UML-VT: A Formal Verification Environment for UML Activity Diagrams

Zamira Daw, John Mangino, and Rance Cleaveland
Department of Computer Science, University of Maryland

Abstract—**This paper introduces a translation tool that supports formal verification of UML activity diagrams using the model checkers: UPPAAL, SPIN, NuSMV and PES. The motivation for this tool arises from the desire to check the properties of a system early in the development process, and the fact that UML is commonly used to describe software models. The tool is implemented as an Eclipse-plugin that automatically translates the UML activities and logical requirements into valid input notation for the model checkers. The automated aspect of the plugin allows users without a background in formal methods to verify the safety and liveness of a system. The translation strategies implemented in this plugin are the result of an experimental study. A tutorial video can be found in https://www.youtube.com/watch?v=AHsih8REUxM.**

I. Introduction

A major concern in system development is the correctness of software components. The benefit of formal methods is that they verify the systems correctness against specific requirements for all possible inputs. In contrast to testing, formal methods allow not only the verification of the presence of an error in a system, but also the verification of the absence of errors. Automated formal verification such as model checking [1] has been improved in the last decades, allowing the verification of more complex software systems. However, the usage of model checking is limited by the required knowledge of formal methods, as well as by the state explosion problem, which restricts the size of systems that are verifiable.

Model-driven development (MDD) has been used in software development in order to handle the development of complex systems. In MDD approaches, designers first build models of the desired software, which can be manually or automatically (code generation) transformed into development artifacts (e.g. source code). The models abstract elements/behaviors of the systems, thereby reducing the complexity of the development and facilitating the understanding of the system. Unified Modeling Language (UML) [2] has attracted substantial attention as a language for MDD. Moreover, UML is a non-proprietary, independently maintained standard, which provides several graphical sublanguages and an extension mechanism (profiling). A UML activity diagram is a behavioral diagram that is generally used to specify the workflow of a system. The presented tool uses UML activity diagrams in order to specify the behavior of the system.

The UML Verification Tool (UML-VT) is meant to support the integration of model checking into a MDD process. The purpose of this integration is to provide formal verification in the early phases of development regardless of ones knowledge of formal methods. Furthermore, the integration leverages the higher abstraction of the UML models in order to reduce the state explosion problem, thereby allowing the verification of more complex systems.

The UML-VT transforms UML activities into the input notation for the model checkers: UPPAAL, SPIN, NuSMV and PES. The used transformation strategies have been chosen based on the results of an experimental study conducted by the authors [3]. This paper presents the capabilities and functionality of the UML-VT as well as references for theoretical and technical work, on which this tool is based. The plugin and source code of the UML-VT can be found at http://www.cs.umd.edu/∼rance/projects/uml-vt/

II. Functionality

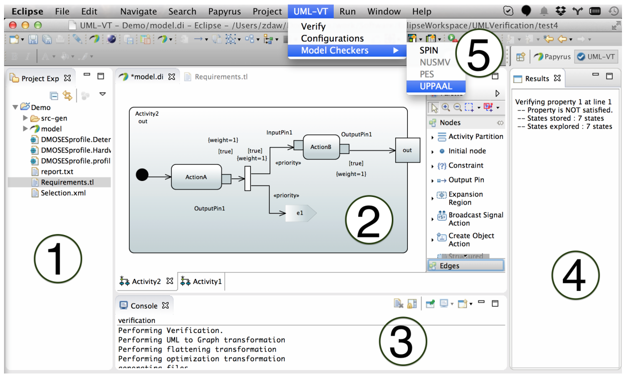
The UML-VT is an open-source Eclipse plug-in that verifies UML activities against given requirements using well-know model checker tools such UPPAAL [4], SPIN [5] and NuSMV [6], and an experimental model checker PES [7]. The inputs of the verification are the UML models and the requirements. The format of the UML models is *.uml, which can be exported from the majority of UML-Tools such as Papyrus[1] or MagicDraw[2]. Requirements have to be specified as a temporal logic formula, which can be created using a Requirement Editor provided by the UML-VT. The output of the verification is a report that shows the satisfiability of the given requirement and counter examples that violate the requirement (depending of the chosen model checker).

The UML-VT also provides an Eclipse Perspective shown in Figure 1 in order to facilitate the verification workflow. The Perspective can be open by the following menu chain: *Window* → *Open Perspective* → *Other* → *UML-VT*. The perspective contains four views and one menu. View 1 shows the Project Explorer, View 2 is reserved for the chosen modeling tool, View 3 shows the Console which will update the user on the current state of the verification process, and View 4 displays the report file, which contains the results of the model checker. The UML-VT Menu allows the user to start the verification (*Verify*), to set the paths of the model checkers executable file (*Configuration*), and to set a default model

Research supported by NSF Grant CCF-0926194

[1]https://eclipse.org/papyrus/
[2]http://www.nomagic.com/products/magicdraw.html

checker (*Model checkers*). This menu is only displayed if the UML-VT Perspective is active.



```
verification
Performing Verification.
Performing UML to Graph transformation
Performing flattening transformation
Performing optimization transformation
```

```
Verifying property 1 at line 1
 -- Property is NOT satisfied.
 -- States stored : 7 states
 -- States explored : 7 states
```

Fig. 1. Eclipse-Plugin of the UML-VT

The verification workflow is shown in Figure 2.

Create project: A new project can be created using *File* → *New* → *Project* → *UML-VT* → *Project*. The created project contains by default a Papyrus model, DMOSES Profile [8], and the Requirement file. The DMOSES Profile is explained in the modeling section.

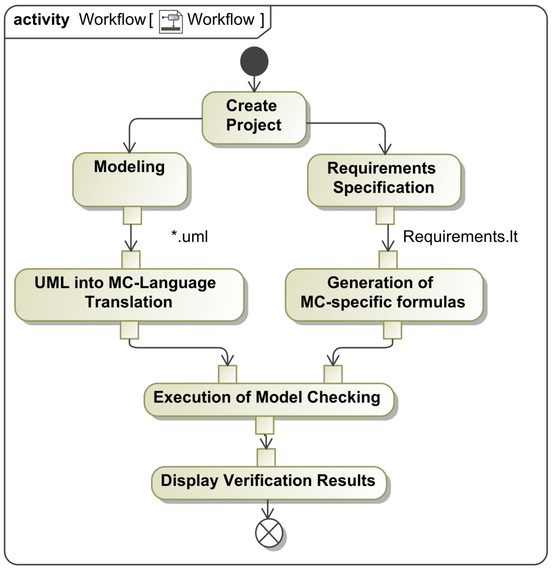

activity Workflow [Workflow]
Create Project
Modeling
Requirements Specification
*.uml
Requirements.lt
UML into MC-Language Translation
Generation of MC-specific formulas
Execution of Model Checking
Display Verification Results

Fig. 2. Workflow of the UML-VT

Modeling: The user can model the system within the Papyrus model or using his/her preferred UML-Tool. However, note that the verification requires the file *.uml. If MagicDraw is used as modeling tool, this file can be obtained by *File* → *Export To* → *Eclipse UML2*. The DMOSES profile is a UML profile that extends UML activity diagrams and state machine diagrams in order to add information regarding execution time, parallelism and priority [8]. This profile gives an unambiguous behavior specification, which is required for the formal verification. An example of an extended UML activity is shown in Figure 3.

Requirements specification: The user can specify the requirements with in the file Requirements.tl using the Requirement Editor. The editor aims to facilitate the specification of temporal logic formulas by highlighting keywords and checking syntax. LTL and CTL formulas are supported. The editor syntax is model checker independent. An example requirement for the activity in Figure 3 can be "In all execution paths, the $ActionB$ should be executed at least one time". The temporal logic formula corresponding to the requirement is:

$$\mathbf{AF}\ Activity2 :: ActionB \tag{1}$$

$$\mathbf{AF}\ Activity1 :: Action2 :: ActionB \tag{2}$$

Note that equation 1 verifies the $ActionB$ if the main activity would be $Activity2$ and equation 1 verifies the same action but if the main activity would be $Activity1$. Requirement Editor provides code completions, which suggests a list of possible names of actions.

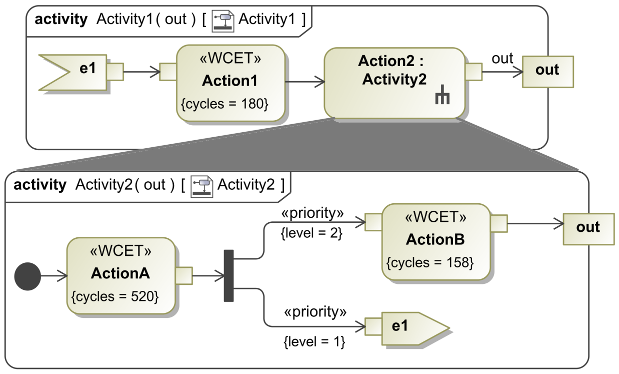


Fig. 3. UML activity example

Generation of model-checker specific properties: Once the Requirements.tl file is saved, model-checker specific formulas are generated in the scr-gen folder according to the chosen model checker. Formulas that are not supported by the given model checker (e.g. UPPAAL only allows only a subset of CTL* [9]) are not transformed into model-checker specific formulas and are shown in the console.

Translation from UML-models into model-checker languages: The translation is started by clicking the Verify button in the UML-VT Menu. Depending on the chosen model checker a different translation is executed. The translations support hierarchical modeling and also apply optimization techniques. The generated files are saved in the scr-gen folder. These files are the input for the model checkers.

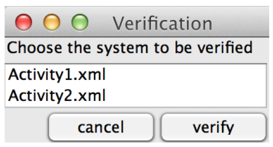
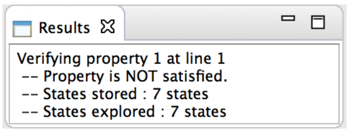


Fig. 4. Possible main activities to verify

Model checking execution: After a successful translation, a window pops up with a list of all possible main activities as shown in Figure 4. Thus, the user can select the main activity of the system, afterwards the chosen model checker is automatically invoked. There is a timeout that limits the verification time.

Display of verification results: The results of the verification are saved in the report.txt file and are displayed after the model checker has finished. Current efforts address a model checker independent report in order to facilitate the understanding of the results. Figure 5 shows the verification result of the example in Figure 3 and the requirement of formula 2. Note that the property is not satisfied, which implies that the action is not executed. This is because the event $e1$ is sent only after it is received. Since there are no other activities that send this event, actions within Figure 3 are never executed. Errors such as this one are not easy to find in a model with multiple activities and multiple hierarchical levels. This example shows how the correctness of the model can be easily verified using the UML-VT.



```
Verifying property 1 at line 1
 -- Property is NOT satisfied.
 -- States stored : 7 states
 -- States explored : 7 states
```

Fig. 5. Verification results of Formula 2 using UPPAAL

III. Architecture

The UML-VT Eclipse-plugin can be divided into four main components: Model Transformation, Code Generation, a Requirement Editor, and a User Interface as is shown in Figure 6. The *User Interface* aims to facilitate the interaction with the user by providing a Perspective, Menus, Project Wizards, Plugin-Installation, Verification Management, etc. This component is primary based on the Rich Client Platform (RCP) provided by Eclipse.

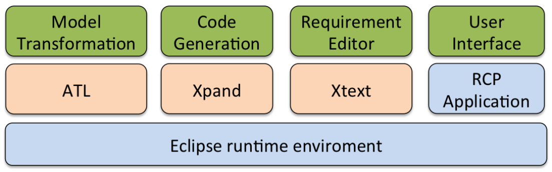


Fig. 6. Architecture of the Eclipse-Plugin. Developed components (green) are based on Eclipse components (blue), and external plugins (orange).

Due to space limitation, technical details of the translations can be found in [3]. The transformation of UML models into model-checker input notation is implemented in two modules: *Model Transformation* and *Code Generation*. The interaction between these parts is shown in Figure 7. Note that an intermediate graph is used. Thus, UML models are transformed into graphs by the *Model Transformation* module, and afterwards, graphs are transformed into model-checker input notation by the *Code Generation* module. The intermediate graph is used in order to facilitate the transformation of multiple diagrams, the generation of multiple model-checker languages, and the implementation of optimization algorithms. The Model Transformation module consists of three components: *Model to Model Transformation*, which transforms UML models into graphs, *Hierarchy Management*, which flattens multiple hierarchical levels (e.g. modeled by using CallBehaviorActions), and *Optimization*, which reduces the number of vertices of the graph by merging sequential vertices and normalizes execution times. These transformations are implemented using an eclipse plug-in called ATL (ATLAS Transformation Language)[3].

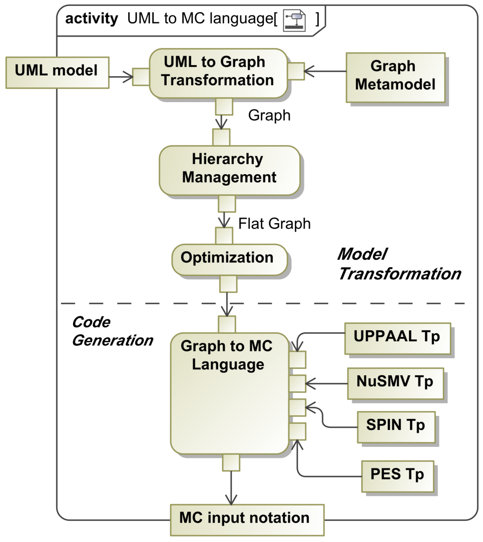


Fig. 7. Workflow transformation from UML models into model-checker input notation

The *Code Generation* module is based on templates that specify the behavior of the different elements of the UML activities using the model-checker languages. For example, for UPPAAL, the template *UPPAAL Tp* specifies the token-based behavior of the UML activities using Timed Automata. Note that the templates encompass sophisticated understanding formal-methods. Since there are multiple ways to specify this behavior using model-checker languages, the authors conducted an experimental study to analyze the influence of different translations strategies on the verification performance [3]. The templates that are used correspond to the best translation strategies for each model-checker in relation to the results of the study. *Code Generation* module uses the plug-in Xpand[4]. The multiple intermediate steps between the UML models and the model-checker input notation are transparent for the user, who only sees the generated files for the model checkers.

The correctness and scalability of the transformations, optimizations and code generation have been tested using a benchmark composed of a set of 67 UML activity diagrams and a model of a medical device case study, an infusion pump [3]. The UPPAAL translation achieves the best performance in the verification of UML activities, in particular for big models, since the variable management of this model checker in the creation of the state space of the system.

The plug-in Xtext[5] is used in the component *Requirement Editor*. The component specifies the domain-specific language of the temporal logic formulas, and implements the editor and the generation of the model-checker specific formula files.

[3]https://eclipse.org/atl/
[4]http://wiki.eclipse.org/Xpand
[5]https://eclipse.org/Xtext/

IV. Related Work

To the best of our knowledge, there is an apparent lack of tools aimed at formal verification of UML activities. Although there are different approaches that propose the verification of UML activities using model checking, summarized in [3], the majority of these approaches do not provide a public tool (or it could not be found). Similar to the presented tool, these approaches use translational strategies to generate model-checker languages. In general, these approaches offer only the usage of one model checker. In contrast, the UML-VT supports four model checkers, and provides an open-source Eclipse plugin.

MADES project propose a tool chain [10] to verify UML Model of embedded systems. MADES uses its own model checker. In [11], dos Santos also presents a formal verification tool for UML behavioral diagrams using NuSMV. Similar to our approach, these tools presents translations from UML model into the model-checker input notation. In contrast to UML-VT, these tools force the user to use only one model checker. Furthermore, the translations provided by the UML-VT are based on an experimental study providing a higher confiability. It is worth mentioning that these tools where not available to download, and therefore the description is based only on academic publications.

V. Discussion

The usage of UML models as an input notation of the verification has three main advantages. 1) It is easier to integrate the verification in the MDD process because these models are also used during the development process. 2) The abstraction of the UML models mitigates the state-explosion problem of the model-checking algorithm. Furthermore, the DMOSES profile adds additional information about the implementation (e.g. execution time) that allows verifying the system taking into account implementation features. The DMOSES profiles also contributes to the reduction of the state-space since this UML extension provides a deterministic behavior and reduces the behavior concurrency by giving a limited processing units. These aspects allow the application of model checking without any further optimization methods (e.g. CEGAR). 3) Since UML models are model-checker independent, model-checker specific translations can be implemented, which facilitate the choice of the model checker. Although the choice of a model checker should primarily depend on its ability to address the system domain or the properties to verify, in practice, the model checker is chosen based on the previous experience of the formal method experts. Thus, the support of multiple model checkers by the UML-VT allows users to use the most adequate model checker for the application area (e.g. UPPAAL for real-time systems or SPIN for distributed systems).

VI. Conclusion and Future Works

The UML-VT enables formal verification of UML activities using model checkers. The tool is implemented in Eclipse, which is already known as a modeling and MDD environment. In order to facilitate the integration to any MDD process, the tool allows the verification of models with an EMF input format, which can be exported from the majority of UML-Tools. The UML-VT supports the verification using the model checkers UPPAAL, SPIN, NuSMV, and PES. The support of multiple model checkers allows the user to choose the most appropriate model checker with respect to the target platform. Generation of model-checker input notation is based on model-to-model transformations, which optimize the space state of the system, and on templates, which encompass the knowledge of a model-checking expert. These templates are also tied to our interpretation of the UML models, which is based on the DMOSES profile. This limitation can be overcome by extending the transformations in order to support other UML profiles or other diagrams.

In our ongoing work, we address this limitation in a more general way by allowing the user to specify its own formal semantics by using an extensible formal semantics [12]. The extensible semantics provides a reference semantics that can be extended according to the interpretation of the UML models. A label transition system (LTS) is generated from the input UML models according to the user-specific semantics. The LTS formally specifies the behavior of the system. We are working in a semantics framework tool that allows specifying the extensible semantics, and provides simulation of the UML models, consistency verification of the semantics (bisimulation) and formal verification (translation from LTS into model-checker input notation) based on the user-specific semantics.

References

[1] C. Baier, J.-P. Katoen *et al.*, *Principles of model checking*. MIT press Cambridge, 2008, vol. 26202649.

[2] OMG, *Unified Modeling Language, Superstructure, Version 2.4.1*, http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF, 2011.

[3] Z. Daw and R. Cleaveland, "Comparing model checkers for timed uml activity diagrams," *Science of Computer Programming*, 2015.

[4] G. Behrmann, A. David, and K. Larsen, *A Tutorial on Uppaal*. Springer Berlin Heidelberg, 2004, vol. 3185, pp. 200–236. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-30080-9_7

[5] G. J. Holzmann, "The model checker spin," *IEEE Transactions on Software Engineering*, vol. 23, no. 5, pp. 279–295, 1997.

[6] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, *NuSMV 2: An OpenSource Tool for Symbolic Model Checking*. Springer Berlin Heidelberg, 2002, vol. 2404, pp. 359–364. [Online]. Available: http://dx.doi.org/10.1007/3-540-45657-0_29

[7] D. Zhang and R. Cleaveland, "Fast on-the-fly parametric real-time model checking," in *Real-Time Systems Symposium, 2005. RTSS 2005. 26th IEEE International*, Dec 2005, pp. 10 pp.–166.

[8] Z. Daw and M. Vetter, *Deterministic UML Models for Interconnected Activities and State Machines*. Springer Berlin Heidelberg, 2009, vol. 5795, pp. 556–570.

[9] A. Pnueli, "The temporal logic of programs," in *Foundations of Computer Science, 18th Annual Symposium on*. IEEE, 1977, pp. 46–57.

[10] A. Radjenovic, N. Matragkas, R. F. Paige, M. Rossi, A. Motta, L. Baresi, and D. S. Kolovos, "Mades: a tool chain for automated verification of uml models of embedded systems," in *Modelling Foundations and Applications*. Springer, 2012, pp. 340–351.

[11] L. B. R. dos Santos, E. R. Eras, V. A. de Santiago Júnior, and N. L. Vijaykumar, "A formal verification tool for uml behavioral diagrams," in *Computational Science and Its Applications–ICCSA 2014*. Springer, 2014, pp. 696–711.

[12] Z. Daw and R. Cleaveland, "An extensible operational semantics for uml activity diagrams," in *International Conference on Software Engineering and Formal Methods*, 2015.