Applying ASP to UML Model Validation

Mario Ornaghi, Camillo Fiorentini, Alberto Momigliano, and Francesco Pagano

Dipartimento di Scienze dell'Informazione, Università degli Studi di Milano, Italy
{ornaghi,fiorenti,momiglia,pagano}@dsi.unimi.it

Abstract. We apply ASP to model validation in a CASE setting, where models are UML class diagrams and object diagrams are called "snapshots". We present the design and implementation of MSG, a snapshot generator for UML models that employs DLV-Complex as a generator engine, the answer sets representing the legal snapshots.

1 Introduction

The object of this research is the application of ASP to model validation in a CASE setting, in particular evaluating the "correctness" of formal specifications (or *models*) with respect to their requirements. Here, models are UML class diagrams [8] with constraints, typically written in OCL: a diagram should represent an abstraction of the problem domain; the objects populating a system state should represent a "snapshot" of a corresponding counterpart in the modeled world. In the UML, snapshots are represented by object diagrams. The legal snapshots are those satisfying the constraints that can be attached to the model to better specify the desired properties. In this context, tools for snapshot generation (SG) are an important part of the "weaponry" of light-weight formal methods. In fact, the relevance of SG for validation and testing in OO software development is widely acknowledged and a relevant part of the recently branded field of "Model-Based Testing" [4]. The latter ranges from model animation to ways of establishing partial certification such as model consistency and constraints independence checking.

This paper presents the design and implementation of a snapshot generator for UML models called "MSG" (read as "Message" and standing for "Milano Snapshot Generator", `cooml.dsi.unimi.it/msg`), which employs DLV-Complex [3] as a generator engine, the answer sets representing the legal snapshots. This is integrated in a system that takes as input any UML class diagram in XMI format and eventually displays back to the user the answers, i.e. the snapshots in the same format. The main theoretical contribution consists in a specialized representation of UML class diagrams into DLV-Complex, tailored to the *fully automatic* generation of *non isomorphic* snapshots The representation makes essential use of DLV-Complex's external functions, but still requires the introduction of an intermediate language (DLVExi) adding polymorphic types and existential quantification [5].

Background on UML. The Unified Modeling Language (UML) comprises a variety of model types for describing system properties, both static (e.g. class models, object models) and dynamic (e.g. state-machines, activity graphs). One of the more prominent

E. Erdem, F. Lin, and T. Schaub (Eds.): LPNMR 2009, LNCS 5753, pp. 457–463, 2009.

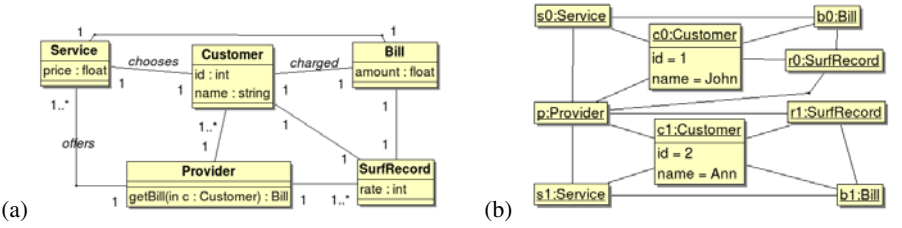


Fig. 1. A class diagram of an Internet Service Provider (a) and a snapshot for it (b)

model types is the class model (visualized as a class diagram) used to represent the underlying data model of a system in an object-oriented manner. A class diagram consists of a set of classes (rectangles) and relations among them, in particular associations (lines connecting rectangles), as shown in our running example (Fig. 1 (a)), a simplified version of the ISP example from `http://www.brucker.ch/projects/hol-ocl/`. In this scenario a `Provider` `offers` some `Service`(s) at a certain `price`. A `Customer` `chooses` one of these services and she is `charged` a `Bill` according to her `SurfRecord` and the download `rate`. According to the UML "type-instance" dichotomy [8], classes, associations and class diagrams represent "types". Classes instantiate over "objects", associations over "links" and class diagrams over object diagrams, called "snapshots". Fig. 1 (b) shows a snapshot, where a provider `p` (precisely, an object `p` of class `Provider`) offers two services `s0`, `s1` and has two clients `c0`, `c1`, where `c0` chooses `s0`, and so on. Association ends are decorated with various multiplicities. For example, the multiplicity `1..*` at the `Service`-end of the association `offers` denotes that a provider may offer one or more services, while `1` at the `Provider`-end indicates that a service is offered by one provider.

We remark that we can omit *oids* (object identifiers) in a snapshot. One way to look at this is considering oids as abstractions of memory addresses, which are transparent in OO programming. Thus, two snapshots of the same class diagram should be considered equal if they are *isomorphic*, where, roughly speaking, a snapshot isomorphism is a bijective map among oids that preserves the navigations. For example, if in the ISP snapshot we map `c0` into `c1` and `c1` into `c0`, we get an isomorphic copy, i.e., morally, the "same" snapshot.

2 Model Validation and Snapshot Generation

Since requirements are informal, model validation can be only empirical, i.e., it is performed by comparing the formal model with the user's expectations. In this context, a snapshot generation tool (SGT) plays an important role. A SGT has two inputs: a model M and a set G of *generation requests* (GR), needed to make the number of snapshots finite. SGT output the set of legal snapshots that satisfy G. It allows us to perform various "experiments". To name one, the mere existence of a snapshot ensures that the model is *consistent* with its constraints.

Fig. 2 shows the architecture and the data-flow of MSG. We start with the open source UML tool BOUML (`http://bouml.free.fr/`) to design diagrams and to generate the

corresponding XMI representation. At the end of SG, BOUML displays the snapshots produced by the *as2xmi* module from the XMI format. The translation from XMI to DLV has been divided in two phases, using an intermediate language DLVExi. The latter allows us to decouple the representation of XMI models in logic from the definition and implementation of the generation request language. The Java component XMI2DLVEXI translates an XMI model M into a DLVExi program E_M, which is a faithful representation of M in the following sense: every legal snapshot of M is represented by an "answer set" of E_M and every "answer set" of E_M represents a legal snapshot of M. The component TODLV translates the program E_M and the generation requirements G into a DLV-Complex program $P_{M,G}$. The answer sets of $P_{M,G}$ are the answer sets of E_M that satisfy G. DLV-Complex [2] is used as the generator engine.

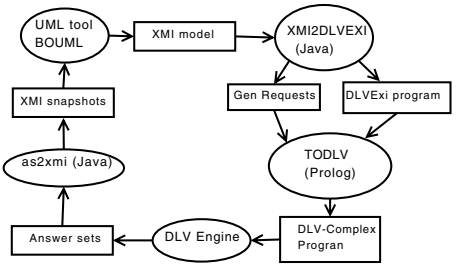


Fig. 2. The data flow view of the system

3 Representing UML into ASP

In this section we discuss some encoding techniques for UML+OCL class diagrams in DLV-Complex. We do this via an intermediate language, DLVExi, which can be seen as an extension of DLV-Complex with a ML-like polymorphic type system and allowing conjunction and existential quantification in the head of clauses, as shown by the following fragment of `trel`, one of the DLVExi encodings of the UML with which we have experimented.

```
type obj(C) --> o(C).   type typeId(X) --> tid.
type mult --> m(int,int); star(int); union(mult,mult).
type association(C1,C2) --> ass(assoc_name). ...
pred object(obj(C)), is_association(association(C1,C2)), is_class(class(C)),
    link(association(C1,C2),obj(C1),obj(C2)), att_rec(obj(C),T),
    mLeft(association(C1,C2),obj(C2),int),
    mRight(association(C1,C2),obj(C1),int),
    leftMult(association(C1,C2),mult),
    rightMult(association(C1,C2),mult),  violates(int,mult), ...
trel(C1:type, C2:type, X:C1, Y:C2, A:association(C1,C2)) isunit {
  object([C1],o(X)) v neg(object([C1],o(X))) if is_class(tid([C1])),   %g1
  link(A,o(X),o(Y)) v neg(link(A,o(X),o(Y))) if                        %g2
          is_association(A) & object(o(X)) & object(o(Y)),
  exi([v], att_rec(o(X),v)) if  object(o(X)),                          %ec
  false if leftMult(A,M)&object(o(Y))&mLeft(A,o(Y),N)&violates(N,M),   %t1
  false if rightMult(A,M)&object(o(X))&mRight(A,o(X),N)&violates(N,M)  %t2}.
```

Before explaining `trel`, some brief comments on the DLVExi language are in order. Types are expressed analogously to `datatype` declarations in functional programming languages, where `-->` productions introduce polymorphic types by listing the

type constructors (also called generators). For example, `mult` (multiplicity) is generated by `m(int,int)`, `star(int)` and, recursively, `union(mult,mult)`. The ground `mult`-terms represent multiplicities, for example `union(m(1,2),star(5))` represents `1..2, 5..*`. One use of (polymorphic) types is as wrappers, abstracting away from the types of the specific UML model. In particular, `obj(C)` is the type of the oids for a class type `C` and `association(C1,C2)` the type of the associations between class `C1` (left hand side) and `C2` (right hand side). For the sake of type safe grounding [5], every ground term must have a unique type. To this aim, [5] introduces annotated functions $f_J(\ldots)$ and predicates $p_J(\ldots)$. In our concrete syntax, the annotations J are enclosed between square brackets. For example, `tid([C1])` is the concrete syntax of $\texttt{tid}_{C1}$, while `object([C1],o(X))` is $\texttt{object}_{C1}\texttt{(o(X))}$. Annotations may be left understood and are reconstructed by the system. If multiple annotations are possible, the system produces an error message. Polymorphic types allow us to decouple the general representation choices from the signature of the specific UML model. In this way, we can represent a UML model M by a DLVExi theory $T_M = R \cup E_M$, where R is a general "representation theory", and an "encoding theory" E_M representing M in R. The theory R does not depend on M, but only on the representation choices and the generation strategy. For example, the above `trel` corresponds to a *relational representation* of the associations, by means of the predicates `object(O)` ("`O` is a live object") and `link(A,O1,O2)` ("`O1` and `O2` are linked by the association `A`"). According to the "guess and test" methodology of DLV [9], live objects and links are *guessed* by the rules `%g1` and `%g2`, while `%t1` and `%t2` "test" the multiplicity constraints. The XMI2DLVEXI component translates M into the encoding theory E_M. For example, part of the ISP-encoding (Fig. 1 (a)) is:

```
type customer. type bill.  ...  % the types for the ISP classes
type customer_atb --> rec(id:int, name:string). % the types for the
type bill_atb --> rec(amount:float).            % attribute-records
... ISP isunit {
   is_class(tid([bill])) if true, % "bill"  identifies a class-type
   ...
   is_association(ass([customer,bill],charged)) if true,
   % "charged" is the name of an association between customer and bill
   ...
   leftMult(ass([customer,bill],charged),m(1,1)) if true,
   % the multiplicity on the customer association end of "charged" is 1
   % ....}
```

Inputs to the TODLV component are the theory $T_M = R \cup E_M$ and a set GR of generation requests. The output is a DLV-Complex program. The generation requests suggest a finite set of possible object identifiers and a finite set of attribute values, in order to get finitely many models. Examples of GR for `customer` and `bill` are:

```
type customer --> c1; c2.  type bill --> b1; b2.            %i
att_rec([customer], c1,V) if  V=rec(0,ted),                 %ii
att_rec([customer], c2,V) if  V=rec(1,mary), att_rec([bill],B,V)
if member(V,[rec(12.3), rec(10.5)]), ...
```

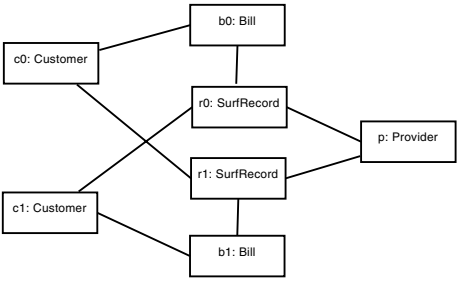
By `%i`, we fix a finite set of possible oids, while `%ii` gives a finite set of "witness-choices" for the existential variable `V` of the clause `%ec`. The TODLV component replaces the

existential formula with a disjunction over the witness-choices, as shown in the clauses `%c4` of the following DLV-Complex program:

```
of(obj(C),o(X)) :- is_class([C],tid([C])), of(C,X). ...                %c1
object([C],o(X)) v -object([C],o(X)) :- of(C,X),is_class([C],tid([C])).
link([C1,C2], A, o(X), o(Y)) v -link([C1,C2],A, o(X), o(Y)))) :-
        is_association([C1,C2],A), object([C1],o(X)), object([C2],o(Y)).
...
is_class([bill],tid([bill])). ...                                      %c2
is_association([customer,bill], ass([customer,bill],charged)).
....
of(customer,c1). of(customer,c2). of(bill,b1). of(bill,b2).            %c3
....
att_rec([customer],c1,rec(0,john)). att_rec([customer],c2,rec(1,mary)).%c4
att_rec([bill], B, rec(12.3)) v att_rec([bill], B, rec(10.5)) :-
      object([bill],o(B)). ...
```

Clauses `c1` are the DLV-Complex translation of `trel` (we use `of(C,X)` as "`X` is of type `C`"), excluding `%ec`. Clauses `c2` comes from the ISP-encoding, `c3` from the generation requests for the oids and `c4` from the existential clause `%ec` and the related generation requests. We remark that TODLV does not perform grounding, which is left to DLV-Complex. Type and annotation reconstruction play a central role, since they enforce the correct grounding of polymorphic clauses. For example, the annotations of `is_association` clauses of `c2` are used to instantiate the type-variables `C1, C2` in the `link`-clause.

Finally we launch DLV-Complex with the above program, and we get back a set of stable models that represent the possible snapshots. Such models can be visualized as object diagrams using BOUML or a graphical tool. Here we show a snapshot that is consistent with the given specifications, yet does not fit with our expectations. This would suggest some problem in the modeling phase. Here, customer c_0 is associated with b_0, and r_1, while c_1 is associated with b_1 and r_0. This is surprising since, for instance, r_1 is the surfrecord of c_0, but r_1 and c_0 refer to different bills. This brings about very well the usefulness of lightweight formal methods and model validation in particular. One could do all sort of heavy functional verification via interactive theorem proving only to discover that the initial model was under-specified and required further constraints. Without some clever notion of proof reuse, this would have meant a lot of wasted effort.



We conclude with a final comment concerning the intermediate language DLVExi. It has been introduced to enhance the expressive power of DLV-Complex, in order to define different representations of UML, of which `trel` is an example, minimizing the impact on the XMI2DLVEXI and TODLV components and on the GR language and its semantics. In particular, we have developed a functional representation `tfun` of

the associations, not explained here for lack of space. The `tfun` representation drastically reduces the number of the generated isomorphic snapshots. For example, without generating the attributes, with 1 provider and at most 2 customers we have 100 snapshots with `trel`, while only 6 with `tfun`. The non isomorphic snapshots are 4.

4 Related and Future Work

Animation tools for UML diagrams such as state-chart, activity etc. are a commercial enterprise. Among academics, the USE tool [6] claims to be the only one supporting automatic SG; differently from us, SG requires the user to write Pascal-like procedures in a dedicated language. The issue of isomorphic models does not seem to be addressed and the performances of USE are very sensitive to the *order* of objects and attribute assignments [1]. Other animation and validation tools support different languages. Alloy [7] is based on first-order relational logic. The Alloy Analyzer compiles a formula in the Alloy language into quantifier-free booleans and feed to a SAT solver. According to [1], the Alloy Analyzer is the leading system for generation of instances of invariants, animation of the execution of operations and checking of user-specified properties. However, Alloy is not formally object-oriented, nor does it support UML and OCL.

We have described the design and implementation of MSG, a tool using ASP for MBT in the context of model validation of UML+OCL class diagrams. While the system is not yet ready to be released our preliminary experiments have shown that it compares favourably with the functionalities and the statistics reported in [1] w.r.t. our main "competitor", USE. Our main theoretical contribution has been the introduction of an intermediate language and of a representation of UML class diagrams tailored to the fully automatic generation of non isomorphic snapshots.

Future work include engineering the implementation, but also improve the representation, especially w.r.t. cyclic structures: the functional encoding yields rational terms. Possible approaches are *coinductive* techniques or identifying isomorphic graphs via classes of equivalence w.r.t. oid and link names, using a *nameless* representation. We plan to integrate one of the available compilers for OCL and address validation of pre/post conditions of methods supporting both *forward* and *backward* animation.

References

1. Aydal, E.G., Utting, M., Woodcock, J.: A comparison of state-based modelling tools for model validation. In: Paige, R.F., Meyer, B. (eds.) TOOLS (46). LNBIP, vol. 11, pp. 278–296. Springer, Heidelberg (2008)
2. Calimeri, F., Cozza, S., Ianni, G.: External sources of knowledge and value invention in logic programming. Ann. Math. Artif. Intell. 50(3-4), 333–361 (2007)
3. Calimeri, F., Cozza, S., Ianni, G., Leone, N.: DLV-Complex, `http://www.mat.unical.it/dlv-complex`
4. Dalal, S.R., et al.: Model-based testing in practice. In: ICSE 1999, pp. 285–294 (1999)
5. Fiorentini, C., Momigliano, A., Ornaghi, M.: Towards a type discipline for answer set programming. In: Berardi, S., Damiani, F., de Liguoro, U. (eds.) TYPES 2008 Post-Proceedings. LNCS, vol. 5497, pp. 117–135. Springer, Heidelberg (2009)

6. Gogolla, M., Bohling, J., Richters, M.: Validating UML and OCL models in USE by automatic snapshot generation. Software and System Modeling 4(4), 386–398 (2005)
7. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. MIT Press, Cambridge (2006)
8. Larman, C.: Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development. Prentice Hall, Upper Saddle River (2004)
9. Leone, N., et al.: The DLV system for knowledge representation and reasoning. ACM TOCL 7(3), 499–562 (2006)