Model Finding and Model Completion with USE

Martin Gogolla[1], Loli Burgueño[2], and Antonio Vallecillo[2]

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

Abstract. This short contribution demonstrates central options 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 and for completing partial models. 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 this paper, we demonstrate the options available in 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 ("Are we building the right product"; aim: build test cases) and verification ("Are we building the product right"; aim: verify a property) 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. One central advantage we see in our approach is that we offer mainstream languages like UML and OCL to formulate models and use these languages also to give feedback.

The structure of the rest of this paper is as follows. Section 2 presents the background work: CTs and the USE model validator completion capabilities. Section 3 shows how a system can be tested with cts and object model completion. Section 4 compares our work to similar related proposals. Finally, Sect. 5 concludes the paper.

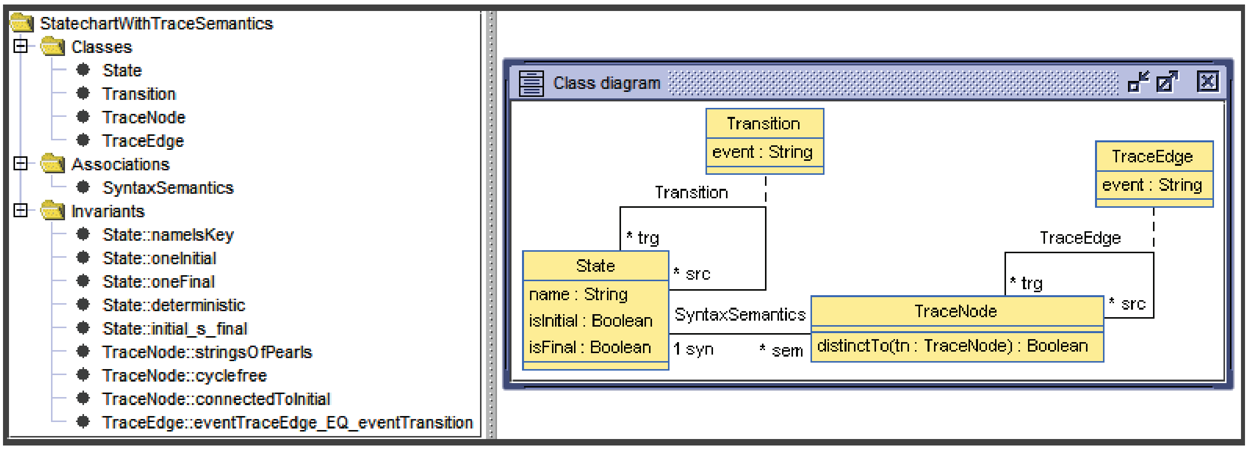

Fig. 1. A metamodel for Statecharts with trace semantics.

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 (`TraceNode` objects linked together looking like a 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 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

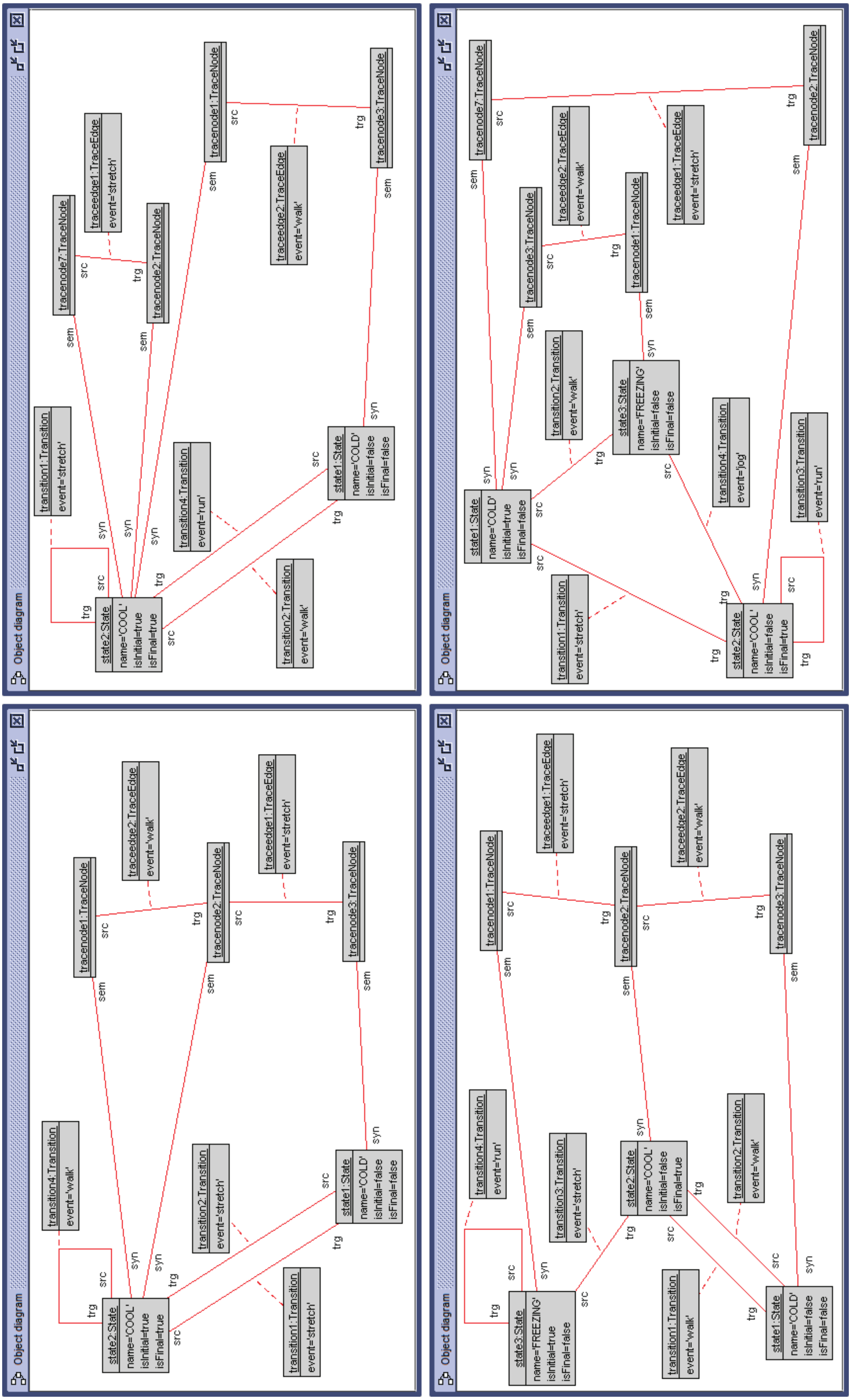

Fig. 2. Four different object models constructed with classifying terms.

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/`.

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

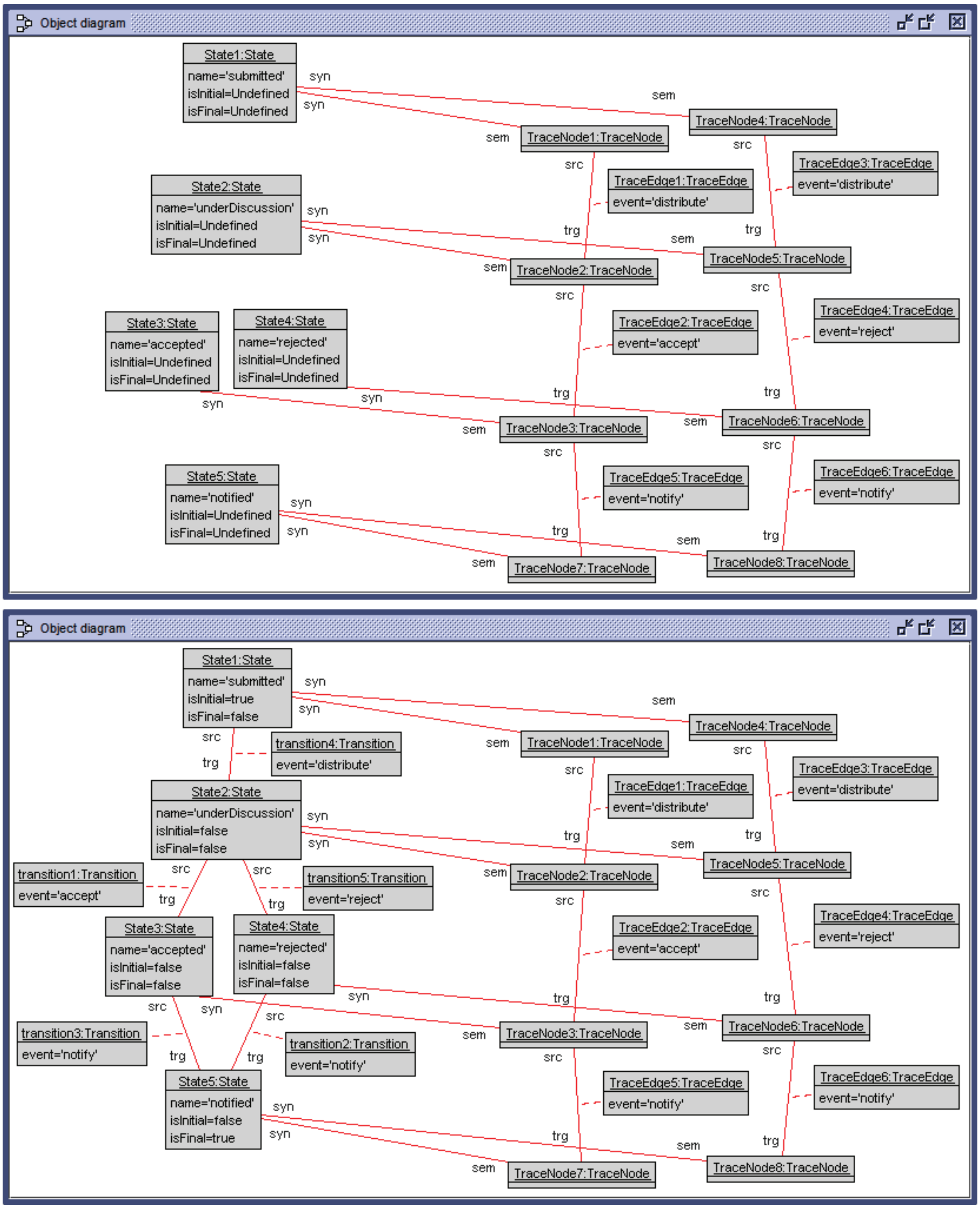

Fig. 3. Completion of an incomplete object model to a full object model.

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):

```
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 `twoStates` and `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 important 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 `Paper` objects submitted to a `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 automatically deduced from two example traces and an incomplete statechart description. The price we have to pay is that we have to present an exhaustive set of invariants. The full models are available.

4 Related work

The USE model validator is based on the transformation [13] of UML and OCL into Kodkod [19]. Related pproaches rely on foundations like logic programming and constraint solving [4,5], relational logic and Alloy [1], term rewriting [16] or graph grammars [8]. To reason about UML/OCL models, there are different alternatives, for instance, translating them into standard first-order logic using theorem provers [2,3,14], or map them to many-sorted first-order logic [7]. There are (semi-)automatic proving approaches for UML class properties based on the basis of description logics [15], on the basis of relational logic and pure Alloy [1] using only a subset of OCL, and focusing on model inconsistencies with Kodkod [18]. The approaches in [6,17] use metamodels and solvers for software improvement. A classification of model checkers with respect to model verification tasks can be found in [9]. None of these approaches offers our options, i.e. to automatically scroll through several valid object models in one verification task. More details of our approach are given in [11].

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

1. Anastasakis, K., Bordbar, B., Georg, G., Ray, I.: On Challenges of Model Transformation from UML to Alloy. Software and System Modeling **9**(1) (2010) 69–86
2. Beckert, B., Keller, U., Schmitt, P.H.: Translating the object constraint language into first-order predicate logic. In: Proc. 2nd Verification WS. (2002) 113–123
3. Brucker, A.D., Wolff, B.: HOL-OCL: A formal proof environment for UML/OCL. In Fiadeiro, J.L., Inverardi, P., eds.: 11th Int. Conf. FASE. LNCS 4961, Springer (2008) 97–100
4. Cabot, J., Clarisó, R., Riera, D.: UMLtoCSP: A Tool for the Formal Verification of UML/OCL Models using Constraint Programming. In: ASE'07. (2007) 547–548
5. Cicchetti, A., Ruscio, D.D., Eramo, R., Pierantonio, A.: JTL: A bidirectional and change propagating transformation language. In Malloy, B.A., Staab, S., van den Brand, M., eds.: 3rd Int. Conf. SLE. LNCS 6563, Springer (2010) 183–202
6. Cuadrado, J.S., Guerra, E., de Lara, J.: Quick fixing ATL model transformations. In Lethbridge, T., et al., eds.: 18th MoDELS, IEEE. (2015) 146–155
7. Dania, C., Clavel, M.: OCL2MSFOL: A mapping to many-sorted first-order logic for efficiently checking the satisfiability of OCL constraints. In: Proc. MODELS'16. (2016) 65–75
8. Ehrig, K., Küster, J.M., Taentzer, G.: Generating instance models from meta models. Software and System Modeling **8** (2009) 479–500
9. Gabmeyer, S., Brosch, P., Seidl, M.: A Classification of Model Checking-Based Verification Approaches for Software Models (2013) Proc. 1st VOLT Workshop.
10. Gogolla, M., Büttner, F., Richters, M.: USE: A UML-based specification environment for validating UML and OCL. Science of Computer Programming **69** (2007) 27–34
11. Gogolla, M., Hilken, F.: Model Validation and Verification Options in a Contemporary UML and OCL Analysis Tool. In Oberweis, A., Reussner, R., eds.: Modellierung (MODELLIERUNG'2016), GI, LNI 254 (2016) 203–218
12. Hilken, F., Gogolla, M., Burgueño, L., Vallecillo, A.: Testing models and model transformations using classifying terms. SoSyM (2016) `http://link.springer.com/article/10.1007%2Fs10270-016-0568-3`.
13. Kuhlmann, M., Gogolla, M.: From UML and OCL to relational logic and back. In: Model Driven Engineering Languages and Systems. LNCS 7590, Springer (2012) 415–431
14. Kyas, M., Fecher, H., de Boer, F.S., Jacob, J., Hooman, J., van der Zwaag, M., Arons, T., Kugler, H.: Formalizing UML models and OCL constraints in PVS. Electr. Notes Theor. Comput. Sci. **115** (2005) 39–47
15. Queralt, A., Artale, A., Calvanese, D., Teniente, E.: OCL-Lite: Finite reasoning on UML/OCL conceptual schemas. Data Knowl. Eng. **73** (2012) 1–22
16. Roldán, M., Durán, F.: Dynamic Validation of OCL Constraints with mOdCL. ECEASST **44** (2011)
17. Steimann, F., Hagemann, J., Ulke, B.: Computing repair alternatives for malformed programs using constraint attribute grammars. In: OOPSLA'16@SPLASH. (2016) 711–730
18. Straeten, R.V.D., Puissant, J.P., Mens, T.: Assessing the Kodkod Model Finder for Resolving Model Inconsistencies. In: ECMFA. (2011) 69–84
19. Torlak, E., Jackson, D.: Kodkod: A relational model finder. In Grumberg, O., Huth, M., eds.: 13th Int. Conf. TACAS. LNCS 4424, Springer (2007) 632–647