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 define snapshots as a formal counterpart of object populations, which are associated to specifications via the constructive interpretation of logical connectives. In this paper, 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 successful termination implies consistency of the system. This is analogous to adequacy results in the theory of CLP's.

1 Introduction

We are developing the Constructive Object Oriented Modeling Language CooML [1] as a specification language for OO systems [23]. Similarly to UML/OCL [27], 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 problem domain and internally-defined elements, the former corresponding to loosely or non-already defined components. This feature permits the possibility to select the appropriate level of abstraction as far as specifications are concerned. For example, in an initial stage, the user may *loosely* specify modules that will be subsequently completely detailed.

CooML positions itself in the family of lightweight formal methods [11]. This means that it is possible to analyse specifications in a modular way, rather than focusing on whole system correctness. In the context of OO modeling, the issue of specification *validation*, including consistency checking, 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 [9], as well as formal specifications in JML based on symbolic animation [5].

However, CooML is based on a semantics related to the constructive explanation of logical connectives (aka the BHK interpretation [26]). Specifically, the truth of a CooML proposition in a given interpretation is explained by a mathematical object we call an *information term*, akin to the way a proof term inhabits a type. The underlying logic

[1] http://cooml.dsi.unimi.it

introduced in [18] is characterized by the different way it gives evidence to an *atomic* formula. If we call a *piece of information* a pair $I : P$, where P is a formula and I is its information term, then for an atom A, $I : A$ represents a piece of information that may be *true* or *false* in a world w. The world w can be viewed as a classical interpretation. 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, T does not contain evidence for F, but it yields a piece of information true in all the models of F. This introduces a novel and flexible way to handle *incomplete* information, compare this to the notorious difficulties partial information encounters in other information systems such as relational databases. For more details on the logical aspects, we refer to [18].

Crucially, in this logic we can identify snapshots with information terms, thusly providing a formal counterpart to object populations. Our notion of snapshot has connections with model generation in Answer Set Programming [16, 21]. We argue that CooML's snapshot generation, with its proof theoretic flavor, can be advantageous with respect to the pure model-theoretic one, especially in cases where not all the information needed to define a model is required. The possibility of treating information in a less committed way with respect to a purely model-theoretic approaches, has effects on the possibility of *selecting only the relevant information*, and possible gain in the efficiency of the *grounding procedure*. Further, models have been proposed for representation and solving combinatorial problems through propositional theories [1]. This suggests the possibility to exploit CooML as a language for representing combinatorial problems, solving them by snapshot generation. This is pursued elsewhere[2].

In this paper, we describe SGA, a snapshot generation algorithm, which has as inputs a CooML theory T, axiomatizing a set of classes in a problem domain PD and the user's generation requirements $\mathscr{G}$, expressed in a parametric logic L. The latter is also the language of T-formulae. Snapshots should be consistent with respect to PD and $\mathscr{G}$. To check consistency, we associate to $I : T$ a set of formulae, representing its information content. More specifically, this can be seen as the *minimum* associated information. We show that the algorithm is sound and complete so that successful termination implies consistency of the system. This is analogous to adequacy results in the theory of CLP's.

Next, Section 2 introduces CooML theories first by example and then formally. Section 3 describes the SG algorithm and the theory behind it, before concluding in Section 4 positioning CooML w.r.t other modeling languages in software engineering as well as answer set programming, in connection with model generation.

2 CooML specifications

In CooML we distinguish the *internal signature* Σ, which includes all the symbols explicitly defined in a CooML specification, and the *global signature* Γ, which extends Σ by including symbols that refer to the external world. We model the internal *system states* as Σ-interpretations, and the global *world states* as Γ-interpretations. The language is organized around classes that are structured in a class hierarchy with single inheritance rooted in *Obj*. All the entities of a CooML system are called *values*. They belong to the general supertype `Value`, which defines the identity relation and the string representation. `Obj` is a subtype of `Value`. The elements of Γ and their interpretation in a world state are defined as follows:

[2] `http://cooml.dsi.unimi.it/papers/modelGeneration06.pdf`

```
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 of %c in %t *);
  Boolean booked(Passenger p, Integer s, Coach c, Trip t) =
                                  (* %p booked seat %s of %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 :- isOf(P,'Passenger',T), guides(P,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{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{trip is Trip(coach); booked(this,seat,coach,trip)}
             }}}
```

Fig. 1: The `coachCompany` package

- *PD Types*. A *Problem Domain Type* extends `Value` by a set of problem domain functions and predicates. We have external *generic PD types* and internal *data types* (`Integer`, `String`, ...). The latter are "statically" defined, i.e., their values do not depend on the current state; they are internal, i.e., CooML assumes that an implementation is given, which evaluates ground terms into values. Nothing instead is assumed on generic PD types. They may be characterized by a set of formal or informal *loose properties* that we call *PD constraints*.
- *Object classes*. Internal. An object class `C(`$\underline{e}$`)` extends `Obj`, by introducing a set of *environment parameters* $\underline{e}$, a class property Pty_C(`this`,$\underline{e}$) [3] and a set of *methods*. The

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

class property establishes the structure and the meaning of the information contained in its objects. Let's denote by $\underline{\mathbf{e}}$ a ground instance of all the environment parameters $\underline{e}$; in particular, `C(`$\underline{\mathbf{e}}$`)` can be used to create new objects. An object **o** created by `C(`$\underline{\mathbf{e}}$`)` has *environment* $\underline{\mathbf{e}}$, contains an *information term I* structured according to $Pty_C(\mathbf{o},\underline{\mathbf{e}})$, and uses the *methods* implemented by `C(`$\underline{\mathbf{e}}$`)`. We write "**x** `is C(`$\underline{\mathbf{e}}$`)`" to indicate that **x** has been created by `C(`$\underline{\mathbf{e}}$`)`, while "**x** `instanceof C(`$\underline{\mathbf{e}}$`)`" means that **x** has environment $\underline{\mathbf{e}}$ and has been created by a subclass `C'` of `C`. We call "`_ is _`" and "`_ instanceof _`" *class predicates*.

In CooML specifications are called *theories* and consist of packages introducing a set of PD types, PD functions, PD constraints and a set of object classes. Here we informally present the language via an example adapted from [4], where we have introduced more details, in order to illustrate the main features of the language. The *problem domain* concerns a simple 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 package in Fig. 1 is a possible CooML specification of this setup.

The `pds` declaration (problem domain specification) introduces the PD type `Person` and the PD functions used in the specification. The informal descriptions `(*...*)` use terms of the global signature provided by the analysis phase [13], where a notation such as `%t` links the parameter `t` of the PD function to the related comment. A `<constr>` declaration introduces a set of PD constraints, that can be expressed in a formal or informal language. PD constraints represent general problem domain properties that are not interpreted by CooML but that could be interpreted by some external tool. In our example PD constraints are expressed in Prolog and formalize the intended meaning of PD elements needed by the snapshot generation algorithm: the first constraint says that a coach seat cannot be vacant and booked at the same time, the second one excludes overbooking (a place can be booked by at most one person), the third one says that a guide cannot be a passenger.

The `pds` part contains our knowledge about the problem domain, but it does not specify the information content of the system. The latter is given, collectively, by the information terms contained in the system objects that are alive in the current state `w`. For example, 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` and, so far, only a passenger `john`. Let *I* be the information term contained in an object with class predicate **o** `is C(`$\underline{\mathbf{e}}$`)` and class property $Pty_C(\mathbf{o},\underline{\mathbf{e}})$. We say that **o** carries the *piece of information* $I : Pty_C(\mathbf{o},\underline{\mathbf{e}})$. We defer the formal development to Section 2.1 and we explain these notions through our example. For class properties, CooML uses a prefix syntax, where formulae may be labeled. Labels are used in fact to refer to subformulae in methods and in the documentation. For example, the label `seats` is used in the `getSeats` method to get `seatsNr`. A piece of information may be:

Simple. `tt:T{`A`}`, where A can be any formula. The `T` operator indicates that we are only interested in the truth value of A. The information *content* is the set $\{A\}$, simply indicating that A is true in the current world state. For example,

```
tt:T{t1 is Trip(mini);t2 is Trip(mini);t3 is Trip(mini)}
```

means that t1,t2,t3 are assigned to mini. The pieces of information

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

say that t2 is a private guided trip (no information about the guide), whereas t3 has no guides. If A is an atomic formula, the operator T can be omitted.

Existential. (**x**, I) :exi{$\tau\, x$; $P(x)$}, where τ is the type of the existential variable x. The term **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 pieces of information, we know the value of x which makes $P(x)$ true.

Universal. ((**x**$_1$,I_1), . . . , (**x**$_n$,I_n)) :for{$\tau\, x$; $G(x)$ → $P(x)$}, where $G(x)$ is an *x-generator*, i.e., a simple property true for finitely many x. The information content is the union of those of $I_1 : P(\mathbf{x}_1), \ldots, I_n : P(\mathbf{x}_n)$ and of the *domain information formula* dom(x; $G(x)$; **x**$_1$, . . . , **x**$_n$), which lists the terms **x**$_1$,. . . ,**x**$_n$ that verify $G(x)$. 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 are, for instance

((4,tt), ((t1,tt), (t2,tt), (t3,tt))) : and{seats(mini) trips(mini)}
IC_1 = {4 = numberOfSeats(mini), dom(trip; trip is Trip(mini); t1,t2,t3)}

Disjunctive. (k,I_k) :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 atom john is Passenger(t1) contains the information term (1,tt), then

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

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

Pieces of information can also be attached to classes and theories.

Class. A class C($\underline{e}$) with property $Pty_C(this, \underline{e})$ is represented by the *class axiom*

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

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:

((mini,CoachInfo)) : for{Obj this; this is Coach() → coachPty(this)}
IC_2 = {dom(this; this is Coach();mini), 4 = numberOfSeats(mini),
dom(trip; trip is Trip(mini); t1,t2,t3)}

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

Theory. A CooML theory is represented by its class axioms and PD constraints. The formers establish the possible pieces of information and the related information contents, the latter do not carry information, but introduce formal or informal "integrity constraints".

We remark that there are two sources of incomplete information. The first one depends on the use of T(e.g., in T{exi{Person p; guides(p,t2)}), while the second

one is due to the fact that only the information explicitly required by class properties belongs to the information content of a snapshot. In a sense, the information content of a snapshot of a theory T could be seen as an "incompletely specified" model of T.

We conclude this section by discussing system snapshots and constraint satisfaction. By $T_P = \langle \mathtt{thAx}, \mathscr{T} \rangle$ we indicate the CooML theory associated with a CooML package P defining classes $C_1, \ldots, C_n$, where $\mathtt{thAx} = \mathtt{and}\{\mathtt{clAx}(C_1) \cdots \mathtt{clAx}(C_n)\}$ is the conjunction of all the class axioms and $\mathscr{T}$ are the PD constraints of P. A piece of information $I : \mathtt{thAx}$ will be called a *system snapshot*. A snapshot for our `coachCompany` system is a piece of information of the form:

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

and possible information terms I_1, I_2, I_3 are

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

where `[...]` denote tuples. The information content is (for conciseness, we show only part of it):

```
dom([x,y]; x is Passenger(y);[john,t1]), dom(x; x is Trip(mini); t1,t2,t3),
nobooking(john,t1), booked(john,2,mini,t1), ...
```

To check the latter against the PD constraints, we have to represent it in the format required by the PD logic. In our example, we have the Prolog clauses:

```
isOf(john, 'Passenger',[t1]).
isOf(T, 'Trip',[mini]) :- T=t1; T=t2; T=t3.
nobooking(john,t1).
booked(john,2,mini,t1).
.....
```

The above piece of information is consistent with respect to the three sample constraints. If instead we add the constraint

```
false :- isOf(T,'Trip',[C]), nobooking(P,T), booked(P,_Seat,C,T),
```

the above piece of information becomes inconsistent and should be discarded.

2.1 Formal definitions

Here we give a formal treatment of the constructive semantics of CooML and discuss the relationship with the classical semantics. The syntax of CooML entities should be clear from the previous informal discussion and it is omitted for the sake of space[4]. We inductively define the set of *information terms* for a property P, denoted by $\mathrm{IT}(P)$, where A stands for any formula, G for a $\underline{x}$-generator and $\underline{\mathbf{x}}$ for values of $\underline{x}$:

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

[4] See `http://cooml.dsi.unimi.it/iclp.html`

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

$$
\begin{array}{lcl}
\mathrm{IC}(\mathtt{tt} : \mathrm{T}\{A\}) & = & \{\mathrm{T}\{A\}\} \\
\mathrm{IC}(\,(I_1,\ldots,I_n) : \mathtt{and}\{P_1 \cdots P_n\}\,) & = & \bigcup_{j=1}^{n} \mathrm{IC}(I_j : P_j) \\
\mathrm{IC}(\,(k,I) : \mathtt{case}\{P_1 \ldots P_n\}\,) & = & \mathrm{IC}(I : P_k) \\
\mathrm{IC}(\,(\underline{\mathbf{x}},I) : \mathtt{exi}\{\underline{\mathtt{T}}\,\underline{x}; P(\underline{x})\}\,) & = & \mathrm{IC}(I : P(\underline{\mathbf{x}})) \\
\mathrm{IC}(\,((\underline{\mathbf{x}}_1,I_1),\ldots,(\underline{\mathbf{x}}_n,I_n)) : \mathtt{for}\{\underline{\mathtt{T}}\,\underline{x}; G(\underline{x}) \rightarrow P(\underline{x})\}\,) & = & (\,\bigcup_{j=1}^{n} \mathrm{IC}(I_j : P(\underline{\mathbf{x}}_j))\,) \\
 & & \cup\, \{\, \mathtt{dom}\,(\underline{x}; G(\underline{x}); \underline{\mathbf{x}}_1,\ldots,\underline{\mathbf{x}}_n)\,\}
\end{array}
$$

The information content $\mathrm{IC}(I : P)$ represents the minimum amount of information needed to get evidence for P according to I.

Definition 1. *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:*

$$
\begin{array}{lll}
\mathscr{C} \rhd \mathtt{tt} : \mathrm{T}\{A\} & \textit{iff} & \mathrm{T}\{A\} \in \mathscr{C} \\
\mathscr{C} \rhd (I_1,\ldots,I_n) : \mathtt{and}\{P_1 \cdots P_n\} & \textit{iff} & \mathscr{C} \rhd i_j : P_j \text{ for all } j \in 1..n \\
\mathscr{C} \rhd (k,I) : \mathtt{case}\{P_1 \ldots P_n\} & \textit{iff} & \mathscr{C} \rhd I : P_k \\
\mathscr{C} \rhd (\underline{\mathbf{x}},I) : \mathtt{exi}\{\underline{\mathtt{T}}\,\underline{x}; P(\underline{x})\} & \textit{iff} & \mathscr{C} \rhd I : P(\underline{\mathbf{x}}) \\
\mathscr{C} \rhd ((\underline{\mathbf{x}}_1,I_1),\ldots,(\underline{\mathbf{x}}_n,I_n)) : \mathtt{for}\{\underline{\mathtt{T}}\,\underline{x}; G(\underline{x}) \rightarrow P(\underline{x})\} & \textit{iff} & \mathtt{dom}\,(\underline{x}; G(\underline{x}); \underline{\mathbf{x}}_1,\ldots,\underline{\mathbf{x}}_n) \in \mathscr{C} \\
 & & \text{and } \mathscr{C} \rhd I_j : P(\underline{\mathbf{x}}_j) \text{ for all } j \in 1..n
\end{array}
$$

Hence a collection $\mathscr{C}$ is a set of ground simple properties $\mathrm{T}\{A\}$ and `dom` predicates, where syntax and semantics of A are left open. This allows us to tune simple properties to PD constraints and its logic. We require only that A can be understood by the final user as an information about the current world state. For example, $\mathrm{T}\{\mathtt{age(john)=5}\}$ tell the user the current age of `john`. This information is equivalent to $\mathrm{T}\{\mathtt{age(john)=5;\ 3+3=6}\}$, since the user is supposed to already know that 3+3=6. To deal with this situation at the proper level of abstraction, we introduce the notion $\mathscr{C} \Rightarrow P$, meaning that P is a *consequence* of $\mathscr{C}$. We leave this notion generic; intuitively, it means that an user will trust P, whenever he trusts $\mathscr{C}$. We only require the following to hold, where $\mathscr{C}^* = \{P \mid \mathscr{C} \Rightarrow P\}$:

$$\mathscr{C} \subseteq \mathscr{C}^* \tag{1}$$

$$\mathscr{C}_1 \subseteq \mathscr{C}_2^* \text{ implies } \mathscr{C}_1^* \subseteq \mathscr{C}_2^* \tag{2}$$

We read $\mathscr{C}_1 \subseteq \mathscr{C}_2$ as "$\mathscr{C}_1$ contains less information than $\mathscr{C}_2$".

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

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

Now we apply the above discussion to the problem of checking snapshots against constraints. We recall that a *snapshot* for a CooML theory $T = \langle \mathtt{thAx}, \mathscr{T} \rangle$ is a piece of information $I : \mathtt{thAx}$. We assume that simple properties directly use the PD language, which contains `false`, and we say that a collection $\mathscr{C}$ of simple properties is *consistent* iff $\mathscr{C} \not\Rightarrow \mathtt{false}$. The notion of $\Rightarrow$ is still generic. Intuitively, consistency means that there is at least a world state satisfying $\mathscr{C}$.

Definition 2. *Let $T = \langle \mathtt{thAx}, \mathscr{T} \rangle$ be a CooML theory, and $I : \mathtt{thAx}$ a snapshot. We say that $I : \mathtt{thAx}$ is T-*consistent *iff* $\textsc{ic}(I : \mathtt{thAx}) \cup \mathscr{T} \not\Rightarrow \mathtt{false}$.

The above definition is justified by the fact that $\textsc{ic}(I : \mathtt{thAx})$ is the minimum information needed to give evidence to $I : \mathtt{thAx}$.

Definition 3. *A CooML theory $T = \langle \mathtt{thAx}, \mathscr{T} \rangle$ is* snapshot-consistent *iff it has at least one T-consistent snapshot.*

This second definition is more problematic than the previous one for an abstract notion of consequence. Thus, we show that by using a PD logic based on classical Γ-interpretations (world-states), where Γ is the global signature, snapshot-consistency coincides with the usual one: a theory is snapshot-consistent iff there is a world state satisfying it. We define the following translation $(\_)^\dagger$ from CooML formulae into first order logic (assuming that simple properties contain first order formulae):

$$\begin{aligned} (\mathtt{T}\{A\})^\dagger &= A \\ (\mathtt{case}\{P_1 \ldots P_n\})^\dagger &= (P_1)^\dagger \vee \ldots \vee (P_n)^\dagger \\ (\mathtt{for}\{\underline{\mathtt{t}}\ \underline{x};\ G(\underline{x}) \to P\})^\dagger &= \exists L.\ \forall \underline{x}.\ (G(\underline{x}) \leftrightarrow \mathit{member}(\underline{x}, L)) \wedge (G(\underline{x}) \to (P)^\dagger) \end{aligned}$$

and so on, omomorphically, for the other connectives. In the latter formula L is of the appropriate list type and $\mathit{member}(\underline{x}, L)$ is true iff $\underline{x}$ is an element of L. By $\mathscr{C} \Rightarrow P$, we mean that for every world state $\mathtt{w}$, $\mathtt{w} \models (\mathscr{C})^\dagger$ entails $\mathtt{w} \models (P)^\dagger$, where world states are Γ-interpretations and the truth relation is the standard one. Finally, we say that a world state is *reachable* iff every individual can be denoted by a ground term of Γ.

Theorem 2. *Let $\mathtt{w}$ be a reachable world state and P a CooML property. Then $\mathtt{w} \models (P)^\dagger$ iff there is a piece of information $I : P$ such that $\mathtt{w} \models (\textsc{ic}(I : P))^\dagger$.*

Henceforth, we will leave the translation $(\ldots)^\dagger$ understood. Reachability is needed to obtain the values required by existential properties and to represent generators domains. Further, it is indeed reasonable to assume that world states are reachable:

Corollary 1. *Let $T = \langle \mathtt{thAx}, \mathscr{T} \rangle$ be a CooML theory. Then T is snapshot-consistent iff $\mathtt{thAx} \cup \mathscr{T} \not\Rightarrow \mathtt{false}$.*

The proof follows from Theorem 2, the reachability of world states, and the fact that $\mathtt{thAx} \cup \mathscr{T} \Rightarrow \mathtt{false}$ iff there is no world state such that $\mathtt{w} \models \mathtt{thAx} \cup \mathscr{T}$. The class of world states and the PD logic are strictly related. In the next section we show a generation algorithm that uses a fragment of classical logic to handle PD constraints and simple properties.

3 A snapshots generation algorithm and its theory

A Snapshot Generation Algorithm (SGA) for a CooML theory $T = \langle \mathtt{thAx}, \mathscr{T} \rangle$ takes as input a set $\mathscr{G}$ of simple properties, representing the user's *generation requirements* and tries to produce T-consistent snapshots that satisfy the generation requirements. Roughly, *generation states* represent incomplete snapshots, e.g. 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 *SGA*, on the top of which sound and complete implementations can be developed.

Notation- $T = \langle \texttt{thAx}, \mathscr{T} \rangle$ will be a CooML theory with global signature Γ, and *world states* of T will be reachable (classical) models of $\mathscr{T}$.

3.1 Consistency checking

To recognize inconsistent attempts, *SGA* uses the information content of (the pieces of information of) the current generation state S, denoted by INFO_S. The syntax of the formulae in INFO_S, $\mathscr{T}$ and $\mathscr{G}$ depends on the PD logic L. Here we consider logic programming based PD logics.

We first consider our Prolog implementation and then we show how it can be extended/modified to include different paradigms, in particular CLP [8,12]. Our implementation, called SnaC, contains a meta-predicate `holds(G)`, which checks the consistency of $\text{INFO}_S \wedge G$. Inconsistency is detected by means of the special atom `ff`: its finite failure signals snapshot consistency, conversely, its success corresponds to inconsistency. Clauses with head `ff` are called *integrity constraints* and `ff` may occur only as such. The formulae of INFO_S and $\mathscr{G}$ are CooML *simple* properties. We limit the form of the latter in order to properly represent them in SnaC. We omit the detailed definition of the *permitted* simple properties; the most significant cases are shown (left hand side) in the following example, together with their (simplified) SnaC representation (on the right).

Example 1.

1) `nobooking(john,t1)`	`nobooking(john,t1).`
2) `T{exi{Person p; guides(p,t2)}}`	`guides(p_1,t2).`
3) `T{not exi{Person p;guides(p,t3)}}`	`ff :- holds(guides(_P,t3)).`
4) `4=numberOfSeats(mini)`	`numberOfSeats(mini,4).`

The atom 1) is collected "as is". The formula 2) is translated into `guides(p_1,t2)`, where `p_1` is a new *Skolem constant*. A Skolem constant arises from a T-existential formula and stands for an unknown element; the introduction of Skolem constants, as well-known, preserves satisfiability. Negative formulae such as 3) are translated into integrity constraints. Atoms containing PD functions, such as 4), are translated into relations; the corresponding functionality constraints are implicit in the `holds` implementation.

In SnaC, $\mathscr{T}$ is $CET \cup \mathscr{P}$, where CET is Clark's Equality Theory and $\mathscr{P}$ is a logic program (possibly) using integrity constraints, as in Fig. 1. A translation from $\mathscr{P}$ in the internal form is needed. For example, the first PD constraint in Fig. 1 becomes

```
ff :- holds((vacant(S,C,T),booked(_P,S,C,T))).
```

We will indicate by $\mathscr{P}^{\#}$ the internal representation of $\mathscr{P}$. In $\mathscr{P}$, the PD predicates can occur only in the body of clauses, and in $\mathscr{P}^{\#}$ within `holds`-atoms. We consider `holds(A)` as equivalent to A, i.e., `holds` *can be ignored in the logical reading of formulae*. It is simply a meta-predicate for invoking the internal procedure checking A in the current information content $\text{INFO}_S^{\#}$. The internal procedure is designed to properly treat the part left *open* by $\mathscr{P}$ (for a discussion about open programs, see [14]), namely the Skolem constants coming from T-existential formulae and the PD predicates. The latter are "closed"

by INFO_S in different ways by different snapshots. Furthermore, we assume that Γ contains a denumerable set of Skolem constants that are not included in CET, because they denote unknown individuals.

A terminating call to `holds(G)` may have two outcomes:

(A). It finitely fails; in this case, $CET \cup (\mathscr{P}^{\#} \cup \text{INFO}_S^{\#})^{\leftrightarrow} \models \neg\exists(G)$, where $(\ldots)^{\leftrightarrow}$ denotes the iff completion.

(B). It succeeds computing an answer substitution σ; in this case, a set $\mathscr{U}$ of "unsolved constraints" is collected, such that $CET \cup (\mathscr{P}^{\#} \cup \text{INFO}_S^{\#}) \models \forall(\mathscr{U} \rightarrow G\sigma)$. We call σ a $\mathscr{U}$-answer, where in this case $\mathscr{U}$ contains constraints on Skolem constants (see Section 3.2), but it could, in general, include any delayed constraint.

We say that I : `thAx` is *consistent w.r.t.* $\mathscr{G}$ if $\text{IC}(I : \texttt{thAx}) \cup \mathscr{G} \not\Rightarrow \texttt{false}$. Since world states that satisfy $\mathscr{G}$ are reachable models of $CET \cup (\mathscr{P} \cup \mathscr{G})^{\#}$ that interpret `ff` as `false` (according to $(\ldots)^{\#}$), one can prove:

Theorem 3. *Let $T = \langle \texttt{thAx}, \mathscr{T} \rangle$ be a CooML theory, $\mathscr{G}$ a set of generation requirements and I : `thAx` a snapshot.*

1) If `ff` finitely fails, then $\text{IC}(I : \texttt{thAx}) \cup \mathscr{G} \not\Rightarrow \texttt{false}$.
2) If `ff` succeeds collecting a (possibly empty) set $\mathscr{U}$ of unsolved constraints, then $\text{IC}(I : \texttt{thAx}) \cup \mathscr{G} \Rightarrow \neg\exists(\mathscr{U})$.

In case 1), SnaC accepts I : `thAx` as a T-consistent snapshots w.r.t. $\mathscr{G}$. Case 2) is used as follows; the snapshot I : `thAx` is not T-consistent w.r.t. $\mathscr{G}$ iff, for every world state `w` such that $w \models \mathscr{G}$, we have $w \models \exists(\mathscr{U})$. In general, the latter fact cannot be decided, but there are cases that can be immediately recognized. When such a case is found, SnaC discards the snapshot. Otherwise, it gives $\neg\exists(\mathscr{U})$ as answer constraint.

The soundness and finite failure properties of answers are akin to similar properties in CLP (soundness of successful derivations and completeness of failed ones [8]), when one sees a CLP system as constituted by a *constraint system*, characterizing the constraint solver and a *calculus*, formalized as a transition system [8]. Roughly, we can consider $\mathscr{T}$ as a program of a CLP system using, as calculus, an extension of the standard logic programming operational semantics and, as constraint system, the Herbrand universe under CET, modified to deal with Skolem constants. So far, our prototype can detect only trivial cases of inconsistency, hence our approach could benefit from the introduction of CLP techniques. This will be briefly addressed in the Conclusions.

3.2 Validating specifications via the SGA

The SGA loads (in a suitable internal representation) a CooML theory $T = \langle \texttt{thAx}, \mathscr{T} \rangle$, a set $\mathscr{G}$ of generation constraints and a list of SG-goals of the form I : `isOf`$(o, C, \underline{e})$, where I is an information term possibly containing variables. Let's look at the `coachCompany` theory with the generation constraints

```
isOf(C,'Coach',[]) :- member(C,[c1,c2]).
isOf(P,'Passenger',[]) :- member(P,[anna,john,ted]).
isOf(T,'Trip',[C]) :- member((T,C), [(t1,c1),(t2,c2),(t3,c1), (t4,c1)]).
numberOfSeats(c1,3).
numberOfSeats(c2,60).
```

A sample SG-goal is:

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

where we use Prolog lists to represent information terms. Since `[3,tt]:seats(C)` has information content `3 = numberOfSeats(C)`, this means that we are looking for the information `Trips:trips(C)` for every coach `C` with 3 seats. More precisely, the SG-goal includes both a generation goal (generate all the coaches `C` with 3 seats that satisfy the generation constraints) and a query (for each `C`, show the information on the trips assigned to it). An answer to the previous goal is:

```
Trips = [ [t1,tt], [t4,tt] ],
C = c1
```

which represents a snapshot where trips `t1` and `t4` have been assigned to coach `c1` (having 3 seats). *SGA* fully generates the snapshot, building information terms for all the classes. An example of complete snapshot, together with part of the related information content and the answer, is

```
[[3,tt], [[t1,tt],[t4,tt]] ] : isOf(c1,'Coach',[])
[2, [[1,tt], [2,[anna,tt]], [3,tt]]] : isOf(t1,'Trip',[c1])
[1, [1,tt]] : isOf(t4,'Trip',[c1])
[2, [[2,c1],tt]] : isOf(anna,'Passenger',t1).
[1, tt] : isOf(ted,'Passenger',t4).
isOf(ted,'Passenger',[t4]), guides(p_1,t4), ...; UNSOLVED: not(ted = p_1)
```

In particular, the unsolved constraint `not(ted = p_1)` comes from the check of the third integrity constraint in Fig.1. The user can ask for the generation of all the (consistent) snapshots and extract information from each of them.

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

Validating Specifications The goal 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 constraints and check if they match the user expectations. To this aim, it's useful to tune the generation constraints to separately consider various aspects that can be understood within a small, "human viable" number of examples, as usual in this context [9]. For instance, 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 taken 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 [22].

Partial and Full Model Checking As traditional in software model checking, here the goal is to show that, under the assumption of the generation constraints, 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 constraints. We may perform full model checking if the set of the generated snapshots is representative of all models of the theory w.r.t. the property under consideration.

3.3 A prototype algorithm

Now we describe a general schema for the Snapshot Generation Algorithm, whereas 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 SG-goals that we call *populations*. The generation process starts from a set P_0 of SG-goals to be solved, i.e. to become grounded. SGA gradually instantiates P_0, possibly generating new goals. It divides the population in two separate sets: ToDo, containing the SG-goals not solved yet, 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 $closed(C, \underline{e})$. Such a predicate is inserted in Closed 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 $\textsc{ToDo} = \emptyset$.
- A state S has *domain* $\mathrm{Dom}(S) = \{\mathtt{isOf}(o, C, \underline{e}) \mid I : \mathtt{isOf}(o, C, \underline{e}) \in \textsc{Done} \cup \textsc{ToDo}\}$.
- $\langle \textsc{Done}_1, \textsc{ToDo}_1, \textsc{Closed}_1, \textsc{Info}_1 \rangle \preceq \langle \textsc{Done}_2, \textsc{ToDo}_2, \textsc{Closed}_2, \textsc{Info}_2 \rangle$ iff:
  1. $\textsc{Done}_1 \subseteq \textsc{Done}_2$, $\mathrm{Dom}(S_1) \subseteq \mathrm{Dom}(S_2)$, $\textsc{Info}_1 \subseteq \textsc{Info}_2$;
  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* $S = \langle \textsc{Done}, \emptyset, \textsc{Closed}, \textsc{Info} \rangle$ 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 $\langle S,\ I : \mathtt{isOf}(o, C, \underline{e}),\ S' \rangle$ s.t.:

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

The listing for a non deterministic SGA based on expansion steps follows:

SGA ($\langle \mathtt{thAx}, \mathscr{T} \rangle$, $\mathscr{G}$, $ToDo_0$)

```
1   Thy = thAx; PDAx = 𝒯 ∪ 𝒢; S = ⟨∅, ToDo_0, ∅, ∅⟩;  UC = {⊤};
2   while ToDo ≠ ∅ do
3       if error(S) fail;
4       else % Generation Step:
5           Choose I : isOf(o, C, e) ∈ ToDo and compute ⟨S, I : isOf(o, C, e), S'⟩;
6           S = S' and UC = update(S, UC);
7   if globalError(S) fail;
8   else return S, UC
```

TODO_0 are the SG-goals to be solved under theory $\langle \texttt{thAx}, \mathscr{T} \rangle$ and generation constraints $\mathscr{G}$. The variable UC stores the unsolved constraints generated by the function calls $\texttt{error}(S)$ and $\texttt{globalError}(S)$. They check consistency against "local" and "global" integrity constraints. The former must be *monotonic*, i.e., $\texttt{error}(S)$ and $S \preceq S'$ entails $\texttt{error}(S')$. The latter applies only to states in solved form. In SnaC, $\texttt{error}(S)$ calls `ff`, obtaining a $\mathscr{U}$; if $\mathscr{U}$ is recognized as satisfiable, then we get inconsistency; otherwise a simplified form of $\mathscr{U}$ is collected. The constraints considered so far are monotonic. A global constraint typically arises using negated `holds`-atoms and `globalff`, for example

```
globalff :- not holds(isOf(_P,'Passenger',[t1])).
```

indicating that trip `t1` must have at least one passenger. It can be used only to discard final solutions, where we assume that all the passengers have been created.

The SGA is a general schema, whose core is the implementation of expansion steps and of the `holds` predicate, required to satisfy the properties in the cases (A) and (B) of Section 3. For lack of space we do not explain further the current implementation of `holds`. At the moment it is rather rough: it detects only the trivial inconsistencies and no simplification is supported. As a result, the content of UC could become difficult to read. Nevertheless, in the experiments carried over so far, one gets useful UC-answers.

To state the adequacy results, we introduce some additional notation, associating class C_j and population P to information terms:

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

where $\{T_{j_1} : \texttt{isOf}(o_{j_1},C_j,\underline{e}_{j_1}),\ldots,T_{j_k} : \texttt{isOf}(o_{j_k},C_j,\underline{e}_{j_k})\}$ is the set of SG-goals of P with class C_j; if no SG-goal with class C_j belongs to P, $\text{ITP}(P,C_j) = [\_\mathit{Tail}]$.

Theorem 4 (Correctness). *Let $S^* = \langle \text{DONE}^*, \emptyset, \text{CLOSED}^*, \text{INFO}^* \rangle$ be a state computed by SGA with theory $T = \langle \texttt{thAx}, \mathscr{T} \rangle$ and generation constraints $\mathscr{G}$, and let $I^* = \text{ITP}(\text{DONE}^*)$ be the information term of the generated population* DONE^*. *Then,* $\text{IC}(I^* : \texttt{thAx}) \cup \mathscr{G} \Rightarrow \neg\exists(UC)$.

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

The proof of soundness requires a correct implementation of the `holds` predicate, while for completeness we need to enforce property 3) of expansion steps.

4 Related work and conclusion

We have presented the semantics of the OO modelling language CooML, a language in the spirit of the UML, but based on a constructive semantics, in particular the BHK explanation of the logical connectives. We have introduced a general 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 a goal in the setting of a CooML specification,

and we have outlined a non-deterministic generation algorithm SGA, showing that finite minimal solutions can be, in principle, generated. One needs a constraint language in order to specify the general properties of the problem domain, as well as the instantiation constraints. In an implementation of SGA, a consistency checking algorithm is assumed, which either establishes the consistency/inconsistency of the current snapshot, or collects a set of unsolved constraints. The relevance of SG for validation and testing in OO software development is widely known. The USE tool [9] for 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 include [5] w.r.t. JML specification. In [2] specification of features models are translated into SAT problems; tentative solutions are propagated with a Truth Maintenance System. If a inconsistency is discovered the TMS explains the causes in view of possible model repair. Related is also [19],where design space specs are seen as tree whose nodes are constrained by OCL statements and BDD's are used to find solutions. Snapshot generation is only one of the aspects of CooML, once we put our software engineering glasses on and see it more generally as a *specification* rather than modeling language [10, 15]. Here we do not have considered, in particular, *methods*. The underlying logic supports a clean notion of correct *query* methods, namely methods 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 our future work.

We 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 prototype, we did not seek any serious attempt to simplify the unsolved constraints. This could be partially solved by the introduction of CLP, in particular over finite domains. More in general, it is desirable to understand the connections between Theorem 3 and the notion of satisfaction-completeness in constraint systems. Another direction comes from the relation between CooML's approach to incomplete information and answer set programming [1, 21], in particular disjunctive LP [16, 25]. A naive extension of SGA to this case would lead to inefficient solutions, yet the literature offers several ways constraints and ASP can interact [6, 17, 20]. We may explore the possibility of combining snapshot generation with SAT provers, to which we may pass ground unsolved constraints when global consistency is checked. There is also the more general issue of the relationships between information terms and stable models, in particular partial stable models [24]), in the context partial of logics [3, 7].

More information about the project can be found at `http://cooml.dsi.unimi.it`, while `http://cooml.dsi.unimi.it/iclp.html` contains additional material pertaining to the present paper.

References

1. C. Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. CUP, 2003.
2. D. S. Batory. Feature models, grammars, and propositional formulas. In J. H. Obbink and K. Pohl, editors, *SPLC*, volume 3714 of *LNCS*, pages 7–20. Springer, 2005.

3. S. Blamey. Partial Logic. In D. M. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic Volume 3: Alternatives To Classical Logic*, pages 1–70. D. Reidel Pub., 1986.
4. A. Boronat, J. Oriente, A. Gómez, I. Ramos, and J. A. Carsí. An algebraic specification of generic OCL queries within the Eclipse modeling framework. In A. Rensink and J. Warmer, editors, *ECMDA-FA*, volume 4066 of *LNCS*, pages 316–330. Springer, 2006.
5. F. Bouquet, F. Dadeau, B. Legeard, and M. Utting. JML-testing-tools: A symbolic animator for JML specifications using CLP. In N. Halbwachs and L. D. Zuck, editors, *TACAS*, volume 3440 of *Lecture Notes in Computer Science*, pages 551–556. Springer, 2005.
6. F. Buccafurri, N. Leone, and P. Rullo. Strong and weak constraints in disjunctive datalog. In J. Dix, U. Furbach, and A. Nerode, editors, *LPNMR*, volume 1265 of *Lecture Notes in Computer Science*, pages 2–17. Springer, 1997.
7. M. Fitting. A deterministic Prolog fixpoint semantics. *J. Log. Program.*, 2(2):111–118, 1985.
8. T. Fruewirth and S. Abdennadher. *Essentials of Constraint Programming*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003.
9. M. Gogolla, J. Bohling, and M. Richters. Validating UML and OCL models in USE by automatic snapshot generation. *Software and System Modeling*, 4(4):386–398, 2005.
10. J. V. Guttag and J. J. Horning. *Larch: languages and tools for formal specification*. Springer-Verlag New York, Inc., New York, NY, USA, 1993.
11. D. Jackson and J. Wing. Lightweight formal method. *IEEE Computer*, April 1996.
12. J. Jaffar and M. J. Maher. Constraint logic programming: A survey. *J. Log. Program.*, 19/20:503–581, 1994.
13. C. Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*. Prentice Hall, Upper Saddle River, NJ, 2004.
14. K.-K. Lau, M. Ornaghi, and S.-Å. Tärnlund. Steadfast logic programs. *J. Log. Program.*, 38(3):259–294, 1999.
15. G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: a behavioral interface specification language for Java. *SIGSOFT Softw. Eng. Notes*, 31(3):1–38, 2006.
16. N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. The dlv system for knowledge representation and reasoning. *ACM Trans. Comput. Log.*, 7(3):499–562, 2006.
17. V. W. Marek, I. Niemelä, and M. Truszczynski. Logic programs with monotone cardinality atoms. In V. Lifschitz and al., editors, *LPNMR*, volume 2923 of *LNCS*, pages 154–166. Springer, 2004.
18. P. Miglioli, U. Moscato, M. Ornaghi, and G. Usberti. A constructivism based on classical truth. *Notre Dame Journal of Formal Logic*, 30(1):67–90, 1989.
19. S. Neema, J. Sztipanovits, G. Karsai, and K. Butts. Constraint-based design-space exploration and model synthesis. In R. Alur and I. Lee, editors, *EMSOFT*, volume 2855 of *Lecture Notes in Computer Science*, pages 290–305. Springer, 2003.
20. I. Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Ann. Math. Artif. Intell.*, 25(3-4):241–273, 1999.
21. I. Niemelä and P. Simons. Smodels - an implementation of the stable model and well-founded semantics for normal LP. In *LPNMR*, pages 421–430, 1997.
22. J. Offutt and A. Abdurazik. Generating tests from UML specifications. In R. France and B. Rumpe, editors, *Proc. of UML'99*, volume 1723 of *LNCS*, pages 416–429. Springer, 1999.
23. M. Ornaghi, M. Benini, M. Ferrari, C. Fiorentini, and A. Momigliano. A constructive object oriented modeling language for information systems. *ENTCS*, 153(1):67–90, 2006.
24. T. C. Przymusinski. Well-founded and stationary models of logic programs. *Ann. Math. Artif. Intell.*, 12(3-4):141–187, 1994.
25. F. Ricca, N. Leone, V. D. Bonis, T. Dell'Armi, S. Galizia, and G. Grasso. A dlp system with object-oriented features. In C. Baral, G. Greco, N. Leone, and G. Terracina, editors, *LPNMR*, volume 3662 of *Lecture Notes in Computer Science*, pages 432–436. Springer, 2005.
26. A. S. Troelstra. From constructivism to computer science. *TCS*, 211(1-2):233–252, 1999.
27. J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modelling with UML*. Object Technology Series. Addison-Wesley, Reading/MA, 1999.