Regression Testing for Visual Models

Ralf Laue[1], Arian Storch[2], and Markus Schnädelbach[3]

[1] University of Applied Sciences Zwickau, Department of Computer Science
Dr.-Friedrichs-Ring 2a, 08056 Zwickau, Germany
ralf.laue@fh-zwickau.de
[2] it factum GmbH, Arnulfstr. 37, 80636 Munich, Germany
arian.storch@bflow.org
[3] Saxon Police Force - Computer and Internet Crime Unit, Germany

Abstract. In this paper, we present a set of *Eclipse* plug-ins which adapts the idea of regression testing for the area of visual modelling in software engineering: Expected properties of models in languages such as UML, BPMN, etc. are stored together with the model (comparable with test cases added to software). With each change of the model, these properties can be checked. The solution should work with any visual modelling language included into *Eclipse* – both for standardised as for domain-specific languages.
The advantage of our approach over current existing solutions is that the process of model checking is completely hidden to the modeller. In particular, it is not necessary for the modeller to learn a formalism for specifying expected properties.

1 Introduction

In this paper, we present a set of *Eclipse* plug-ins that aim to improve the quality of visual models in languages such as UML, SysML or BPMN, but also in domain-specific visual languages. For this purpose, the idea of regression testing which is a well-known paradigm in software engineering is adapted for the domain of visual modelling. In software engineering, regression testing makes sure that software still works correctly after it has been changed. This avoids that desirable properties (expressed as test cases) get lost with a software change. It is good practice to run such tests after each software change. If the results for all test case executions are "green" (Fig. 1), the software developer knows that the software still has the behaviour which is expected by the test cases.

We would like to achieve the same for the domain of visual models in software engineering: Expected properties are stored together with the models and can be tested with every change of the model. However, here we face a different situation: While a typical software developer is able to write test cases (which are executable computer programs as well), we cannot expect that every modeller can formulate desirable model properties in a formal language that is needed for model verification. In [1], Visser et al. point out that hiding the formal models and specification formalisms from the end-user is paramount to the success of

X. Franch, J. Ralyté, R. Matulevičius, C. Salinesi, and R. Wieringa (Eds.):
CAiSE 2017 Forum and Doctoral Consortium Papers, pp. 89-95, 2017.

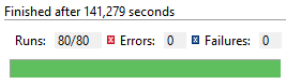


Fig. 1. "All tests green" in *JUnit*

formal verification. With our *Eclipse* plug-ins, we provide the possibility to hide both the formal model that is needed by the model checker as the formalism needed for specifying properties. This enables end users who are not experts for formal methods to use the "regression testing" feature for visual models.

2 Challenges of "Hiding" Formal Methods

The usual way for verifying properties of visual models in software engineering starts with transforming the model into another (formal) model that can be analysed by a verification tool. This formal model can be e.g. a Petri net or an abstract automaton that can be processed by a model checker. In the most cases, the verification tool (a model checker, a SAT solver, a natural language processing tool, etc.) already exists independently from the modelling tool. If we want to use such a tool for "regression testing" of visual models, we face the following challenges (cf. [1]):

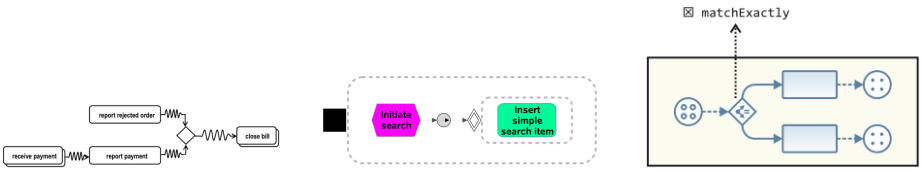
1. The visual model has to be transformed into the formal language the tool can work with.
2. The properties to be verified have to be transformed into a formal language understood by the tool.
3. Starting the verification tool should be possible directly from the modelling tool's user interface.
4. The information which of the checked properties are fulfilled should be shown in the modelling tool's user interface.
5. If a property is not fulfilled, an explanation of the reasons (such as a counterexample found by a model checker) should be transferred back to the visual model.
6. The tested properties have to be bound to the model such that they can be checked again with the next change of the model.

Our set of *Eclipse* plug-ins, called the *bflow* Hive*, provides a solution to those challenges. Originally the plug-ins have been developed for the *bflow* Toolbox* [2], an *Eclipse*-based open-source tool for business process modelling with Event-Driven Process Chains. Now, the current version of our *Eclipse* plug-ins should work with *every* graphical modelling tool that is based on *Eclipse* using *GMF* or *Graphiti*.

The basic steps for integrating model verification into the user interface of a modelling tool (model transformation into a formal language, starting external verification tool and visualising its results in the modelling tool) are described

$$\Box(solvency\_check\_done \vee \neg\langle offer\_premium\_status\rangle\top)$$

Fig. 2. Property Specification with Temporal Logic (example taken from [4])



(a) source: [5] (b) source: [6] (c) source: [7]

Fig. 3. Property Specification with Visual Languages

in [3]. Here, we would like to discuss a recent enhancement that deals with point (2) from the above list – a point which is often neglected by current tools. For our discussion, we use examples from the area of business process modelling. However, we want to emphasise that our solution is independent from modelling languages and application domains.

Point (2) from the above list - specifying the properties to test - is not yet satisfactory implemented by several current tools that verify properties of visual models. In the area of business process modelling and compliance checking, we often encounter the need to specify the properties as temporal logic formulae (see Fig. 2) or as OCL constraints.

Other authors suggested graphical languages for specifying the expected properties; some examples are shown in Fig. 3. The disadvantage is that modellers have to learn one more visual language.

Our approach for specifying properties is to use patterns for frequently occurring model properties. This idea goes back to the work of Dwyer et al. [8] who developed a set of property specification patterns such as "Response" (A must always be followed by B) or "Absence" (A must never happen). In the spirit of this work, specific property specification patterns for business processes have been developed [9, 10]. In [10], the participants of a study were able to express 72 out of 82 business compliance rules by means of these patterns.

Wong and Gibbons [11] showed how Dwyer's patterns can be used for verifying properties of BPMN models. This means that the theory of using specification patterns for verifying properties of business process models is well-researched. However - to the best of our knowledge - this method was rarely included into publicly available modelling tools. Notable exceptions are [9] as one of the first works on applying patterns for property verification and the work by the groups who have developed the specification patterns (see e.g. [12] and [10]).

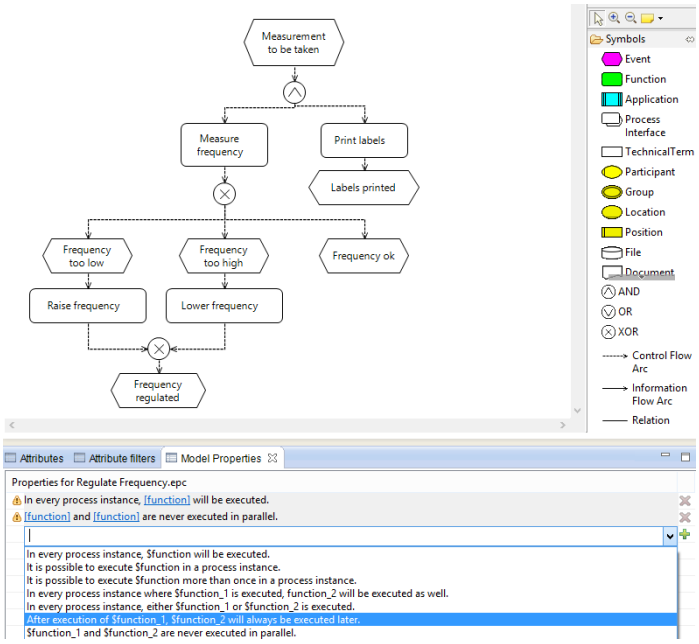


Fig. 4. Selecting the Properties to Verify from a List of Property Patterns

3 Example

Business process models often have to adhere to certain compliance rules such as "After each customer request we receive, an offer will be sent". To ensure a high quality of the model or the adherence to compliance rules, it is desirable to verify such properties after each change of the model. Using the terminology from [8], the mentioned property can be described by the specification pattern "Response". A modeller who wants to verify this property in our tool *bflow* Toolbox* opens the *Eclipse* view "Model properties" (which is provided by our plug-ins) and selects the appropriate pattern from a menu as shown in Fig. 4. Next, the placeholders in the property specification are bound to shapes in the diagram (i.e. to tasks in the depicted business process) by firstly clicking on the hyperlink for the placeholder and secondly clicking on the shape. When all properties are bound to concrete model elements, both the model to be verified as its expected properties are transformed into the input language of a formal tool. For demonstrating the approach, we transform a business process model in the language Event-Driven Process Chains into a network of automata that can be verified by the model checker *UPPAAL*. In addition, the properties to be verified are translated into a temporal logic formula as well. For this purpose, a template has to be provided which lists the property specification patterns as well as their formal representation in the language of the model checker. Of course, an expert in formal methods is needed for creating these templates.

However, once the templates exist, end-users can use them without any deeper knowledge in this area. For example, for the mentioned property, this template looks like this:

```
After execution of $function_1 , $function_2 will always be
executed later >>>i_$function_1.working-->i_$function_2.working
```

Next, the tool for verification (in our case, the model checker *UPPAAL*) is called, its results are analysed and shown in the Model Properties view in the same way. The properties are marked as "red" or "green" (see Fig. 5) in the same way as in *Eclipse*'s *JUnit* plug-in (Fig. 1).

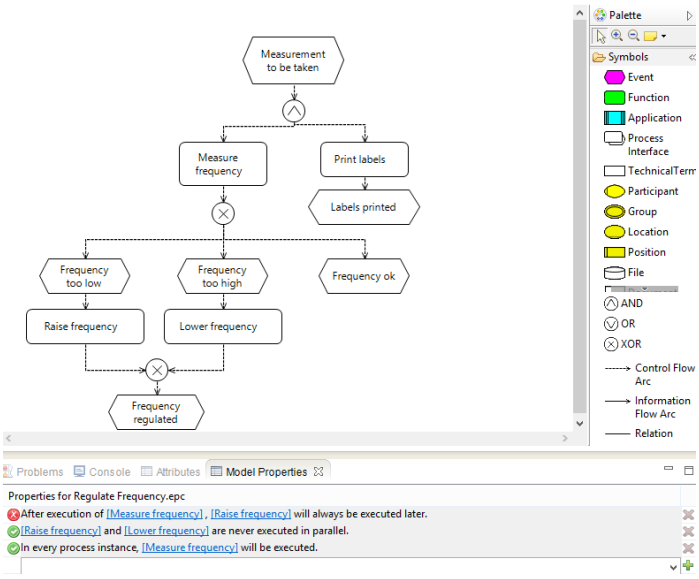


Fig. 5. The Results of the Analysis are Shown as "Red" or "Green"

4 Using our Plug-ins in Eclipse-Based Modelling Tools

While the example in the previous section deals with business process models, the pattern-based approach can be used for other types of modelling as well. For example, property specification patterns have been used for verifying properties of UML state charts [13, 14] or goal models [15]. [16] discusses the development for property specification patterns for arbitrary domain-specific languages. To sum up, we believe that the *bflow* Hive* can be useful for a great variety of different types of modelling, including domain-specific approaches. To give an example, Fig. 6 shows how our plug-ins have been used for verifying typical properties of models in the goal-oriented modelling language i^* .

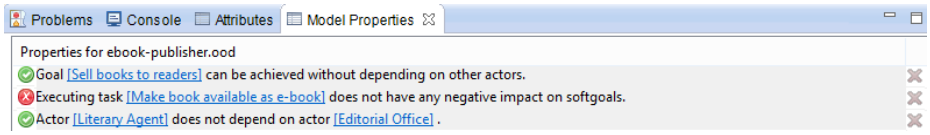


Fig. 6. Property Verification in an i^* Model

Our collection of plug-ins, called the *bflow* Hive*, is already integrated into our own EPC modelling tool *bflow* Toolbox*[1] and into UPROM [17] (a modelling tool that allows functional software size estimation) and (partly) in the i^* modelling tool *openOME*[2].

As already mentioned, modelling tool developers can include the functionality described in this article into their own *Eclipse*-based modelling tool just by adding our plug-ins to the tool.

Compiled binaries and the source code can be obtained from the repository-https://github.com/bflowtoolbox/ where all plug-ins named org.bflow.toolbox.hive.* are part of the tool-independent *bflow* Hive*. The easiest way for a user to profit from the features provided by the *bflow* Hive* in the modelling language of his or her choice is to download the most recent version of the *bflow* Toolbox* from the web site mentioned above and to use *Eclipse*'s update mechanism for adding support for other modelling languages.

We are looking forward to reports from people who made use of the regression testing feature or other *bflow* Hive* features into their tools.

Any questions related to our plug-ins are welcome to `bflow@bflow.org`.

References

1. Visser, W., Dwyer, M.B., Whalen, M.W.: The hidden models of model checking. Software and System Modeling **11** (2012) 541–555
2. Böhme, C., Hartmann, J., Kern, H., Kühne, S., Laue, R., Nüttgens, M., Rump, F.J., Storch, A.: bflow* Toolbox - an open-source business process modelling tool. In: Proceedings of the Business Process Management 2010 Demonstration Track. (2010) 46–51
3. Laue, R., Storch, A., Höß, F.: The bflow* Hive - adding functionality to Eclipse-based modelling tools. In: Proceedings of the BPM Demo Session 2015. (2015) 120–124
4. Giordano, L., Martelli, A., Spiotta, M., Dupre, D.T.: Business processes verification with temporal answer set programming. In: Proceedings of the 1st International Workshop on Knowledge-intensive Business Processes. Volume 861 of CEUR Workshop Proceedings. (2012) 48–59
5. Forster, A., Engels, G., Schattkowsky, T., Van Der Straeten, R.: Verification of business process quality constraints based on visual process patterns. Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering (2007) 197–208

[1] http://bflow.org
[2] http://sourceforge.net/projects/openome/

6. Speck, A., Feja, S., Witt, S., Pulvermüller, E., Schulz, M.: Formalizing business process specifications. Comput. Sci. Inf. Syst. **8** (2011) 427–446
7. Müller, J.: Strukturbasierte Verifikation von BPMN-Modellen. Vieweg+Teubner, Wiesbaden (2011)
8. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Property specification patterns for finite-state verification. In: Proceedings of the second workshop on Formal methods in software practice, ACM Press (1998) 7–15
9. Janssen, W., Mateescu, R., Mauw, S., Fennema, P., van der Stappen, P.: Model checking for managers. In: 5th and 6th International SPIN Workshops. (1999) 92–107
10. Elgammal, A., Türetken, O., van den Heuvel, W., Papazoglou, M.P.: Formalizing and appling compliance patterns for business process compliance. Software and System Modeling **15** (2016) 119–146
11. Wong, P.Y., Gibbons, J.: Property specifications for workflow modelling. Science of Computer Programming **76** (2011) 942 – 967 Integrated Formal Methods (iFM09).
12. Clarke, L.A., Avrunin, G.S., Osterweil, L.J.: Using software engineering technology to improve the quality of medical processes. In: 30th International Conference on Software Engineering (ICSE 2008), Companion Volume, ACM (2008) 889–898
13. Lopez, K., Cheng, B.H., Konrad, S., Goldsby, H.: Visualizing requirements in UML models. First International Workshop on Requirements Engineering Visualization) (2006)
14. Leitner-Fischer, F., Leue, S.: Quantum: Quantitative safety analysis of UML models. In: Proceedings Ninth Workshop on Quantitative Aspects of Programming Languages. Volume 57 of EPTCS. (2011) 16–30
15. Van Lamsweerde, A.: Elaborating security requirements by construction of intentional anti-models. In: Proceedings of the 26th International Conference on Software Engineering, IEEE Computer Society (2004) 148–157
16. Meyers, B., Deshayes, R., Lucio, L., Syriani, E., Vangheluwe, H., Wimmer, M.: ProMoBox: A framework for generating domain-specific property languages. In: Software Language Engineering - 7th International Conference. Volume 8706 of LNCS., Springer (2014) 1–20
17. Aysolmaz, B., Demirörs, O.: Automated functional size estimation using business process models with UPROM method. In: 2014 Joint Conference of the International Workshop on Software Measurement and the International Conference on Software Process and Product Measurement. (2014) 114–124