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

LPNMR'09 – Potsdam, September 15

Motivations

We apply ASP to model validation in a CASE setting.

The aim is to evaluate the *correctness* of formal specifications (models) with respect to their requirements.

- Model: UML class diagrams with constraints (e.g. in OCL)
  - A diagram represents an abstraction of the problem domain
  - An objects diagram represents a snapshot of the system
  - legal snapshots: snapshots that satisfy the constraints
- In general, model validation can be only *empirical*: it is performed by comparing the formal model with the user's expectations.
- Tools for snapshot generation are crucial for model validation.

Our contribution: the MSG generator

We are developing a snapshot generator for UML models called "MSG" ("Milano Snapshot Generator").

- It employs DLV-Complex as a generator engine (answer sets represent the legal snapshots).

$$\text{DLV-Complex} \quad = \quad \text{DLV} \; + \; \text{external functions, lists, } \ldots$$

- To represent UML class diagrams, we introduce the intermediate language

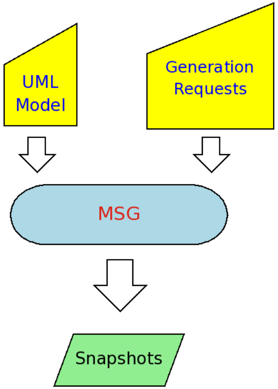
$$\begin{aligned} \text{DLVExi} \quad = \quad \text{DLV-Complex} \quad & + \quad \text{polymorphic types} \\ & + \quad \text{existential clauses} \end{aligned}$$

- Our aim is to minimize the generation of *isomorphic* snapshots

Validation by MSG: a quick overview

Given a UML model M an a set of Generation Requests G, MSG outputs all the legal snapshots of M satisfying G.

- The GR are needed to make the number of snapshots finite.
- The intermediate language DLVExi allows us to *decouple* the representation of UML models from the GR language.

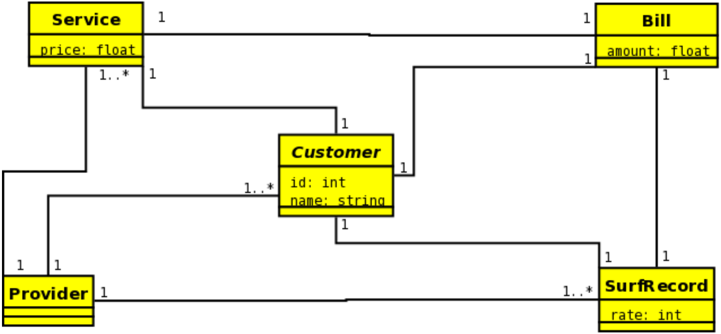


Example: the Internet System Provider

A `Provider` *offers some* `Service`*(s) at a certain price.*
A `Customer` *chooses one of these services and he is charged a* `Bill` *according to his* `SurfRecord` *and the download rate.*

The UML model

Only multiplicity constraints are used



Example: the Internet System Provider

Some Generation Requests

The GR suggest a finite set of possible object identifiers (OID) and a finite set of attribute values, in order to get finitely many models.

Requests on OID

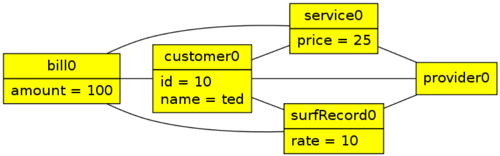
Possible providers:	`provider0`
Possible customers:	`customer1`, `customer2`
Possible bills:	`bill1`, `bill2`
...	

Requests on attribute values

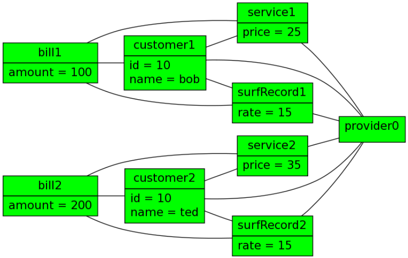
$$\forall c : \texttt{Customer}\ (\, \texttt{c.id} \in \{\, 10, 20 \,\}\,)$$
$$\forall c : \texttt{Customer}\ (\, \texttt{c.name} \in \{\, \texttt{bob}, \texttt{ted} \,\}\,)$$
$$\ldots$$

Example: the Internet System Provider

Some (non-isomorphic) snapshots generated by MSG



Snapshot 1



Snapshot 2

Validation

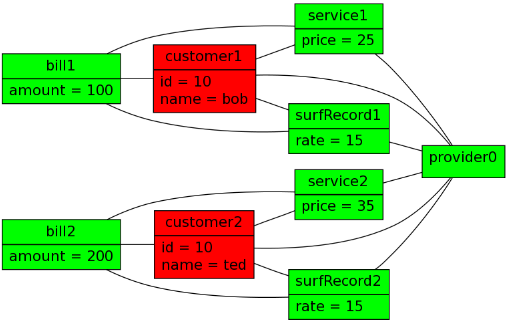
Do the generated snapshots fit with user's expecations?

Yes	$\Longrightarrow$	the specification is OK
No	$\Longrightarrow$	the specification must be revised (add new constraints, ...)

Note: If no snapshots are generated, the UML model is *inconsistent*.

Example: the Internet System Provider

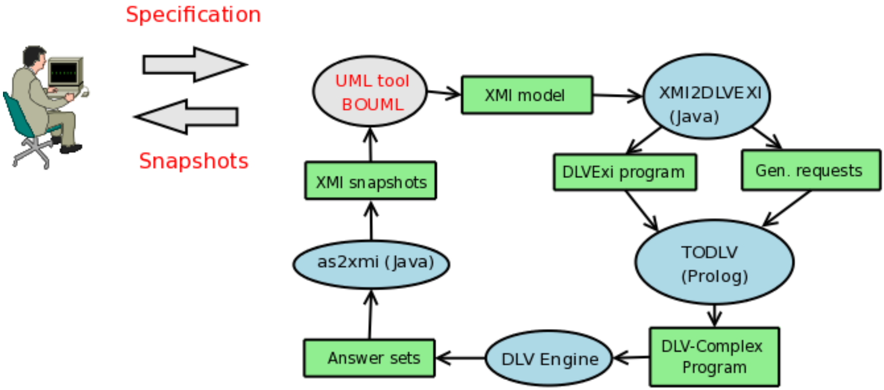
In snapshot 2, the two customers have the same id:



Fix: add to the UML model the constraint

$$\forall c_1, c_2 : \texttt{Customer} \left(c_1.id = c_2.id \implies c_1 = c_2 \right)$$

MSG architecture



MSG architecture

- BOUML
  Used to design UML diagrams with constraints and to generate the corresponding XMI representations.

- XMI2DLVEXI (Java)
  It translates an XMI model M into a DLVExi program E_M, which is a *faithful* representation of M

  > *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.*

- TODLV (Prolog)
  It 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.*

Related work

- USE
  - Snapshot generation requires the user to write Pascal-like procedures in a dedicated language.
  - The issue of isomorphic models does not seem to be addressed
  - The performances are sensitive to the order of objects and attribute assignments
- Alloy
  - It is based on first-order relational logic.
  - A specification is translated into quantifier-free boolean formulla and feed to a SAT solver.
  - Alloy is not formally object-oriented, nor does it support UML and OCL.

Future work

MSG is not yet ready to be released, but preliminary experiments have shown that it compares favourably w.r.t. USE.

Future work

- Engineering the implementation
- Improve the representation, in order to reduce the generation of isomorphic snapshots.
- Validation of pre/post conditions of methods supporting both *forward* and *backward* animation.

A naive encoding of the ISP system in DLVExi

- Type declarations

  The --> symbol introduce polymorphic types by listing the type constructors (also called generators).

```
 %% GENERAL ENCODING
type meta_type(X) --> type.
type obj(C) --> o(int).
type association(C1,C2) --> ass(assoc_name).
type mult -->  m(int,int) ; star(int).

 %% ISP ENCODING
type bill.   type customer.   type provider.   type service.   type surfRecord.
type attributes(C)  -->  rec(float) ; rec(int,string) ; rec(int).
type assoc_name -->  nn.
type attribute_value -->  amount(float) ; id(int) ; name(string) ;
                          price(float) ;   rate(int).
```

On polymotphic types

- Polymorphic types allow us to *decouple* the general representation choices from the signature of the specific UML model
- An UML model M can be represented by a DLVExi theory

$$T_M = R \cup E_M$$

  where R is a general "representation theory" and E_M encodes M in R.
- Since every ground term must have a unique type, we introduces *annotated* functions $f_J(\dots)$ and predicates $p_J(\dots)$.

  In the concrete syntax, the annotations J are enclosed between square brackets.

- We provide a type reconstruction algorithm to find out the annotations. If multiple annotations are possible, the system produces an error message.

A naive encoding: Guess and Test

Live objects and links are *guessed* by the rules g1 and g2, while t1 and t2 *test* the multiplicity constraints.

```
 %% GENERAL ENCODING
pred object(obj(C)).     %% predicate definition
pred is_class(meta_type(C)).
pred link(association(C1,C2), obj(C1),obj(C2)).
pred is_association(association(C1,C2)).
pred mLeft(association(C1,C2),obj(C2),int).
pred mRight(association(C1,C2),obj(C1),int).
pred leftMult(association(C1,C2), mult).
pred rightMult(association(C1,C2), mult).
pred violates(int,mult).

encoding(C1:type, C2:type, C:type, O:obj(C), O1:obj(C1), O2:obj(C2),
   A:association(C1,C2), M:mult, N:int)  isunit
 { %% module definition
   object(O) v neg(object(O)) if is_class(type([C])).  %% g1
   link(A,O1,O2) v neg(link(A,O1,O2)) if     %% g2
         is_association([C1,C2], A) &  object(O1) & object(O2)
   exi([x], att_rec(O,x)) if  object(O).
   false if    %% t1
     leftMult([C1,C2],A,M) & object(O2) & mLeft([C1,C2],A,O2,N) & violates(N,M).
   false if    %% t2
     rightMult([C1,C2],A,M) & object(O1) & mRight([C1,C2],A,O1,N) & violates(N,M).
 }
```

A naive encoding: the ISP system

```
 %% ISP ENCODING

%% classes
is_class(type([bill])) if true.
is_class(type([customer])) if true.
is_class(type([provider])) if true.
...

%% associations
is_association(ass([bill,customer],nn)) if true.
is_association(ass([bill,service],nn)) if true.
is_association(ass([bill,surfRecord],nn)) if true.
...

%% attributes
value([bill], This, amount(F)) if  att_rec(This,rec([bill],F)).
value([customer], This, id(I)) if  att_rec(This,rec([customer],I,S)).
value([customer], This, name(S)) if  att_rec(This,rec([customer],I,S)).
...
%% multiplicities

leftMult(ass([bill,customer],nn),m(1,1)) if true.
rightMult(ass([bill,customer],nn),m(1,1)) if true.
leftMult(ass([customer,provider],nn),star(1)) if true.    %%  1..*
rightMult(ass([customer,provider],nn),m(1,1)) if true.
...
```

Some Generation Requests for the ISP system

- Object identifiers are chosen by means of the oid Prolog predicate.

```
oid(provider,0).  %% at most  provider0
oid(bill,I) :- member(I,[1,2]).    %% at most bill1 and bill2
oid(customer,I) :- member(I,[1,2]).
oid(service,I) :- member(I,[1,2]).
oid(surfRecord,I) :- member(I,[1,2]).
```

- Attributes are settled by the attribute Prolog predicate.

```
attributes(bill, Obj, rec([bill], 100)).
  %% for ever Obj of class bill, 100 is a possible value of the  amount of Obj
attributes(bill, Obj, rec([bill],200)).
attributes(customer, Obj, rec([customer],10,ted)).
attributes(service, Obj, rec([service],25)).
attributes(surfRecord, Obj, rec([surfRecord],10)).
 ...
```

The DLV-Complex translation

```
of(o([C],I),obj(C)) :-   %%  o([C],I) has type obj(C)
  is_type(obj(C)), oid(C,I).

is_type(obj(C)):-
  is_class_type(C).

is_class_type(C):-
  is_class([C],type([C])).

is_class([bill], type([bill])).
is_class([customer], type([customer])).
....
oid(bill, 1).   oid(bill, 2).  oid(customer, 1).  oid(customer, 2).
...
%% GUESS CLAUSES
object([C], O ) v -object([C], O) :-
    is_class([C], type([C])), is_type(C), of(O, obj(C)).

link([C1, C2], A, X3, X4) v -link([C1, C2], A, O1, O2) :-
    is_association([C1, C2], A), object([C1], O1), object([C2], O2).
```

Type and annotation reconstruction play a central role, since they enforce the correct grounding of polymorphic clauses

The translation of existential clauses

An existential formula is replaced with a disjunction over the "witness-choices" settled by the generation requests

Example

The existential clause

```
exi([x], att_rec(Obj,x)) if  object(Obj).
```

for `Obj`= `bill` is translated as

```
att_rec([bill], Obj, rec([bill], 100)) v att_rec([bill], Obj, rec([bill], 200)) :-
    object([bill], Obj).
```

since the choiches of the `rec` values for `bill` are:

```
attributes(bill, Obj, rec([bill], 100)).
attributes(bill, Obj, rec([bill], 200)).
```