Snapshot Generation via Constructive Logic

Mario Ornaghi, Camillo Fiorentini, Alberto Momigliano

Dip. Scienze dell'Informazione
Università degli Studi di Milano (Italy)

MoVeLog'05, Sitges, October 5, 2005

Preview

- We introduce COOML (Constructive OO Modeling Language), an OO modeling&specification language with a constructive semantics that allows us to generate snapshots (system states that satisfy a specification)
- We propose snapshots generation as a way of checking and understanding COOML specifications:
  - Formal verifications are frequently wrong
    - inconsistencies
    - wrong axiomatizations of a problem domain
  - Thus, even in presence of formal verification, testing (of specifications) cannot be eliminated

Overview

- Motivations of COOML
- The COOML specification&modeling language
- A tool for generating snapshots
  - understanding /validating specifications
- Conclusions

The COOML language

Two abstractions:

- data types: data + operations
  - specifying (ADT, programs, ...)
- data models: *meaning of structured data*
  - modeling (the "real world": ER models, OO models, ...)

Modeling languages: based on non-domain-specific data models.

- relational data model in DB (recently, XML data bases)
- UML in OO
- semantics nets in AI
- RDF, DAML+OIL (semantics Web)
- ...

The COOML language

COOML is a modeling&specification language with:

- a constructive layer (modeling), with decoupled
  - pieces of information (data and computing)
  - formulas (meaning and reasoning)
  - decoupling allows us to deal with partial information and multiple meanings and to partially overcome the trade-off expressive power/computability
- a domain logic layer (specification), for expressing properties of the problem domain and reasoning on it

The Problem Domain Specification (PDS) layer

- Open choice of the PDS (syntax and semantic entailment)

$$I \models F$$

  - the interpretation I is an abstraction of the "real word",
  - F is a formal or informal statement, expressing a property of the real world.

- For snapshot generation, we have used the following PDS:
  - Atoms and literals: as usual
  - Iterators: e.g. $sum(X, G(X), T)$, with generator $G(X)$
  - $false \leftarrow L_1 \wedge \ldots \wedge L_n$ ($L_1, \ldots, L_n$ literals)
  - $A \leftarrow L_1 \wedge \ldots \wedge L_n$,
  - consistency if *false* fails
  - Extension to CLP possible

The COOML Modeling layer

- Based on a predicative extension of E^*, a decidable intermediate constructive logic [Miglioli'89].
- The bridge to the PDS layer: atoms of the form
  - true$\{F\}$ (or simply F), F any complex PDS formula.
- The Java-like syntax of COOML specifications:
  - Atoms: true$\{F\}$
  - Properties: $\textsc{and}\{\ldots\}$
    $\textsc{or}\{\ldots\}$
    $\textsc{exi}\{\,Type\ x : \ldots\}$
    $\textsc{for}\{\,Type\ x : G(x) \mid \ldots\}$

    Remark: bounded universal quantification with generator $G(x)$.
  - Class specifications: explained later.

Semantic 1: the information values

We define the $\textsc{it}(S)$ for every specification S.

$$\begin{array}{rcl}
\textsc{it}(A) & = & \{\, true \,\},\ \text{where } A \text{ is an AT} \\
\textsc{it}(\textsc{and}\{P_1\ \ldots\ P_n\}) & = & \{\, (i_1, \ldots, i_n) \mid i_1 \in \textsc{it}(P_1), \ldots, i_n \in \textsc{it}(P_n) \,\} \\
\textsc{it}(\textsc{or}\{P_1\ \ldots\ P_n\}) & = & \{\, (k, i) \mid k \in \{1, \ldots, n\} \text{ and } i \in \textsc{it}(P_k) \,\} \\
\textsc{it}(\textsc{exi}\{\,Type\ x : P\}) & = & \{\, (c, i) \mid c \text{ has type } Type \text{ and } i \in \textsc{it}(P) \,\} \\
\textsc{it}(\textsc{for}\{x : G(x) \mid P\}) & = & \{\, (\,(c_1, i_1), \ldots, (c_m, i_m)\,) \mid i_1 \in \textsc{it}(P), \ldots, i_m \in \textsc{it}(P)\}
\end{array}$$

Semantics 2: the data model of the pieces of information

Let S be a ground specification, let $d \in \mathrm{IT}(S)$ and let I be a classical interpretation. We define

$$I \mid\models d : S$$

$I \mid\models \mathit{true} : A$	IFF	A holds in I, where A is an AT
$I \mid\models (i_1, \dots, i_n) : \mathrm{AND}\{P_1 \ \dots \ P_n\}$	IFF	$I \mid\models i_1 : P_1, \dots, I \mid\models i_n : P_n$
$I \mid\models (k, i) : \mathrm{OR}\{P_1 \ \dots \ P_n\}$	IFF	$I \mid\models i : P_k$
$I \mid\models (c, i) : \mathrm{EXI}\{\mathit{Type}\ x : P(x)\}$	IFF	$I \mid\models i : P(c)$
$I \mid\models L : \mathrm{FOR}\{x : G(x) \mid P(x)\}$	IFF	$L = ((c_1, i_1), \dots, (c_m, i_m))$ and $\mathit{Dom}(G) = \{c_1, \dots, c_m\}$ and $I \mid\models i_1 : P(c_1), \dots, I \mid\models i_m : P(c_m)$

Example

- A piece of information:

  `[[for(linux),[1,true]], [for(snoopy), [2,true]]]:`
  $\mathrm{FOR}\{\mathit{Obj}\ x : x.\mathit{inThisRoom} \mid \mathrm{OR}\{\mathit{person}(x),\ \mathit{dog}(x)\}\}$

  - $\mathit{Dom}(x.\mathit{inThisRoom}) = \{\mathit{linux}, \mathit{snoopy}\}$
  - for $x = \mathit{linux}$ `[1,true]`:$\mathrm{OR}\{\mathit{person}(x),\ \mathit{dog}(x)\}\}$, i.e., $\mathit{person}(\mathit{linux})$
  - similarly, $\mathit{dog}(\mathit{snoopy})$

- Multiple meanings:

  `[[for(linux),[1,true]], [for(snoopy), [2,true]]]:`
  $\mathrm{FOR}\{\mathit{Obj}\ x : x.\mathit{inThisRoom} \mid \mathrm{OR}\{\mathit{person}(x),\ \neg\mathit{person}(x)\}\}$

  We get $\neg\mathit{person}(\mathit{snoopy})$ instead of $\mathit{dog}(\mathit{snoopy})$

COOML class specification

Class $C\{$
  $\mathrm{ENV}\{\mathit{Types}\ \underline{e} : E_C(\mathit{this}, \underline{e})\}$
  $\mathit{PtyName} :\ S_C(\mathit{this}, \underline{e})$
$\}$

- Class axiom
  The spec. $S_C(\mathit{this}, \underline{e})$ describes the structure and the meaning of the information values stored by the instances of C:

  ClassAx(C) : $\mathrm{FOR}\{\mathit{Types}\ \underline{e} : \mathit{this}.C(\underline{e}) \mid S_C(\mathit{this}, \underline{e})\}$

- Environment constraint
  $S_C(\mathit{this}, \underline{e})$ may depend on *environment variables* $\underline{e}$, and the problem formula $E_C(\mathit{this}, \underline{e})$ links *this* to its environment $\underline{e}$.

  EnvConstr(C) : $\forall(\mathit{this}.C(\underline{e}) \rightarrow E_C(\mathit{this}, \underline{e}))$

Populations

- A *population* $\mathcal{P}_C$ of a class C is an information value

  $$(\dots((o_j, \underline{e}_j), d_j)\dots) : \mathrm{FOR}\{\mathit{Types}\ \underline{e} : \mathit{this}.C(\underline{e}) \mid S_C(\mathit{this}, \underline{e})\}$$

- The populations $\mathcal{P}$ of a system S are obtained by the populations of the classes of S.
- Let I be an interpretation. $I \mid\models \mathcal{P} : S$ (I is a model of $\mathcal{P} : S$) iff, for every class C of S:
  (i) Every constraint axiom of C holds in I.
  (ii) $I \mid\models \mathcal{P}_C$: **ClassAx**(C).
- $\mathcal{P} : S$ represents system snapshot. A system S is consistent if there exists a population for S with at least one model.

The Main theorem

Snapshot generation is based on the following theorem:

Theorem

Let S be a COOML specification and I be a reachable interpretation of the problem domain.
Then:

(i). $I \models S$ *iff there is a population* $\mathcal{P}$ *for* S *such that* $I \mid\models \mathcal{P} : S$;

(ii). *for every* $\mathcal{P}$ *for* S, *we can extract a set* $\mathcal{IC}(\mathcal{P}, S)$ (*information content*) *of ground atoms such that*

$$I \mid\models \mathcal{P} : S \quad \textit{iff} \quad I \models \mathcal{IC}(\mathcal{P}, S)$$

An example: the CashRegister system

Class CashRegister{
CashRegisterPty: $\textsc{or}\{$ $\textsc{exi}\{Receipt\ r : r.Receipt(this)\}$
$empty(this)\}$
}

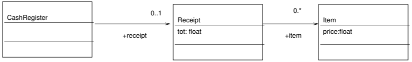
Class Receipt{
$\textsc{env}\ \{CashRegister\ c : true\}$
ReceiptPty: $\textsc{and}\{$ $\textsc{for}\{\ Item\ i : i.Item(this) \mid i.inCatalog()\}$
$\textsc{exi}\{\text{float}\ tot : this.grandtot(tot)\}$
}

Class Item{
$\textsc{env}\ \{Receipt\ r : true\}$
ItemPty: $\textsc{exi}\{\text{float}\ p : this.price(p)\}$
}

Problem domain specification

$$false \leftarrow empty(C) \wedge R.Receipt(C)$$
$$false \leftarrow grandtot(R,T) \wedge sum(P, itemPrice(R,P), T_1) \wedge \neg(T = T_1)$$
$$itemPrice(R,P) \leftarrow I.Item(R) \wedge price(I,P)$$

UML class diagram



Class CashRegister{
CashRegisterPty: $\textsc{or}\{$ $\textsc{exi}\{Receipt\ r : r.Receipt(this)\}$
$empty(this)\}$
}

Class Receipt{
$\textsc{env}\ \{CashRegister\ c : true\}$
ReceiptPty: $\textsc{and}\{$ $\textsc{for}\{\ Item\ i : i.Item(this) \mid i.inCatalog()\}$
$\textsc{exi}\{\text{float}\ tot : this.grandtot(tot)\}$
}

Class Item{
$\textsc{env}\ \{Receipt\ r : true\}$
ItemPty: $\textsc{exi}\{\text{float}\ p : this.price(p)\}$
}

Class Receipt{
$\textsc{env}\ \{CashRegister\ c : true\}$
ReceiptPty: $\textsc{and}\{$ $\textsc{for}\{\ Item\ i : i.Item(this) \mid i.inCatalog()\}$
$\textsc{exi}\{\text{float}\ tot : this.grandtot(tot)\}$
}

$I \mid\models$ `[[for(it1) true], [for(it2), true], [for(it3), true]]` :
$\textsc{for}\{\ Item\ i : i.Item(r) \mid i.inCatalog()\}$

IFF

in the interpretation I the items of the receipt `r` are `it1`, `it2` and `it3` and all the items are in the catalog.

$I \mid\models$ `[exi(1000), true]` :
$\textsc{exi}\{\text{float}\ tot : r.grandtot(tot)\}$

IFF

in the interpretation I the grandtotal of the receipt `r` is 1000.

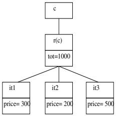
Populations as snapshots

A snapshot for the Cash Register system, in our Prolog notation, is

```
[  [exi(300), true]:class(item, it1, [r(c)]),
   [exi(200), true]:class(item, it2, [r(c)]),
   [exi(500), true]:class(item, it3, [r(c)]),
   [[[for(it1), true], [for(it2), true], [for(it3), true]],
    [exi(1000),true]] : class(receipt, r(c), [c]),
   [1, [exi(r(c)), true]] : class(cashregister, c, [])
]
```

with information content $\mathcal{IC}$:

```
price(300, it1),  price(200, it2),   price(500, it3),
grandtot(r(c), 1000),
inCatalog(it3), inCatalog(it2), inCatalog(it1), receipt(r(c), c)
closed(item, [r(c)])
   % it.Item(r(c)) holds  IFF it=it1 OR  it=it2 OR  it=it3
...
```



The generation algorithm and its use

- We introduce the generarion algorithm (implemented in Prolog)
- We discuss by an example the testing a model for consistency and against the informal undertanding of the problem domain
- Testing of method specifications is in progress

Generating populations

COOML classes are represented by the following Prolog predicates:

- `pty(class(C,I,E),Pty)`
  representing the class axioms
- `class(C,I,E) >> [EnvConstr1, ..., EnvConstrn]`
  representing the environment constraints
- `isAtom(...)`
  representing the COOML atoms used in the classes

The user *chooses* the possible object identifiers to be used in the generation, by defining the predicate

`id(Class, Id, Env)`

Moreover, the user can update the `isAtom` predicates to fix the possible values of attributes.

The algorithm

- Initialization
  - Store the user's choices.
  - *ToDo* $::= \{id_1.C_1(\underline{e}_1), \ldots, id_n.C_n(\underline{e}_n)\}$ (obj. *to be generated*)
  - *Pop* $::= \emptyset$ (generated *population*)
  - *Constr* $::= \emptyset$ (*constraints* about generated objects).
- Generation step

  For each $id.C(\underline{e}) \in ToDo$:
  - Find an information value for **ClassAx**(C).
  - Update *ToDo*, *Pop*, *Constr*, according to the environment constraints.
  - Check Problem Domain constraints.

  If *ToDo* $\neq \emptyset$, do a new generation step.

At the end of the computation, *Pop* is a consistent completely generated population.

An example

Initialization

```
%%%  OBJECT IDENTIFIERS

id(cashregister,c,[]). % a cashregister c

id(receipt,r(C),[C]):- id(cashregister,C,_).
%a cashregister C has at most one receipt r(C)

id(item,it1,[R]) :- id(receipt,R,_).
id(item,it2,[R]) :- id(receipt,R,_).
id(item,it3,[R]) :- id(receipt,R,_).

%%% CHOOSE ATTRIBUTES

isAtom(grandtot(_,1000)).
isAtom(price(500,_)).
isAtom(price(200,_)).
isAtom(price(300,_)).
```

Problem domain constraints

The problem domain constraints for the cashregister system are represented using the predicate `holds(Atom,State)` (`State` is the generation state).

- **Local error**
  Can be checked at *any* moment of a generation step.

$$false \leftarrow empty(C) \wedge R.Receipt(C)$$

In the Prolog program:

```
localErr(empty(C),State) :-
        holds(receipt(_,_,[C]), State).

localErr(receipt(_,C),State) :-
       holds(empty(C),State).
```

- **Global error**
  Can be checked *only* at the end of a generation step

$$false \leftarrow grandtot(R,T) \wedge sum(P.itemPrice(R,P),T_1) \wedge \neg(T = T_1)$$

$$itemPrice(R,P) \leftarrow I.Item(R) \wedge price(I,P)$$

In the Prolog program:

```
globalErr(State) :-
       holds(grandtot(R,T),State),
       holds(closed(item,[R]),State),
       sum(P,price(P,R,State),T1),
       not(T=T1).
```

First generation step

Class CashRegister{
CashRegisterPty: OR{ EXI{*Receipt r* : *r.Receipt*(*this*)}
empty(*this*)}
}

We do a generation step of an object c of class `cashregister`

```
?- build(1, [class(cashregister,c,[])], [Pop, ToDo, Constr]).
```

Two solutions:

```
Pop =  [[1, [exi(r(c)), true]] : class(cashregister, c, [])]

ToDo = [class(receipt, r(c), [c])]

Constr = []
------------------------------

Pop = [[2, true]:class(cashregister, c, [])]
ToDo = []
Constr = [empty(c)]
```

Second generation step

Class Receipt{
ENV {*CashRegister c* : *true*}
ReceiptPty: AND{ FOR{ *Item i* : *i.Item*(*this*) | *i.inCatalog*()}
EXI{float *tot* : *this.grandtot*(*tot*)}
}

```
Pop = [[[[for(it1), true]], [exi(1000), true]] : class(receipt, r(c), [c]),
       [1, [exi(r(c)), true]] : class(cashregister, c, [])]

ToDo = [closed(item, [r(c)]), class(item, it1, [r(c)])]
       % closed(item, [r(c)] : no items can be generated in next steps

Constr = [grandtot(r(c), 1000), inCatalog(it1), receipt(r(c), c)]
------------------------------------------------------------------------------

Pop =[[[[for(it1), true], [for(it3), true]],
       [exi(1000), true]] : class(receipt,r(c), [c]),
      [1, [exi(r(c)), true]] : class(cashregister, c, []) ]

ToDo = [closed(item, [r(c)]), class(item, it3, [r(c)]), class(item, it1, [r(c)])]

Constr = [grandtot(r(c), 1000), inCatalog(it3), inCatalog(it1), receipt(r(c), c)]
------------------------------------------------------------------------------

Pop = [[[[for(it1), true], [for(it2), true], [for(it3), true]],
        [exi(1000),true]] : class(receipt, r(c), [c]),
       [1, [exi(r(c)), true]] : class(cashregister, c, []) ]

ToDo = [closed(item, [r(c)]), class(item, it3, [r(c)]), class(item, it2, [r(c)]),
        class(item, it1, [r(c)])]

Constr = [grandtot(r(c), 1000), inCatalog(it3), inCatalog(it2), inCatalog(it1), receipt(r(c), c)]
......
```

Third generation step

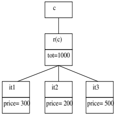
Class Item{
ENV {*Receipt r* : *true*}
ItemPty: EXI{float *p* : *this.price*(*p*)}
}

```
Pop =
[[exi(300), true]:class(item, it1, [r(c)]),
 [exi(200), true]:class(item, it2, [r(c)]),
 [exi(500), true]:class(item, it3, [r(c)]),
 [[[for(it1), true], [for(it2), true], [for(it3), true]], [exi(1000),true]]
   : class(receipt, r(c), [c]),
  [1, [exi(r(c)), true]] : class(cashregister, c, [])]

ToDo = [closed(item, [r(c)])] % nothing to do

Constr = [price(300, it1), price(200, it2), price(500, it3),
grandtot(r(c), 1000), inCatalog(it3), inCatalog(it2), inCatalog(it1), receipt(r(c), c)]

....
```



Testing the specifications: two goals

- Ensure the consistency of a specification: it suffices that at least one snapshot is generated.
  This is done also by UML-OCL snapshot generation tools, e.g. USE
- Check a formal model&specification with respect to the (informal) problem domain.
  In this case it is useful to generate many snapshots. The idea is to fix the generation data (possible object identifiers and values of the attributes) and generate all the corresponding snapshots.

In our cashregister example, if we generate all the snapshots with one cashregister, all (and only) the expected snapshots are generated.

But with two casregisters c1 and c2 . . .

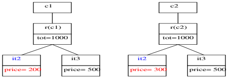
```
build(3,[class(cashregister,c1,[]),class(cashregister,c2,[])],...)
```

Population

```
[[exi(200), true]:class(item, it2, [r(c1)]),    ---->  price of it2 in r(c1)
 [exi(500), true]:class(item, it3, [r(c1)]),
 [exi(300), true]:class(item, it2, [r(c2)]),    ---->  price of it2 in r(c2)
 [exi(500), true]:class(item, it3, [r(c2)]),
  [[[for(it2), true], [for(it3), true]], [exi(1000), true]]:class(receipt, r(c1),[c1]),
 [[[for(it2), true], [for(it3), true]], [exi(1000), true]]:class(receipt, r(c2),[c2]),
  [1, [exi(r(c2)), true]]:class(cashregister, c2, []),
  [1, [exi(r(c1)), true]]:class(cashregister, c1, [])]
```

Constraints

```
price(300, it2), price(200, it2), price(500, it3),
grandtot(r(c1), 1000), grandtot(r(c2), 1000), ........
```



The total is wrong!

Correcting the specifications

- The problem:
  The item `it2` has two differen prices.

  $\Longrightarrow$ Change the specifications!

- Solution 1
  Add the unicity constraint for the prices

$$false \leftarrow empty(C) \wedge R.Receipt(C) \quad \texttt{\%local}$$
$$false \leftarrow grandtot(R, T) \wedge sum(P.itemPrice(R, P), T_1) \wedge \neg(T = T_1) \quad \texttt{\%global}$$
$$false \leftarrow price(I, P_1) \wedge price(I, P_2) \wedge \neg(P_1 = P_2) \quad \texttt{\%local}$$

- Solution 2
  Change the price definition

$$price(Item, P)$$

  into

$$price(Item, Receipt, P)$$

Conclusions

- COOML: work in progress
  Tools: java implementation, snapshot generation, proofs as programs support, ...
- We have presented snapshot generation for specification understanding and validation.
  - Related work: snapshot generation for UML-OCL specifications, in particular USE [Gogolla et al., 2003].
- So far we have developed a prototype in Prolog and we have tested it with various specifications, e.g.:
  - The 8-queen problem
  - Circular lists
  - ...

  We have devised specification mistakes and suggestions to correct them.
- We are developing the testing of method specifications by pre and post conditions (requiring the generation of pairs of snapshots).