Available online at www.sciencedirect.com
ScienceDirect
Science of Computer Programming 69 (2007) 27–34
Science of Computer Programming
www.elsevier.com/locate/scico

USE: A UML-based specification environment for validating UML and OCL

Martin Gogolla[a,*], Fabian Büttner[a], Mark Richters[b]

[a] *University of Bremen, Bremen, Germany*
[b] *EADS Space Transportation, Bremen, Germany*

Received 2 November 2005; received in revised form 6 November 2006; accepted 16 January 2007
Available online 10 October 2007

Abstract

The Unified Modeling Language (UML) is accepted today as an important standard for developing software. UML tools however provide little support for validating and checking models in early development phases. There is also no substantial support for the Object Constraint Language (OCL). We present an approach for the validation of UML models and OCL constraints based on animation and certification. The USE tool (UML-based Specification Environment) supports analysts, designers and developers in executing UML models and checking OCL constraints and thus enables them to employ model-driven techniques for software production.

Keywords: UML; OCL; Model; Constraint; Invariant; Pre- and post-conditions; Model validation; Model certification; Model execution

1. Introduction

UML [13] is accepted today as a de facto standard for developing software. UML and its sub-language OCL [15] are regarded as central ingredients of model-centric software production.

To assure model quality, model verification and model validation have been proposed. Here, we concentrate on model validation, i.e., on checking that a model meets informal requirements a developer has in mind. Checking also involves that the model satisfies particular properties, for example, that certain consequences can be proved or at least certified by a model inspection process.

In order to assist developers in model-driven techniques, we put forward the tool USE (UML-based Specification Environment). USE basically is an interpreter for a sub-set of UML and OCL. An OCL constraint is either an invariant or a pre- or post-condition. USE started as a dissertation project [11] with its first version being available already in 1998. Work around USE combined efforts to define the formal semantics of OCL hand in hand with a Java implementation [12] based on an OCL metamodel. Early USE versions were developed further by diploma theses and other student projects. The original USE tool was extended by a snapshot generator [7] and better support for

* Corresponding address: Mathematik/Informatik, Universitaet Bremen, Bremen, Germany.
E-mail address: gogolla@informatik.uni-bremen.de (M. Gogolla).

0167-6423/$ - see front matter © 2007 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2007.01.013

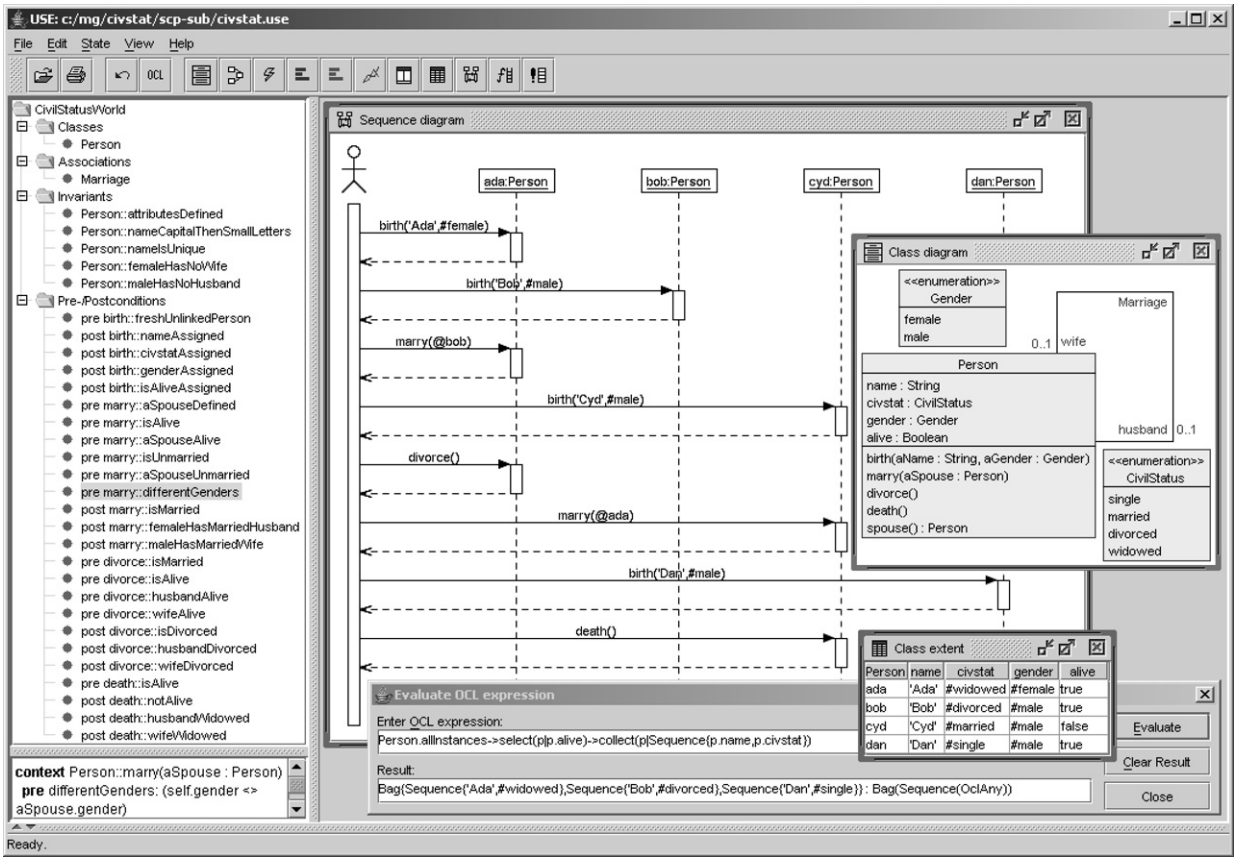

Fig. 1. Class diagram, sequence diagram, class extent, and OCL evaluation.

class diagrams. Over the time, small bugs were eliminated and extensions, for example in the user interface, were added on the basis of feedback from USE users. Future versions of USE will provide better support for (1) sequence diagrams, the (2) evaluation browser (used as an OCL expression debugger), (3) XMI import and export, and (4) model refactoring. The latter may be regarded as a special Model Driven Architecture (MDA) extension of USE [5].

USE has many installations outside of Bremen. USE has been utilized in a number of case studies and teaching and development projects, among other places at the MIT, Cambridge, MA, at the University of Edinburgh, Scotland, at the University of Colorado, CO, and the University of Lisbon, Portugal.

The structure of the rest of this paper is as follows. Section 2 discusses the concepts behind USE. Section 3 concentrates on related approaches and Section 4 very shortly mentions the relevant theory. The paper ends with concluding remarks in Section 5.

2. Concepts of the UML-based specification environment (USE)

Model validation through exploring properties of models is a significant task within model-based software development. The USE system supports developers in analyzing the model structure (classes, associations, attributes, and invariants) and the model behavior (operations and pre- and post-conditions) by generating typical snapshots (system states) and by executing typical operation sequences (scenarios). Developers can formally check constraints (invariants and pre- and post-conditions) against their expectations and can, to a certain extent, derive formal model properties.

The USE approach is explained here by a small example model describing persons and the advance of their civil status. The two USE screenshots in Figs. 1 and 2 show on the left side the examined model: The model browser overview window in the top left displays all model elements (classes, associations, invariants, and pre- and post-

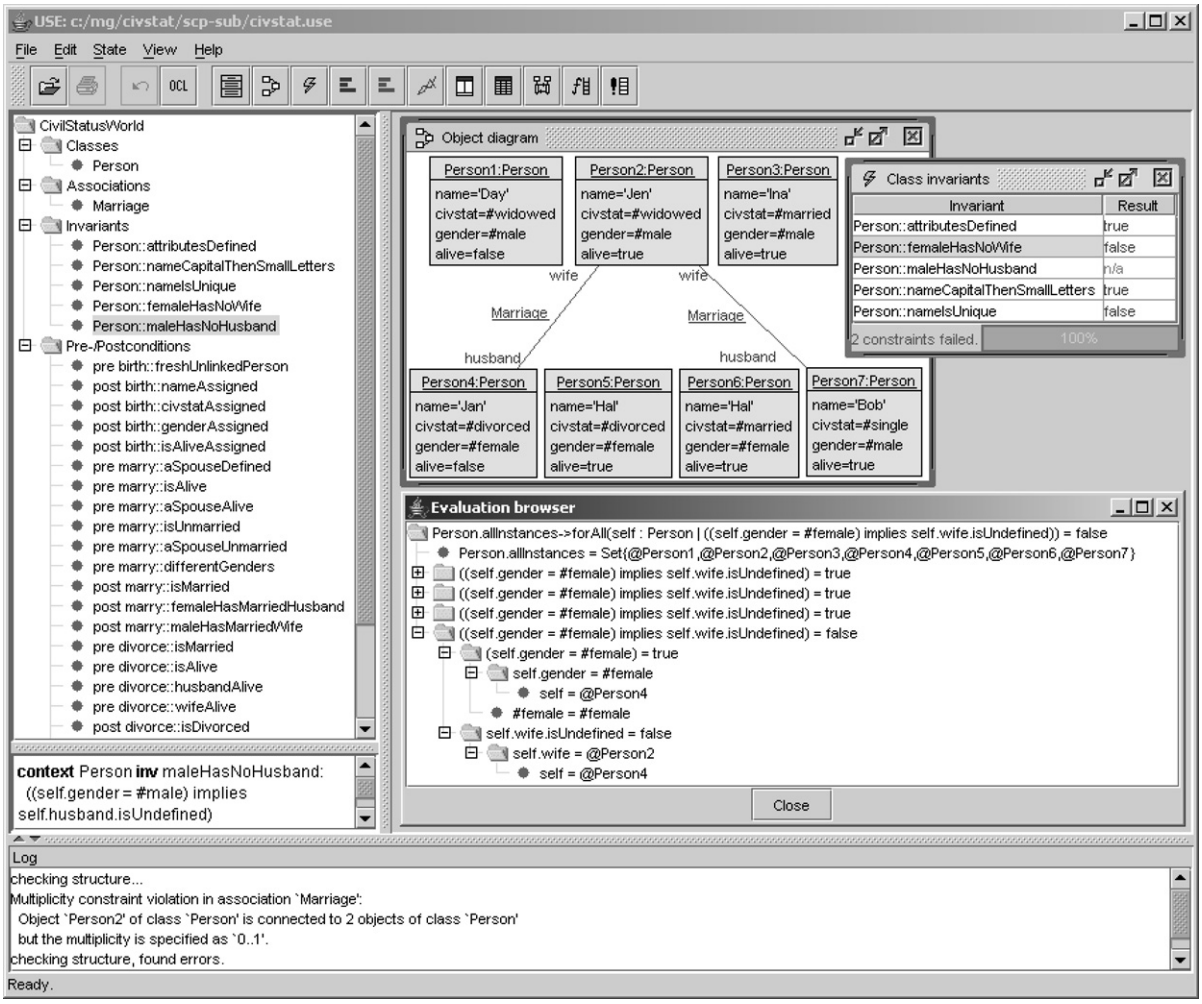

Fig. 2. Object diagram, class invariants, evaluation browser, and log.

conditions), and the window below the browser shows details about the selected model element (highlighted with grey color). Both figures introduce further functionalities in the right part of the window which are explained below.

2.1. Basic concepts

The first screenshot in Fig. 1 introduces the class diagram, the sequence diagram, the class extent, and the evaluation of OCL expressions.

Class diagram: A UML model is given to USE in textual form. The central parts of the example model are shown in the class diagram: One class with attributes and operations, one association with role names and multiplicities, and two enumerations. Structural restrictions (invariants) which determine the allowed object diagrams and behavioral restrictions (pre- and post-conditions) which narrow the allowed operation calls and operation returns are present in the model browser, but are not shown in the class diagram.

Sequence diagram: Execution of operations is indicated in the sequence diagram. The operation sequence including operation parameters is given to USE in a command shell. In the current operation sequence shown in the screenshot, all pre- and post-conditions are satisfied. Failing pre- or post-condition would have been highlighted and could be analyzed in more detail on the command shell. As an example for a complete operation description with pre- and post-conditions we show the operation `marry`.

```
marry(aSpouse:Person)
pre  aSpouseDefined: aSpouse.isDefined
pre  isAlive: alive
pre  aSpouseAlive: aSpouse.alive
pre  isUnmarried: civstat<>#married
pre  aSpouseUnmarried: aSpouse.civstat<>#married
pre  differentGenders: gender<>aSpouse.gender
post isMarried: civstat=#married
post femaleHasMarriedHusband: gender=#female implies
     husband=aSpouse and husband.civstat=#married
post maleHasMarriedWife: gender=#male implies
     wife=aSpouse and wife.civstat=#married
```

We use many small pre- and post-conditions (and invariants) with explicit names. Such small conditions give fine grained information explaining possible pre- or post-condition or invariant failure. Meaningful names help us to understand constraint failure. But operations are not only characterized descriptively by pre- and post-conditions. In USE, operations can be detailed also imperatively by so-called procedures in a language close to the Action Semantics language of UML with a Pascal-like syntax. As an example, we show the realization of operation `marry`. Expressions in brackets (like `[self.gender=#female]`) indicate parts containing possibly complex OCL expressions.

```
procedure Person_marry(self:Person,aSpouse:Person)
begin
  [self].civstat:=[#married]; [aSpouse].civstat:=[#married];
  if [self.gender=#female] then
    begin Insert(Marriage,[self],[aSpouse]); end
  else -- [self.gender=#male]
    begin Insert(Marriage,[aSpouse],[self]); end;
end;
```

The procedure shows how new links relating two persons are inserted into the Marriage association. The role names in Marriage possess the order (wife, husband) which has to be respected when manipulating links. A comment is indicated by `--`.

Class extent: The extent of class Person after the last operation has been executed indicates the current object identities (ada, bob, etc.) and the values the attributes (name, civstat, etc.) currently take. In this situation, the class extent uniquely determines a UML object diagram (with no links) characterizing the current system state. The class extent may optionally be shown with invariants evaluated separately for each object.

OCL expression evaluation: USE makes it possible to query the current system state by evaluating OCL expressions in an ad hoc manner. The example query retrieves the name and civil status of alive persons. In SQL, this query would be stated as `select name, civstat from Person where alive=true`. Thus, SQL-like query formulation and evaluation is possible in USE. The class extent window and the evaluate OCL expression window are two ways to inspect the current system state. The developer has the freedom to choose the most appropriate one.

2.2. Advanced concepts

The second screenshot in Fig. 2 presents the object diagram, the class invariant evaluation, the evaluation browser, and the log protocol.

Object diagram: A complete system state can be captured in an object diagram presenting objects with attribute values and object links. The object diagram in the example is generated by executing the procedure crowd shown below with appropriate parameters: crowd(3, 4, 2) (3 female persons, 4 male, 2 marriages). The procedure generates the object diagram with randomly chosen attribute values and links established also in a random way.

```
procedure crowd(numFem:Integer, numMale:Integer, numMarr:Integer)
var theFemales: Sequence(Person), theMales: Sequence(Person),
    f: Person, m: Person;
```

```
begin
  theFemales:=CreateN(Person,[numFem]);
  theMales:=CreateN(Person,[numMale]);
  for i:Integer in [Sequence{1..numFem}] begin
    [theFemales->at(i)].name:=Any([Sequence{'Ada','Bel','Cam',
      'Day','Eva','Flo','Gen','Hao','Ina','Jen'}]);
    [theFemales->at(i)].civstat:=
      Any([Sequence{#single,#married,#divorced,#widowed}]);
    [theFemales->at(i)].gender:=Any([Sequence{#female,#male}]);
    [theFemales->at(i)].alive:=Any([Sequence{false,true}]);
    end;
  for i:Integer in [Sequence{1..numMale}] begin
    [theMales->at(i)].name:=Any([Sequence{'Ali','Bob','Cyd',
      'Dan','Eli','Fox','Gil','Hal','Ike','Jan'}]);
    [theMales->at(i)].civstat:=
      Any([Sequence{#single,#married,#divorced,#widowed}]);
    [theMales->at(i)].gender:=Any([Sequence{#female,#male}]);
    [theMales->at(i)].alive:=Any([Sequence{false,true}]);
    end;
  for i:Integer in [Sequence{1..numMarr}] begin
    f:=Any([theFemales]); m:=Any([theMales]);
    Insert(Marriage,[f],[m]);
    end;
end;
```

The resulting object diagram does not conform to the multiplicities stated in the class diagram nor to the explicitly given invariants. This can formally be explored and explained by the following USE functionalities.

Class invariants: The window with the class invariants shows their evaluation in the current system state. An invariant can be evaluated to true, false, or can be not applicable (n/a). One has not to worry about a true invariant. A false invariant indicates an invalid system state which can be inspected by double-clicking the invariant name (here femaleHasNoWife), and this opens the Evaluation browser explained further down. If an invariant is not applicable, the Log window detailed below will tell us about the underlying reason.

Evaluation browser: The evaluation browser allows us to take a detailed view on a chosen invariant. The browser focuses on the invariant evaluation in order to display variable assignments. This allows us to understand invariant failure and to detect the violating parts of the object diagram. In the example, we see the explicit form of the invariant femaleHasNoWife in the first line. We further inspect the case where the implication premise is true and the implication consequence is false. This points to Person4 being a female but having Person2 as a wife which is forbidden by the invariant femaleHasNoWife. Thus the evaluation browser may be understood as a means to detect the invariant violators in the system state, that is, the objects leading to invariant failure.

Log protocol: The Log protocol reports on the validity of the multiplicity constraints from the class diagram. If a multiplicity constraint is violated, the violating object together with the actual violating multiplicity and the allowed class diagram multiplicity is stated. In the example, we learn that Person2 offends the multiplicity constraint 0..1 on the husband side of the Marriage association, because Person2 is connected to two husbands. The invariant maleHasNoHusband is classified as not applicable in the Class invariant window, because the term self.husband being part of the constraint is expected to return a single (or no) person, but in this state it returns a proper, two-element set of persons.

An object diagram like the one from above helps the developer in understanding the model in at least two ways: (1) It may show formally valid objects and links. This confirms the developer's choice of constraints or this leads to a tightening of the constraints. (2) It may show invalid objects and links, and with this the developer is again either confirmed in the constraint choice or this leads to a weakening of the constraints. In both cases, the obtained object diagrams may be regarded as positive or negative test cases which may even be transformed to other later software development phases.

As indicated by the top icons in the screenshots, USE provides other options apart from those we have shown but which we do not discuss here.

2.3. Certification

Above we have explained how USE can be employed for validating models through inspection. We now turn to handle formal properties by USE.

Consistency: USE can check the consistency of models. If a valid object diagram can be constructed, then the invariants are not contradictory.

Independence: Independence of a single invariant, that is the single invariant does not follow from another invariant or combination of other invariants, may be shown by constructing a system state which satisfies all other invariants but does not satisfy the single invariant whose independence has to be proved.

Implication: Implications of the actual invariants may be derived by showing that it is not possible to construct an object diagram for the invariants, if the negation of the implication to be proved is added to the current invariants.

Let us consider the following example which proves that our model implies the absence of bigamy. This is certified by the following command line protocol.

```
use> gen load bigamy.invs
     Added invariants: Person::bigamy
use> gen start civstat.assl attemptBigamy()
use> gen result
     Random number generator was initialized with 5649.
     Checked 663552 snapshots. Result: No valid state found.
```

In this command line protocol, first the following new invariant bigamy is added to the already present invariants.

```
context Person inv bigamy: wife.isDefined and husband.isDefined
```

Afterwards the procedure attemptBigamy() is called and USE reports that is has inspected 663552 snapshots (system states) but no valid state has been found. Why does USE inspect exactly this number of snapshots? In order to understand this, we first have to take a look at the procedure `attemptBigamy`.

```
procedure attemptBigamy()
var p: Person, w: Person, h:Person, thePersons: Sequence(Person);
begin
  thePersons:=CreateN(Person,[3]);
  for i:Integer in [Sequence{1..3}] begin
    [thePersons->at(i)].name:=Try([Sequence{'A','B','C'}]);
    [thePersons->at(i)].civstat:=
      Try([Sequence{#single,#married,#divorced,#widowed}]);
    [thePersons->at(i)].gender:=Try([Sequence{#female,#male}]);
    [thePersons->at(i)].alive:=Try([Sequence{false,true}]);
    end;
  p:=Try([thePersons]); w:=Try([thePersons->excluding(p)]);
  h:=Try([thePersons->excluding(p)->excluding(w)]);
  Insert(Marriage,[w],[p]); Insert(Marriage,[p],[h]);
end;
```

The procedure tries to construct a bigamy snapshot (recall that we have loaded the invariant bigamy) by creating three persons and two Marriage links and where a single person plays both the role of a wife and the role of a husband. The procedure tries all possibilities for all attributes and all roles but does not succeed. But, again, why does USE come across 663 552 snapshots?

A single person can take 3 name values (‘A’, ‘B’, ‘C’), 4 civil status values (single, married, divorced, widowed), 2 gender values (female, male), and 2 alive values (false, true). These together give 3*4*2*2 = 48 possibilities for a single person. We consider 3 persons and therefore get 48*48*48 = 110 592 choices. For the marriage

links (assignments to p, w, h), we obtain 3*2*1 = 6 configurations, and this makes 6*110 592 = 663 552 possibilities. This shows: Our model guarantees the absence of bigamy. We are aware of the fact that this is not a complete formal proof, and that the result only holds as long as all assumptions which we have made in connection with the procedure attemptBigamy() are true. But in this case, the procedure makes no inadmissible assumptions, and this certifies the desired property.

3. Comparison with related tools and systems

A comparison of tools for OCL can be found in [14]. Relevant related work includes the Dresden OCL compiler [8] compiling OCL into Java code, the OCLE system having a similar scope as USE but no automatic snapshot facility [6], the Kent Modeling Framework KMF [2] allowing to use OCL for its own Java projects, the Octopus [10] OCL 2.0 syntax checker, BoldSoft's tool ModelRun [4], the KeY system [1] based on TogetherJ and allowing interactive verification of OCL properties, a recent approach compiling OCL to C# [3], and work translating (a simplified version of) OCL into the theorem prover PVS [9]. Few commercial UML tools (e.g., Poseidon, MagicDraw, MaxUML, Together, XMF-Mosaic) provide basic OCL support.

The first version of USE was released in 1998. Since then, it has become a useful and mature tool. The theoretical work in connection with USE proposing an OCL metamodel and defining the formal semantics of OCL found its way into the OCL 2.0 OMG standard. USE is the only system allowing snapshots to be generated automatically. The combination of validation and certification techniques also seems to be quite unique.

4. Relevant theory

The syntax of UML and OCL is defined with a metamodel using UML (MOF). The semantics of OCL and the needed class diagram features of UML is expressed in terms of plain set theory in [11] which has been implemented in the USE system hand in hand with the formal semantics. This set-theoretic semantics of OCL is part of the OMG standard and additionally expressed in that standard in terms of a metamodel.

5. Conclusion

We have described the tool USE which allows UML models with OCL constraints (invariants and pre- and post-conditions) to be validated against developer's assumptions. USE allows to a certain extent the checking of formal properties. USE permits us to review the consistency of UML models and the independence of constraints. USE makes it possible to certify properties. It can be shown that under particular assumptions certain constraints are logical consequences of a given UML model. Thus, USE supports developers in analyzing the model structure and behavior and in exploring properties of models. USE (as all other OCL systems we are aware of) does not allow full automatic formal verification of arbitrary properties formulated in OCL.

USE has been employed in a number of larger case studies which are described on the publication page of our group. USE is currently utilized is a model transformation project where XML and documentation is generated from models. Our experience shows that large models (e.g., an integrated UML and OCL metamodel) and larger system states (e.g., several hundreds of objects) can be tackled with USE.

The latest version improves the modeling possibilities for sequence diagrams, and the next version will enhance the evaluation browser. We plan to upgrade the ease of using USE. In particular, the following topics will be treated: (A) Object layout (choosing from different automatic layout algorithms) and presentation possibilities (e.g., filtering of displayed objects by queries) in the object diagram, (B) adjusting the command line interface and the graphical user interface, (C) revising the implicit defaults and providing the possibility for storing named configurations and preferences (e.g., for class, object and sequence diagram and evaluation browser properties).

Further development will extend USE to make model refactorization and transformation possible. This is a first step towards a tool supporting model-driven engineering. Apart from these conceptional extensions, work has to take into account continuous integration of smaller changes, improvements and the elimination of bugs.

Supplementary data

Supplementary data associated with this article can be found, in the online version, at doi:10.1016/j.scico.2007.01.013.

References

[1] W. Ahrendt, T. Baar, B. Beckert, M. Giese, E. Habermalz, R. Hähnle, W. Menzel, P.H. Schmitt, The KeY approach: Integrating object oriented design and formal verification, in: M. Ojeda-Aciego, I. de Guzmán, G. Brewka, L.M. Pereira (Eds.), Proc. 8th European Workshop Logics in AI, JELIA'2000, in: LNCS, vol. 1919, Springer, 2000, pp. 21–36.
[2] D. Akehurst, O. Patrascoiu, The Kent Modeling Framework (KMF), http://www.cs.kent.ac.uk/projects/ocl, University of Kent, 2005.
[3] D. Arnold, OCL/C# Compiler, www.ewebsimplex.net/csocl/, ewebsimplex, 2005.
[4] Boldsoft, Boldsoft OCL Tool Model Run, www.boldsoft.com, Boldsoft, Stockholm, 2002.
[5] F. Büttner, H. Bauerdick, M. Gogolla, Towards transformation of integrity constraints and database states, in: D.C. Martin (Ed.), Proc. Dexa'2005 Workshop Logical Aspects and Applications of Integrity Constraints, LAAIC'2005, IEEE, Los Alamitos, 2005, pp. 823–828.
[6] D. Chiorean, Using OCL beyond specifications. In: A. Evans, R. France, A. Moreira, B. Rumpe (Eds.), Proc. UML'2001 Workshop Rigorous Development, LNI, GI, Bonn, 2001, pp. 57–68.
[7] M. Gogolla, J. Bohling, M. Richters, Validating UML and OCL models in USE by automatic snapshot generation, Journal on Software and System Modeling (2005).
[8] H. Hussmann, B. Demuth, F. Finger, Modular architecture for a toolset supporting OCL, in: A. Evans, S. Kent, B. Selic (Eds.), Proc. 3rd Int. Conf. Unified Modeling Language, UML'2000, in: LNCS, vol. 1939, Springer, 2000, pp. 278–293.
[9] M. Kyas, H. Fecher, F.S. de Boer, J. Jacob, J. Hooman, M. van der Zwaag, T. Arons, H. Kugler, Formalizing UML models and OCL constraints in PVS, Electronic Notes in Theoretical Computer Science 115 (2005) 39–47.
[10] Klasse Objecten, The Klasse Objecten OCL Checker Octopus, www.klasse.nl/english/research/octopus-intro.html, Klasse Objecten, 2005.
[11] M. Richters, A precise approach to validating UML models and OCL constraints, Ph.D. Thesis, Universität Bremen, Fachbereich Mathematik und Informatik, Logos Verlag, Berlin, BISS Monographs, No. 14, 2002.
[12] M. Richters, M. Gogolla, OCL - syntax, semantics and tools, in: T. Clark, J. Warmer (Eds.), Advances in Object Modelling with the OCL, in: LNCS, vol. 2263, Springer, Berlin, 2001, pp. 43–69.
[13] J. Rumbaugh, G. Booch, I. Jacobson, The Unified Modeling Language 2.0 Reference Manual, Addison-Wesley, Reading, 2003.
[14] A. Toval, V. Requena, J. Fernandez, Emerging OCL tools, Software and Systems Modeling 2 (4) (2003) 248–261.
[15] J. Warmer, A. Kleppe, The Object Constraint Language: Precise Modeling with UML, 2nd edition, Addison-Wesley, 2003.