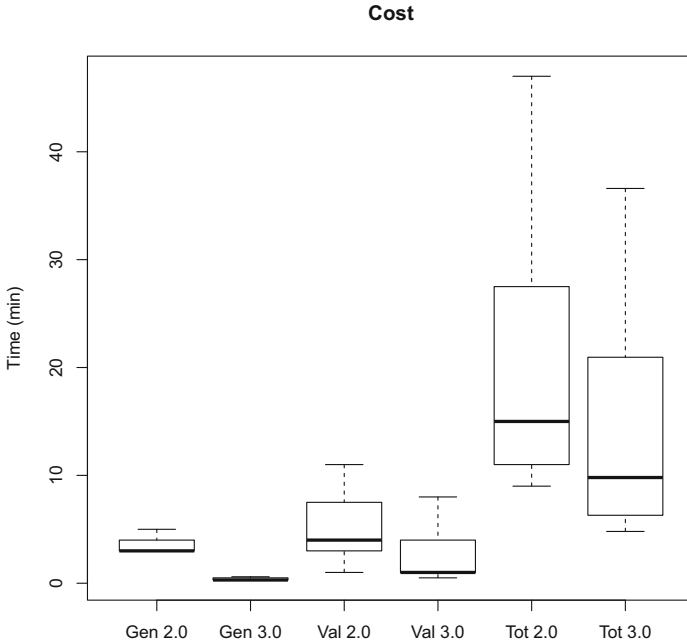


Fig. 5. Cost (time spent in min) due to each of the activities: boxplots.

4 Related Work

This section presents some of the research literature related to this paper (not exhaustive), showing works that deal with software inspections using software support. First, we show some approaches that deal with inspection-aided tools related to software design. Then, works which encompass controlled experiments applied to inspection techniques are addressed.

Taba and Ow [24] developed a web-based tool, ArSeC, to support their proposed model. It is designed to detect and remove the defects in the first two phases of software development. The model suggests designing and using a comprehensive database that contains potential defects and their causes stored. This engine alerts the inspectors about the possible defects and shows the possible causes. The work [25] presents AutoInspect, a tool for semi-automated inspection of design documents. The tool was developed in the context of Turkish Aerospace Industries. The tool facilitates and increases the inspection efficiency in the tasks that in their experience has shown to be tedious and effort intensive, e.g., browsing through a long design Word document and checking formats, etc. As outputs, a list of defects found during inspection, and a design verification report (in PDF format) are created.

A tool-based approach that tries to identify potential document quality defects is presented in [26]. This tool-based analysis relies on best practices for software documentation. The tool checks software development documents against implemented document quality rules. It covers the range from requirements across system, architecture and design, up to test specifications. Sinha et al. [27] have developed an approach for automated and ‘edittime’ inspection of use cases based on the construction and analysis of models of use cases. The models contain linguistic properties of the use case text along with the functional properties of the system under discussion. The author of the use case enters Natural Language text and issues found with the underlying model are listed by the tool. Li and Liu [28] designed methods of deriving functional scenarios and generating inspection tasks by applying consistency properties to each scenario. They implemented these specific methods in a support tool. The inspector has to derive functional scenarios from specification first. Then, the consistency properties need to be applied to each functional scenario to create inspection tasks, which should be examined by the inspector to ensure the consistency.

The papers presented so far show tools that were developed to support particular techniques/methodologies, normally to deal with issues on specific contexts. We did not find any available tool that could be adapted to our needs. We are interested in 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. The complexity for the use of formal methods (Model Checking, in this case) is almost completely hidden from the practitioner, and thus it has a high potential to be applied in practice. SOLIMVA 3.0 can be applied to any software product that uses UML as the modeling specification language.

[29] compared two Inspection Technique, MIT 1 - Model Inspection Technique and UCE - Use Case Evaluation, for Usability Evaluation by means of a controlled experiment that measured efficiency, effectiveness, perceived ease of use, and perceived usefulness. The two techniques aim to support the identification of usability defects through the evaluation of use cases specifications. The results indicate that MIT 1 allows users to find more usability defects in less time than UCE. However, UCE was considered easiest to use and more useful than MIT 1. The work [30] proposed a metric based reading (MBR) technique used for requirements inspections, whose main goal is to identify specific types of defects in use cases. They performed a controlled experiment to ascertain if the usage of MBR really helps in the detection of defects in comparison with a simple Checklist technique. The experiment result revealed that MBR reviewers were more effective at detecting defects than Checklist reviewers, but they were not more efficient, because MBR reviewers took longer than Checklist reviewers on average.

The last two works presented controlled experiments, each one comparing two techniques, the first for usability defects in use cases and the second comparing the reading technique to checklist technique, also related to use cases. We didn't find experiments dealing with Model Checking aiming inspection as our approach does. Besides, SOLIMVA proposes to check requirements not only from use cases, but also requirements expressed in natural language.

5 Conclusions

In this work we presented a controlled experiment aiming at assessing the benefits of usability improvements for software inspection methodologies based on Model Checking. For this, we compared the two versions (2.0 and 3.0) of XMITS, the tool developed to support the SOLIMVA 3.0 methodology, which uses Model Checking to help in the inspection of software designs. The results backed by statistical analysis showed that the cost to perform the Modeling activity of SOLIMVA using the new version of XMITS (3.0) was better than XMITS 2.0. We considered 20 sets of UML behavioral diagrams from two different space application systems and the ATM system to perform the experiment. This fact suggests that usability improvements on software products are valuable and can produce significant impact on process such as inspection, which is strongly dependent on the human factor.

Future directions include advances on usability issues of XMITS, but regarding to another activity of SOLIMVA, the Execution of Model Checking. Such improvements include to inform the user the type of error that is occurring when some inconsistency is found in the model when applying Model Checking. In the current version, the encountered problem is not detailed. Another constraint is related to the counterexample generated, 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 where this inconsistency was found and, more than that, the exact

point in this diagram where the property was not satisfied. In addition, we will perform another controlled experiment or quasi-experiment, considering these improvements, addressing not only the cost but also the effectiveness relating to the number of encountered defects, since effectiveness is a very important question to be answered.

References

1. Sjøberg, D.I.K., Hannay, J.E., Hansen, O., ByKampenes, V., Karahasanovic, A., Liborg, N.-K., Rekdal, A.C.: A survey of controlled experiments in software engineering. *IEEE Trans. Softw. Eng.* **31**(9), 733–753 (2005)
2. Zannier, C., Melnik, G., Maurer, F.: On the success of empirical studies in the international conference on software engineering. In: *Proceedings of the 28th International Conference on Software Engineering*, pp. 341–350. ACM (2006)
3. Lemos, O.A.L., Ferrari, F.C., Eler, M.M., Maldonado, J.C., Masiero, P.C.: Evaluation studies of software testing research in Brazil and in the world: a survey of two premier software engineering conferences. *J. Syst. Softw.* **86**(4), 951–969 (2013)
4. Travassos, G.H.: Forrest Shull, Jeffrey Carver, and Victor Basili. *Reading techniques for OO design inspections*. Technical report (2002)
5. Baier, C., Katoen, J.-P.: *Principles of Model Checking*. MIT Press, Cambridge (2008)
6. Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using branching time temporal logic. In: Grumberg, O., Veith, H. (eds.) *25 Years of Model Checking*. LNCS, vol. 5000, pp. 196–215. Springer, Heidelberg (2008). doi:[10.1007/978-3-540-69850-0_12](https://doi.org/10.1007/978-3-540-69850-0_12)
7. Queille, J.P., Sifakis, J.: Specification and verification of concurrent systems in CESAR. In: Dezanı-Cıancaglıni, M., Montanari, U. (eds.) *Programming 1982*. LNCS, vol. 137, pp. 337–351. Springer, Heidelberg (1982). doi:[10.1007/3-540-11494-7_22](https://doi.org/10.1007/3-540-11494-7_22)
8. The Object Management Group (OMG), Needham, MA, USA. *OMG Unified Modeling Language (OMG UML), Superstructure, V2.4.1* (2011)
9. dos Santos, L.B.R., de Santiago, Jr., V.A., Vijaykumar, N.L.: Transformation of UML behavioral diagrams to support software model checking. *Electron. Proc. Theor. Comput. Sci.* **147**, 133–142 (2014)
10. Eras, E.R., dos Santos, L.B.R., Santiago Júnior, V.A., Vijaykumar, N.L.: Towards a wide acceptance of formal methods to the design of safety critical software: an approach based on UML and model checking. In: Gervasi, O., Murgante, B., Misra, S., Gavrilova, M.L., Rocha, A.M.A.C., Torre, C., Taniar, D., Apduhan, B.O. (eds.) *ICCSA 2015*. LNCS, vol. 9158, pp. 612–627. Springer, Cham (2015). doi:[10.1007/978-3-319-21410-8_47](https://doi.org/10.1007/978-3-319-21410-8_47)
11. Fondazione Bruno Kessler. *NuSMV home page* (2015)
12. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: *Proceedings of the International Conference on Software Engineering*, pp. 411–420. ACM, New York (1999)
13. Modeliosoft. *Modelio open source community* (2011)
14. Weaver, J.L., Gao, W., Chin, S., Iverson, D., Costa, A.G.M.: *Plataforma Pro JavaFX Desenvolvimento de RIA para Dispositivos Móveis e para Área de Trabalho por Scripts com a Tecnologia Java*, p. 619. *Ciência Moderna LTDA*, Rio de Janeiro (2010)
15. GRAPHVIZ. *Graph visualization software* (2017)

16. Gansner, E., Koutsofios, E., North, S.: Drawing graphs with dot: dot users manual (2006)
17. JRSOFTWARE. Inno setup (2016)
18. Wohlin, C., Runeson, P., Host, M., Ohlsson, M.C., Regnell, B., Wesslen, A.: Experimentation in Software Engineering: An Introduction (2000)
19. Campanha, D.N., Souza, S.R.S., Maldonado, J.C.: Mutation testing in procedural and object-oriented paradigms: an evaluation of data structure programs. In: 2010 Brazilian Symposium on Software Engineering (SBES), pp. 90–99. IEEE (2010)
20. Shapiro, S.S., Wilk, M.B.: An analysis of variance test for normality (complete samples). *Biometrika* **52**(3–4), 591–611 (1965)
21. Stephens, M.A.: Tests Based on EDF Statistics. Marcel Dekker, New York (1986)
22. Balera, J.M., Santiago Júnior, V.A.: A controlled experiment for combinatorial testing. In: Proceedings of the 1st Brazilian Symposium on Systematic and Automated Software Testing (SAST 2016), pp. 1–10. ACM (2016)
23. Kohl, M.: Introduction to Statistical Data Analysis with R. Bookboon.com, London (2015)
24. Taba, N., Ow, S.: A new model for software inspection at the requirements analysis and design phases of software development. *Int. Arab J. Inf. Technol. (IAJIT)* **13**(6), 51–57 (2016)
25. Coskun, M.E., Ceylan, M.M., Yigitözü, K., Garousi, V.: A tool for automated inspection of software design documents and its empirical evaluation in an aviation industry setting. In: 2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops (ICSTW), pp. 287–294. IEEE (2016)
26. Dautovic, A., Plösch, R., Saft, M.: Automated quality defect detection in software development documents. In: First International Workshop on Model-Driven Software Migration (MDSM 2011), p. 29 (2011)
27. Sinha, A., Sutton, Jr., S.M., Paradkar, A.: Text2Test: automated inspection of natural language use cases. In: 2010 Third International Conference on Software Testing, Verification and Validation (ICST), pp. 155–164. IEEE (2010)
28. Li, M., Liu, S.: Tool support for rigorous formal specification inspection. In: 2014 IEEE 17th International Conference on Computational Science and Engineering (CSE), pp. 729–734. IEEE (2014)
29. Valentim, N.M.C., Rabelo, J., Oran, A.C., Conte, T., Marczak, S.: A controlled experiment with usability inspection techniques applied to use case specifications: comparing the MIT 1 and the UCE techniques. In: 2015 ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS), pp. 206–215. IEEE (2015)
30. Bernárdez, B., Genero, M., Durán, A., Toro, M.: A controlled experiment for evaluating a metric-based reading technique for requirements inspection. In: Proceedings of the 10th International Symposium on Software Metrics, pp. 257–268. IEEE (2004)