Formal Verification of UML-based Software

Luciana Brasil Rebelo dos Santos[1], **Valdivino Alexandre de Santiago Júnior**[2], **Nandamudi Lankalapalli Vijaykumar**[3]

[1]Programa de Doutorado em Computação Aplicada – CAP
Instituto Nacional de Pesquisas Espaciais – INPE

[2]Coordenação Geral de Ciências Espaciais e Atmosféricas – CEA
Instituto Nacional de Pesquisas Espaciais – INPE

[3]Laboratório Associado de Computação e Matemática Aplicada – LAC
Instituto Nacional de Pesquisas Espaciais – INPE

{luciana.santos,vijay}@lac.inpe.br {valdivino}@das.inpe.br

Abstract. *Formal Verification methods, such as Model Checking, are best applied in early stages of system design, when costs are low and benefits can be high, increasing the quality of systems, when they are completed. The Unified Modeling Language - (UML) is currently accepted as the de facto standard for modeling (object-oriented) software, and its use is increasing in the aerospace industry. This work describes how UML diagrams created in the early phases of software development, such as sequence, activity, state machines (variation of Harel's Statecharts) diagrams, can be transformed into a finite-state model to support Model Checking of UML-based software. The results of this work will improve SOLIMVA, a methodology initially developed to generate model-based system and acceptance test cases considering Natural Language requirements artifacts.*

Key words: *Formal Verification, Model Checking, UML, SOLIMVA Methodology.*

1. Introduction

Critical systems require high reliable software, and it is essential to ensure that the software has the fewest number of defects when is released for use. Software Assurance (SA), according to the National Aeronautics and Space Administration (NASA) [NASA 2009], includes several disciplines, to name a few: Software Quality; Software Safety; Software Reliability; Software **Verification and Validation**; and Software Independent **Verification and Validation**. Hence, Verification and Validation (V&V) play a key role of getting quality and has been gaining much importance in the academia and private sector. V&V activities are usually time-consuming, specially if complex systems are considered. Techniques are developed to facilitate and make the efforts easier with these tasks.

Formal methods offer a large potential to obtain an early integration of verification in the design process, and to provide more effective verification techniques. Besides, formal verification methods, such as Model Checking, are best applied in early stages of system design, when costs are low and benefits can be high, increasing the quality of systems. Adoption of formal methods will be easier when they can be applied

within standard development processes and when they are based on standard notation [KNAPP and MERZ 2002].

The Unified Modeling Language - (UML) is currently accepted as the *de facto* standard for modeling (object-oriented) software, and its use is increasing in the aerospace industry. It presents diagrams that represent the static structure of a system, and also defines diagrams to model the dynamic behavior of systems. In particular, dynamic aspects of system behavior can be specified by interactions (i.e. sequence diagrams). UML behavioral state machines (variant of Harel's Statecharts) and activity diagrams give a view of the system that is associated with instances of classes. These types of diagrams represent complementary views of the system, but, at the same time, hide redundant descriptions of the same aspects of the system. This gives the opportunity for V&V techniques to ensure the consistency of these descriptions [KNAPP and MERZ 2002]. Nevertheless, V&V of complex software developed according to UML is not trivial due to complexity of the software itself, and the several different UML models/diagrams that can be used to model behavior and structure of the software.

Therefore, this work proposes the automated translation of UML diagrams, used to model the behavior of the system, into finite-state model to support Model Checking of UML-based software. Considering "properties" generated from use case diagrams, which represent the requirements and the "finite-state model" automatically translated from the behavioral diagrams (sequence, activity, and state machines), Model Checking can be used to ensure that the behavior of the system satisfies the requirements, that is, whether the property holds for all states in the finite-state model.

It is important to mention that such a model will have a unified view of different perspectives of behavioral modeling of the system obtained by using various UML diagrams. Besides, this work extends the SOLIMVA methodology [SANTIAGO JÚNIOR 2011] [SANTIAGO JÚNIOR and VIJAYKUMAR 2012] initially developed to generate model-based system and acceptance test cases considering Natural Language requirements artifacts. By including Formal Verification in the SOLIMVA methodology, it completes a full cycle, addressing not only Testing and Inspection but also Formal Verification.

This work is organized as follows. Section 2 presents the proposal itself. Section 3 shows a case study and preliminary results. Final remarks are in Section 4.

2. Proposal for Model Checking UML-based software

In this paper, verification consists of sequence of scenarios to be checked. Basically, scenarios focus on how the system behaves to implement its functionalities. In this work, a scenario is considered an instance of a use case, i.e., one of the paths a use case can have. The core issue of the proposed approach is to provide means for verification based on UML documentation. The properties are extracted from requirements, which, in turn, are taken from use cases. The system view used to compose the model is captured from three different behavioral diagrams of UML. Specifically, the sequence, activity, and behavioral state machines diagrams are considered.

Sequence diagram belongs to the class of interaction diagrams, which describe how the objects of a system may interact by exchanging events/messages. Activity diagrams

model the control and object flows from one activity to another activity. They evaluate better the conditions by which the instances come to certain decisions. Behavioral state machine diagrams specify the states an object can be in and describe its reactions to incoming events. These type of diagrams represent complementary views of system behavior and are often used in different phases of software specification and design. Based on the behavioral UML diagrams discussed above, a finite-state model will be generated to be checked, in order to detect problems in the system being analyzed.

The extension of SOLIMVA methodology to address Formal Verification is illustrated in the activity diagram of Figure 1. The new activities created to address Formal Verification of UML-based software are shown in dashed line of Figure 1. It is worth mentioning that the same activities, with the same features, present in older versions of the SOLIMVA methodology are also present in the new version, as well as the workflow, that is the same. The only difference is that it is possible to execute the new activities in parallel with the older activities. In practical terms, this extension of SOLIMVA proposes that the activities of testing/inspection and Formal Verification can be performed independently by different teams and even at different phases of the software development lifecycle.

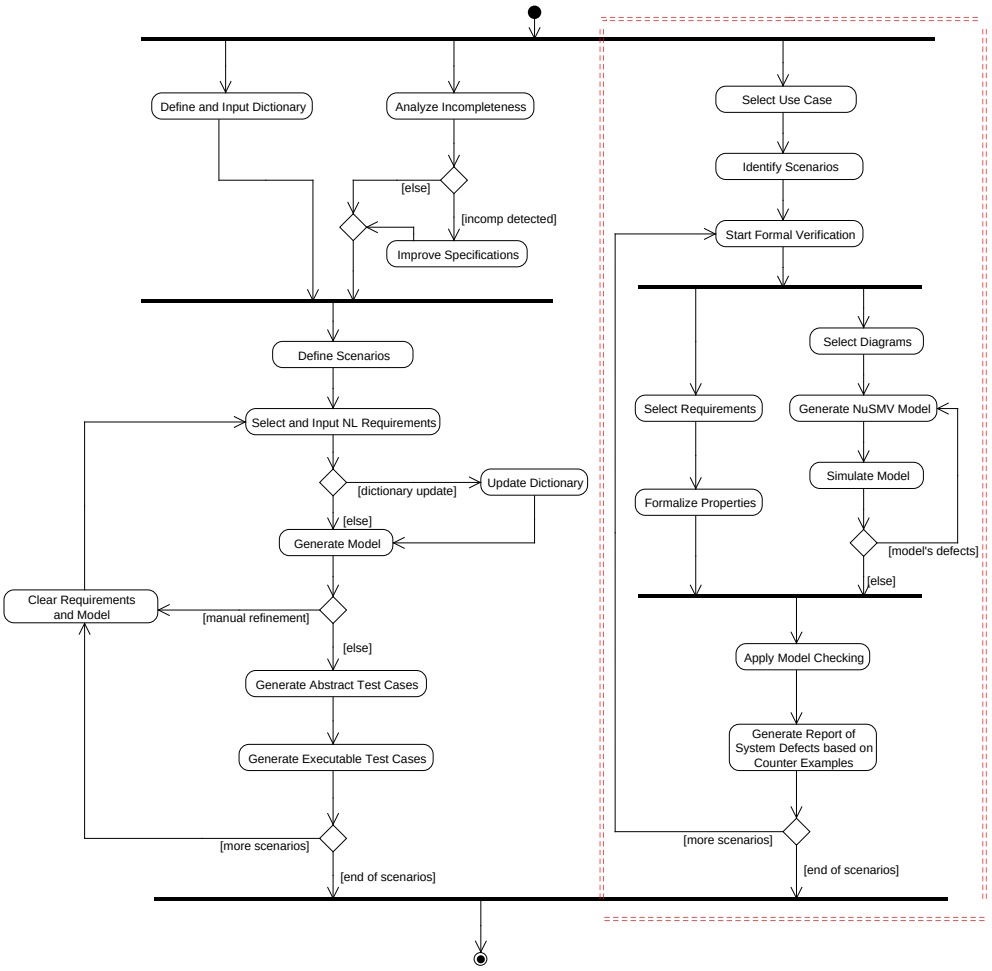

Figure 1. SOLIMVA Methodology Extension

The first new activity in the new workflow is *Select Use Case*. In this step, the user

selects the use cases which are representative and must be checked. After a use case is selected, various scenarios can be identified. The user must select the relevant ones. This is the second activity: *Identify Scenarios*. Once the scenario is identified, it is time to begin, in fact, the Formal Verification, which is the third activity. The activity *Start Formal Verification* means that at this moment, everything has to be prepared to start Model Checking, and therefore all the formalizations have to be made, as well as the system model to be checked must be generated. This can be observed in the next activities, which can be executed in parallel.

The activity *Select Requirements* is where the user should identify the suitable requirements which will be verified in the system model during the Model Checking process. Once the requirements are selected, it is time to formalize the properties. Here, the properties are formalized using temporal logic, such as LTL or CTL. Dwyer [DWYER et al. 1999] proposed a system of property specification patterns for finite-state verification. They proposed 8 patterns and 5 pattern scopes. Hence, based on a requirement, one identifies a pattern and the scope within the pattern that mostly characterize such requirement. Having decided which is the pattern and scope, they proposed a template to generate the properties in LTL, CTL, and Quantified Regular Expressions.

The activities related to model creation, which are *Select Diagrams*, *Generate NuSMV Model*, and *Simulate Model*, can be executed in parallel with the *Select Requirements* and *Formalize Properties* activities. The first activity related to model creation, *Select Diagrams*, is when the respective diagrams that represent the behavior relating to the use case selected are identified. Eventually, the use case selected does not have a representation in all the diagrams that this approach is intended to use. For example, a use case can be associated with a sequence and activity diagram but not with a state machine diagram; or there exists only the sequence and state machine diagram for that use case but not the activity. In these cases, the model must be generated from the available diagrams for that use case.

In the activity *Generate NuSMV Model* the finite-state model is generated from the diagrams. Here, the UML diagrams are automatically translated into a finite-state model. NuSMV is the model checker used in this work. It is open, flexible, and a documented platform. NuSMV supports both LTL and CTL temporal logic. The model of the system is simulated prior to Model Checking (*Simulate Model* activity) in order to get rid of modeling defects. If more model defects are identified then the workflow returns to the *Generate NuSMV model* activity and restart from this point.

When there is no more remaining defect in the model and all properties are created, Model Checking can be applied (*Apply Model Checking* activity). Detected system defects are then reported (*Generate Report of System Defects* activity). Having generated the report for a single scenario, the user starts again selecting the next scenario. This process must be repeated until there is no more scenario and the process is finalized.

3. Case Study: ATM example

This section presents a case study to illustrate the new activities proposed in the extension of the SOLIMVA methodology. Consider the automated teller machine (ATM) classical example. In accordance with our approach, the first activity is *Select Use Case*. Let's consider the use case *Session Use Case* that states [BJORK 2012]: "A session is started when

a customer inserts an ATM card into the card reader slot of the machine. The ATM pulls the card into the machine and reads it. (If the reader cannot read the card due to improper insertion or a damaged stripe, the card is ejected, an error screen is displayed, and the session is aborted.) The customer is asked to enter his/her Personal Identification Number (PIN), and is then allowed to perform one or more transactions. After each transaction, the customer is asked whether he/she would like to perform another transaction. When the customer is through performing transactions, the card is ejected from the machine and the session ends. The customer may abort the session by pressing the Cancel key when entering a PIN or choosing a transaction type".

Two possible scenarios for this use case are: "the customer is allowed to perform transactions". The other scenario is "the customer is not allowed to perform transactions".

Several requirements can be observed in the identified scenarios. Assuming that the following requirements were chosen to be checked:

1. Requirement: *the customer can perform transactions only if he/she has a valid card and a valid personal identification number (PIN)*. The requirement formalized as a property is: *the ATM cannot allow the user to request an operation if either the card or the PIN is not valid*. This property can be formalized using the **Absence Pattern and Scope After Q** proposed by [DWYER et al. 1999], in LTL, as follows:
   $$\Box((\neg CardOK || \neg PinOK) \rightarrow \Box(\neg y))$$
   where *CardOK* means the card is valid, *PinOK* means the identifier is valid, and *y* means "transaction performed".
2. Requirement: *The customer can cancel the operation at any time*. The property is formalized as: *whenever the client push the cancel button, the card must be returned to him and all the operation is aborted*. This property can be formalized using the **Response Chain Pattern and Globally Scope** proposed by [DWYER et al. 1999], in LTL, as follows:
   $$\Box(ButtonCancelled \rightarrow \Diamond(CardEjected \wedge \bigcirc\Diamond x))$$
   where *ButtonCancelled* means the customer pressed the key buttonCancel, *CardEjected* means the card is returned to the customer, and *x* means "operation aborted".

To create the model, the sequence, activity and behavioral state machine diagrams must be selected. Figure 2 shows, respectively, the sequence and behavioral state machine diagrams for the ATM system. There is no activity diagram for the use case selected, so only the sequence and behavioral state machine will be used.

3.1. Preliminary Results

Considering the diagrams and the properties to be checked, the finite-state model visualized in Figure 3 is generated. It is important to note that the finite-state model can be simpler than the UML diagrams, even though it has been built based on these UML behavioral diagrams. Only the states and variables that are really important for analyzing the specified scenario are considered.

Three main variables that characterize the model were identified. The first one is *act*, which represents the physical actions that can be performed. It is defined as follows:

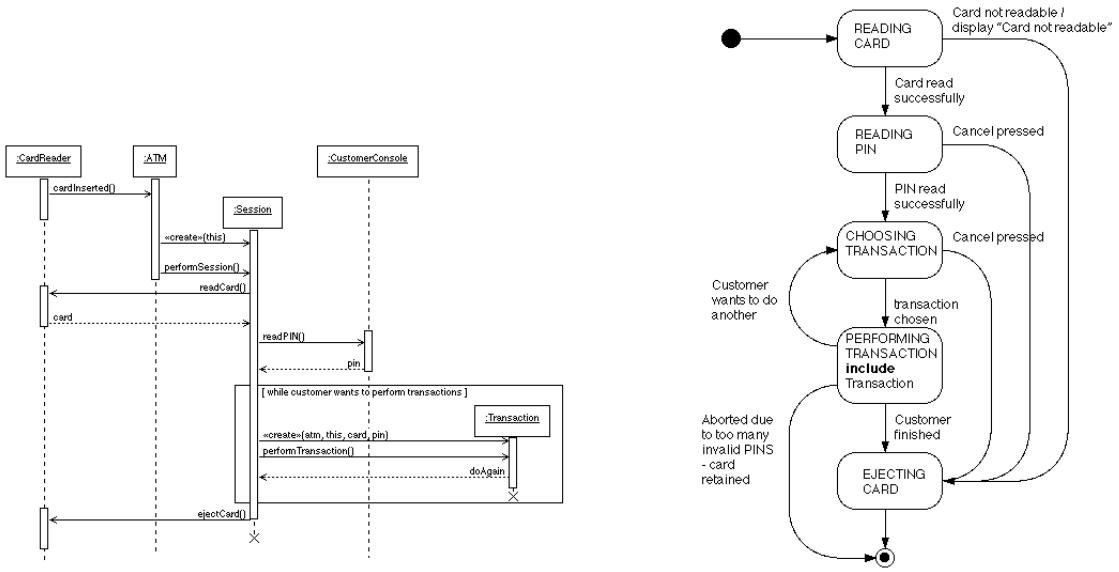

Figure 2. Sequence and Bahavioral State Machine Diagrams for Example ATM System. FONTE: [BJORK 2012]

$$act = \{noAction,\ cardInserted,\ ButtonCancelled,\ CardEjected\},$$

where *noAction* means that no actions were taken yet; *cardInserted* means the customer has inserted the card into the machine; *ButtonCancelled* means the customer pressed the key CancelButton; *CardEjected* means the card was returned to the customer.

The second variable is *rsp*, which represents the machine responses. It is defined as follows:

$$rsp = \{noResponse,\ CardOK,\ CardError,\ PinOK,\ PinError\},$$

where *noResponse* means there is no response to give; *CardOK* means the card is authenticated; *CardError* means the card authentication has failed; *PinOK* means the PIN is authenticated; *PinError* means the PIN authentication has failed.

The third variable is *Status*, which represents the status of the system in every moment. It is defined as follows:

$$Status = \{waitingCard,\ waitingAuthCard,\ waitingAuthPin,\ readyTransaction,\ TransactionPerformed,\ TransactionFinished\},$$

where *waitingCard* means the system is available, waiting the card; *waitingAuthCard* means the system is waiting for the card authentication; *waitingAuthPin* means the system is waiting for the PIN authentication; *readyTransaction* means the system is ready to perform any transaction; *TransactionPerformed* means a transaction has been performed by the customer; *TransactionFinished* means the action was finished. Note that, in this status, it is possible that the customer has been performed a transaction, but it is also possible he/she has not been performed any transaction, as the key CancelButton may have been pressed.

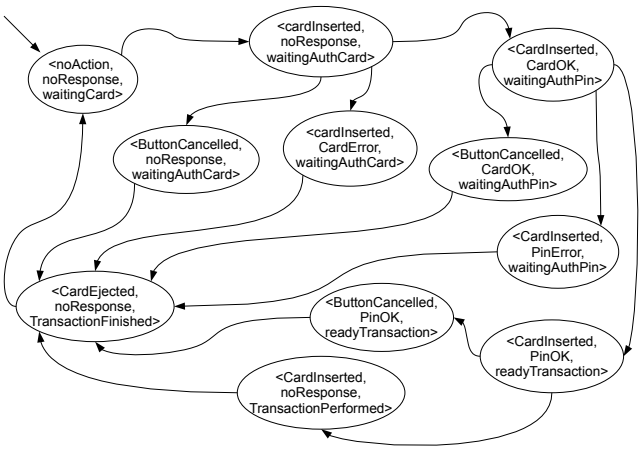

Figure 3. Finite-State Model for Example ATM System

The finite-state model contains eleven states. Each state is characterized by the values of the variables: the states are labeled according to the variables. In each state there are the values that the variables take. For instance, in the upper leftmost state: *act = noAction, rsp = noResponse, and Status = waitingCard*. A very simple choice is to let each value of each variable acts as an atomic proposition. Hence, *noAction* is considered as an atomic proposition, it can be *true* or *false*, as well as all other values of the variables [SANTIAGO JÚNIOR 2011]. In this sense, the set of atomic propositions, *AP*, is:

> *AP* = {*noAction, cardInserted, ButtonCancelled, CardEjected, noResponse, CardOK, CardError, PinOK, PinError, waitingCard, waitingAuthCard, waitingAuthPin, readyTransaction, TransactionPerformed, TransactionFinished* }

Figure 4 shows the NuSMV source code of the model presented in Figure 3. NuSMV describes the finite-state model as a set of variables and predicates on these variables. In the illustrated example, there are three enumerative variables, which represent the actions to be performed, the responses the machine may give, and the status that the system can assume according to the inputs. The output is the resulting status. *Init* keyword specifies the initial status, representing that the system is avaiable for use. *Next* specifies the possible status the system can assume, according to the actions and responses it receives.

4. Final Remarks

This work presented an extension of the SOLIMVA methodology, which aims to automatically translate UML diagrams into finite-state model to support Model Checking of UML-based software. The workflow basically consists of identifying scenarios, which are instances of a use case. Then, selecting requirements (and further the properties) and identifying the behavior diagrams relating to the scenario chosen are the next activites that must be performed. The properties are formalized using LTL or CTL and the finite-state model is generated based on the sequence, activity, and behavioral state machine diagrams. In this step, Model Checking is applied and a report of system defects is generated.

```
MODULE main
VAR
 act:   {noAction, cardInserted, ButtonCancelled, CardEjected};
 rsp:   {noResponse, CardOK, CardError, PinOK, PinError};
 Status:    {waitingCard, waitingAuthCard, waitingAuthPin,
            readyTransaction, TransactionPerformed,
            TransactionFinished};

ASSIGN
 init(Status) := waitingCard;
 next(Status) := case
  Status = waitingCard & rsp = noResponse & act = CardInserted : waitingAuthCard;
  Status = waitingAuthCard & rsp = CardOK & act = CardInserted : waitingAuthPin;
  Status = waitingAuthPin & rsp = PinOK & act = CardInserted : readyTransaction;
  Status = waitingAuthPin & rsp = PinOK & act = CardInserted : readyTransaction;
  Status = readyTransaction & rsp = noResponse & act = CardInserted :
                                                    TransactionPerformed;
  Status = TransactionPerformed & rsp = noResponse & act = CardEjected :
                                                    TransactionFinished;
  Status = TransactionFinished & rsp = noResponse & act = noAction : waitingCard;

  act = ButtonCancelled : TransactionFinished;
  rsp = CardError : TransactionFinished;
  rsp = PinError : TransactionFinished;

  TRUE : Status;
  esac;
```

Figure 4. ATM System represented as NuSMV Model

The preliminary results presented in this initial study showed that it is viable to achieve the finite-state model and its NuSMV source code, based on the behavioral UML diagrams, as well as the properties, extracted from the requirements presented in the use cases. This first study was accomplished manually. The future directions are the development of the supporting tool for automating the translation of UML diagrams into finite-state model.

References

BJORK, R. C. (2012). A complete example of object-oriented analysis, design, and programming applied to a moderate size problem: the simulation of an automated teller machine.

NASA (2009). Nasa software assurance: Software assurance definitions.

SANTIAGO JÚNIOR, V. A. (2011). *SOLIMVA: A methodology for generating model-based test cases from natural language requirements and detecting incompleteness in software specifications*. Thesis (phd in applied computing), Instituto Nacional de Pesquisas Espaciais (INPE), São José dos Campos, SP, Brazil. (sid.inpe.br/mtc-m19/2011/11.07.23.30-TDI).

SANTIAGO JÚNIOR, V. A. and VIJAYKUMAR, N. L. (2012). Generating model-based test cases from natural language requirements for space application software. *Software Quality Control*, 20(1):77–143.

DWYER, M., AVRUNIN, G., and CORBETT, J. (1999). Patterns in property specifications for finite-state verification. In *Software Engineering, 1999. Proceedings of the 1999 International Conference on*, pages 411–420. IEEE.

KNAPP, A. and MERZ, S. (2002). Model checking and code generation for uml state machines and collaborations. In *Proceedings of 5th Workshop on Tools for System Design and Verification, Technical Report*, volume 11, pages 59–64.