Transformation of UML Behavioral Diagrams to Support Software Model Checking

Luciana Brasil Rebelo dos Santos Valdivino Alexandre de Santiago Júnior
Nandamudi Lankalapalli Vijaykumar
Laboratório Associado de Computação e Matemática Aplicada - LAC
Instituto Nacional de Pesquisas Espaciais - INPE
São José dos Campos - SP, Brazil
luciana.santos@lac.inpe.br valdivino.santiago@inpe.br vijay.nl@inpe.br

Unified Modeling Language (UML) is currently accepted as the standard for modeling (object-oriented) software, and its use is increasing in the aerospace industry. Verification and Validation 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. This paper presents an approach to transform up to three different UML behavioral diagrams (sequence, behavioral state machines, and activity) into a single Transition System to support Model Checking of software developed in accordance with UML. In our approach, properties are formalized based on use case descriptions. The transformation is done for the NuSMV model checker, but we see the possibility in using other model checkers, such as SPIN. The main contribution of our work is the transformation of a non-formal language (UML) to a formal language (language of the NuSMV model checker) towards a greater adoption in practice of formal methods in software development.

1 Introduction

Verification and Validation (V&V) play a key role of getting quality and has been gaining much importance in the academia and private sector. Formal methods offer a large potential to obtain an early integration of verification in the design process, and to provide more effective verification techniques [2]. However, formal methods require mathematical background and their use is restricted, as users prefer the simplicity of other notations, rather than more formal means. Thus, adoption of formal methods will be easier when they can be applied within standard development process and when they are based on standard notation [20].

Unified Modeling Language - (UML) [18] is currently accepted as the standard for modeling (object-oriented) software. 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 and the system requirements.

This paper presents an approach related to Model-Driven Development and Formal Verification. Our approach transforms up to three different UML behavioral diagrams (sequence, activity, behavioral state machines) into a single Transition System (TS) to support Model Checking of software developed in accordance with UML. We consider properties generated from use case descriptions, which represent the

Submitted to:
FESCA2014

requirements, and the TS translated from the behavioral diagrams. Our verification essentially consists of sequence of scenarios to be checked. Besides, the single TS has a unified view of different perspectives of behavioral modeling of the system, obtained by using various UML diagrams. The main contribution of our work is the transformation of a non-formal language (UML) to a formal language (language of the NuSMV model checker) towards a greater adoption in practice of formal methods in software development.

The rest of the paper is organized as follows. In the next section we explain our approach to apply Formal Verification. This section also shows the details for constructing the TS, and for generating the model checker notation. In Section 3 we apply our approach to a case study and shows preliminary results. Related work is discussed in Section 4. Finally, Section 5 concludes the paper.

2 Transforming UML behavioral diagrams into Transition Systems (TS)

A prerequisite for model checking is a property to be checked and a model of the system under consideration. In Figure 1, we show our approach aiming at Model Checking of software developed in accordance with UML. In the following, we explain the main activities that the workflow performs:

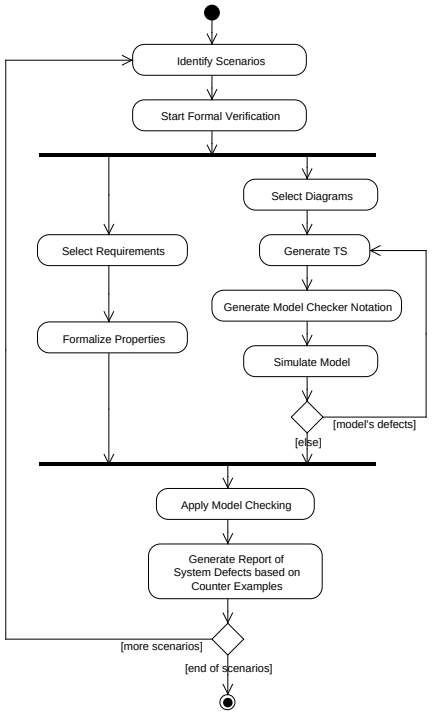

Figure 1: Workflow of the proposed approach

(i) **identify scenarios** by looking at use case models. A use case can be viewed as a scenario. Each scenario is a set of related subscenarios tied together by a common goal. The mainline sequence ('main success scenario' [5]) and each of the variations ('extensions and sub-variations') are the scenarios identified by our approach; (ii) **formalize properties**. For each selected scenario, we extract requirements from the textual description of use cases. After that, we formalize properties by means of specification

patterns [9]; (iii) **generate TS**. Based on the available UML behavioral diagrams, we generate a single TS (finite-state model). This is the main activity to achieve the objective of our work. As will be explained later, our approach does not demand that all three UML behavioral diagrams (sequence, activity, behavioral state machines) exist: it is enough to have the sequence diagram and one of the other two to generate the TS; (iv) **generate model checker notation**. Then, we translate the created TS to a model checker. Our first idea is to use the NuSMV model checker [13]; (v) Finally, we **apply Model Checking** to realize about defects in the behavioral description of the system represented by the UML diagrams. We repeat activities from (ii) to (v) for each selected scenario. Also note that activities (ii) and (iii)/(iv) may be accomplished in parallel.

In the next subsection we will detail the main activity of our approach, i.e. a solution to generate a single TS based on UML behavioral diagrams. We will focus on activity (iii) described above. It works in two phases: in the first phase, the single TS from each one of the diagrams is generated separately. Due to space constraints, we refrain to show the construction of individual TSs from each one of the diagrams. We focus on the second phase, which shows how to obtain the definitive/unified TS from the combination of the individual TSs. Finally, in Section 2.2 we present directives for the generation of the NuSMV notation (activity (iv)).

2.1 The Unified Transition System

In this subsection, we show how to generate the final/unified TS. Some definitions and notations used in our approach are given in the sequence. For brevity, we denote Sequence diagram as SD, Behavioral State Machine diagram as SMD, and Activity diagram as AD. As previously mentioned, a scenario is identified by looking at use case models. We take for granted that every use case has at least one SD describing it. We also assume that a scenario must have at least two diagrams describing it: one is mandatory, the SD , and the other one can be any of the other two diagrams.

A state in the TS is identified by a tuple $\ll (Message, State, Activity),\ g0, g1, ..., gn \gg$ where *Message* is from SD; *State* is from SMD; and *Activity* from AD; *g0,g1,...,gn* represent all the existing guards in the diagrams along with their respective values. At the beginning, all guard values are assigned to '*dc*', which means 'do not care', because at the beginning we don't know the guard values. If there are parallel messages, states, or activities, an '*and*' is inserted in the tuple, such as $\ll (Msg1 and Msg2, State1 and State2, Activity1 and Activity2), ... \gg$, which means that *Msg1* and *Msg2* are sent in parallel *State1* and *State2* are parallel states, as well as *Activity1* and *Activity2*. When a diagram is missing, its part in its corresponding state in the TS is substituted by '-'. It also happens when there is some behavior that is not modeled in one of the diagrams, that is, when the diagrams are incoherent or when there is a more detailed description in one of them. In these situations, states are described as $\ll (Message, State, -), g0, g1, ..., gn \gg$ or $\ll (Message, -, State),\ g0, g1, ..., gn \gg$, or *≪(Message-—-—),g0,g1,...,gn≫*.

The main challenge to combine the three diagrams is to define the correspondence among messages, events, and activities. For instance, an event for a transition of the SMD can be a message of the SD, while the message can be a call to an activity modeled by an AD. Nevertheless, finding where a specific message of the SD is modeled in the SMD or in the AD is very hard. Instead, we build the final TS using the individual TSs. We look over the TSs starting from their initial states. The correspondence among states of each TS is assigned by using the flow of transitions. At each iteration, we parse the three TSs and construct the new states by taking the next possible transitions in each TS.

The flow of transitions, alone, is not enough to construct the final TS, as each TS contains multiple paths. In addition, we use guard values to help find the correspondence among states. As we have already

stated, the SD is mandatory and conducts the diagrams merging. The basic elements that delineate the possibility of paths in the SDs are guards in the combined fragments. Thus, we adopt the guard values along with the flow of transitions to construct the final TS states. If we think of an algorithm, at each iteration, the algorithm seeks the next possible transitions in each TS and its guard values. To find out states that have the same guard values, we have created the *gvs* (guard value structure) which represents the values of each one of all available guards on each state. Every time, for each possible next transitions of the TSs created from the SD, SMD, and AD, their *gvs* are verified and a match of its values is created. Thereafter, new states are generated.

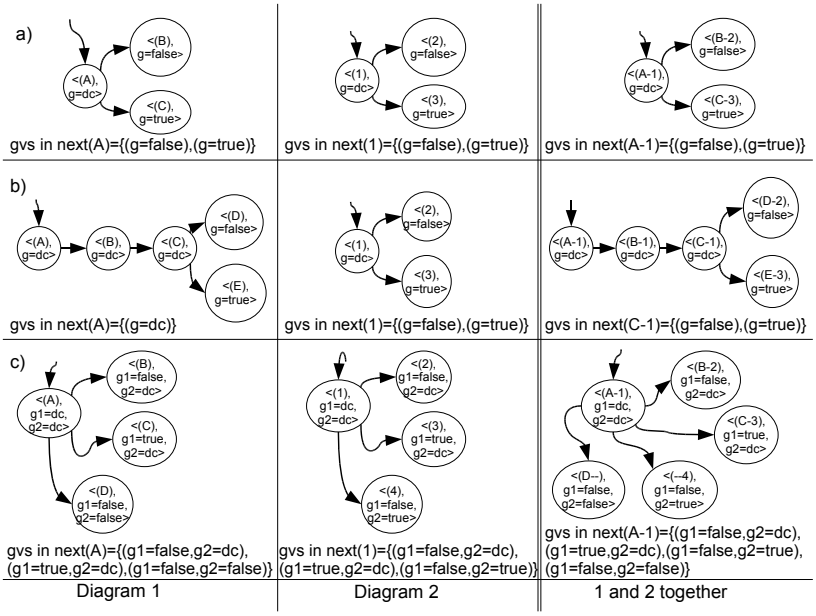

Figure 2: Possible situations to generate the unified TS and its respective gvs

Figure 2 shows the main possible situations that may occur to generate a state in the final TS. The diagram in column 3 is the combination of the diagrams in columns 1 and 2. Situation a) is straightforward to see. Situation b) shows how the algorithm handles cases when in at least one of the next possible states, the value of its *gvs* is the same of the current *gvs* value. This occurs because the diagrams do not model the same behavior during their evolution. An SMD may contain more details than an SD, and so on. In this case, the TSs have different lengths. Situation c) shows an example of inconsistent states, due to inconsistent diagrams. As previously discussed, this may happen when one of the diagrams models a behavior that is not modeled in the other diagram and vice versa. New transitions are created, as many as necessary to cover all possibilities of *gvs* values.

There is still a situation we did not mention in Figure 2: TSs with no guards. Here we have two possibilities. The first one is the case of all diagrams have no guards. We assume that when a diagram has no guards, this means that there is no multiple paths. For example, a sequence diagram with no combined fragments, or with only parallel combined fragment. All other combined fragments require guards. Or an activity diagram with no decision node. In the absence of multiple paths, it is quite easy to construct the final TS: we only need follow the flow of transitions. The second one is the case of a diagram has guards and the other one does not. At the beginning, the algorithm seeks all guards in all diagrams. Suppose TS1 has guards *g1* and *g2* and TS2 has no guards. The *gvs* will be composed by

$\{g1, g2\}$ and it is attributed to TS2. As originally TS2 did not contain *g1,g2*, their values are assigned as "*dc*" and will never change. Thenceforth the situations are covered in Figure 2.

2.2 Generation of model checker notation

After its creation, the unified TS can be used to systematically generate its corresponding encoding into the model checker input language by constructing declarative divisions [7]. It is important to emphasize that once a formal unified TS was generated from UML behavioral diagrams, we see the possibility of transforming it into several different languages of available model checkers such as SPIN [11], UPPAAL [4], and NuSMV [13].

In our current approach, we chose the NuSMV model checker because it is open source, it has a widespread use in academia, and it accepts properties formalized not only in Computation Tree Logic (CTL) but also in Linear Temporal Logic (LTL) [2], two logics that are well known and have mapping defined in the specification patterns [9].

Considering the NuSMV model checker, declaration of variables is relatively easy. One variable we call *State* which is related to the element $\ll$*(Message-State-Activity)*$\gg$ of the tuple that identifies a certain state of the TS. In addition, there will be as many variables as the guards identified within the UML behavioral diagrams, i.e. $g0$ is transformed into a variable $vg0$, $g1$ into a variable $vg1$, and so on. All these variables derived from the guards will be enumerated with the following values: $\{dc, false, true\}$. The dc value is important because in many occasions and especially at the beginning of the behavioral system modeling, the values of certain guards or do not care or do not make sense be true or false. Another remark is that we use only state variables of NuSMV.

The initial value of the *State* variable of the final TS in NuSMV is always *Start-Start-Start*. Note that this is not the "full" state characterization of the initial state of the NuSMV model because we depend on the values of the variables representing the guards to know this. To generate the model in NuSMV we simply follow the transitions of the unified TS, making appropriate assignments to the *State* variable and for each variable derived from the guards.

3 Preliminary Results

This section presents a case study to illustrate the feasibility of our approach. We consider more thoroughly the ATM case study, where the ATM interacts with a customer via a specific interface and communicates with the bank over an appropriate communication link. The initial part of the sequence, activity and behavioral state machine diagrams for ATM are illustrated in Figure 3.

In accordance with our approach, we must **identify the scenarios**, observing the use cases. Then, we **start Formal Verification**. For this, we **generate TS** and **select requirements**. Applying the suggested approach manually on the three diagrams, we achieve the TS shown in Figure 4. Actually, Figure 4 exhibits only part of the unified TS (only the part related with Figure 3). In total, we have 756 states, 86 of which are reachable states, when running NuSMV model checker. Each state is characterized by the values of the variables. We have identified four main variables that characterize the TS: (i) *State* = {*Start-Start-Start,...,End-Idle-End*}; (ii) *CardOk* = {*dc,false,true*}. *CardOk* represents the card validation; (iii) *PinOk* = {*dc,false,true*}. *PinOk* represents the PIN validation; and (iv) *BalOk* = {*dc,false,true*}. *BalOk* represents the available amount of the customer account. *CardOk, PinOK* and *BalOk* comes from the guards identified within the UML diagrams.

Continuing the approach, considering the use case *Perform Transaction*, we can extract two relevant

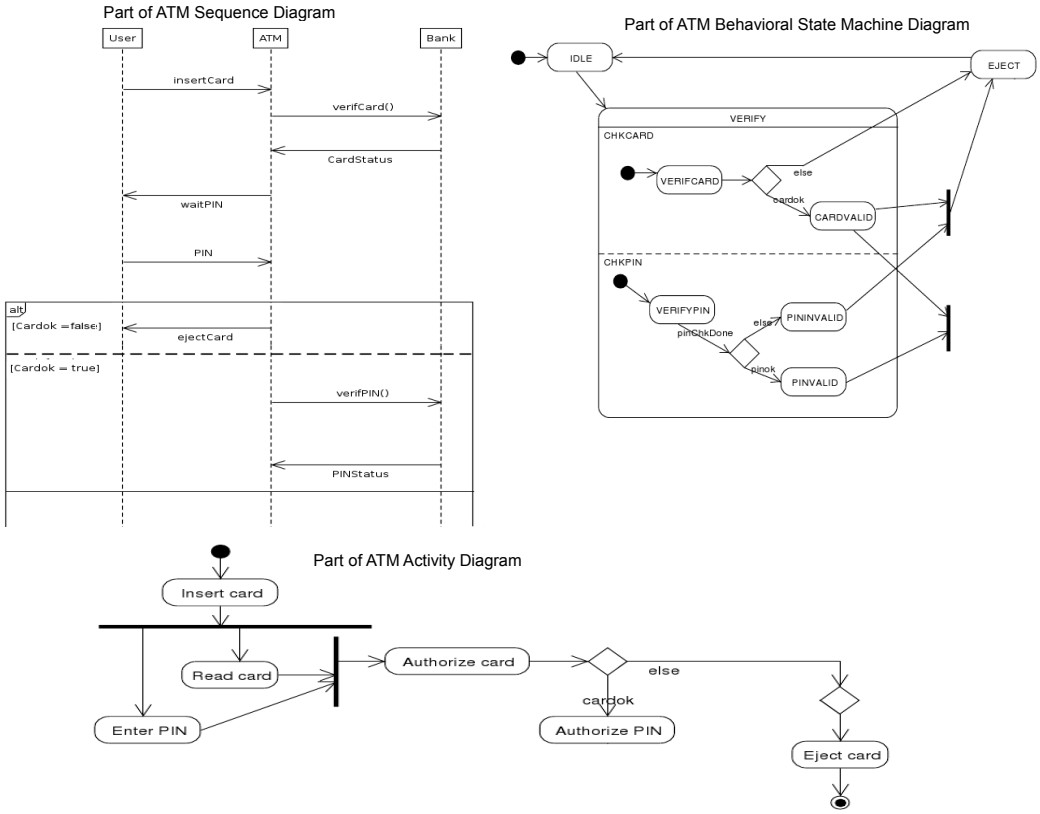

Figure 3: Initial part of UML diagrams for ATM case study

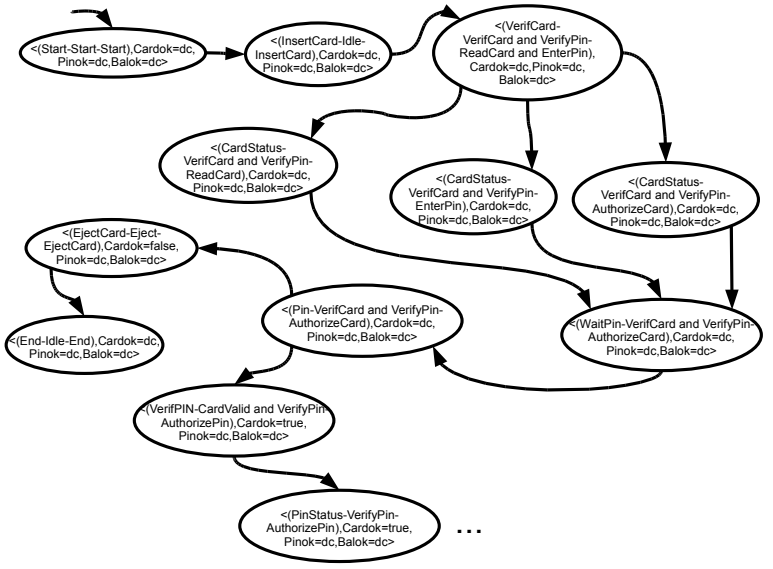

Figure 4: Part of the final TS obtained from the three diagrams

user-defined properties related to requirements. To proceed with the Formal Verification, it is necessary to **formalize the properties** to be checked. We chose Computation Tree Logic (CTL) [2] to formalize the properties. Note that the properties could be formalized using LTL as well, considering that NuSMV support such logic.

Requirement 1: *the customer can perform transactions only if he/she has a valid card and a valid personal identification number (PIN). Otherwise, he/ she can not perform any kind of transaction*. This property can be formalized using the **Absence Pattern and Scope After Q** proposed by [9], in CTL, as follows[1]:

$$AG((CardOK = false \quad | \quad PinOK = false) \rightarrow AG(!(State = WaitAccount - CardValidandPinValid - InitiateTransaction)))$$

CardOk and *PinOk* refer to the validation of the customer credentials. The first state on the final TS which can be reached only when *CardOk* and *PinOk* are *true* is *WaitAccount-CardValidandPinValid-InitiateTransaction*, that is, when the custumer has card and PIN valids. If we observe the individual TSs, we can see that state *WaitAccount* for the SD is the first reachable state when both *CardOk* and *PinOk* are *true*, for SMD *CardValidandPinValid*, and also *InitiateTransaction* for AD.

Requirement 2: *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 property can be formalized using the **Existence Pattern and Scope After Q** proposed by [9], in CTL, as follows:

$$A[!(State = InsuffFunds - Modify - ShowBalance)W((State = InsuffFunds - Modify - ShowBalance)\&AF(State = CashAdvance - Chkbal - CheckBalance))])$$

The state that allows the customer to perform a cash operation is *CashAdvance-Chkbal-CheckBalance* and the state that indicates that the available funds are insufficient is *InsuffFunds-Modify-ShowBalance*. To formalize the properties, it is necessary to see the final TS and its variables.

The next task is to **generate the model checker model** based on the previous generated final TS. The guidelines to do this were presented in Section 2.2. The model checker code will be automatically generated when the approach is implemented. A small part of the NuSMV source code is shown below for this case study.

```
MODULE main
VAR
  State:  {Start-Start-Start,InsertCard-Idle-InsertCard,
           VerifCard-VerifCardandVerifyPin-ReadCardandEnterPin,...,End-Idle-End};
  CardOk:  {dc,false,true};
  PinOk:  {dc,false,true};
  BalOk:  {dc,false,true};
ASSIGN
  init(State):= Start-Start-Start;
  next(State):= case
    State=Start-Start-Start & CardOk=dc & PinOk=dc & BalOk=dc :
           InsertCard-Idle-InsertCard;
    State=InsertCard-Idle-InsertCard & CardOk=dc & PinOk=dc
            & BalOk=dc :  VerifCard-VerifCardandVerifyPin-ReadCardandEnterPin;
          ...
  esac;
```

After applying Model Checking, the results indicate that the first propriety is satisfied. However, the second property is violated, generating a counterexample. When analyzing the counterexample, one can note that it is not possible to reach state *CashAdvance-ChkBal-CheckBalance* (where is possible to

[1] We used the NUSMV's syntax to write the property in CTL.

request a new cash advance operation) from state *InsuffFunds-Modify-ShowBalance* (where available funds are insufficient).

In this section, we have shown the feasibility of our approach, by means of a traditional case study performed manually, to generate the unified TS from the various UML behavioral diagrams, and we introduced policies to transform the TS into the NuSMV input language. Besides, after running the model checker, we have found that the behavioral modeling described in the UML diagrams does not comply with all the specified requirements.

4 Related Work

This section presents some of the research literature related to this paper, showing (not exhaustive) approaches that use Formal Verification and UML.

Knapp [20] used two complementary UML notations for the specification of dynamic system behavior - state machines and collaborations - to automatically verify whether the interactions expressed by a collaboration can indeed be accomplished by a set of state machines. The approach applied consistency checking between diagrams.

Mikk [17] and Latella [15] translated Statecharts into PROMELA, the input language of SPIN verification system. Lam [14] formally analyzed activity diagrams using NuSMV model checker. The objective was determining the correctness of activity diagrams. Eshuis [10] presented two translations from activity diagrams to NuSMV. The aim was to assess the activity diagrams from the point of view of requirements and also from the point of view of implementation, which represents the current system behavior. In [1], Anderson translates the specification of TCAS (Traffic Alert and Collision Avoidance System), which is specified in RSML (Requirements State Machine Language) into the input language of NuSMV. The objective was to investigate if Model Checking could be used in large software specifications. Dubrovin [8] implemented a tool that translates UML hierarchical state machine models to the input language of NuSMV too. Uchitel [21] proposes translation of scenarios, specified as Message Sequence Charts (MSCs), into a specification in the form of Finite Sequential Processes. This can then be fed to the Labelled Transition System Analyser model checker to support system requirements validation. All these previous studies deal with a single UML or UML-like diagram to perform Formal Verification. Rather, our approach allows to work with up to three UML behavioral diagrams. In addition, it is not clear if in the previous studies the authors used specification patterns to formalize the properties. Specification patterns provide clear guidelines to such formalization. We are proposing not only a tool to translate UML diagrams into a unified TS to support Model Checking but also a full approach to detect defects in the design of software developed in accordance with UML.

Baresi [3] developed a tool to carry out Formal Verification of UML-based models, mainly interested in the timing aspects of systems. It is composed by: static part (class diagrams); dynamic aspects and behavior are rendered through: (a) state diagrams and activity; (b) sequence diagrams; and (c) interaction overview diagrams, used to relate different sequence diagrams; Clocks (and time diagrams) are used to add the time dimension to systems. All these diagrams seem to be required to construct the approach.

Cortellessa [6] proposes a methodology Performance Incremental Validation in UML (PRIMA-UML) aimed at generating a queueing network based performance model from UML diagrams that are usually available early in the software lifecycle (use case, sequence, and deployment). Bernardi [16] translated sequence and statechart diagrams into Generalized Stochastic Petri Nets. Both works aimed analyze performance aspects of systems. Our approach is more related to functional aspects of the software product.

Calinescu [12] used a probabilistic model checker (PRISM) to verify critical systems, after changes. Verifying these software systems only at design time is insufficient, they have to be reverified after each change. They did not work with UML diagrams, but with components and deterministic finite automata.

The main motivation of our approach is the practical use of formal methods in software development, through automation. The idea is that the user feeds the tool with UML behavioral diagrams and it shows the defects. This can be done throughout the lifecycle, even before software coding. [6] suggested an interesting approach to encompass performance validation task as an integrated activity within the development process. We aim at detecting design defects within the solution, but considering functional requirements of the software product. Besides, our approach suggests to get a single vision of the system captured from three different diagrams (sequence, activity, and behavioral state machines). These diagrams represent complementary views of system behavior and are often used in different phases of software specification and design, allowing thus a wider system range to be verified. Most of the works we mentioned deal with a specific type of UML diagram. [3] seems to require an assorted number of diagrams (structural, behavioral and time diagrams), which are not always available on the documentation. Adversely, our proposal requires two diagrams as mandatory (one is sequence and another activity or behavioral state machine), which provides a higher chance of being used in real applications.

5 Conclusions

In this paper we presented an approach that, ultimately, is another initiative in order to facilitate and thus increase the use of formal methods in software real projects. We draw on two facts to the development of this work. First, UML is a language widely used in various application domains, including the aerospace one. Second, Formal Verification and formal methods in general, despite all the benefits already presented by academic community, have not seen widespread adoption as a routine part of systems development practice [22].

We presented an approach that transforms up to three UML behavioral diagrams (sequence, activity, behavioral state machines) into a single Transition System to support Model Checking. Our proposal requires only 1 diagram as mandatory (sequence) and another (activity or behavioral state machine) in order to create the TS. If there are the 3 diagrams, our approach can also generate the unified model (TS). Thus, we believe that it has a great potential to be used in real systems development because not always a significant variety of UML diagrams is within the artifacts (requirements specification, software design document) created to develop certain type of software.

Future directions follow. We are currently developing a tool to support this transformation. We have already developed the transformation from SD into a single corresponding TS, and we are working at the AD transformation. When SMD transformation is finished we will begin the construction of the final part, the unified TS generation. Another direction is to automatically identify in the UML diagrams a problem (inconsistency between diagrams, incorrect behavior) when a counterexample is detected by running NuSMV. We will dedicate efforts in transforming the unified TS to other model checkers such as SPIN and UPPAAL. More complex case studies in several domains such as space software [19] and general purpose software are in our targets.

References

[1] Richard J Anderson, Paul Beame, Steve Burns, William Chan, Francesmary Modugno, David Notkin & Jon D Reese (1996): *Model checking large software specifications*. In: *ACM SIGSOFT Software Engineering*

Notes, 21, ACM, pp. 156–166.

[2] C. Baier & J.-P. Katoen (2008): *Principles of model checking*. MIT Press, Cambridge, MA, USA. 975 p.

[3] L. Baresi, A. Morzenti, A. Motta & M. Rossi (2012): *Towards the UML-Based Formal Verification of Timed Systems*. In: *Formal Methods for Components and Objects*, Springer, pp. 267–286.

[4] G. Behrmann, A. David & K. Larsen (2004): *A tutorial on uppaal*. *Formal methods for the design of real-time systems*, pp. 33–35.

[5] Alistair Cockburn (2000): *Writing Effective Use Cases*. Addison-Wesley Professional, US. 304 p.

[6] Vittorio Cortellessa & Raffaela Mirandola (2002): *PRIMA-UML: a performance validation incremental methodology on early UML diagrams*. *Science of Computer Programming* 44(1), pp. 101–129.

[7] M. Debbabi, F. Hassaïne, Y. Jarraya, A. Soeanu & L. Alawneh (2010): *Verification and Validation in Systems Engineering*. Springer, Berlin, Heidelberg - Germany. 270 p.

[8] J. Dubrovin & T. Junttila (2008): *Symbolic model checking of hierarchical UML state machines*. In: *Application of Concurrency to System Design, 2008. 8th International Conference on*, IEEE, pp. 108–117.

[9] M. B. Dwyer, G. S. Avrunin & J. C. Corbett (1999): *Patterns in property specifications for finite-state verification*. In: *Proceedings...*, International Conference on Software Engineering (ICSE), ACM, New York, NY, USA, pp. 411–420.

[10] R. Eshuis (2006): *Symbolic model checking of UML activity diagrams*. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 15(1), pp. 1–38.

[11] G.J. Holzmann (2004): *The SPIN model checker: Primer and reference manual*. 1003, Addison-Wesley.

[12] Kenneth Johnson, Radu Calinescu & Shinji Kikuchi (2013): *An Incremental Verification Framework for Component-Based Software Systems*.

[13] FONDAZIONE BRUNO KESSLER (2011): *NuSMV Home Page*. Available at `http://nusmv.fbk.eu/`.

[14] V.S.W. Lam (2007): *A formalism for reasoning about UML activity diagrams*. *Nordic Journal of Computing* 14(1), pp. 43–64.

[15] D. Latella, I. Majzik & M. Massink (1999): *Automatic verification of a behavioural subset of UML statechart diagrams using the SPIN model-checker*. *Formal Aspects of Computing* 11(6), pp. 637–664.

[16] José Merseguer, Javier Campos, Simona Bernardi & Susanna Donatelli (2002): *A compositional semantics for UML state machines aimed at performance evaluation*. In: *Discrete Event Systems, 2002. Proceedings. Sixth International Workshop on*, IEEE, pp. 295–302.

[17] E. Mikk, Y. Lakhnech, M. Siegel & G.J. Holzmann (1998): *Implementing statecharts in PROMELA/SPIN*. In: *Industrial Strength Formal Specification Techniques, 1998.*, IEEE, pp. 90–101.

[18] The Object Management Group OMG (1997): *OMG - Unified Modeling Language (OMG UML)*. Available at `http://www.uml.org/`.

[19] V. A. Santiago Júnior & N. L. Vijaykumar (2012): *Generating Model-Based Test Cases from Natural Language Requirements for Space Application Software*. *Software Quality Journal* 20(1), pp. 77–143. DOI: 10.1007/s11219-011-9155-6.

[20] T. Schäfer, A. Knapp & S. Merz (2001): *Model checking UML state machines and collaborations*. *Electronic Notes in Theoretical Computer Science* 55(3), pp. 357–369.

[21] Sebastian Uchitel & Jeff Kramer (2001): *A workbench for synthesising behaviour models from scenarios*. In: *Proceedings of the 23rd intern. conference on Software engineering*, IEEE Computer Society, pp. 188–197.

[22] J. Woodcock, P. G. Larsen, J. Bicarregui & J. Fitzgerald (2009): *Formal methods: Practice and experience*. *ACM Computing Surveys* 41(4), pp. 19:1–19:36.