Towards Model Validation and Verification with SAT Techniques

Martin Gogolla

Universität Bremen, Informatik, AG Datenbanksysteme, D-28334 Bremen

Abstract. After sketching how system development and the UML (Unified Modeling Language) and the OCL (Object Constraint Language) are related, validation and verification with the tool USE (UML-based Specification Environment) is demonstrated. As a more efficient alternative for verification tasks, two approaches using SAT-based techniques are put forward: First, a direct encoding of UML and OCL with Boolean variables and propositional formulas, and second, an encoding employing an intermediate, higher-level language (Kodkod, stongly connected to Alloy). A number of further, presently not realized verification and validation tasks and the transformation of higher-level modeling concepts into simple UML/OCL models, which are checkable with SAT-based techniques, are shortly discussed. Finally, the potential of SAT-based techniques for model development is again emphasized.[1]

1 Introduction

We assume system development is done in an object-oriented way by using a modeling language like UML [RBJ05]. In the first sections of this paper, we restrict ourselves to a very small, but precise subset of UML, namely class diagrams and OCL [WK03] invariants and OCL pre- and postconditions. In a later section, we will explain how other UML diagrams like statechart or use case diagrams can be integrated into the approach.

In the early phases of system development, the model under development will be explored by and worked out on the basis of so-called scenarios as pictured in Fig. 1. The class diagram together with the invariants determines the system states, also called snapshots or, in UML terms, object diagrams. A system state encloses objects, connections to other objects (in UML terms links) and object properties (attribute values). The state transitions are determined by the operations which are described by pre- and postconditions. Operations are realized by so-called command sequences which may create or delete objects or links and may modify attribute values. A scenario is manifested by a collection of system states and state transitions.

[1] Partially joint work with Rolf Drechsler, Mirco Kuhlmann, Mathias Soeken, and Robert Wille from the University of Bremen.

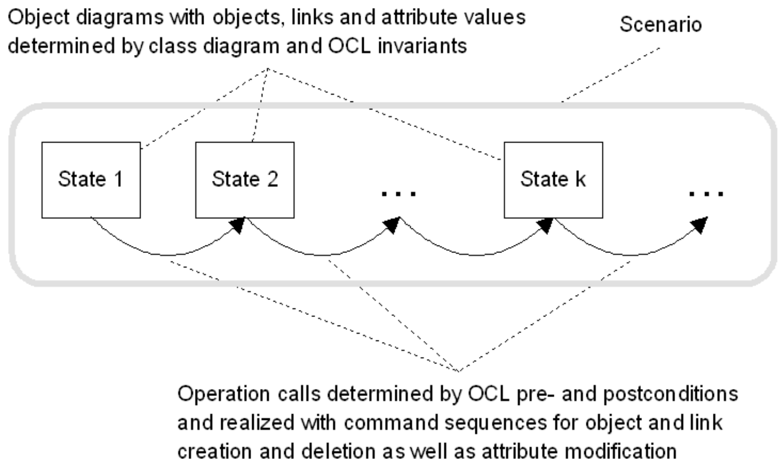

Fig. 1. General Components of a Scenario

2 Validation and Verification with USE

Let us consider an example. The class diagram in Fig. 2 describes persons and their civil status. The class diagram will be extended below by OCL invariants and the operations will be detailed with OCL pre- and postconditions. Such a model can be validated and animated with the UML-based Specification Envi-

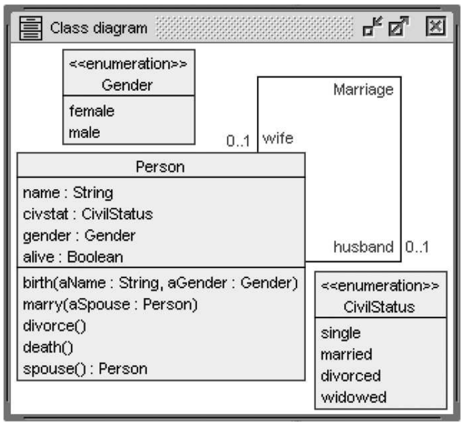

Fig. 2. Example Class Diagram

ronment USE [GBR05,GBR07]. Figure 3 shows two screenshot from USE and illustrates the model with one scenario: The upper part of Fig. 3 represents the beginning of the scenario with five operation calls and the lower part of Fig. 3 adds another operation call. Both stages of the scenario are portayed with five USE windows providing different functionality.

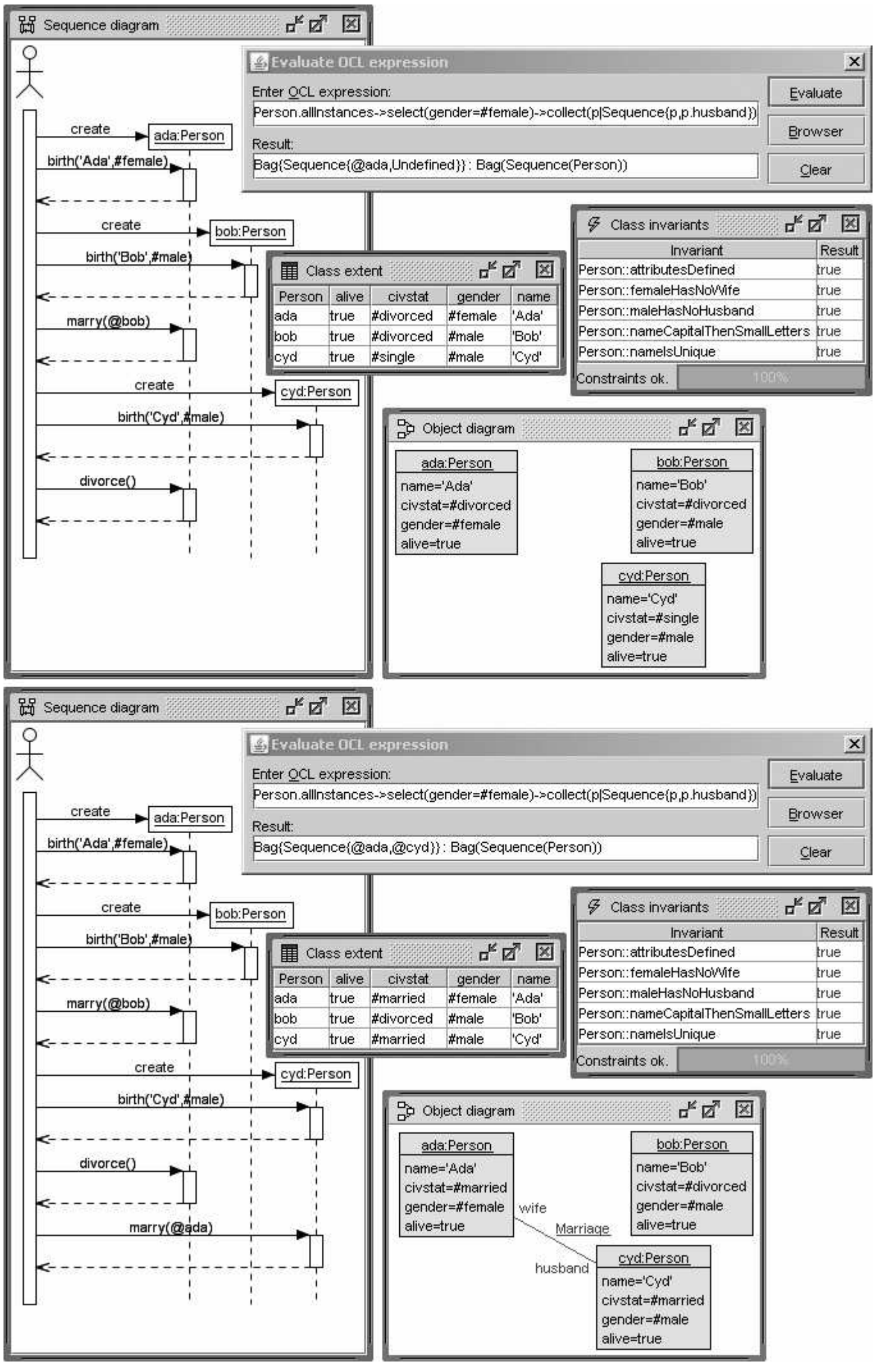

Fig. 3. Example Scenario Marriage-Divorce-Marriage

Sequence diagram: The objects together with their lifelines and the operation calls detailed with the actual parameter values are displayed.

Object diagram: The objects with their attribute values and their links are shown in the state that is reached after the last operation from the sequence diagram has been completed.

Class extent: The objects and the object attribute values are represented in a relation-like form.

Class invariants: The current state of the five defined class invariants is stated. An invariant can evaluate to false, true or not-applicable. This can happen, if the actual multiplicities in the object diagram do not meet the required multiplicities from the class diagram.

Evaluate OCL expression: The current state can be inspected with OCL expressions. The OCL expression is evaluated in the current system state.

The five invariants state that all attributes are different from undefined, that strings for names have a particular format, that names are unique, that females do not have a wife and that males do not have a husband.[2]

```
constraints
context self:Person
  inv attributesDefined: name.isDefined and civstat.isDefined and
      gender.isDefined and alive.isDefined
  inv nameCapitalThenSmallLetters:
      let small:Set(String)=Set{'a','b','c',...,'x','y','z'} in
      let capital:Set(String)=Set{'A','B','C',...,'X','Y','Z'} in
      capital->includes(name.substring(1,1)) and
      Set{2..name.size}->forAll(i |
        small->includes(name.substring(i,i)))
  inv nameIsUnique: Person.allInstances->forAll(self2|
      self<>self2 implies self.name<>self2.name)
  inv femaleHasNoWife: gender=#female implies wife.isUndefined
  inv maleHasNoHusband: gender=#male implies husband.isUndefined
```

Regarding pre- and postconditions, we only show these for the operation marry. The other pre- and postconditions look similar.

```
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
```

[2] This conservative view on marriages originates from the payer of the development and does not represent the view of the developer.

```
     husband=aSpouse and husband.civstat=#married
post maleHasMarriedWife: gender=#male implies
     wife=aSpouse and wife.civstat=#married
```

The realization of an operation call can be done in USE in different ways: It can be done in an interactive way on the graphical user interface, it can be done with a command sequence on the command line interface or it can be done with the script language ASSL (A Snapshot Sequence Language) [GBR05]. Below you see the realization of the operation marry as a command sequence and as an ASSL procedure.

```
-- cmd file -----------------------------------------------------------
-- Person::marry(aSpouse:Person) to be called with self:Person ---------
!set self.civstat:=#married
!set aSpouse.civstat:=#married
!insert
  (if self.gender=#female then self else aSpouse endif,
   if self.gender=#female then aSpouse else self endif) into Marriage

-- ASSL procedure -----------------------------------------------------
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 original purpose of ASSL was however to search through a class of object diagrams and to check whether an object diagram satisfyling particular properties can be found. Consider the following ASSL procedure which tries to construct an object diagram where a single person is married twice taking the role of the husband in the first marriage and taking the role of the wife in the second marriage.

```
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 above ASSL procedure is used within a context where the following additional invariant bigamy is known.

```
context Person inv bigamy: Person.allInstances->exists(p|
  p.wife.isDefined and p.husband.isDefined)
```

The following command line protocol shows that the additional invariant is added to the model and that the attemptBigamy procedure is called without success. The procedure searches all system states with three persons where all possibilities for attributes and links are tried, but does not succeed. This proves that in the considered search space the model is bigamy free.

```
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.
```

3 Validation and Verification Approaches using SAT-based Techniques

We are currently working on two approaches in which we apply SAT-based techniques for model validation and verification tasks. One approach uses a direct encoding of models and model properties into a language which can be fed into a SAT solver directly. Some details of this work can be found in [SWK+10]. The other approach employs an intermediate language for the representation of models and model properties and is based on Kodkod [TJ07], an API for Alloy [Jac06].

3.1 Direct Encoding

In Fig. 4 we see an example model from [SWK+10]. The model is represented directly with boolean variables and is handed then to an off-the-shelf SAT solver. In Fig. 5 we identify the representation of a system state with three register objects and one processor object. The model attributes are given as boolean vectors of appropriate length named α for attribute. The model links are also given as boolean vectors named λ for link. Details of this approach can be found in [SWK+10].

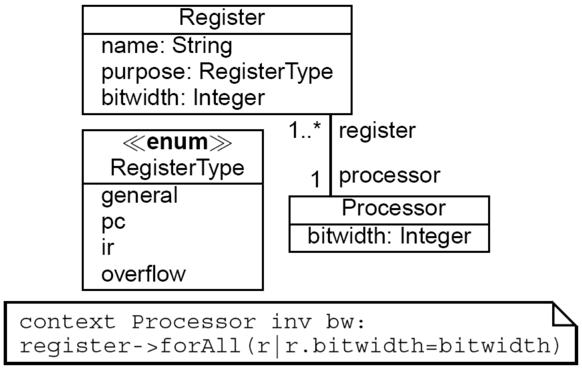

Fig. 4. Example Model for Direct SAT Representation

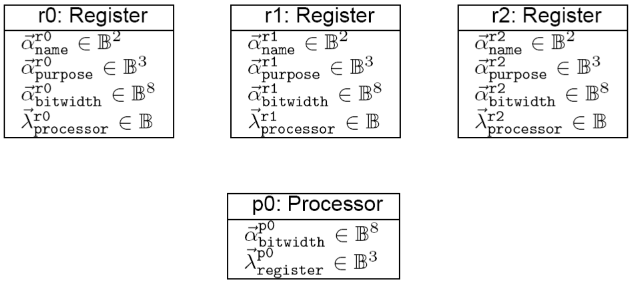

Fig. 5. Example System State for Direct SAT Representation

3.2 Encoding using an Intermediate Language

Our encoding of UML and OCL in Kodkod, an API for Alloy, first defines a so-called universe of constants over which formulas may be quantified. The universe has to take into account that a particular system state is going to be explored. For our civil status example, let assume we want to construct a system state with three Person objects and three String values for the name attribute. Then we define three constants P1, P2, P3 and three constants for the different names. Furthermore, constants for the enumerations and for the datatype Boolean have to be provided. Note that the universe is partly determined by the system state and partly by the features from the class diagram.

```
{P1, P2, P3,
 S_Ada, S_Bob, S_Cyd,
 C_single, C_married, C_divorced, C_widowed,
 G_female, G_male,
 B_true, B_false}
```

Kodkod as well as Alloy is based on relations and a relational logic. Therefore we define a unary relation Person expressing the typing information. Kodkod allows a relation to be defined with a lower bound and an upper bound for the set of tuples included in the relation. For Person, lower and upper bound coincide. This is not the case for the binary relation name expressing with the lower bound that no name information is allowed and expressing with the upper bound each Person can take any name in principle. Analogously the binary relation marriage expresses that no marriages are allowed as the lower bound and that any person can be married to anybody else as the upper bound. Analogous definitions must be given for the other attributes from the class diagram.

```
Person :1 [{<P1>,<P2>,<P3>},
           {<P1>,<P2>,<P3>}]

String :1 [{<S_Ada>,<S_Bob>,<S_Cyd>},
           {<S_Ada>,<S_Bob>,<S_Cyd>}]


name :2 [{},{<P1,S_Ada>,<P1,S_Bob>,<P1,S_Cyd>,
             <P2,S_Ada>,<P2,S_Bob>,<P2,S_Cyd>,
             <P3,S_Ada>,<P3,S_Bob>,<P3,S_Cyd>}]

marriage :2 [{},{<P1,P1>,<P1,P2>,<P1,P3>,
                 <P2,P1>,<P2,P2>,<P2,P3>,
                 <P3,P1>,<P3,P2>,<P3,P3>}]
```

An essential and elegant feature of the relational logic of Kodkod and Alloy is the use of the relational join operator to express the access to a relation. For example, if we have a Person variable p then we write in OCL p.husband or p.wife to access the marriage association. In Kodkod we would express this as p.marriage or marriage.p assuming that the first parameter in the relation is used for wife and the second parameter for the husband.

Typing information, multiplicities and constraints must be expressed in the relational logic. Below we show some examples for the civil status class diagram. The description is not complete. More formulas are needed. The last formula expresses the bigamy constraint. Thus our expectation would be that Kodkod replies with the answer that the formula is not satisfiable.

```
-- name defined on Person, yields String and total function
(all u:Universe |
  name.u in Person and u.name in String and
  lone u.name and u in Person equiv one u.name)

-- marriage reflexive association on Person with 0..1 multiplicities
(all u:Universe |
  u.marriage in Person and marriage.u in Person and
  lone u.marriage and lone marriage.u)

-- nameIsUnique
(all p1:Person | (all p2:Person |
  p1<>p2 implies p1.name<>p2.name))

-- femaleHasNoWife
(all p:Person |
  p.gender=G_female implies no marriage.p)

-- bigamy
(some p:Person | one p.marriage and one marriage.p)
```

4 Verification and Validation Tasks

Below we indicate a collection of questions which one can pass over to a UML and OCL verification system on the basis of given set of invariants and a given set of pre- and postconditions.

The first set of questions relates to logical deductions.

- Are the invariants consistent?
- Are the invariants independent?
- Is a given constraint deducible from the invariants?
- Is an operation precondition consistent to the invariants?
- Is an operation precondition deducible from the invariants?
- Is an operation precondition independent from the invariants?
- Is a given constraint deducible from an operation precondition?
- Are there operations which exclude each other w.r.t. preconditions?
- Is an operation postcondition consistent to the invariants?
- Is an operation postcondition deducible from the invariants?

– Is an operation postcondition independent from the invariants?
– Is a given constraint deducible from an operation postcondition?
– Are the invariants respected by the operations?

The second set of questions relates to scenarios.

– Are the constraints too strong or too weak?
– Are there scenarios which should be accepted but are not?
– Are there scenarios which are accepted but should not?
– Is there a scenario with a finite number of states?
– Is there a scenario with an infinite number of states?
– Are there states from which no transition can be made?
– Is there a scenario leading from stateBegin to stateEnd where states are explicitly given?
– Is there a scenario leading from stateBegin to stateEnd in which OCLexpressionBegin and OCLexpressionEnd are valid
– Is there a scenario with non-empty class populations?
– Is there a scenario with all classes/attributes/roles/operations involved?
– Starting from the empty state and executing operations whose pre- and postconditions are satisfied, does one reach only states satisfying the invariants?
– Starting from a given state satisfying the invariants and executing operations whose pre- and postconditions are satisfied, does one reach only states satisfying the invariants?

5 Transforming UML Concepts into Object Models and OCL

The approach presented so far covers class diagrams and object diagrams which are detailed by OCL constraints. However, other UML or SysML diagrams can be integrated.

– One can encode statechart diagrams as OCL pre- and postconditions as it has been discussed in [GR02].
– One can understand sequence diagrams and communication diagrams as representations of scenarios. This is already partly reflected in USE in form of sequence diagrams. The representation with communication diagrams is under development.
– Activity diagrams may represent operation implementations.
– Use cases and use case diagrams can be understood as system operations, i.e., an arrangement of calls to objects with parameters.

6 Conclusion

The proposed approach formally describes system structure and behavior based on concepts currently used in UML, but these concepts can used for DSM (Domain-Specific Modeling) as well. We discussed here only questions on the model level, but analogous questions could also be asked on the meta-model level. System quality assurance has been put forward through verification and validation tasks with a mixture of proof- and scenario-based quality checks. Verification tasks are expected to be performed very efficiently with SAT tools even for large scenarios.

References

[GBR05] Martin Gogolla, Jörn Bohling, and Mark Richters. Validating UML and OCL Models in USE by Automatic Snapshot Generation. *Journal on Software and System Modeling*, 4(4):386–398, 2005.

[GBR07] Martin Gogolla, Fabian Büttner, and Mark Richters. USE: A UML-Based Specification Environment for Validating UML and OCL. *Science of Computer Programming*, 69:27–34, 2007.

[GR02] Martin Gogolla and Mark Richters. Development of UML Descriptions with USE. In Hassan Shafazand and A Min Tjoa, editors, *Proc. 1st Eurasian Conf. Information and Communication Technology (EURASIA'2002)*, pages 228–238. Springer, Berlin, LNCS 2510, 2002.

[Jac06] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, 2006.

[RBJ05] J. Rumbaugh, G. Booch, and I. Jacobson. *The Unified Modeling Language Reference Manual, Second Edition*. Addison-Wesley, Reading, 2005.

[SWK+10] Mathias Soeken, Robert Wille, Mirco Kuhlmann, Martin Gogolla, and Rolf Drechsler. Verifying UML/OCL Models Using Boolean Satisfiability. In WolfgangMüller, editor, *Proc. Design, Automation and Test in Europe (DATE'2010)*. IEEE, 2010.

[TJ07] Emina Torlak and Daniel Jackson. Kodkod: A relational model finder. In Orna Grumberg and Michael Huth, editors, *TACAS*, LNCS 4424, pages 632–647. Springer, 2007.

[WK03] J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, 2003. 2nd Edition.