Regression Testing for Properties of Evolving i^* Models

Ralf Laue[1] and Arian Storch[2]

[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

Abstract. In software engineering, regression testing allows to validate expected properties of a software after each change of the code. In this paper, we present a tool that transfers this idea to the area of visual modelling, in particular to i^* modelling. The modeller can select typical properties (expressed in natural language) from a menu. Then these properties will be stored together with the i^* model and can be verified with every change of the model.

1 Introduction

Up to a certain degree, model quality can be supported by automatic tests. In [1] we demonstrated how non-trivial syntactic requirements of the language i^* can be validated. This validation feature has later been released in the public version of the open-source i^* modelling tool *openOME*. In addition to those syntactic checks, a few rather simple checks of textual quality have been integrated as well. By means of these checks we are able to find certain common errors such as goals that have wrongly been modelled as tasks. The analysis of labels was continued in [2] where we used natural language processing tools for assessing the adherence of labels in an i^* model to naming conventions.

In this paper, we would like to present a tool (more specifically: a set of Eclipse plug-ins) that transfers the idea of regression testing to the domain of 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", 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. To support the modeller, we allow to select the properties from a list of predefined typical model properties.

2 Tool Description

The basic aim of our Eclipse plug-ins, the *bflow* Hive* (formerly known as *Eclipse Modeling Toolbox*), is to make it easy for researchers and practitioners to extend

the functionality of Eclipse-based editors for visual languages. Originally the plug-ins have been developed for the *bflow* Toolbox* [3], an Eclipse-based open-source tool for business process modelling with Event-Driven Process Chains. Now we strive for providing the Eclipse plug-ins such that they work with every graphical modelling tool that is based on Eclipse using EMF and GMF or Graphiti.

We provide an interface that allows modellers and researchers to add their own extensions to those tools without having to become familiar with Eclipse programming and the source code of the modelling tool. Before describing the regression testing feature that was recently added to the *bflow* Hive*, we briefly summarise the other features of the *bflow* Hive* plug-ins that have already been described in previous work [4, 5]. Our Eclipse plug-ins allow to:

- define file transformations for exporting and importing of models into/from other file formats,
- add user-defined attributes to shapes and edges in the diagram,
- decorate shapes and edges in the diagram with user-defined visual annotations (i.e. small icons) - depending on the user-defined attributes mentioned above,
- run validation rules in background to ensure desirable constraints (such as the absence of an inheritance cycle in a UML class diagram),
- equip the tool with add-ons that allow to transform the diagram into the input language of an external program, to run this program and to transform its results back to the graphical interface of the modelling tool.

Once again, we would like to emphasize that all these things can be achieved without having to edit or compile the source code of the modelling tool.

3 Regression Testing for *i** Models

With the features described in the last section, it is possible to analyse a graphical model by means of an external program (e.g. a model checker, a simulation tool or a SAT solver). This is quite useful for purposes such as assessing the syntactical quality of *i** models as described in [1]. In this use case, the required properties to be validated are the same for all diagrams - the models simply have to adhere to the *i** syntax rules.

In [6], Amyot and Yan discuss several examples where the use of models for specific purposes means that further constraints have to be checked. Examples are when a certain style of the modelling language is used or when a certain structure of the model is a prerequisite for applying analysis tools or for using automatic model transformations. [6] presents a solution to this problem in *jUCMNav*, an open-source Eclipse-based tool for goal and scenario modelling: Users can define situation-specific constraints as OCL expressions and check the compliance of the models to these constraints. Alternatively, it is possible to select existing constraints from a pre-defined list. The functionality of our regression testing plug-ins follows a similar purpose - it allows to add tests to a

model. However, compared to the solution presented in [6], our solution provides the following additional features:

- It is independent from tools and modelling languages (requiring just Eclipse using EMF and GMF or Graphiti).
- It allows to use arbitrary tools for validating or analysing models.
- It allows various ways for communicating the result of the analysis to the modeler (for example it is possible to change the colour of an involved shape or to add a small icon at one of its corners).
- It allows binding a set of model-specific properties to a model.

While the first three items in this list just refer to the technical realisation, the last one means a consideration in the conceptual design. By assigning a set of expected properties to a model, we allow those properties to be checked after each change of the model - just in the same way as regression testing works for software.

Kolliadis et al. [7] describe two scenarios where this is important for i^* models which are used for business process development: First, changed requirements can lead to a change of the i^* model. In particular [7] discusses the case that new actors, goals, dependencies, etc. arise. And second, the business process may be changed, e.g. by adding a new actor or task which in consequence also means a change of the corresponding i^* model. This leads to the question whether properties that have been fulfilled in the previous version of the model still hold.

In the scenario discussed in the previous paragraph, regression testing is used when the processes depicted in the i^* diagram are already operating. In a similar way, this kind of testing can be useful for assessing variants of an i^* model in the early design phase. After finding a variant that has certain desirable properties, these properties should be preserved when the model is changed - or at least we want to be aware of the fact that some change breaks a property that has already been fulfilled in another variant of the model.

For using the regression testing feature, the basic course of action in this view is as follows: First, we need an external tool that runs some validation. The only assumption is that the command line can be used for starting the tool and for providing its input parameters. In order to check a property of a model, two kinds of input parameters have to be provided: First, an export of the model to a language that the model checker understands. For example, this can be a network of automata to be used by a model checker. Our plug-ins for writing user-defined model exports can be used for generating such an export. And second, we need a representation of the property to test, once again in a language that is understandable by the external tool. For example, this can be an OCL constraint or a logic formula to test. As we want to make the regression testing feature available to modellers who are not experts for temporal logic or similar formalisms, we use a template approach for this purpose: A template file lists the property description in natural language and assigns a formal expression in the tool's language to it. Placeholders can be substituted by references to model elements. For example, a line in the template file can look like this:

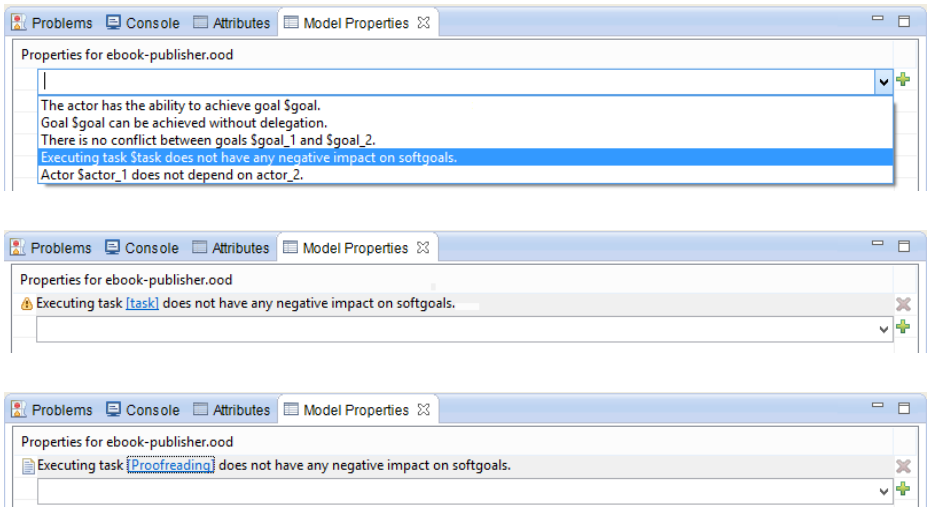

Fig. 1. Defining expected properties in the Model Properties view

`There is no conflict between the goals $goal_1 and $goal_2 >>> no_conflict($goal_1,$goal_2).`

Before the external validation tool is called, the placeholders $goal_1 and $goal_2 have to be replaced by references to model element IDs.

Fig. 1 shows the steps for assigning an expected property to an i^* model using the "Model Properties" view that was introduced with our plug-ins. First, one of the properties defined in the template has to be selected. It is shown with a yellow warning sign because the property contains a placeholder which is not yet bound to a model element. By clicking first on the placeholder and second on a model element, the placeholder is substituted by the ID of an actual model element. When the validation tool is called now, it will be equipped with the necessary inputs (model and property translated to a formal language that the tool understands). We require that the tool writes its result either to a file or to the console output. The output is read and commands for our plug-ins will be generated. Those commands will cause the properties to be marked green (fulfilled) or red (not fulfilled) in the Model Properties view.

For demonstration purposes, we use some example properties that can be checked by simple label propagation algorithms [8]. As described in [1], we use Prolog for describing the model and its expected properties in a formal way and for verifying the model properties. The result of such a validation is shown in Fig. 2. While the first and third property is fulfilled (which is indicated by a green marker at the left), the second one is violated (and gets a red marker). Example properties that can be checked this way is that goals are achievable, that dependencies have reciprocal dependencies, that a failure of one actor does not have effects on another actor, etc.

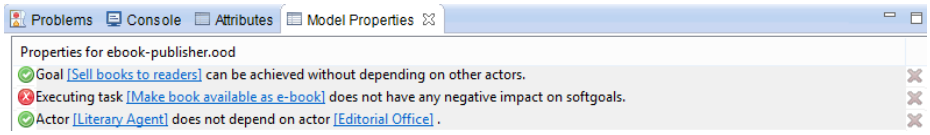

Fig. 2. Results are shown as "green" or "red" in the Model Properties view

Other authors described other properties that can be checked automatically. E. g. Liu et. al [9] apply the model checker *Alloy* to test expected access permission properties ("Least privilege" and "Separation of duties") in an i^* model. In addition to properties that can be verified by formal logic, it is also possible to check whether certain model metrics are within a certain expected range. An example for the use of such metrics is discussed by Franch and Maiden [10] in the context of reasoning about software architecture.

This way, at least a set of basic properties can be expressed and checked by means of templates. While these templates have to be defined by an expert in formal methods, they can be used without any knowledge in this area. Developing a set of useful templates for frequently occurring expected model properties will be an interesting challenge for future research.

We acknowledge that our approach has its limits: By restricting to templates for common properties, it will be impossible (or at least very difficult) to reach an expressiveness of rich specification languages such as Formal Tropos [11]. Furthermore, there are many situations where human judgment is necessary when analysing i^* models. In case of conflicting contributions, human judgment may be necessary for assessing the contribution from means to ends [8]. In the same way, only humans can decide that issues can be regarded as settled in case of conflicting interests. In such a case, automated "regression tests" would not be possible. This means that regression testing as described in this paper is a help, but no substitution for educated reasoning about goal models.

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

Our collection of plug-ins, called the *bflow* Hive*, is already integrated into our own EPC modelling tool *bflow* Toolbox*[1] and into *UPROM* [12] (a modelling tool that allows functional software size estimation). Previously developed plug-ins are already part of the i^* modelling tool *openOME*[2]. Unfortunately, *openOME* is based on an older Eclipse version than the one we used for our plug-ins. Therefore, instead of just adding our plug-ins to a compiled *openOME*, we had to compile *openOME* together with our additional plug-ins using a newer Eclipse version (Kepler) for making the regression testing feature available for *openOME*.

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

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.

The source code can be obtained from https://github.com/bflowtoolbox/app 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 *bflow* Hive* features 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. Laue, R., Storch, A.: A flexible approach for validating i^* models. In: Proceedings of the 5th International i^* Workshop. (2011)
2. Storch, A., Laue, R., Gruhn, V.: Analysing the style of textual labels in i^* models. In: Proceedings of the 7th International i^* Workshop. (2014)
3. 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
4. Laue, R., Storch, A.: Adding functionality to openOME for everyone. In: Proceedings of the 5th International i^* Workshop. (2011) 169–171
5. Laue, R., Storch, A., Höß, F.: The bflow* Hive - adding functionality to Eclipse-based modelling tools. In: Proceedings of the Business Process Management Demo Session 2015. (2015) 120–124
6. Amyot, D., Yan, J.B.: Flexible verification of user-defined semantic constraints in modelling tools. In: Proceedings of the 2008 Conference of the Center for Advanced Studies on Collaborative Eesearch: Meeting of Minds. (2008) 7:81–7:95
7. Koliadis, G., Vranesevic, A., Bhuiyan, M., Krishna, A., Ghose, A.K.: Combining i^* and BPMN for business process model lifecycle management. In: Business Process Management Workshops. Volume 4103 of LNCS., Springer (2006) 416–427
8. Horkoff, J., Yu, E.: Interactive goal model analysis for early requirements engineering. Requirements Engineering **21** (2016) 29–61
9. Liu, L., Yu, E., Mylopoulos, J.: Security and privacy requirements analysis within a social setting. In: Proceedings of the 11th IEEE International Conference on Requirements Engineering, Washington, DC, USA, IEEE Computer Society (2003)
10. Franch, X., Maiden, N.A.M.: Modelling component dependencies to inform their selection. In: COTS-Based Software Systems, Second International Conference. Volume 2580 of LNCS., Springer (2003) 81–91
11. Fuxman, A., Mylopoulos, J., Pistore, M., Traverso, P.: Model checking early requirements specifications in Tropos. In: 5th IEEE International Symposium on Requirements Engineering, IEEE Computer Society (2001) 174–181
12. 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