UML-Checker: An Approach for Verifying UML Behavioral Diagrams

Flávio Fernandes, Mark Song
Computer Science Department,
Pontifícia Universidade Católica de Minas Gerais
Belo Horizonte, MG, Brazil
Email: flavio.fernandes@sga.pucminas.br, song@pucminas.br

Abstract— **UML is a visual modeling language used for specifying, visualizing, constructing, and documenting software artifacts. Despite having many features to model systems, conducting verifications and validations on UML models is not an easy task. In this paper, the problem of performing verification of UML models is discussed through a translation of UML behavioral diagrams into formal models to be verified by a symbolic model checker. An approach and tool (UML checker) is presented in order to conduct verifications on UML diagrams. The main ideas are: (a) provide an approach to perform the translation of activity, state and use case diagrams to the formal input language of the NuSMV checker; (b) automate the translation of UML diagrams to a formal language; and (c) provide a set of predefined validations that are used to check the diagrams.**

Index Terms— **methods, model checking, UML behavioral diagrams**

I. INTRODUCTION

The Unified Model Language (UML) is a visual modeling language used in software development. It provides resources to specify, visualize, construct and document systems through diagrams. The UML provides structural and behavioral diagrams that allow the creation of individual profiles for a given system. Structural diagrams are designed to visualize and document the static aspects of systems, while the behavioral diagrams aims at visualizing the dynamic aspects.

Due to the features offered, the UML has unquestionable advantages as a technique for visual modeling. With the explosion of UML, several support tools have been developed. However, the existing tools do not guarantee that the generated models are correct.

Human errors can introduce defects at any stage of software development, including during the modeling cycle. As a result, the cost to detect and remove such defects increases considerably through the software development [1, 2, 3, 4].

A diagram without errors has important benefits such as the possibility of raising the accuracy of an implementation. One way to raise this accuracy is through the process of Verification & Validation (V&V) [5, 6, 7]. Inspections and testing are activities of V&V process that could assist in verification of UML diagrams.

However, checking all combinations of input and pre-conditions is not feasible except for trivial cases [8, 9, 10]. The verification of a system is laborious and requires specialized tools such as theorem provers [11].

The use of formal techniques force a detailed analysis of the specifications, which may reveal potential inconsistencies or omissions that would otherwise remain unnoticed until the system operates [12]. Formal methods enable the software engineer to specify, develop and verify a computer system applying a rigorous mathematical notation [13].

Among the formal method techniques there is the Symbolic Model Checking approach, which is used to check whether a finite state model satisfies certain properties. The purpose of the technique is to provide a framework to evaluate expressions in order to confirm or obtain counterexamples (in case of failures) [14, 15, 16].

This paper presents an approach to support the verification of UML diagrams using formal methods. It provides a method for automatic translation of activity, state and use case UML diagrams to Symbolic Model Checking, and also, a way to carry out checks on diagrams from pre-defined validations.

In the next sections we present the UML behavioral diagrams and discuss the Model Checking approach.

II . UML BEHAVIORAL DIAGRAMS

A diagram is a graphical presentation of a set of elements, usually represented as graphs. They are designed to allow viewing the system from different perspectives - in this sense, a diagram is a projection of a given system.

In all the systems, except for the most trivial, a diagram represents a partial view of the system components. UML has 13 diagrams: Class Diagram, Component Diagram, Object Diagram, Composite Structure Diagram, Deployment Diagram, Package Diagram, Activity Diagram, Use Case Diagram, State Machine Diagram, Sequence Diagram, Interaction Overview Diagram, Communication Diagram and Timing Diagram.

UML diagrams are organized into two major groups: Structural Diagrams and Behavioral Diagrams. Behavioral Diagrams have a specific group of diagrams: State Machine, Activity Diagram and Use Case Diagram.

A State Machine Diagram (Figure 1) is a behavioral diagram that specifies the sequences of states through which an object goes through during its lifetime in response to events, together with their responses to those events.

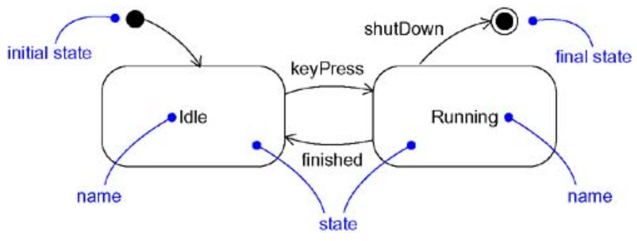

Figure 1. State Machine Diagram.

An Activity Diagram (Figure 2) is a special type of State Machine Diagram where activity states are represented instead of object states. It is used to represent the flow from one activity to another. Unlike the state machine diagram that are event-driven, activity diagrams are control-flow-oriented.

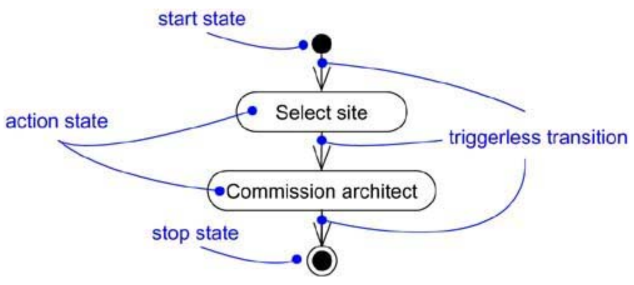

Figure 2. Activity Diagram.

An activity is an ongoing execution. The activities result in some action, formed by the computations that result in a system state change or a value returned. The actions include a call to other operations, sending a signal, creating or destroying an object or some pure computation, such as the computation of an expression.

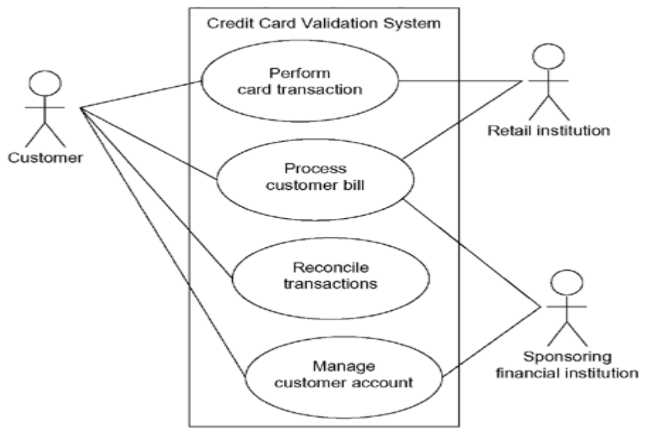

Figure 3. Use Case Diagram.

The Use Case Diagram (Figure 3) is intended to help the communication between analysts and clients. A use case diagram describes a scenario that shows the system`s features from the users point of view. It corresponds to an external view of the system and graphically represents *actors*, *use cases* and *relationships* between these elements. It is especially important to organize and model the system behavior.

III. Model Checking

Model checking and more specifically Symbolic Model Checking (SMC) is a technique to explore finite applications described as a graph. Each graph containing states and transitions which are validated throw temporal logic. It is possible to verify if a transition occurs based on Boolean formulas and temporal logic (linear or branching logic).

Clarke has introduced SMC to verify hardware designs and identify errors based on properties described in temporal logic [13]. SMC has been applied also to software designs in order to check if a sequence of states matches the restrictions described by a set of temporal clauses.

This approach, unlike the traditional ones such as test, simulation and theorem proving is completely automatic. It also explores all state-space system searching for errors.

The system is described as a 4-tuple graph M (*States, Initials, Actions*, Δ). Each *state* $\in$ *States* refers to the present condition of a system. *Initials* $\subseteq$ *States* is a subset defining the initial states. $\Delta \subseteq$ *States* x *Actions* x *State* is a total relation (the set of transitions). One computation of M is $p = s_0, s_1 \ldots s_n$ of states ($s_0 \in$ *Initials*).

The properties to be checked can be described in some temporal logic, which is a formalism to describe computation (transitions between states). This logic can be used to describe events occurring in the system.

The temporal logic describes linear or branching time events occurring in continuous or discrete time. In the branching-time logic, the time has many possible future paths in a given instance.

The time can be defined discrete if between two instances of time no other instance is found; otherwise it is assumed as continuous. In our work the Computation Tree Logic (CTL) is used. CTL describes an infinite tree where each path in the tree describes one possible computation of the system.

CTL operators must be applied in all paths to describe **one** or **all** possible computation (path quantifiers). The **A** operator describes **all** possible paths, whereas the **E** operator, **some** path. Also, a temporal operator must be applied to define behaviors that are expected along a path described by formula *f*.

The main temporal operators describe future event (**F –** *f* must be valid in some state path); events that holds forever (**G -** *f* must be valid in all states); next state (**X** – formula f must be valid in the next state based on the current state of the path).

Following are some CTL formulas. For example, EF(*required* $\rightarrow$ EF *acknowledge*) describes that if eventually *required* occurs, then *acknowledge* will also occurs in, at least, one path. Another example: AG(*empty*

∧ *waiting*) check if in all states *empty* must be true and so, the system status must be *waiting*.

IV. A Formal Methodology to Perform UML Diagrams Verification

This section presents a formal methodology developed to perform validations of activity, state and use case diagrams using Symbolic Model Checking. Figure 4 presents the methodology, which is divided into three phases: specification, verification and results.

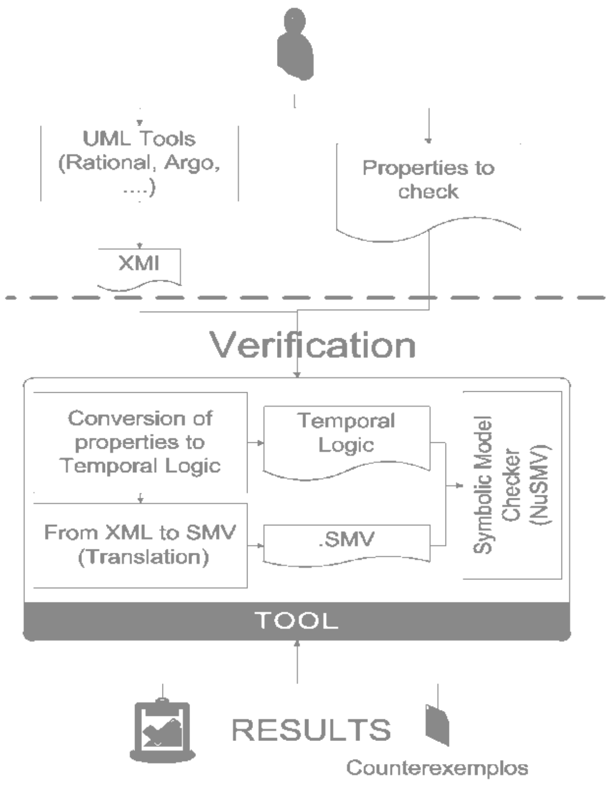
UML Tools (Rational, Argo, ...)
Properties to check
XMI
Verification
Conversion of properties to Temporal Logic
Temporal Logic
From XML to SMV (Translation)
.SMV
Symbolic Model Checker (NuSMV)
TOOL
RESULTS
Counterexemplos

Figure 4. Methodology.

A. Specification

The specification phase is performed by a Software Engineer that, using UML tools available (e.g., Rational, Astah), creates the diagrams to be checked. After, one generates a description of the diagrams in XMI format using UML tools output (the XMI is an OMG standard for exchanging metadata information via XML). Moreover, in this phase, the properties (rules to be verified in a diagram) are selected.

B. Verification

The next phase, Verification, comprises: the XMI translation to SMV file, the conversion of properties to the temporal logic file and the automatic verification. This phase begins when the framework receives, from the Specification phase, the XMI file and the Properties to be checked.

With such information, the translation of the XMI files (which represents the diagrams) to the formal model is performed, thereby generating the SMV file.

Then, the properties are converted to temporal logic, generating the Temporal Logic file. Finally, both files (the SMV and Temporal Logic files) are linked in order to carry out the checking.

C. Results

In the last phase (which is automatically performed by the tool), the results are presented. If the diagram does not satisfy the properties specified then counterexamples are generated - in this case the counterexample describes a computation that does not satisfy the properties checked.

V. The Translation Process

A. Translation of State Machine Diagram

In the translation of the State Machine Diagram, the *states* are grouped into enumerations. Each *event/action* is translated to a boolean variable and each *transition* is represented by the *CASE* statement.

TABLE 1.
TRANSLATION OF STATE MACHINE DIAGRAM TO SMV.

UML	SMV	Initial value
State	Element of enumeration type	
Event/Action	Boolean variable	False
Transition	CASE statements	

It is noteworthy that the enumeration will always be initialized with the initial state to mark the starting point of the diagram.

Every event is initialized with false value and can be activated with the transitions between states. To illustrate the proposed translation we use a simple state machine diagram (Figure 1).

Applying the proposed changes one can obtain the following code:

```
MODULE main
VAR
 keyPress:boolean;
 finished:boolean;
 shutdown:boolean;
 states:{ idle, running, final};
ASSIGN
 init(states):= idle;
 next(states):= case
   keyPress : running;
   finished : idle;
   shutdown : final;
   1: states;
 Esac
```

B. Translation of Activity Diagram

In the translation of the Activity Diagram, each *activity*, *initial state* and *final state* are mapped as logical variables describing whether an activity was executed or not.

All variables begin as false (not running activity), except for the *initial state* that assumes the value true. After the declaration of variables and their initial values, for each element the implementing rules are defined.

To translate the *transition* behavior of the activities some specific cases were identified.

The first one is a simple sequential transitions (Figure 2). They are common, and occur when a state is complete - the control flow passes immediately to the next action or activity node.

In this case, the execution of the evaluated activity is post-condition to the execution of the previous ones. So if the activity *Commission architect* has already run, it will continue with the value true.

If the activity *Select site* has been performed, but *Commission architect* has not, then it will assume in that transition the true value.

In all other cases it holds false. As the initial state has no dependence and only determines the start of the activities flow, it will always start with the true value.

Therefore, the next activity after the initial state will have no dependency to be marked as true. The following code shows the translation in this case:

```
MODULE main
VAR
  initial:boolean;
  selectSite:boolean;
  commissionArchitect:boolean;
  final:boolean;
ASSIGN
  init(initial):= 1;
  next(selectSite):= 1;
  next(commissionArchitect):= case
    commissionArchitect = 1 : 1;
    commissionArchitect = 0 & selectSite = 1 : 1;
    1 : 0;
  esac;
  next(final):= case
    commissionArchitect = 1 : 1;
    commissionArchitect = 0 & selectSite = 1 : 1;
    1 : 0;
  esac;
```

The second case occurs when the activity diagram contain branches, specifying alternative paths based on some boolean expression, as shown in Figure 5. Therefore, the verifier shall evaluate the validity of logical-temporal assertions considering all paths from the branch.

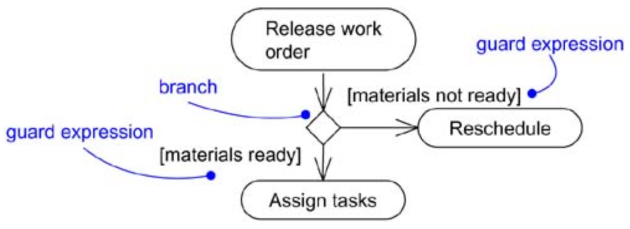

Figure 5. Branches.

When evaluated elements are involved in branches, nondeterminism features must be introduced on the choice (according to the code below).

It may be noted the adoption of rules to ensure the permanence of values previously assigned - and the nondeterminism being indicated for other cases.

```
next(assignTasks):= case
  assignTasks = 1 : 1;
  releaseWork = 1 & reschedule = 1 : 0;
  releaseWork = 1  : {0, 1};
  1 : 0;
esac;
next(reschedule):= case
  reprogramar = 1 : 1;
  releaseWork = 1 & (assignTasks = 1
                     | next(assignTasks = 1)) : 0;
  releaseWork = 1 & assignTasks = 0 : 1;
  1 : 0; esac;
```

The third case refers to the flow with parallel execution, as demonstrated in Figure 6.

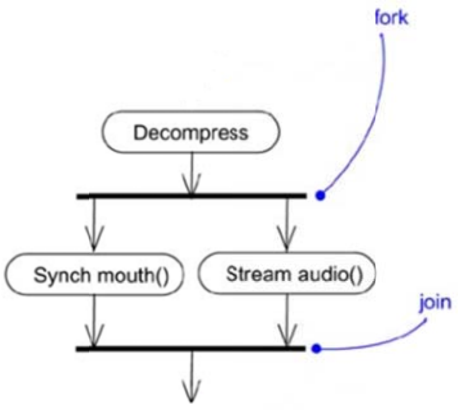

Figure 6. Parallel Execution.

This case deals with both the initial activities of the parallelism and the activities in which the parallelism ends.

Such occurrences are the only ones dependent of a specific treatment, and the intermediate elements of the parallel paths are generically handled as follows:

```
next(synchMouth):= case
  decompress = 1 : 1;
  1 : 0; esac;
next(streamAudio):= case
  decompress = 1 : 1;
  1 : 0; esac;
```

C. *Translation of Use case diagram*

In the Use Case Diagram (Figure 3), each *actor* will be mapped as a boolean variable, *use cases* are mapped as an enumeration and *include* and *extend* relationships with other enumerations as shown Table 2.

TABLE 2.
TRANSLATION OF USE CASE DIAGRAM TO SMV.

UML	SMV
Actor	Boolean variable
Use case/Include/Extend/ Generalization	Element of enumeration type

There are two specific cases in the translation of the use case diagram: (1) the first case is *Association*, which simply makes the connection between actors and use cases, (2) the second case is the representation of relationships: *include*, *extend*, or *generalization*. For each relationship is generated an *enumeration* containing the use cases and actors who are part of the relationship:

```
MODULE main
VAR
  customer:boolean;
  retailInstitution:boolean;
  sponsoring:boolean;
  usecase: {perform, process, reconcile, manage};
ASSIGN
  next(usecase):= case
    customer: {perform, process, reconcile, manage};
    retailInstitution: {perform, process};
    sponsoring: {reconcile, manage};
  esac;
```

VI. Verifications

In this section, we briefly present how properties can be specified in order to apply and verify the UML activity diagram through formal methods. A detailed description can be found in [17].

In the state and activity diagrams it is possible to check whether states or activities have been executed or not in a given time.

To verify that the flow can always reach its end: if there is no cycles without execution limits, that indicates one can always reach the end of a flow. We have listed some possible verification to be executed from the translation made.

One can verify if "*an action holds another will hold*". Figure 7 represents the state transitions required for the verification of this type of validation.

Considering that the variable *First* represents the first action and *Second* the second one, the *status 1* of an action represents the execution while *0* represents that it was not executed.

Thus, in the *initial state,* indicated by an upper arc, both activities were not executed. Following, the action *First* should be performed while the action *Second* remains unexecuted.

Later, at some point, the state in which both actions are carried out must be achieved and not abandoned until the end of the flow.

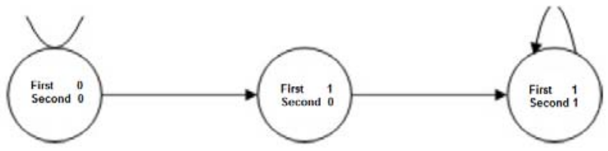

Figure 7. An acion implies in another future action.

The following temporal logic formula, where *First* and *Second* represent components of the flow describes the SMV verification.

Basically, "*In every action/states of the model where the action First is performed, one will always execute Second*":

SPEC AG(*First* -> AF (*Second*));

Another test can describe (Figure 8) that "A*n action implies on not running another one"*. The following temporal logic formula describes the SMV verification:

SPEC AG(*First* -> AG(!*Second*));

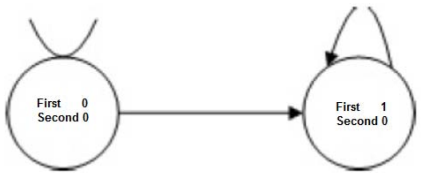

Figure 8. An action implies on not running another one.

VII. The UML Checker

In order to validate the proposed methodology, we created the UML Checker that automates the entire process, offers a set of rules for evaluating diagrams (presented in the last section), performs validations according to the rule set and displays the results obtained, which are positive or negative. In the case of a negative counterexample appears to justify the answer.

Figure 9 presents an overview of the tool architecture that was developed using Java SE libraries. The open-source JDOM library was also used, which is a library that allows reading and writing files in XML format. In addition, we introduced into the tool the symbolic model checker NuSMV which is responsible for performing validations.

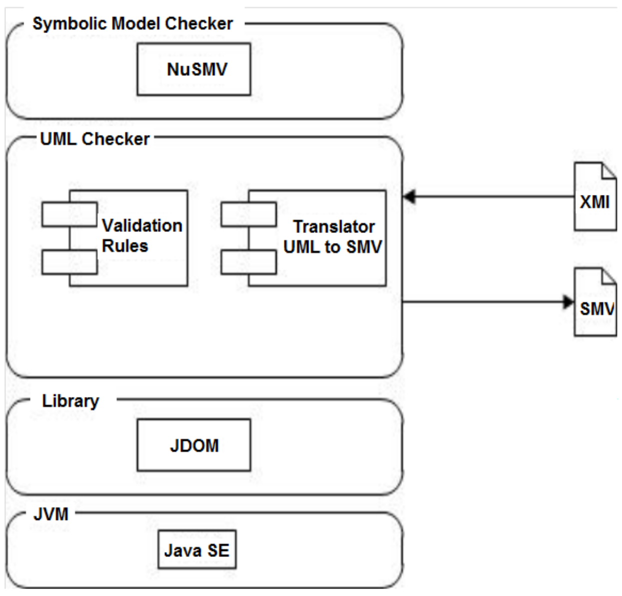

Figure 9. Architecture.

The tool has two interfaces. The first one, called "Translator" is responsible for translating the diagrams. The second, called "Checker" is responsible for verifying the models.

After the execution, the interface is responsible for analyzing the responses and displaying them to the user in a simplified manner.

The tool takes as input the model diagram in XMI format. The translator is responsible for going through the input file translating it to the verification model.

After the translation and together with the properties to be validated (as informed by the user) the diagram is checked by NuSMV.

VIII. EVALUATIONS

In order to evaluate the tool's ability to detect problems and effectively assist in validating the diagrams we attempted to get real modeling, their previous reviews and test them to draw a comparison.

All properties identified were tested, and in the case of negative responses a counterexample is presented. We privileged tests that used the original diagram (i.e., without changes), when this was possible. Otherwise, it was decided to insert errors. Due to space limitations will present only part of the activity diagram evaluations.

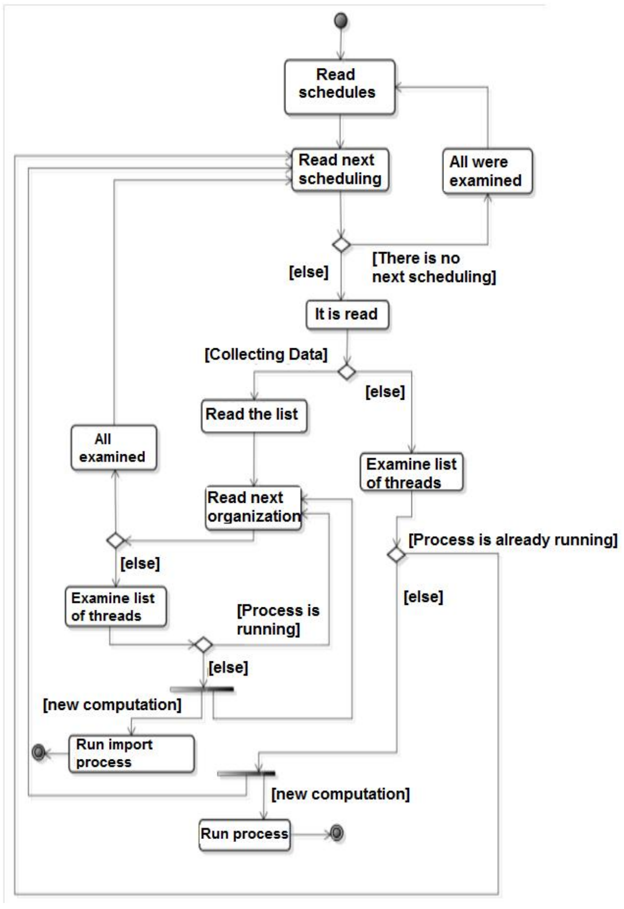

Figure 10. Scheduler Diagram.

In the case study to be presented, the diagrams (a total of 15 diagrams) describe a tool for extraction, transformation and loading of data between two repositories.

The primary purpose of the tool is the implementation of an automated charging process to an intermediate data model, providing consistent data for the process of data collection.

The first activity diagram is the Scheduler, shown in Figure 10. This diagram was used to model the activities performed during the execution of scheduled tasks to import the project data.

The second one is the Data Exclusion (Figure 11), which represents the flow of all activities that are performed by the system to delete a database.

One can use the activity diagram Scheduler. In this diagram we tested the rule that whenever the action *"Read schedules"* is performed (Figure 12), the action *"Read next scheduling on the list"* will hold. The answer is true for both activities.

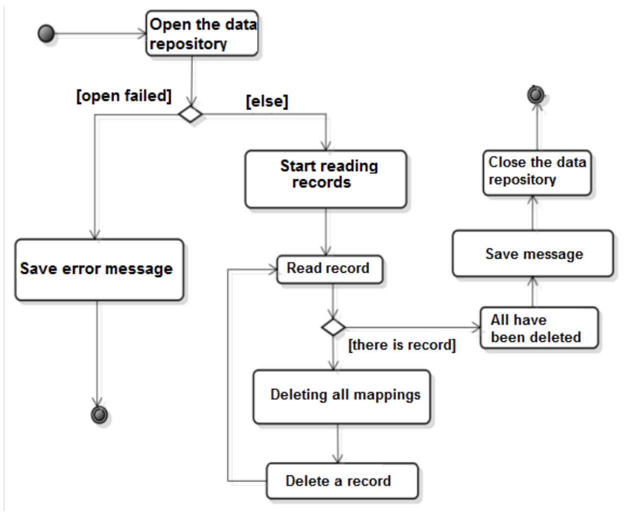

Figure 11. Data Exclusion Diagram.

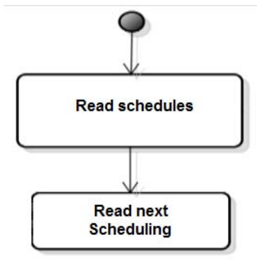

Figure 12. Read Schedules.

Checking the statement, if action *"Read schedules"* holds, the action *"All scheduling from the list were examined"* does not hold (Figure 13), we received a correct answer (no) once the second action may not occur. Note that, the second action depends on the branch to be performed.

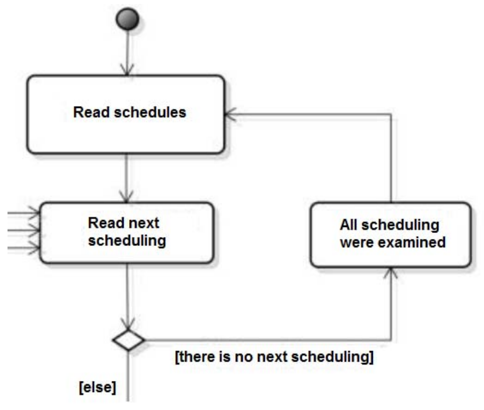

Figure 13. Loop Read Schedules.

Using the activity diagram Data Exclusion in Figure 11, one can observe that there is no possible computation in which action *"Save error message"* holds and also *"Reads record"* action holds – an expected result.

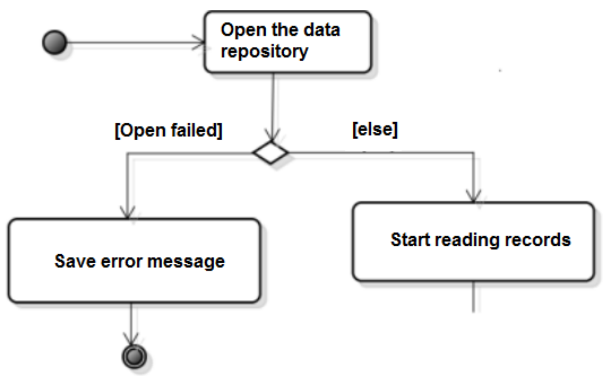

Figure 14. Open the data repository.

Note that, not always *"Read next scheduling on the list"* holding will imply in *"Read the list of organization"* action – here an uninterrupted loop will result in, at least, one computation not reaching the second action.

Note the relationship between the activities *"Read schedules"* and *"All scheduling from the list were examined"* in Figure 13.

One can check if the first action holds the second one will not. The result is negative because if the second action is not performed when the first is, it does not mean that the second one will never be.

One can check if there is a some sort of dependency between action. Take as example, activities *"Read schedules"* and *"All scheduling from the list were examined"*. We checked if *"All scheduling from the list were examined"* can be only executed if *"Read schedules"* holds. As expected, a positive answer was generated.

We check for activities that do not hold. One can think as an error, but that is not always the case. Given the activities from Figure 14 *"Save error message" and "Start reading records"* a true answer was generated. Note that all activities depend on the condition evaluation.

We also tested whether the tool correctly interprets the parallelism rules, not applying the same logic of conditional objects.

To do so we used the activities *"Read next organization"* and *"Run the process"* that are positioned in the activity diagram Scheduler (Figure 15) immediately after the parallelism object and a correct negative response was obtained.

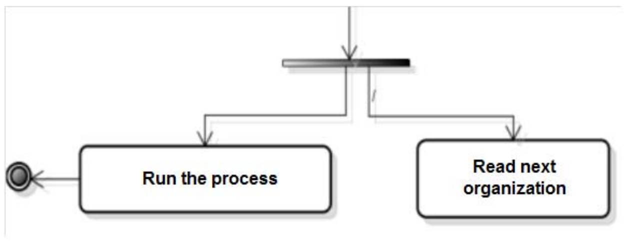

Figure 15. Run the import process.

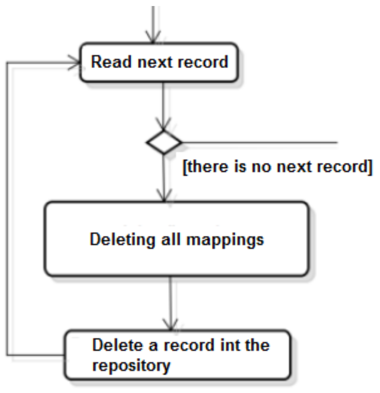

Figure 16. Read next record.

In the case of Figure 16, we tested if introducing an error (disconnecting an element) then a true response would be obtained.

We tested the action *"Deletes a record in the repository"* from the Data Exclusion diagram, before and after the deleting of the entry transition.

Thus, the test performed produced a negative answer. Therefore, there is at least one way in which the activity *"Deletes a record in the repository"* will be performed.

Then, we deleted the entry transition for the action in question and the tool gave a positive response, as expected, because the activity has no connection with another activity of the diagram, so it is impossible that such action is performed.

IX. Conclusion

In order to conduct verifications in UML diagrams, many techniques have been proposed to transform UML diagrams into formal specifications. For example, one can mention the works done by [18, 19, 20, 21].

The work done by [18] presented a new method to automatically transform a UML system model to the formal method RTPA, contributing to formalization of UML diagrams.

The paper [19] presents a Petri Nets model that can express the formal semantic of State Diagrams, especially for features like transitions conflict. A study done by [21] uses Symbolic Model Verifier (SMV) to verify State charts Diagrams showing that formal verification can be used to verify UML models.

Through the related works found it was possible to identify some of their limitations. The works focus on the translation of UML state and activity diagrams because of the proximity to the check pattern.

Furthermore, these works do not describe the patterns used to translate the UML diagrams to Symbolic Model Checking.

Given the need to ensure better quality of UML models this work developed a framework to perform automatic checking of UML models that have not been addressed by these works.

Our work contributes by providing a standard for the translation of UML behavioral diagrams to Symbolic Model Checking.

ACKNOWLEDGMENT

The authors thank CAPES and FAPEMIG for the support to their research.

REFERENCES

[1] V. R. Basili, B. T. Perricone, Software errors and complexity: an empirical investigation. Communications of the ACM, v.27, p.42-52, January, 1984.

[2] C. Kaner, J. L. Falk, H. Q. Nguyen, Testing computer software. New York: John Wiley & Sons, 1999.

[3] N. Fenton, M. Neil, A critique of software defect prediction models. IEEE Transactions on Software Engineering, v.25, p.675-689, 1999.

[4] T. J. Ostrand, E. J. Weyuker, R. M. Bell, Where the bugs are. ACM Sigsoft International Symposium on Software Testing and Analysis, 2004. Proceedings of the ISSTA '04. Boston: ACM, 2004. p.86-96.

[5] R. M. Bell, T. J. Ostrand, E. J. Weyuker, Looking for bugs in all the right places. International Symposium on Software Testing and Analysis, 2006. Proceedings of ISSTA. Maine: ACM, 2006. p.61-72.

[6] T. Ostrand, E. Weyuker, R. Bell, Predicting the location and number of faults in large software systems. IEEE Transactions on Software Engineering, v.31, n.4, p.340-355, Abril, 2005.

[7] R. S. Pressman, Software Engineering: A Practitioners Approach. 7th ed. New York: McGraw Hill Science/Engineering/Math, 2009.

[8] I. Sommerville, Software Engineering. 7th ed. Boston: Pearson Addison Wesley, 2004.

[9] R. G. Sargent, Verification and validation of simulation models. In: The Winter Simulation Conference, 2007. Proceedings of the WSC '07. Washington: IEEE Computing Society, 2007. p.124-137.

[10] D. Graham, E. Veenendaal, I. Evans, R. Black, Foundations of software Testing: ISTQB certification. Derby: Int. Thomson Business, 2008.

[11] G. Fraser, F. Wotawa, P. E. Ammann, Testing with model checkers: a survey. Software Testing, Verification and Reliability, v.19, p.215-261, September, 2009.

[12] E. M. Clarke, F. Lerda, Model Checking: Software and Beyond. Journal of Universal Computer Science, v.13, p.639-649, 2007.

[13] E. M. Clarke, O. Grumberg, D. A. Peled, Model Checking. Cambridge, Massachusetts: The MIT Press, 1999.

[14] K. L. McMillan, Symbolic Model Checking. Norwell: Kluwer Academic Publishers, 1993.

[15] E. M. Clarke, O. Grumberg, K. L. McMillan, X. Zhao, Efficient generation of counterexamples and witnesses in symbolic model checking. ACM/IEEE Design Automation Conference, 32, 1995. Proceedings of DAC'95. San Francisco: ACM/IEEE, 1995. p.427-432.

[16] A. Biere, A. Cimatti, E. M. Clarke, Y. Zhu, Symbolic Model checking without BDDs. International Conference On Tools And Algorithms For Construction And Analysis Of Systems, 1999. Proceedings TACAS. Cleveland: Carnegie Mellon University, 1999. p.193-207.

[17] Barros, Cristiano de Magalhães; Song, Mark Alan Junho. Automatized Checking of Business Rules for Activity Execution Sequence in Workflows. Journal of Software, v. 7, p. 374-381, 2012.

[18] Tian, Y., Wang, Y.; 2007. Transformation of UML Models into Formal RTPA Specifications. In: Electrical and Computer Engineering, CCECE.

[19] Guo, F., Zhang, M.; 2009. Translating UML Statechart Diagrams to X-Nets. In: 1st Information Science and Engineering (ICISE), pp.5279-5282, IEEE Press.

[20] Bruel, J., France, R.; 1998. Transforming UML Models to Formal Specifications. In: Proceedings of UML'98-Beyond the notation, Lecture Notes in Computer Science. Springer-Verlag.

[21] Hai-yan, C., Wei, D., Ji, W., Huo-wang, C.; 2001. Verify UML statecharts with SMV. In: Wuhan University Journal of Natural Sciences, LNCS, Vol. 6, pp. 183-190. Springer.