Enhancing Development and Consistency of UML Models and Model Executions with USE Studio

Marcel Schäfer
University of Bremen
Bremen, Germany
marcel.schaefer@yahoo.de

Martin Gogolla
University of Bremen
Bremen, Germany
gogolla@informatik.uni-bremen.de

ABSTRACT

The UML and OCL tool USE (UML-based Specification Environment) has been developed over more than one decade offering domain-specific languages for describing (1) UML class and statechart models, (2) OCL constraints for invariants (on classes and states) and pre- and postconditions (on operations and transitions), and (3) SOIL (Simple Ocl-like Imperative Language) command sequences for (3a) operation implementations and (3b) executions of model test cases. The three languages have been originally developed as independent textual languages intended for conventional editing. This contribution introduces a new integrated development environment for the three languages to give the developer projectional editing features. We discuss a number of advantages for model development in a developer interface called USE Studio[1]: (1) completion mechanisms for language syntax elements and already defined developer model elements, (2) structured, focused views on related language elements (e.g., one common view on all model associations), (3) consistency guarantees between the underlying model and model executions, and (4) basic common refactorings for the model and model executions.

CCS CONCEPTS

• **Software and its engineering → Unified Modeling Language (UML)**; **Abstraction, modeling and modularity**; • **General and reference → Design**; **Validation**.

KEYWORDS

Modeling tool, UML, OCL, Domain-Specific Language, Projectional editing, Model view, Model consistency

ACM Reference Format:
Marcel Schäfer and Martin Gogolla. 2020. Enhancing Development and Consistency of UML Models and Model Executions with USE Studio. In *ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems (MODELS '20 Companion), October 18–23, 2020, Virtual Event, Canada.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3417990.3422011

[1] A demo video is available at: https://vimeo.com/441356590

MODELS '20 Companion, October 18–23, 2020, Virtual Event, Canada

ACM ISBN 978-1-4503-8135-2/20/10.
https://doi.org/10.1145/3417990.3422011

1 INTRODUCTION

The tool USE (UML-based Specification Environment) [1, 2] supports model development by offering three different languages for describing (a) UML class and statechart models, (b) constraints, and (c) operation implementations and executions of model tests. These textual languages were originally established having in mind conventional editors for building textual artefacts.

The focus of this paper is the discussion of the concepts for an exclusive integrated development environment (IDE) for USE, designed from the demand to connect the three languages within USE in the development process more closely. By regarding the three USE languages as domain-specific languages (DSLs) we were able to implement them in the language workbench Jetbrains MPS. In order to provide the three languages to the developer, a standalone IDE has been built as a projectional editor. Projectional editing [6] allows for development and direct manipulations of language artefacts based on the abstract syntax tree (AST) of a textual artefact, completely omitting the need for a parser for a language. The underlying textual artefact in form of an AST can be projected in a wide range of styles such as text, symbols, graphics or tables. Any change in a projection will consequently result in a change of the underlying artefact. Choosing a projectional editor is especially encouraged by the desire to build selective views on a USE model, because projections can cover a designated set of elements from the underlying textual artefact. We willl demonstrate with examples that the choice of a projectional editor shows significant benefits in many respects. Interestingly, the advantages of projectional editing have been recently discussed in related works in the context of EMF and SPL models [3, 4]

The structure of the rest of this paper is as follows. Section 2 gives an overview on our IDE USE Studio. Section 3 discusses the IDE support for class models, Sect. 4 handles model executions and Sect. 5 describes common aspects of models and model executions, in particular questions concerning consistency. Section 6 ends the paper with conclusions and future work.

2 OVERVIEW

The tool USE was designed for modeling and verification of systems. As pictured in Fig. 1, USE is based on three domain-specific languages: (a) the elements defined in the model are presented in form of textual class and statechart models; (b) additional textual restrictions can be formulated with invariants and pre- and postconditions using OCL; (c) operation implementations and executable test cases are described in SOIL (Simple Ocl-like Imperative Language). A SOIL command sequence always refers to exactly one model. This

dependency, however, leads to the problem of maintaining consistency. If the underlying model is changed, the SOIL files referring to it may become invalid as a result of the modification.

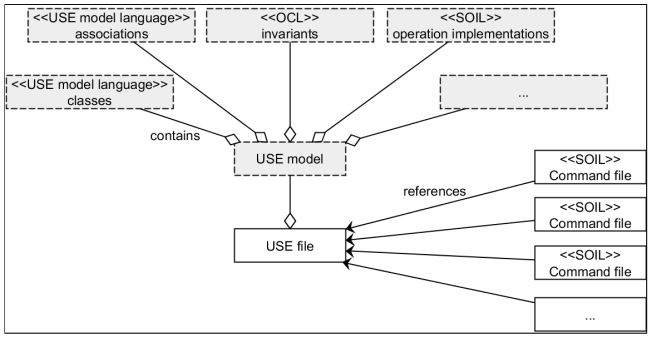

Figure 1: Languages put into context: USE, SOIL and OCL.

For example, in the context of a lecture, a model may be translated from English to Spanish (e.g., class and operation names). This would automatically invalidate any SOIL file that uses the old class names and manual corrections would have to be performed. This can cause much work and is error-prone, in particular if the number of SOIL files is high or if the SOIL files are complex. Thus, our goal was to design a dedicated IDE for the three DSLs applied in USE. The main focus was to provide supportive tooling and a solution for the mentioned consistency problem. We called the resulting tool 'USE Studio'. In order to introduce the IDE and highlight its features, we will show the design process of a sample USE model and some dedicated SOIL command sequences. The class model in Fig. 2 will be the description which we aim to achieve. The class model offers a good overview, as it covers many modeling elements. The model consists of enumerations, several classes, one inheritance relationship (as the class Training extends the class Project), associations, compositions, and a single aggregation. We emphasize that in order to demonstrate particular features of USE Studio during development, we will partly show how elements of the example class model are incrementally built up.

3 USE MODELS

A USE Model is defined by a USE file. The model contains elements defined with the three mentioned DSLs: the language for defining the class and statechart model, OCL for the constraints, and SOIL for the operation implementations. With OCL, modeling details like invariants and pre- and postconditions is possible. This allows the developer to perform model validation and verification tasks.

We assume that we start with a new model called 'Project' that will include our class model, and that we already have created the classes and enumerations. As indicated in Fig. 3, when creating a new attribute the developer first defines an attribute name, here 'name', and after the colon, the type of the attribute has to be determined. By opening the model completion, the developer is offered a list of all valid types. This list consists of OCL standard types like the basic datatypes Boolean, Integer, Real, and String and the different collection kinds (Bag, OrderedSet, Set, Sequence).

While we tried to give the developer a text-like look and feel, the way a projectional editor works is different to the traditional textual approach. As indicated in Fig. 4, by choosing an entry from the completion list, or just typing out a keyword (like 'composition'), a complete language unit is created automatically leaving just a few editable fields for the developer to modify (indicated by the red highlighted fields). All keywords for the 'composition' declaration are inserted automatically, leaving just the developer definable fields editable. The first line of the composition body has already been completed to show the overall correct syntax. The first field contains a reference to a class that should be part of the composition. The developer again is supported by the model completion to show a list of valid classes. The next field represents the multiplicity. The 'role' keyword gives the developer the option to define the name for a specific association end.

The developer has probably to get acquainted with this way of creating artefacts. However, this method promises the comfort of not having to write longer syntactical constructs on her own and syntax errors are completely eliminated. Since the mandatory keywords of the respective construct cannot be edited, errors such as omitted brackets or spelling mistakes can no longer occur [5]. That should save both time and frustration. Another big advantage of the model completion mechanism is on the one hand the comfort to get an overview of the structure and contents of the model, and on the other hand it helps to avoid errors in the developer definable fields. Let us take the example of the class Training (see Figure 5). In the model, Training extends the class Project. When showing the model completion, the developer is presented all possible, valid classes. The class itself is not listed in this context, thus cyclic inheritance is ruled out (Training cannot inherit from Training).

Another case in which model completion stands out is a mechanism for filtering possible operation calls on a given argument. As indicated in Fig. 6, let us take the parameter 'qs' of type 'Set(Worker)' in the operation 'createWorker()' as an example. The type Set belongs to the group of collection kinds of OCL. Set is like Bag an unordered collection type, while Sequence and OrderedSet are ordered collections. The offered choice in Fig. 6 shows all possible operations for the type Set. However, operations like 'reverse()' or 'first()' are not listed, as they can only be applied to ordered collections.

This feature gives the developer a view of all valid operations in the current context and thus avoids possible errors. It also allows the developer to become familiar with the types and their operations more quickly.

Depending on their complexity, USE models can quickly become quite involved. This is also the case with our 'ProjectWorld' example. In a project of this size, it is sometimes difficult to find relevant places in the textual model. Due to this issue, a view mechanism with the additional option to edit the view might come to mind. As indicated in Fig. 7, by applying a tabbed bar at the bottom of the editor, the developer can choose a view option suiting the current requirements. In our example, we might want to edit explicitly the associations between our classes. By choosing the associations tab, we obtain the desired result.

Another selective view is an invariant view as show in Fig. 8. In this view all invariants are shown and details of a particular invariant can be edited and changed.

This feature allows quick access to the different, distinct sections of the model and supports the developer in the development process.

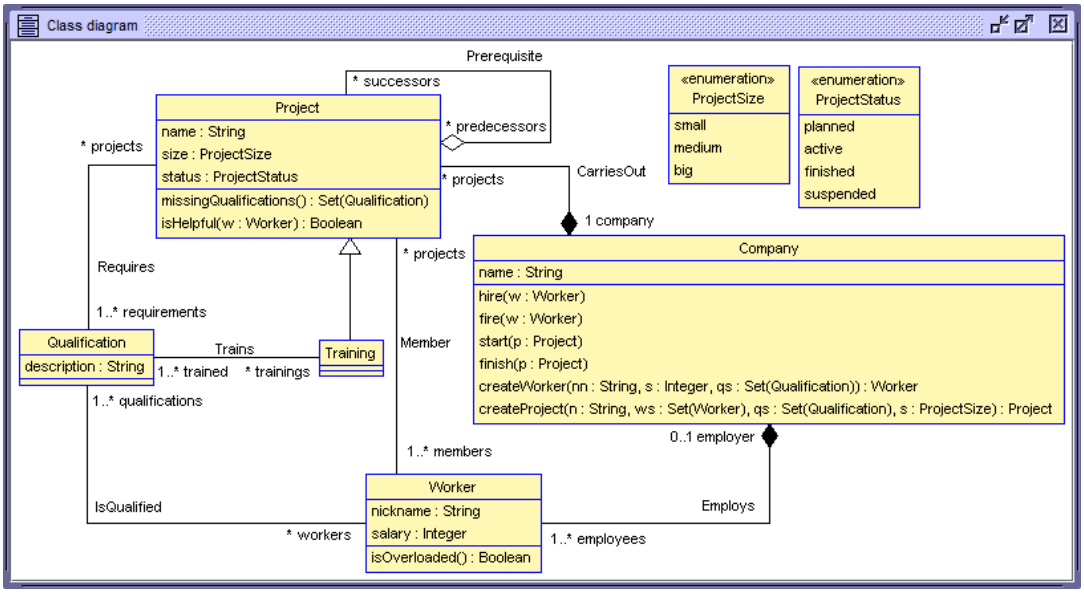

Figure 2: Class model of Example 'ProjectWorld'.

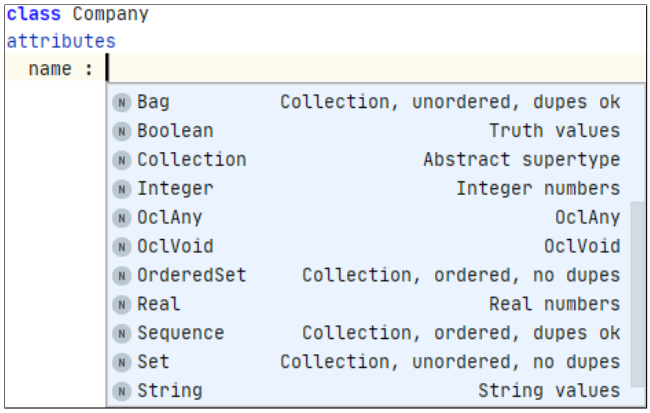

Figure 3: Model completion on standard OCL types.

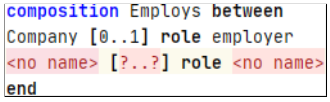

Figure 4: Syntactical directives for 'composition' declaration.

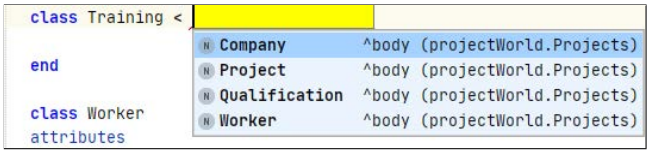

Figure 5: Model completion on inheritance.

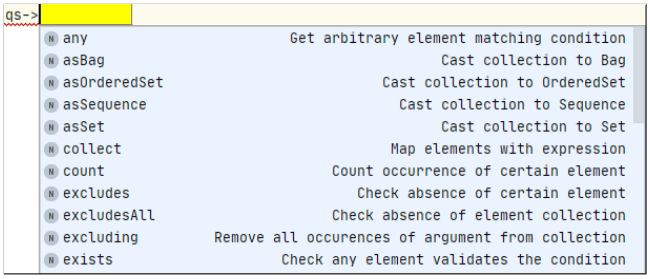

Figure 6: Model completion for valid operations.

4 SOIL EXECUTIONS

As mentioned above, the language SOIL is used to allow for operation implementations and for executions of model test cases. Prerequisite for test cases is a USE model. The test cases are typically defined in a separate SOIL file, but SOIL command sequences can also be applied at run time on the terminal. Among the possible commands are the creation and deletion of objects, attribute value manipulation and link creation and deletion. A SOIL command sequence refers to a concrete USE model. This way we determine the context in which we work. As Fig. 9 shows, with model completion lists the classes that are defined in the USE model can be applied.

```
model Projects

composition CarriesOut between
Company [1] role company
Project [*] role projects
end
composition Employs between
Company [0..1] role employer
Worker [1..*] role employees
end
association IsQualified between
Worker [*] role workers
Qualification [1..*] role qualifications
end
association Member between
Project [*] role projects
Worker [1..*] role members
end
aggregation Prerequisite between
Project [*] role predecessors
Project [*] role successors
end
```

All | Associations | Classes | Invariants | Operations | Pre- & Postconditions | Statemachines

Figure 7: Selective view on model associations.

```
--SOIL for Specification:  Projects

!create IBM : Company
```

Company	^body (projectWorld.Projects)
Project	^body (projectWorld.Projects)
Qualification	^body (projectWorld.Projects)
Training	^body (projectWorld.Projects)
Worker	^body (projectWorld.Projects)

Figure 9: SOIL command sequence referring to classes from a USE model.

Another powerful feature is model completion for developer defined operations. As Fig. 10 shows, the object IBM of type Company offers the listed operations. This is especially useful, as SOIL command sequences and the USE model are in separate files. Using model completion, the developer is always of the latest modifications of the model and does not need to inspect the underlying model every time anew.

```
!Ada := IBM.createWorker('Ada', 6000, Set{Programming, Documentation, Medicine});
```

createProject	^operations (projectWorld.Projects)
createWorker	^operations (projectWorld.Projects)
finish	^operations (projectWorld.Projects)
fire	^operations (projectWorld.Projects)
hire	^operations (projectWorld.Projects)
start	^operations (projectWorld.Projects)

Figure 10: Model completion for developer defined operations in SOIL

5 GUARANTEEING CONSISTENCY OF MODELS AND COMMAND SEQUENCES WITH SMART REFACTORING

The USE Studio was built on the basis of the language workbench Jetbrains MPS. This tool allows for the generation of a standalone IDE to ship the DSLs along with standard tooling to the developer. Jetbrains MPS and the IDEs created from it are projectional editors. This means that the model is stored in a format whose structure can immediately be interpreted as an abstract syntax tree. Each node of the tree has a unique identity that can be used to easily identify a node. Due to this characteristic we immediately have a very good solution for the mentioned consistency problem. Consider our USE model for the running example. This model is persisted as an abstract syntax tree. Any references to class names, for example, can be traced back via a unique node identities. This means that every SOIL file which refers to a specification does so via the unique node identities (e.g., of a class). The information like the name is retrieved at run time. Contrary to a plain textual variant, where model or information are stored purely statically for each file, this dependency essentially solves the problem of consistency on its own.

Figure 11 shows the class definition of class 'Qualification'. Lets say we would like to rename this class into something shorter like 'Skill'. In the traditional textual approach we would have to locate each SOIL file associated with this model and change each occurrence of the class name 'Qualification' to 'Skill'. This is quite time consuming and might cause errors or one may leave out some class name occurrences.

As all of these occurrences are actually references, the IDE manages this refactoring on the fly for each and every reference, may it be in the model itself (as in the operation signatures or operation bodies) or in the additional SOIL command sequences (like in the various create commands). Summarizing, we state that this approach ensures project-wide consistency with no extra effort for the developer, while on the other side a sense for consistency of models and command sequences is successfully established.

6 CONCLUSION AND FUTURE WORK

We designed a dedicated IDE for developing USE models and model executions. The focus was on the creation of a developer-friendly IDE with basic tooling supporting developers. The requirement of viewing the model for certain aspects (such as only classes, only pre- and postconditions) was solved as an extension of the IDE in form of developing a plugin. Depending on the selection of the tab bar, the developer is shown only the desired model parts. This provides comfort and enhances the productivity in the development process. In particular, model completion and on-the-fly syntax checking of the model improves productivity and increases the motivation of the developer to develop in these languages.

A number of improvements can be done. Ultimately, it would even be possible to integrate the entire tool USE into the IDE and thus achieve a seamless link. Since we have a projectional editor, the idea arises to include further visualizations in addition to the plain text option. Currently, there are already two plug-ins in the Jetbrains Marketplace that support common UML diagrams such as

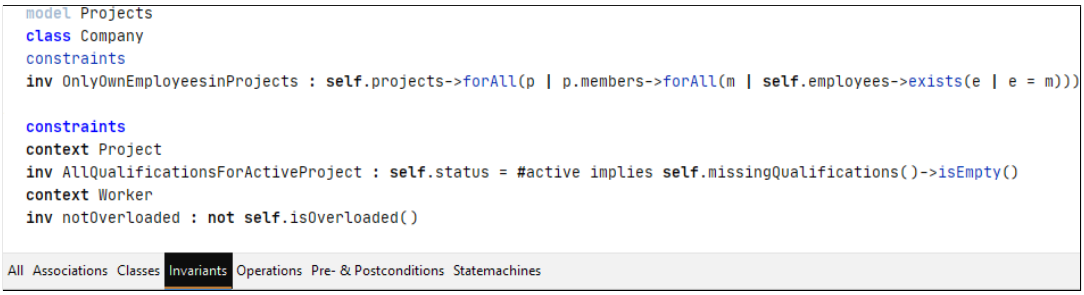

Figure 8: Selective view on model invariants.

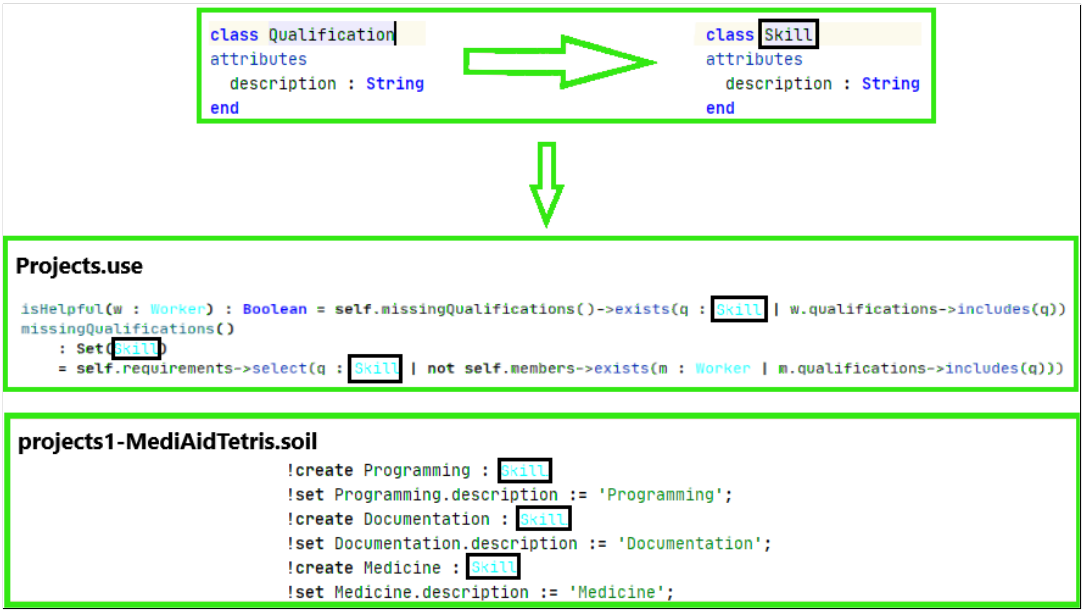

Figure 11: Ensuring consistency by project-wide refactoring on-the-fly.

those mentioned above. By adapting the DSLs to the plugins specifications, these diagrams could be generated. A key feature is that modifying the diagrams also leads to changes in the model. Thus, the flow logic of a state machine could be graphically edited while the corresponding model is generated in the background. We believe that the ideas outlined above offer great potential to further improve the developer experience. Last but not least larger case studies must be carried out in order to gain more insight into developer needs and requirements.

REFERENCES

[1] Martin Gogolla, Fabian Büttner, and Mark Richters. 2007. USE: A UML-Based Specification Environment for Validating UML and OCL. *Journal on Science of Computer Programming, Elsevier NL* 69 (2007), 27–34.

[2] Martin Gogolla, Frank Hilken, and Khanh-Hoang Doan. 2018. Achieving Model Quality through Model Validation, Verification and Exploration. *Journal on Computer Languages, Systems and Structures, Elsevier, NL* 54 (2018), 474–511. Online 2017-12-02.

[3] Dennis Reuling, Christopher Pietsch, Udo Kelter, and Timo Kehrer. 2020. Towards projectional editing for model-based SPLs. In *VaMoS '20: 14th International Working Conference on Variability Modelling of Software-Intensive Systems, Magdeburg Germany, February 5-7, 2020*, Maxime Cordy, Mathieu Acher, Danilo Beuche, and Gunter Saake (Eds.). ACM, 25:1–25:10. https://doi.org/10.1145/3377024.3377030

[4] Johannes Schröpfer, Thomas Buchmann, and Bernhard Westfechtel. 2020. A Generic Projectional Editor for EMF Models. In *Proceedings of the 8th International Conference on Model-Driven Engineering and Software Development, MODELSWARD 2020, Valletta, Malta, February 25-27, 2020*, Slimane Hammoudi, Luís Ferreira Pires, and Bran Selic (Eds.). SCITEPRESS, 381–392. https://doi.org/10.5220/0008971003810392

[5] Friedrich Steimann, Marcus Frenkel, and Markus Voelter. 2017. Robust projectional editing. In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*. 79–90.

[6] Markus Voelter, Janet Siegmund, Thorsten Berger, and Bernd Kolb. 2014. Towards user-friendly projectional editors. In *International Conference on Software Language Engineering*. Springer, 41–61.