Snapshot Generation in a Constructive Object-Oriented Modeling Language

Mauro Ferrari[1], Camillo Fiorentini[2], Alberto Momigliano[2], and Mario Ornaghi[2]

[1] Dipartimento di Informatica e Comunicazione, Università degli Studi dell'Insubria, Italy
mauro.ferrari@uninsubria.it
[2] Dipartimento di Scienze dell'Informazione, Università degli Studi di Milano, Italy
{fiorenti,momiglia,ornaghi}@dsi.unimi.it

Abstract. CooML is an object-oriented modeling language where specifications are theories in a constructive logic designed to handle incomplete information. In this logic we view snapshots as a formal counterpart of object populations, which are associated with specifications via the constructive interpretation of logical connectives. In this paper, we introduce the "snapshot semantics" of CooML and we describe a snapshot generation (SG) algorithm, which can be applied to validate specifications in the spirit of OCL-like constraints over UML models. Differently from the latter and from the standard BHK semantics, the logic allows us to exploit a notion of partial validation that is appropriate to encodings characterised by incomplete information. SG is akin to model generation in answer set programming. We show that the algorithm is sound and complete so that its successful termination implies consistency of the system.

1 Introduction

We are developing the constructive object-oriented modeling language CooML [19] (`http://cooml.dsi.unimi.it`), a specification language for OO systems. Similarly to UML/OCL [23], CooML provides a framework for the design of system specifications in the early stages of the development process. The language allows the user to distinguish between internally defined elements and the problem domain, which may involve loosely or incompletely defined components. This encourages the selection of the appropriate level of abstraction w.r.t. specifications.

CooML follows the spirit of lightweight formal methods [10]: it does not focus on full formalization, nor on whole system correctness, but emphasizes *partiality* in analysis and specification. In particular, in the context of OO modeling, both the validation of a specification and the check of its consistency can be achieved via the notion of *snapshot*, i.e. a population of objects in a given system state that satisfies the specification. Previous work has used snapshots for validation of UML/OCL models [8] and specifications in JML [4].

The novelty of CooML's approach resides in its semantics, which is related to the constructive explanation of logical connectives (a.k.a. the BHK interpretation [22]). Specifically, the truth of a CooML proposition in a given interpretation is explained by a mathematical object that we call an *information term*. For the time being, the latter can be visualized as a sort of *proof term* inhabiting a type/formula. The underlying logic is

King, A. (Ed.): LOPSTR 2007, LNCS 4915, pp. 169–184, 2008.

characterized by how classical and constructive information co-exists, the main "entry" point being the different way in which an *atomic* formula A is given evidence (for more details we refer the kind reader to the original formulation of the logic in [15]). If we call the pair $I : P$ a *piece of information*, where P is a formula and I is its information term, then $I : P$ may be *true* or *false* in a classical interpretation w, called a *world*. Thus, we have a notion of a *model* of a piece of information based on classical logic. In particular, we use $\mathtt{T}\{F\}$ to indicate the truth of F; in fact, $\mathtt{T}$ does not contain evidence for F, but it yields a trivial piece of information true in all the models of F. This introduces a novel and flexible way to handle *incomplete* information, a notorious difficulty in information systems such as relational databases.

Crucially, the constructive side of the logic allows the *identification of snapshots with information terms*, thus providing a formal counterpart to the intuitive notion of object populations. We argue that CooML's proof-theoretic snapshot generation may be advantageous in comparison to a model-theoretic one, especially in cases where not all the information required to define a model is even present. The possibility of treating information in this less committed way means that we can select only the relevant information; this may have a cascade of benefits in terms of conciseness of the representation.

The contributions of this paper are twofold. First, we extend the semantics developed in purely logical terms in [15] to object oriented modeling languages. We regard an OO system specification as a CooML theory T, the system snapshots as the pieces of information $I : T$, and the related information content as a suitable set of formulae. We show that the latter can be seen as the *minimum* information needed to give evidence to snapshots and we relate that to snapshot consistency. Secondly, we describe (and implement) a snapshot generation algorithm (SGA), taking as inputs: (i) a CooML theory T, axiomatizing a set of classes in a problem domain PD; (ii) the user's generation requirements $\mathscr{G}$– they serve an analogous purpose of domain predicates in the grounding phase of ASP's [17]. As snapshots should be consistent with respect to PD and $\mathscr{G}$, we prove that consistency checking is sound and that snapshot generation is complete, i.e. if a consistent snapshot satisfying the generation requirements exists, it will be generated. This is loosely connected to adequacy results in the theory of CLP's [7].

2 CooML Specifications

In this section we informally present the language via an example (adapted from [3]), while we defer the formal exposition to Section 2.1. The problem domain concerns a small coach company. Each coach has a specified number of seats and can be used for regular or private trips. In a regular trip, each passenger has its own ticket and seat number. In a private trip, the whole coach is rented and there may be a guide. The corresponding CooML specification is contained in the package `coachCompany` (Fig. 1). To explain our example we need to introduce CooML types system. We distinguish among *data types* (in our example, `Integer` and `Boolean`), *PD types* (`Person`), and *object types* (`Coach`, `Trip`, `Passenger`). They all inherit from the top type `Value` the identity relation and the string representation. Data types are "statically" defined, i.e., their values do not depend on the current state. CooML assumes the

```
package coachCompany;
pds{type Person;
  Integer numberOfSeats(Coach c) = (* the number of seats of c *);
  Boolean guides(Person p, Trip t) = (* p guides trip t *);
  Boolean nobooking(Passenger p, Trip t) = (* p has no booking in t *);
  Boolean vacant(Integer s, Coach c, Trip t) =
                                  (* s is a vacant seat on c in t *);
  Boolean booked(Passenger p, Integer s, Coach c, Trip t) =
                                  (* p has booked seat s on c in  t *);
  <constr name=bookingConstraints  language=prolog>
    false :- vacant(S,C,T), booked(_P,S,C,T).
    false :- booked(P1,S,C,T), booked(P2,S,C,T), not(P1==P2).
    false :- nobooking(P,T), booked(P,_Seat,_Coach,T).
  </constr>
  }
class Coach{
 coachPty: and{
     seats: exi{Integer seatsNr; seatsNr = numberOfSeats(this)}
     trips: for{Trip trip; trip is Trip(this) --> true} }
 Integer getSeats(){  return seats.seatNr  }
          }
class Trip{ env(Coach coach)
 TripPty: case{private: case{T{exi{Person p; guides(p,this)}}
                            T{not exi{Person p; guides(p,this)}}}
               regular: for{Integer seat; (seat in 1..coach.getSeats()) -->
                  case{vacant: vacant(seat,coach,this)
                       booked: exi{Passenger p; T{and{p is Passenger(this)
                                                      booked(p,seat,coach,this)}}
      }}}}}
class Passenger{ env(Trip trip)
 PsngrPty: case{c1: nobooking(this,trip)
                c2: exi{Integer seat, Coach coach;
                        T{and{trip is Trip(coach)
                              booked(this,seat,coach,trip)}}
      }}}
```

Fig. 1. The `coachCompany` package

existence of an implementation that evaluates ground terms to values. A PD type extends `Value` with a set of problem domain functions.

Nothing is assumed about PD types; they may be characterized by a set of formal or loose properties that we call *PD constraints*, introduced by the tag `<constr>`.

The special subtype `Obj` of `Value` introduces object identities. Objects are created by CooML classes, which are structured in a single inheritance hierarchy rooted in `Obj`. The definition of a class C may depend on some *environment* parameters; namely `C(`$\underline{e}$`)` is a class with environment parameters $\underline{e}$. If $\underline{\mathbf{e}}$ is a ground instance of the environment parameters $\underline{e}$, then `C(`$\underline{\mathbf{e}}$`)` can be used to create new objects. We write "**o** `is C(`$\underline{\mathbf{e}}$`)`" to indicate that **o** has been created by `C(`$\underline{\mathbf{e}}$`)`, while "**o** `instanceof C(`$\underline{\mathbf{e}}$`)`" means that **o** has environment $\underline{\mathbf{e}}$ and has been created by a subclass `C'` of `C`. We call those *class predicates*.

In a package: (i) data types are assumed to be externally implemented; (ii) PD types are defined in the pds (problem domain specification) section; (iii) classes are introduced by suitable class declarations.

pds declaration and world states. The pds section specifies our general knowledge of the problem domain. It introduces PD types, functions and predicates using data and class types. In our example we introduce the PD type Person and functions numberOfSeats, guides, ... The informal descriptions (*...*) use terms of the global signature provided by the analysis phase [11]. A <constr> declaration introduces a set of PD constraints representing general problem domain properties that are not interpreted by CooML, but possibly by some external tool. In the example, PD constraints are expressed in Prolog assisting the SG algorithm in filtering out undesired snapshots. The class predicate "o is C($\underline{e}$)" is represented by the Prolog predicate isOf(o, C, [$\underline{e}$]), while "o istanceOf C($\underline{e}$)" is translated into instanceOf(o, C, [$\underline{e}$]). The first constraint says that a coach seat cannot be vacant and booked at the same time, the second one excludes overbooking (a seat can be booked by at most one person), while the third says that the predicate nobooking(P,T) holds if person P has not booked a seat on the coach associated with trip T. In this paper, we assume that the signature Σ_T of a CooML theory T (including PD types, data types and classes) is first order and that we can represent the possible states of the "real world" by *reachable* Σ_T-interpretations, dubbed *world states*. Reachability means that each element of the interpretation domains is represented by some ground terms, in our case CooML values. In a world state, PD symbols are interpreted over the external world, data types are interpreted according to their implementation, and class predicates represent the current system objects. For instance the class predicates

```
mini is Coach(),  t1 is Trip(mini),  t2 is Trip(mini),  t3 is Trip(mini),
john is Passenger(t1)
```

represent a small company with a single mini-bus mini, three trips t1,t2,t3 operated by mini and, so far, only one passenger john associated with trip t1.

class declarations and properties. A class declaration introduces the name C of the class, its (possible) environment parameters $\underline{e}$, its property $Pty_C(\texttt{this}, \underline{e})$, and its methods [1]. An object **o** created by C($\underline{\mathbf{e}}$) stores a *piece of information* structured according to $Pty_C(\mathbf{o}, \underline{\mathbf{e}})$, and uses the methods implemented by C($\underline{\mathbf{e}}$).

For class properties, CooML uses a prefix syntax, where formulas may be labeled. Labels are used to refer to subformulae. For example, the label seats is used in the getSeats method to refer to seatsNr. A *class property* P is an atomic formula over Σ_T, or (recursively) a formula of the form and$\{P_1 \ \ldots \ P_n\}$, case$\{P_1 \ \ldots \ P_n\}$, exi$\{\underline{\tau}\ \underline{x}; P\}$, for$\{\underline{\tau}\ \underline{x}; G \rightarrow P\}$ and T$\{P^{ext}\}$, where P^{ext} is a property that may also use negation not and implication imp. We stress that not and imp cannot be used outside T.

In CooML's semantics, a property P defines a set of possible pieces of information of the form $I : P$, where I is an *information term*, that is a structure justifying the truth of P. Each piece of information $I : P$ has an *information content*, i.e. a set of simple properties intuitively representing the minimum amount of information needed to justify P according to I. A *simple property* is either an atom or of the form T$\{P^{ext}\}$. A simple property S represents a basic information unit, i.e., it has a unique information term tt

[1] We use the self-reference this as in Java.

where `tt` is a constant. This means that the only information we have is the *truth* of S, and that the associated information *content* is simply the set $\{S\}$. Exemplifying,

```
tt : t1 is Trip(mini)
```

has information content {`t1 is Trip(mini)`} and means that the trip `t1` is assigned to the coach `mini` in the current world state.

The operator `T` may enclose a complex property P and indicates that we are interested only in its truth. Let us consider

```
tt: T{exi{Person p; guides(p,t2)}}    tt: T{not exi{Person p; guides(p,t3)}}
```

The first piece of information says that `t2` is a guided trip without indicating who the guide is; the second one says that `t3` has no guide.

By default [2] the truth of a simple property S in a world state `w` (denoted `w` $\models S$) is defined as in classical logic, by ignoring `T` (i.e., `w` $\models$ `T{`P`}` iff `w` $\models P$) and interpreting `case` as $\vee$, `and` as $\wedge$, `not` as $\neg$, `imp` as $\rightarrow$, `exi` as $\exists$ and `for{`$\underline{\tau}\ \underline{x}$`;`$G(\underline{x})\rightarrow P(\underline{x})$`}` as $\forall \underline{x}(G(\underline{x}) \rightarrow P(\underline{x}))$.

In contrast, non-simple properties are interpreted constructively, by means of information terms. A piece of information $I : P$ may have one of the following forms:

Existential. `(`$\mathbf{x}$`,`I`):exi{`$\tau\ x$`;`$\ P(x)$`}`, where τ is the type of the existential variable x. The term $\mathbf{x}$ is a *witness* for x and the information content is the one of $I : P(\mathbf{x})$. For example,

```
(4,tt) : exi{Integer seatNr; seatNr = numberOfSeats(mini)}
```

has witness `4` and information content {`4 = numberOfSeats(mini)`}, signifying that our mini-bus has 4 passenger seats. Note that, differently from the case of simple properties, we know the value of x that makes $P(x)$ true.

Universal. `((`$\mathbf{x}_1, I_1$`),...,(`$\mathbf{x}_n, I_n$`)):for{`$\tau\ x;\ G(x) \rightarrow P(x)$`}`, where $G(x)$ is an *x-generator*, i.e. a formula true for finitely many x [3]. The information content is the union of those of $I_1 : P(\mathbf{x}_1)$, ..., $I_n : P(\mathbf{x}_n)$ and of the *domain property* `dom(`$x;\ G(x);$`[`$\mathbf{x}_1,\ldots,\mathbf{x}_n$`])`, a special simple property interpreted as $\forall x(G(x) \leftrightarrow member(x, [\mathbf{x}_1, \ldots, \mathbf{x}_n]))$. For example, the information content of

```
((t1,tt),(t2,tt),(t3,tt)) : for{Trip trip; trip is Trip(mini) → true}
```

is {`dom(trip; trip is Trip(mini); [t1,t2,t3])`}, showing that the domain of the `trip`-generator "`trip is Trip(mini)`" is {`t1,t2,t3`}. Since the atomic formula `true` corresponds to no information, it can be ignored.

Conjunctive. $(I_1, \ldots, I_n)$ `: and{`$P_1 \ldots P_n$`}`. The information content is the union of those of $I_j : P_j$, for all $j \in 1..n$. For instance, a piece of information for the class property `coachPty(mini)` and the related information content IC_1 is

```
   ((4,tt), ((t1,tt), (t2,tt), (t3,tt))) : and{seats(mini) trips(mini)}
IC1 = {4 = numberOfSeats(mini), dom(trip; trip is Trip(mini); [t1,t2,t3])}
```

[2] But one can change this, although we do not discuss it for lack of space.

[3] We omit here the precise syntax of generators.

Disjunctive. $\texttt{(k,}I_k\texttt{):case}\{P_1 \ldots P_n\}$. The selector $k \in 1..n$ points to the true subformula P_k and the information content is $I_k : P_k$'s. For example, if the object `john` with class predicate `john is Passenger(t1)` contains the information term `(1,tt)`, then

```
(1,tt) : case{c1:nobooking(john,t1) c2: ...}
```

selects the first sub-property of `PsngrPty`, with information content {`nobooking (john,t1)`}, i.e. `john` has no booking in trip `t1` in the current state.

The information content of classes. Let $\texttt{C(}\underline{e}\texttt{)}$ be a class with property $Pty_C(\texttt{this},\underline{e})$. We associate with C the *class axiom*

$$\texttt{clAx(C): for}\{\texttt{Obj this, }\underline{\tau}\ \underline{e}\texttt{; this is C(}\underline{e}\texttt{)} \rightarrow Pty_C(\texttt{this},\underline{e})\}$$

The corresponding pieces of information and information content are those for universal properties. The piece of information for class `Coach` and its information content IC_2 is:

$$\texttt{((mini,CoachInfo)) : for}\{\texttt{Obj this; this is Coach()} \rightarrow \texttt{coachPty(this)}\}$$

$$IC_2 = \{\texttt{dom(this; this is Coach();[mini]), 4 = numberOfSeats(mini),}$$
$$\texttt{dom(trip; trip is Trip(mini); [t1,t2,t3])}\}$$

where `CoachInfo:coachPty(mini)` is defined as in the conjunctive case.

System snapshots and their information content. Let P be a package introducing a set of constraints $\mathscr{T}$ and the CooML classes $C_1, \ldots, C_n$. We associate with P a CooML theory $T_P = \langle \texttt{thAx}, \mathscr{T} \rangle$, where $\texttt{thAx} = \texttt{and}\{\texttt{clAx}(C_1) \cdots \texttt{clAx}(C_n)\}$.

A piece of information I : `thAx` represents the information content of the whole system. We call it a *system snapshot*, to emphasise that the system may evolve through a sequence $I_0 : \texttt{thAx}, \ldots, I_n : \texttt{thAx}, \ldots$. A snapshot for our `coachCompany` system is of the form:

$$\texttt{(}I_1\texttt{,}I_2\texttt{,}I_3\texttt{) : and}\{\texttt{clAx(Coach) clAx(Passenger) clAx(Trip)}\}$$

and possible information terms I_1, I_2, I_3 are

```
I1 = ((mini,CoachInfo)),  I2 = (([john,t1],(1,tt)),([ted,t2],(1,tt)))
I3 = (([t1,mini],(2,((1,tt), (2,(john,tt)),(3,(1,tt)),(4,(1,tt))))),
      ([t2,mini],(1,(1,tt))),
      ([t3,mini],(1,(2,tt))))
```

where `[...]` denote tuples. A relevant part of the information content for `coachCompany` is given in Fig. 2.

The above information content could be seen as an "incompletely specified" model of the `coachCompany` theory, where `numberOfSeats`, `nobooking`, `vacant`, `booked` and class predicates are completely specified, while for `guides` we have only some partial knowledge, expressed by the `T`-properties, and moreover nothing is said about `Person`. The relationship with classical models can be better explained by comparing the constructive and classical reading of CooML properties.

```
dom(o; o is Coach(); [mini]),  dom(o; o is Trip(mini); [t1,t2,t3]),
dom([o,t]; o is Passenger(t); [[john,t1],[ted,t2]]),
dom([o,c]; o is Trip(c); [[t1,mini],[t2,mini],[t3,mini]]),
4=numberOfSeats(mini),  nobooking(john,t1),  vacant(1,mini,t1),
booked(john,2,mini,t1), vacant(3,mini,t1),  vacant(4,mini,t1),
T{exi{Person p; guides(p,t2)}},  T{not exi{Person p; guides(p,t3)}}
```

Fig. 2. Part of the information content of `coachCompany`

Let $T = \langle \texttt{thAx}, \mathscr{T} \rangle$ be a CooML theory. We can switch to the classical interpretation of `thAx` simply by using the T operator, i.e. by considering the simple property $\texttt{T}\{\texttt{thAx}\}$. One can prove that $\texttt{T}\{\texttt{thAx}\}$ has a reachable model if and only if so does $\textsc{ic}(I : \texttt{thAx})$, for at least one piece of information $I : \texttt{thAx}$. Furthermore, one can prove that $\textsc{ic}(I : \texttt{thAx})$ is the minimum set of simple formulas that justifies I as an explanation of `thAx`.

In this context we are mainly interested in the notion of consistency with respect to the PD constraints, assuming that the latter can be interpreted as first order sentences. In our example, we interpret a program clause $H :- B_1, \ldots, B_n$ as the universal closure of $B_1 \wedge \ldots \wedge B_n \rightarrow H$, as usual. A system snapshot $I : \texttt{thAx}$ for a theory $T = \langle \texttt{thAx}, \mathscr{T} \rangle$ is *consistent* if its information content $\textsc{ic}(I : \texttt{thAx})$ is true in a reachable classical model of $\mathscr{T}$; T is consistent if there is a consistent snapshot for it. For example, the above snapshot (I_1, I_2, I_3) is consistent with respect to the first and second constraint of the `pds` section, but not with the third, since both `nobooking(john,t1)` and `booked(john,2,mini,t1)` belong to the information content of `coachCompany` (Fig. 2).

2.1 Formal Definitions

Let $T = \langle \texttt{thAx}, \mathscr{T} \rangle$ be a CooML theory and Σ_T the associated first order signature. The set of *information terms* for a property P, $\textsc{it}(P)$, is inductively defined as follows, where $\underline{\mathbf{x}}$ stands for values of $\underline{x}$ of the appropriate type:

$$
\begin{array}{lcl}
\textsc{it}(P) & = & \{\, \texttt{tt} \,\}, \text{ if } P \text{ is simple} \\
\textsc{it}(\texttt{and}\{P_1 \cdots P_n\}) & = & \{\, (I_1, \ldots, I_n) \mid I_j \in \textsc{it}(P_j) \text{ for all } j \in 1..n \,\} \\
\textsc{it}(\texttt{case}\{P_1 \cdots P_n\}) & = & \{\, (k, I) \mid 1 \leq k \leq n \text{ and } I \in \textsc{it}(P_k) \,\} \\
\textsc{it}(\texttt{exi}\{\underline{\tau}\ \underline{x}; P\}) & = & \{\, (\underline{\mathbf{x}}, I) \mid I \in \textsc{it}(P) \,\} \\
\textsc{it}(\texttt{for}\{\underline{\tau}\ \underline{x}; G(\underline{x}) \rightarrow P\,\}) & = & \{\, ((\underline{\mathbf{x}}_1, I_1), \ldots, (\underline{\mathbf{x}}_n, I_n)) \mid I_j \in \textsc{it}(P) \text{ for all } j \in 1..n \,\}
\end{array}
$$

A *piece of information* for a closed property P is a pair $I : P$, with $I \in \textsc{it}(P)$. A *collection* is a set of closed simple properties. The *information content* $\textsc{ic}(I : P)$ is the collection inductively defined as follows:

$$
\begin{array}{lcl}
\textsc{ic}(\texttt{tt} : P) & = & \{P\}, \text{ where } P \text{ is a simple property} \\
\textsc{ic}(\,(I_1, \ldots, I_n) : \texttt{and}\{P_1 \cdots P_n\}\,) & = & \bigcup_{j=1}^{n} \textsc{ic}(I_j : P_j) \\
\textsc{ic}(\,(k, I) : \texttt{case}\{P_1 \ldots P_n\}\,) & = & \textsc{ic}(I : P_k) \\
\textsc{ic}(\,(\underline{\mathbf{x}}, I) : \texttt{exi}\{\underline{\tau}\ \underline{x};\ P(\underline{x})\}\,) & = & \textsc{ic}(I : P(\underline{\mathbf{x}})) \\
\textsc{ic}(((\underline{\mathbf{x}}_1, I_1), \ldots, (\underline{\mathbf{x}}_n, I_n)) : \texttt{for}\{\underline{\tau}\ \underline{x}; G(\underline{x}) \rightarrow P(\underline{x})\}\,) & = & \bigcup_{j=1}^{n} \textsc{ic}(I_j : P(\underline{\mathbf{x}}_j)) \\
 & & \cup\, \{\, \texttt{dom}(\underline{x}; G(\underline{x}); [\underline{\mathbf{x}}_1, \ldots, \underline{\mathbf{x}}_n]) \,\}
\end{array}
$$

The information content $\text{IC}(I : P)$ represents the minimum amount of information needed to get evidence for P according to I. We say that a collection $\mathscr{C}$ *gives evidence to* $I : P$, and we write $\mathscr{C} \rhd I : P$, iff one of the following clauses holds:

$\mathscr{C} \rhd \texttt{tt} : P$	iff $P \in \mathscr{C}$
$\mathscr{C} \rhd (I_1, \ldots, I_n) : \texttt{and}\{P_1 \cdots P_n\}$	iff $\mathscr{C} \rhd I_j : P_j$ *for all* $j \in 1..n$
$\mathscr{C} \rhd (k, I) : \texttt{case}\{P_1 \ldots P_n\}$	iff $\mathscr{C} \rhd I : P_k$
$\mathscr{C} \rhd (\underline{\mathbf{x}}, I) : \texttt{exi}\{\tau\, \underline{x};\, P(\underline{x})\}$	iff $\mathscr{C} \rhd I : P(\underline{\mathbf{x}})$
$\mathscr{C} \rhd ((\underline{\mathbf{x}}_1, I_1), \ldots, (\underline{\mathbf{x}}_n, I_n)) : \texttt{for}\{\tau\, \underline{x}; G(\underline{x}) \rightarrow P(\underline{x})\}$	iff $\texttt{dom}(\underline{x}; G(\underline{x}); [\underline{\mathbf{x}}_1, \ldots, \underline{\mathbf{x}}_n]) \in \mathscr{C}$ *and* $\mathscr{C} \rhd I_j : P(\underline{\mathbf{x}}_j)$ *for all* $j \in 1..n$

The information content $\text{IC}(I : P)$ represents an information about the current world state. We define the information content of $\mathscr{C}$ as its closure under (classical) logical consequence, for $\mathscr{C}^* = \{P \mid \mathscr{C} \models P\}$. We say that $\mathscr{C}_1$ *contains less information* than $\mathscr{C}_2$ (written $\mathscr{C}_1 \sqsubseteq \mathscr{C}_2$) iff $\mathscr{C}_1^* \subseteq \mathscr{C}_2^*$. Intuitively, the definition of $\sqsubseteq$ is justified by the fact that an user will "trust" $\mathscr{C}^*$, whenever he trusts $\mathscr{C}$. We could use a different trust-relation, considering different logics. We only need this to hold:

(1). $\mathscr{C} \subseteq \mathscr{C}^*$;
(2). $\mathscr{C}_1 \subseteq \mathscr{C}_2^*$ implies $\mathscr{C}_1 \sqsubseteq \mathscr{C}_2$.

Using the above properties, we can establishes the minimality of $\text{IC}(I : P)$ with respect to $\sqsubseteq$:

Theorem 1. *Let* $I : P$ *be a piece of information:*

1. $\text{IC}(I : P) \rhd I : P$
2. *For every collection* $\mathscr{C}$, $\mathscr{C} \rhd I : P$ *implies* $\text{IC}(I : P) \sqsubseteq \mathscr{C}$.

Now we can apply the above discussion to the problem of checking snapshots against constraints. Let $T = \langle \texttt{thAx}, \mathscr{T} \rangle$ be a CooML theory. We recall that a *snapshot* for T is a piece of information $I : \texttt{thAx}$. We introduce two notions of consistency for snapshots.

- A snapshot $I : \texttt{thAx}$ is consistent with respect to the constraints $\mathscr{T}$ ($\mathscr{T}$*-consistent*) iff there exists a reachable model of $\text{IC}(I : \texttt{thAx}) \cup \mathscr{T}$.
- T is *snapshot-consistent* iff there is at least one snapshot $I : \texttt{thAx}$ such that $I : \texttt{thAx}$ is $\mathscr{T}$-consistent.

The latter definition is related to classical consistency by the following result:

Theorem 2. *Let* $T = \langle \texttt{thAx}, \mathscr{T} \rangle$ *be a CooML theory.* T *is snapshot-consistent iff there is a reachable model of* $\text{T}\{\texttt{thAx}\} \cup \mathscr{T}$.

3 A Snapshots Generation Algorithm and Its Theory

A snapshot generation algorithm (SGA) for a CooML theory $T = \langle \texttt{thAx}, \mathscr{T} \rangle$ takes as input the user's *generation requirements* and tries to produce $\mathscr{T}$-consistent snapshots that satisfy such requirements. Roughly, *generation states* represent incomplete

snapshots, i.e. in logic programming parlance, partially instantiated terms; inconsistent attempts are pruned, when recognized as such during generation.

Consistency checking plays a central role. It depends on the PD logic and it is discussed next. In Subsection 3.2 we illustrate the use of snapshot generation for validating CooML specifications. Finally, in Subsection 3.3 we briefly outline a non deterministic algorithm based on which one may develop sound and complete implementations.

3.1 Consistency Checking

Here we briefly discuss a simplified version of consistency checking in our Prolog implementation, called SnaC. To recognize inconsistent attempts, SGA uses an internal representation of the information content of the current generation state S, denoted by INFO_S. Let P_S be the internal Prolog translation of the information content INFO_S. For this simplified version, we assume that P_S is executed by a suitable *meta-interpreter*. Without giving the formal details, we notice that INFO_S consists either of ground facts, clauses of the form H `:- eq(t1,t2)` or `false :-` B, where:

- We use `eq` to avoid Prolog's standard unification interfering with Skolem constants. Indeed, the latter represent unknown values originating from the translation of `T{exi{...}}`, where different constants may represent the same value. In this simplified account, the `eq` atoms are just residuated by the meta-interpreter in a list of "unsolved equations".
- The reserved atom `false` is adopted to detect inconsistency: its finite failure signals snapshot consistency, conversely, its success corresponds to inconsistency.

Clauses whose head is`false` are called *integrity constraints* and `false` may occur only as such. A SnaC representation P_S has the following property: if the meta-interpretation of a goal G succeeds from P_S with answer σ and a list L of unsolved equations, then $G\sigma$ is a logical consequence of $P_S \cup L$. Furthermore, consistency is preserved and the models of P_S are models of INFO_S (in the declarative reading of P_S, where we interpret `eq` as equality and `false` as falsehood). As an example, let us consider the SnaC representation P_{cComp} in Fig. 3 of the information content of the `coachCompany` package (Fig. 2).

```
isOf(mini,'Coach',[]).  false :- isOf(O,'Coach',[]), not(member(O,[mini]).
isOf(john 'Passenger',[t1]). isOf(ted 'Passenger',[t2]). ...
numberOfSeats(mini,4). nobooking(john,t1). booked(john,2,mini,t1).
vacant(1,mini,t1). vacant(3,mini,t1). vacant(4,mini,t1).
guides(P,t2):- eq(P,p0).
false :- guides(P,t3).
```

Fig. 3. The SnaC representation P_{cComp}

The facts and the constraint in the first lines come from the translation of domain properties. For example, the first row contains the translation of `dom{o; o is Coach(); [mini])}`. The other facts come from the translation of atoms. The clause `guides(P,t2):- eq(P,p0)` is the translation of

`T{exi{Person p; guides(p,t2)}}`, where `p0` is a fresh Skolem constant. Finally, `false :- guides(P,t3)` is the translation of `T{not exi{Person p; guides(p,t3)}}`.

Let us analyse the three possible outcomes of consistency checking starting from the example in Fig. 3:

(a) `false` finitely fails from the program P_{cComp}. This entails that `false` does not belong to the minimum model $\mathscr{M}$ of $P_{cComp} \cup \{$`eq(X,X)`$\}$. The latter contains all the ground atoms in Fig. 3 as well as `guides(p0,t2)`. Since $\mathscr{M}$ is a model of P_{cComp}, it is also a model of the information content of the `coachCompany` package thanks to the properties of the translation.

(b) If we add to P_{cComp} the constraint

```
c1)   false :- nobooking(P,T), booked(P,_S,_C,T).
```

now the goal `false` succeeds from program $P_{cComp} \cup \{$`c1`$\}$, residuating the empty list. This implies that the snapshot corresponding to the information content of `coachCompany` is inconsistent w.r.t. `c1`.

(c) If we instead add the constraint

```
c2)   false :- guides(P,T), isOf(P,'Passenger',[T]).
```

the goal `false` succeeds from program $P_{cComp} \cup \{$`c2`$\}$, residuating `[eq(ted,p0)]`. This implies that `false` belongs to the minimum model $\mathscr{M}$ of $P_{cComp} \cup \{$`c2,eq(ted,p0)`$\}$. The equality `eq(ted,p0)` is returned to the user as a source of inconsistency.

The above discussion is reflected in the following theorem:

Theorem 3. *Let $T = \langle$ `thAx`$, \mathscr{T}\rangle$ be a CooML theory, I : `thAx` a snapshot and P a program containing the translation of* $\text{IC}(I$: `thAx`$)$ *and of the PD constraints $\mathscr{T}$.*

1. *If* `false` *finitely fails from P, then I : `thAx` is $\mathscr{T}$-consistent.*
2. *If* `false` *succeeds from P residuating a set of constraints $\mathscr{U}$, then I : `thAx` is inconsistent with respect to $\mathscr{T} \cup \mathscr{U}$.*

In the first case, SnaC accepts I : `thAx` as a $\mathscr{T}$-consistent snapshot. In the second, $\mathscr{U}$ being empty signals inconsistency. If $\mathscr{U}$ is not empty, it is returned as an answer.

A more general result can be established admitting a larger class of simple properties and PD constraints, via techniques similar to those used in CLP, such as *constraint systems* [7]. Roughly, we can consider $\mathscr{T}$ as a program of a CLP system whose calculus is an extension of the standard logic programming operational semantics and where the constraint system is the Herbrand universe under CET, modified to deal with Skolem constants.

3.2 Validating Specifications Via SG

One of the purposes of snapshot generation is understanding and validating a CooML specification. To this aim, the user can specify suitable *generation requirements* in order

to reduce the number of generated examples to a manageable size and show only the aspects he is interested in. We explain the language of generation requirements and its semantics through our example. It may be helpful to keep in mind the analogy with the behaviour of an *answer set* program during grounding.

In the implementation, the number of generated snapshots can be limited by means of the the special atom `choice(`A`)`. This plays the role of *domain* predicates in ASP. The SG algorithm will instantiate A according to its axiomatisation. For example:

```
choice(isOf(C,'Coach',[])) :- member(C,[c1,c2]).
choice(isOf(P,'Passenger',[T])) :- member(P,[anna,john,ted]).
choice(isOf(T,'Trip',[C])) :- member((T,C), [(t1,c1),(t2,c2),(t3,c1)]).
choice(numberOfSeats(c1,3)).
choice(numberOfSeats(c2,60)).
```

instructs SG to generate one coach `c1` with 3 seats and possible trips `t1`, `t3`, and another `c2` with 60 seats and trip `t2`. The declarative meaning of `choice` is given by the axiom schema $A \rightarrow$ `choice(`A`)`, which, together with the user definition of `choice`, sets up the generation requirements. The generated snapshots will satisfy the PD constraints, as well as the generation requirements.

Once the SG algorithm loads a CooML theory and the user generation requirements, it can be queried with *generation goals* (*G-goals*). A sample G-goal is:

```
(g1)  [ [3,tt], Trips ] : isOf(C,'Coach',[]).
```

Since `[3,tt]:seats(C)` has information content `3 = numberOfSeats(C)`, the query looks for the information `Trips:trips(C)` for every coach `C` with 3 seats. More precisely, the G-goal includes both a generation goal ("generate all the coaches `C` with 3 seats that satisfy the generation requirements") and a query ("for each `C`, show the information on the trips assigned to it"). An answer to `g1` is:

```
Trips = [ [t1,tt] ] and C = c1
```

with information content

```
isOf(c1,'Coach',[]),isOf(t1, 'Trip', [c1])
```

The rest of the snapshot, including information terms for all classes in the package, is omitted for the sake of space. If the user asks for more solutions, all possible snapshots will be shown. In the above example, there are two more solutions, where `c1` has two trip assigned or none.

We now sketch some ways in which SG can be used in the process of system specification and development. This will be the focus of future work.

Validating specifications. The goal here is to show that a CooML theory "correctly" models the problem domain. Validation is empirical by nature: it relates the theory to the modeled world. The idea is to generate models that satisfy given generation requirements and check whether they match the user expectations. To this aim, it is useful to tune the generation requirements to consider separately various aspects that can be understood within a small, "human viable" number of examples, as usual in this context [8]. For instance, we may concentrate on the validation of the booking part of the

`CoachCompany` package. In particular, we can find some supporting evidence of the correctness of the specification in a match between the expected and actual number of snapshots, where parameters of the latter are chosen as small as possible, while preserving meaningfulness. Naturally, snapshots can be used as inputs to tools for automatic, specification-based testing generation, in the spirit of [18].

Partial and full model checking. As traditional in software model checking, here the goal is to show that, under the assumptions of the generation requirements, no snapshot satisfies an undesired property. This is obtained if the SGA finds a snapshot-inconsistency, i.e. it halts without exhibiting any snapshot. Equivalently, one can prove that every snapshot satisfies a given property by showing that its negation is snapshot-inconsistent. We call this approach *partial* model checking, because in general snapshot consistency may depend on the selection of generation requirements. We may perform full model checking if the set of generated snapshots is representative of all models of the theory w.r.t. the property under consideration.

3.3 A Schematic Algorithm

We now describe a general schema for the snapshot generation algorithm, of which SnaC is just a first rough implementation. Let $T = \langle \mathtt{thAx}, \mathscr{T} \rangle$ be a CooML theory, where $\mathtt{thAx} = \mathtt{and}\{\mathtt{clAx}(C_1), \ldots, \mathtt{clAx}(C_n)\}$. Its information terms are represented by sets of G-goals that we call *populations*. The generation process starts from a set P_0 of G-goals to be solved, i.e. to become ground. The SGA gradually instantiates P_0, possibly generating new G-goals. It divides the population in two separate sets: ToDo, containing the G-goals not solved yet and Done, containing the solved ones. A *generation state* has the form $S = \langle \textsc{Done}, \textsc{ToDo}, \textsc{Closed}, \textsc{Info} \rangle$, where:

- Closed is a set of predicates $closed(C, \underline{e})$, which is extended when all the objects with creation class $C(\underline{e})$ have been generated. It prevents the creation of new objects of class $C(\underline{e})$ in subsequent steps.
- Info is the representation in the PD language of the information content of Done, i.e. for every $I : \mathtt{isOf}(o, C, [\underline{e}]) \in \textsc{Done}$, $\textsc{ic}(I : Pty_C(o, \underline{e})) \subseteq \textsc{Info}$.

The following definitions are in order:

- A state S is *in solved form* if $\textsc{ToDo} = \emptyset$.
- $\mathrm{Dom}(S) = \{\, \mathtt{isOf}(o, C, [\underline{e}]) \mid I : \mathtt{isOf}(o, C, [\underline{e}]) \in \textsc{Done} \cup \textsc{ToDo} \,\}$.
- $S_1 \preceq S_2$ for $S_i = \langle \textsc{Done}_i, \textsc{ToDo}_i, \textsc{Closed}_i, \textsc{Info}_i \rangle$ iff
  1. $\textsc{Done}_1 \subseteq \textsc{Done}_2$, $\mathrm{Dom}(S_1) \subseteq \mathrm{Dom}(S_2)$ and $\textsc{Info}_1 \subseteq \textsc{Info}_2$;
  2. If $closed(C, \underline{e}) \in \textsc{Closed}_1$, then
     $\mathtt{isOf}(o, C, [\underline{e}]) \in \mathrm{Dom}(S_1)$ iff $\mathtt{isOf}(o, C, [\underline{e}]) \in \mathrm{Dom}(S_2)$.

The SGA starts from initial state $S_0 = \langle \emptyset, \textsc{ToDo}_0, \emptyset, \emptyset \rangle$ and yields a *solution state* $S = \langle \textsc{Done}, \emptyset, \textsc{Closed}, \textsc{Info} \rangle$ such that $S_0 \preceq S$; since $\textsc{ToDo} = \emptyset$, for every $I : \mathtt{isOf}(o, C, [\underline{e}]) \in \textsc{ToDo}_0$, Done contains a ground information term $(I : \mathtt{isOf}(o, C, [\underline{e}]))\sigma$ solving it. The algorithm computes a solution of S_0 that is minimal with respect to $\preceq$ through a sequence of *expansion steps*. The latter are triples $\langle S,\ I : \mathtt{isOf}(o, C, [\underline{e}]),\ S' \rangle$ such that:

p1. $I : \texttt{isOf}(o, C, [\underline{e}]) \in$ ToDo (the selected goal);
p2. $(I : \texttt{isOf}(o, C, [\underline{e}]))\sigma \in$ Done$'$ and $I : \texttt{isOf}(o, C, [\underline{e}]) \notin$ ToDo$'$ (it has been solved);
p3. $S \prec S'$ and, for every S^* in solved form, $S \prec S^* \preceq S'$ entails $S^* = S'$ (no solution is ignored).

The high-level code for a non deterministic SGA based on expansion steps is listed in Fig.4, where ToDo$_0$ are the G-goals to be solved under theory $\langle \texttt{thAx}, \mathscr{T} \rangle$ and generation requirements $\mathscr{G}$. The SGA is a general schema, whose core is the implementation of the expansion steps, predicates $\texttt{error}(S)$ and $\texttt{globalError}(S)$. The latter are based on the ideas presented in Section 3.1. They use the integrity constraints `false :- B` to detect inconsistency and store in the variable UC the "unsolved constraints". To ensure the correctness of SG, an implementation has to guarantee properties p1, p2, p3 of expansion steps as well as the following requirements:

(i) When new objects or new witnesses (for `exi`) are generated in an expansion step, they are chosen according to the generation requirements, in such a way that Info$_S \models \mathscr{G}$ for every generated state S.
(ii) When `error(S)` returns "true", then Info$_{S'}$ is inconsistent w.r.t. $\mathscr{T}$ for every S' such that $S \preceq S'$ (S included).
(iii) If `globalError(S)` returns "true", then Info$_S$ is inconsistent with respect to $\mathscr{T}$. If it returns "false", then either UC is empty and Info$_S \cup \mathscr{T}$ is consistent or Info$_S \cup \mathscr{T} \cup UC$ is inconsistent.

SG ($\langle$thAx, $\mathscr{T}\rangle$, $\mathscr{G}$, *ToDo*$_0$)
1 *Thy* = thAx; *PDAx* = $\mathscr{T} \cup \mathscr{G}$; $S = \langle \emptyset, \mathit{ToDo}_0, \emptyset, \emptyset \rangle$; $UC = \emptyset$;
2 **while** *ToDo* $\neq \emptyset$ **do**
3 **if** error(S) **fail**;
4 **else** % *Generation Step:*
5 Choose I : isOf(o,C,[$\underline{e}$]) $\in$ *ToDo* and compute $\langle S, I : \texttt{isOf}(o, C, [\underline{e}]), S' \rangle$;
6 $S = S'$;
7 **if** globalError(S) **fail**;
8 **else return** S, UC

Fig. 4. The SG Algorithm

The current implementation is essentially based on a refinement of the meta-interpreter considered in Section 3.1. It could be improved, namely in detecting more than trivial inconsistencies; indeed, no constraint simplification is supported.

To state the adequacy results, we introduce some additional notation (ITP) in order to associate a class C_j and population P with their information terms:

$$\begin{aligned} \text{ITP}(P, C_j) &= [\ [[o_{j_1}, \underline{e}_{j_1}], I_{j_1}]\ ,\ \ldots\ ,\ [[o_{j_k}, \underline{e}_{j_k}], I_{j_k}]\ ] \\ \text{ITP}(P) &= [\ \text{ITP}(P, C_1)\ ,\ \ldots\ ,\ \text{ITP}(P, C_n)\ ] \end{aligned}$$

where, $I_{j_1} : \texttt{isOf}(o_{j_1}, C_j, [\underline{e}_{j_1}]), \ldots, I_{j_k} : \texttt{isOf}(o_{j_k}, C_j, [\underline{e}_{j_k}])$ are the G-goals of P with class C_j ($1 \leq j \leq n$); if no G-goal with class C_j belongs to P, then $\text{ITP}(P, C_j)$ is the empty list.

Theorem 4 (Correctness). *Let $S^* = \langle \text{DONE}^*, \emptyset, \text{CLOSED}^*, \text{INFO}^* \rangle$ be a state computed by SG with theory $T = \langle \texttt{thAx}, \mathscr{T} \rangle$ and generation requirements $\mathscr{G}$, and let $I^* = \text{ITP}(\text{DONE}^*)$ be the information term of the population DONE^*. Then, either UC is empty and $I^* : \texttt{thAx}$ is $\mathscr{G} \cup \mathscr{T}$-consistent, or $I^* : \texttt{thAx}$ is inconsistent with respect to $\mathscr{G} \cup \mathscr{T} \cup UC$.*

The proof follows from properties (i), (ii) and (iii).

Theorem 5 (Completeness). *Let $S_0 = \langle \emptyset, \text{TODO}_0, \emptyset, \emptyset \rangle$ be an initial state of SG with theory T and generation requirements $\mathscr{G}$. If there is a state $S = \langle \text{DONE}, \emptyset, \text{CLOSED}, \text{INFO} \rangle$ such that $S_0 \preceq S$, then the SGA reaches a state S^* in solved form such that $S_0 \preceq S^* \preceq S$.*

The proof follows from properties p1, p2 and p3.

4 Related Work and Conclusion

We have presented some features of the object-oriented modeling language CooML, a language in the spirit of UML, but based on a constructive semantics, in particular the BHK explanation of logical correctives. We have introduced a proof-theoretic notion of snapshot based on populations of objects and information terms, from which snapshot generation algorithms can be designed. More technically, we have introduced generation goals and the notion of minimal solution of such goals in the setting of a CooML specification, and we have outlined a non-deterministic generation algorithm, showing how finite minimal solutions can be, in principle, generated. We use a constraint language in order to specify the general properties of the problem domain, as well as the generation requirements. In an implementation of the SGA we assume a consistency checking algorithm, which either establishes the (in)consistency of the current snapshot, or residuates a set of unsolved constraints.

The relevance of SG for validation and testing in OO software development is widely acknowledged. The USE tool [8] for validation of UML/OCL models has been recently extended with a SG mechanism; differently from us, this is achieved via a procedural language. Other animation tools [4] are based on JML specification. In [2] the specification of features models are translated into SAT problems; tentative solutions are then propagated with a Truth Maintenance System. If a inconsistency is discovered the TMS explains the causes in view of possible model repair. Related work includes also [16], where design space specs are seen as trees whose nodes are constrained by OCL statements and BDD's are used to find solutions.

Snapshot generation is only one of CooML's aspects, once we put our software engineering glasses on and see it more generally as a *specification* rather than modeling language [12, 9]. In this paper we have not considered *methods*, although the underlying logic supports a clean notion of (correct) *query* methods, namely those that do not

update the system state, but extract pieces of information from it. The existence of a method M answering P (i.e., computing $I : P$) is guaranteed when P is a constructive logical consequence of `thAx`. Moreover, M can be extracted from a constructive proof of P. The implementation of query and update methods is a crucial part of future work.

We also plan to improve and extend the snapshot generation algorithm. There are two directions that we can pursue; first, we can fully embrace CLP as a PD logic, strengthening the connection that we have only scratched in Section 3.1. In the current prototype there is little emphasis on the simplification of unsolved constraints. This could be partially ameliorated by adopting CLP, in particular over finite domains. More in general, it is desirable to relate Theorem 3 with the notion of satisfaction-completeness in constraint systems [7]. Another direction comes from the relation between CooML's approach to incomplete information and answer set programming [1, 17], in particular disjunctive LP [13]. A naive extension of the SGA to this case would yield inefficient solutions, yet the literature offers several ways constraints and ASP may interact [14, 5]. We may explore the possibility of combining snapshot generation with SAT provers, to which we may pass ground unsolved constraints in order to check global consistency. Finally we intend to explore the more general issue of the relationships between information terms and stable models, in particular partial stable models [21]; some initial results are presented in [20].

References

1. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. In: CUP (2003)
2. Batory, D.S.: Feature models, grammars, and propositional formulas. In: Obbink, H., Pohl, K. (eds.) SPLC 2005. LNCS, vol. 3714, pp. 7–20. Springer, Heidelberg (2005)
3. Boronat, A., Oriente, J., Gómez, A., Ramos, I., Carsí, J.A.: An algebraic specification of generic OCL queries within the Eclipse modeling framework. In: Rensink, A., Warmer, J. (eds.) ECMDA-FA 2006. LNCS, vol. 4066, pp. 316–330. Springer, Heidelberg (2006)
4. Bouquet, F., Dadeau, F., Legeard, B., Utting, M.: JML-testing-tools: A symbolic animator for JML specifications using CLP. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 551–556. Springer, Heidelberg (2005)
5. Buccafurri, F., et al.: Strong and weak constraints in disjunctive Datalog. In: Dix et al. [6]. pp. 2–17.
6. Dix, J., Furbach, U., Nerode, A. (eds.): LPNMR 1997. LNCS, vol. 1265. Springer, Heidelberg (1997)
7. Fruewirth, T., Abdennadher, S.: Essentials of Constraint Programming. Springer, New York (2003)
8. 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)
9. Guttag, J.V., Horning, J.J.: Larch: languages and tools for formal specification. Springer, New York, Inc., New York, NY, USA (1993)
10. Jackson, D., Wing, J.: Lightweight formal method. IEEE Computer, Los Alamitos (1996)
11. Larman, C.: Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development. Prentice Hall, Upper Saddle River, NJ (2004)
12. Leavens, G.T., Baker, A.L., Ruby, C.: Preliminary design of JML: A behavioral interface specification language for Java. SIGSOFT Softw. Eng. Notes 31(3), 1–38 (2006)

13. Leone, N., et al.: The DLV system for knowledge representation and reasoning. ACM Trans. Comput. Log. 7(3), 499–562 (2006)
14. Marek, V.W., et al.: Logic programs with monotone cardinality atoms. In: Lifschitz, V., Niemelä, I. (eds.) LPNMR 2004. LNCS (LNAI), vol. 2923, pp. 154–166. Springer, Heidelberg (2003)
15. Miglioli, P., Moscato, U., Ornaghi, M., Usberti, G.: A constructivism based on classical truth. Notre Dame Journal of Formal Logic 30(1), 67–90 (1989)
16. Neema, S., et al.: Constraint-based design-space exploration and model synthesis. In: Alur, R., Lee, I. (eds.) EMSOFT 2003. LNCS, vol. 2855, pp. 290–305. Springer, Heidelberg (2003)
17. Niemelä, I., Simons, P.: Smodels - an implementation of the stable model and well-founded semantics for normal lp. In: Dix et al. [6]. pp. 421–430
18. Offutt, J., Abdurazik, A.: Generating tests from UML specifications. In: France, R.B., Rumpe, B. (eds.) UML 1999. LNCS, vol. 1723, pp. 416–429. Springer, Heidelberg (1999)
19. Ornaghi, M., Benini, M., Ferrari, M., Fiorentini, C., Momigliano, A.: A constructive object oriented modeling language for information systems. ENTCS 153(1), 67–90 (2006)
20. Ornaghi, M., Fiorentini, C.: Answer set semantics vs. information term semantics. In: Informal Proceedings of ASP 2007: Answer Set Programming: Advances in Theory and Implementation, `http://cooml.dsi.unimi.it/papers/asp.pdf`
21. Przymusinski, T.C.: Well-founded and stationary models of logic programs. Ann. Math. Artif. Intell. 12(3–4), 141–187 (1994)
22. Troelstra, A.S.: From constructivism to computer science. TCS 211(1–2), 233–252 (1999)
23. Warmer, J., Kleppe, A.: The Object Constraint Language: Precise Modelling with UML. In: Object Technology Series, Addison-Wesley, Reading/MA (1999)