Model Checking and Model Finding with USE

Martin Gogolla¹, Loli Burgueño², and Antonio Vallecillo²

¹ University of Bremen, Germany. gogolla@informatik.uni-bremen.de
² Universidad de Málaga, Spain. {loli,av}@lcc.uma.es

Abstract. This contribution demonstrates central options available in the tool USE (Uml-based Specification Environment) for exploring UML models within software development. It particularly uses so-called classifying OCL terms for building validation and verification scenarios. The contribution demonstrates the tool’s options with an example: statecharts together with a simple syntax model and a model for capturing finite fractions of the statechart semantics.

1 Introduction

In general, exploring the execution space of any non-trivial system for checking whether a property holds or not, is a difficult task. One of the common problems of model checking tools and processes is the combinatorial explosion of the state-space. It is normally due to the intrinsic very large size of the execution tree, but it often gets aggravated by the fact that many of the solutions found during the exploration process are very similar—if not identical—from a structural point of view. This forces model analysts to cope with a large volume of solutions that provide little or no benefit with respect to the ones already found, slowing down the analysis process and introducing even more accidental complexity to the already complex problem of finding counterexamples, or their absence.

The paper layout follows the LNCS style. Only the white margins have been reduced.

Here, we apply the tool USE [10,11] (Uml-based Specification Environment) for UML and OCL models and the concept of classifying terms (CTs) [12], which permit generating relevant and distinguished sample object models for a given specification, together with the completion capabilities of the USE model validator for specifying particular validation and verification scenarios. With them we are able to quickly develop distinguishable and structurally non-equivalent object models that satisfy certain system properties. More precisely, classifying OCL terms permit defining equivalence classes with those models that, from the modeller’s perspective, are equivalent. Then, the USE model validator is able to generate one representative object model for each equivalence class, hence significantly simplifying the number of test cases, and improving the effectiveness of the model checking process. In this contribution we illustrate these ideas for exploring models within software development. We demonstrate the tool options with a simple example: statecharts together with a simple syntax model and a model for capturing finite fractions of the statechart semantics.

The structure of the rest of this paper is as follows. Section 2 presents the background work on which our proposal is based: CTs and the USE model validator completion capabilities. It also presents the running example that serves
Section 3 shows how a system can be tested with classifying OCL terms and object model completion. Section 4 compares our work to similar related proposals. Finally, Sect. 5 concludes the paper and outlines some future work.

2 Modeling with USE

2.1 Running example

Our running example is shown in Fig. 1 with a class model and names of needed OCL invariants. The model serves to describe the syntax of statecharts (left part of the class model) and a semantics for statecharts in terms of a finite number of traces (right part). The top left object model in Fig. 2 shows on the left an example for a statechart and on the right an example trace. One could also say that the left side represents design time elements and the right side runtime items. The OCL invariants for the syntax part require unique state names, existence of a single initial and a single final state, deterministic transitions, and each state to lie between the initial and the final state. The OCL invariants for the semantics part require each trace to be a cyclefree string of pearls, to be connected to the initial state and to show events corresponding to the transition events. Our view on the model is that we have specified syntax and semantics of a particular statechart language. Our tool allows the developer to systematically explore the model by building test cases in form of object models and thereby to validate and verify model properties and get confidence into the model.

2.2 Classifying terms

Usual approaches to generate object models from a metamodel explore the state space looking for different solutions. The problem is that many of these solutions are in fact very similar, only incorporating small changes in the values of attributes and hence “equivalent” from a conceptual or structural point of view.

Classifying terms (CTs) [12] constitute a technique for developing test cases for UML and OCL models. CTs are arbitrary OCL queries on a class model.
Fig. 2. Four different object models constructed with classifying terms.
calculating a characteristic value for each object model. Each expression can be boolean, allowing the definition of up to two equivalence classes, or of type integer, where each resulting number defines one equivalence class. Each equivalence class is defined by the set of object models with identical characteristic values, selecting one canonical representative object model. Hence, the resulting set of object models is composed from one object model per equivalence class, and therefore they represent significantly different test cases. Besides, they partition the full input space. For example, the following classifying terms could be defined for our metamodel.

**context** State **inv** initialEqFinal:
let INI=State.allInstances->any(s | s.isInitial=true ) in
let FIN=State.allInstances->any(s | s.isFinal=true ) in
INI=FIN -- invariants oneInitial and oneFinal give determinateness

**context** Transition **inv** twoThreeOrFourEvents:
let numEvents=Transition.allInstances-> -- collect yields Bag
collect(t | t.event)->asSet()->size() in
numEvents=2 or numEvents=3 or numEvents=4

Each of these two CTs may be true or false. Together, they define four equivalence classes. First, we want to distinguish between statecharts in which the initial and the final state coincide, and others with different initial and final states. Second, we want to have some sample models in which there are 2, 3 or 4 events, and other models in which there could be less than 2 or more than 4 events. The model validator, which is the tool in charge of exploring the search space and generating object models, will simply return one representative of each of the four equivalence classes. Note that CTs do not always pretend to generate models that are representative of the complete metamodel, they might be used to generate models that contain interesting features w.r.t. concrete scenarios of interest to the modeller, and which are only relevant in a sub-part of the given specification. They are also useful for finding object models that should not happen in theory, i.e. counterexamples for our specification.

### 2.3 The USE model validator

Object models are automatically generated from a set of CTs by the USE model validator, which builds and inspects object models and selects one representative for each equivalence class. For this, as described in [12], each CT is assigned a boolean or an integer value, and the values of the CTs are stored for each solution. Using the CTs and these values, constraints are created and given to the Kodkod solver [19] along with the class model during the validation process. The solver prunes all object models that belong to the equivalence classes for which there is already a representative element. The construction process always terminates and yields a finite number of representative object models.

The validator has to be given a so-called ‘configuration’ that determines how the classes, associations, data types and attributes are populated. In particular, for every class a mandatory upper bound for the number of objects must be stated. Both the USE tool and the model validator plugin are available for download from [http://sourceforge.net/projects/useocl/](http://sourceforge.net/projects/useocl/).
3 Systematically exploring a model with USE

In order to illustrate our proposal, let us consider the object models shown in Fig. 2. These structurally different object models have been automatically generated by using 4 boolean-valued CTs (and a configuration fixing attribute values, links, and lower and upper bounds for the number of objects in a class):

```plaintext
context State inv twoStates: State.allInstances->size=2
context State inv threeStates: State.allInstances->size=3
context TraceNode inv oneTrace: TraceNode.allInstances->
    select(tn | tn.src->isEmpty() and tn.trg->notEmpty)->size=1
context TraceNode inv twoTraces: TraceNode.allInstances->
    select(tn | tn.src->isEmpty() and tn.trg->notEmpty)->size=2
```
In principle, \(2^4 = 16\) equivalence classes are possible. However, this number will not be reached, because, for example, the classifying terms \texttt{twoStates} and \texttt{threeStates} cannot be true at the same time. Figure 2 shows 4 found solutions. Essentially, the upper row shows the solutions with 2 states and the lower row the solutions with 3 states; and the left column displays the solutions having 1 trace and the right column the solutions with 2 traces.

Another option in USE is to specify an incomplete object model with missing attribute values, objects or links, and to ask the USE model validator to complete the incomplete model into a full model. Fig. 3 shows an example. In the right side of the upper part, two example traces for two \texttt{Paper} objects submitted to a \texttt{Conference} together with an incomplete statechart in the left are shown. In the lower part the result of asking the model validator to complete the object model is pictured. The model validator has found attribute values and link-objects for association classes in order to satisfy all model constraints. Our view is that the full statechart has been deduced from example traces and an incomplete statechart description on the basis of an exhaustive set of invariants.

4 Related work

The USE model validator is based on the transformation \cite{13} of UML and OCL into Kodkod \cite{19}. Approaches related to ours rely on different foundations like logic programming and constraint solving \cite{4,5}, relational logic and Alloy \cite{1}, term rewriting \cite{16} or graph grammars \cite{8}. None of these approaches offers to automatically scroll through several valid object models in one verification task. To reason about UML/OCL models, there are different alternatives, for instance, translating them into standard first-order logic using theorem provers \cite{2,3,14}, or map them to many-sorted first-order logic \cite{7}. There are (semi-)automatic proving approaches for UML class properties based on the basis of description logics \cite{15}, on the basis of relational logic and pure Alloy \cite{1} using only a subset of OCL, and focusing on model inconsistencies with Kodkod \cite{18}. The approaches in \cite{6,17} use metamodels and solvers for software improvement. A classification of model checkers with respect to model verification tasks can be found in \cite{9}.

5 Conclusions and future work

Exploring the execution space of any non-trivial system is a difficult task. In this paper we have shown how the tool USE can be employed, in conjunction with classifying terms, to specify particular validation and verification scenarios, allowing system analysts to look for object models that satisfy certain properties, or their absence. There are several lines of work that we plan to address next. First, we would like to validate our proposal with more examples, in order to gain a better understanding of its advantages and limitations, and to identify different contexts of use in which our approach works well and others in which the results are not satisfactory (and why). Second, we plan to improve tool support to further automate all tests, so human intervention is kept to the minimum. Finally, we need to define a systematic approach of defining classifying terms for exploring object models using the outlined ideas.
References