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 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 (`http://cooml.dsi.unimi.it`), a specification language for OO systems [21]. Similarly to UML/OCL [25], 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 (PD), which may involve loosely or incompletely defined components. This encourages the selection of the appropriate level of abstraction as far as specifications are concerned.

CooML follows the spirit of lightweight formal methods [11]: it does not focus on full formalization, nor on whole system correctness, but emphasizes partiality in analysis and specification. In the context of OO modeling, both the *validation* of a specification and 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 specifications in JML based on symbolic animation [5].

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 [24]). 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 characterized by the way classical and constructive information co-exist, the main “entry” point being the different way an *atomic* formula A is given evidence (for more details we kindly refer the reader to the original formulation of the logic in [16]). If we call a *piece of information* the pair $I : P$, where P is a formula and I is its information term, then $I : P$ is a particular piece of information, which may be *true* or *false* in a classical interpretation w that we call 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 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 other information systems such as relational databases.

Crucially, the constructive side of the logic allows the *identification of snapshots with information terms*, thusly providing a formal counterpart to the intuitive notion of object populations. We argue that CooML’s proof theoretic snapshot generation may be advantageous w.r.t. 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 and this may have a cascade of benefits in terms of efficiency of the representation.

The contribution of this paper is twofold. First, we apply the semantics developed in purely logical terms in [16] to object oriented modeling languages. We model 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 that it is related to snapshot consistency. Secondly, we describe (and implement) a snapshot generation algorithm (SG); the latter takes as inputs a CooML theory T, axiomatizing a set of classes in a problem domain PD and the user’s generation requirements $\mathcal{G}$, which serve an analogous purpose of domain predicates in the grounding phase of ASP’s [19]. As snapshots should be consistent with respect to PD and $\mathcal{G}$, we prove that consistency checking is sound (this is not too faraway from adequacy results in the theory of CLP’s) and that SG is complete (i.e., if a consistent snapshot satisfying the generation requirements exists, it will be generated).

In the rest of the paper Section 2 introduces CooML theories first by example and then formally, while Section 3 describes the SG algorithm and the theory behind it. We conclude in Section 4 with some very preliminary connections with other modeling languages and model generation tools.

2 CooML specifications

In this section we informally present the language via an example adapted from [4], while we defer to Section 2.1 the formal treatment. 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` in Fig. 1.

```
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(T,'Trip',[C]), nobooking(P,T), booked(P,_Seat,C,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

To explain our example we need to introduce CooML types. We distinguish among *data types* (in our example, `Integer` and `Boolean`), *PD types* (`Person`), and *object types* (`Coach`, `Trip`, `Passenger`). They 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 existence of an implementation that evaluates ground terms into 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 informal *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 data types are assumed to be externally implemented, PD types are defined in the `pds` (Problem Domain Specification) section and 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 [12], 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 representing general problem domain properties that are not interpreted by CooML, but that could be interpreted 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)` cannot hold if person `P` has booked a seat in 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 $Pty_C($`this`$,\underline{\mathbf{e}})$, 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 subformulas. For example, the label `seats` is used in the `getSeats` method to get `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`$\{\underline{\tau}\ \underline{x}; G \rightarrow P\}$, `T`$\{P^{ext}\}$, where P^{ext} is a property which 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

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

of P. Each piece of information $I : P$ for P has an *information content*, a set of simple properties intuitively representing the minimum amount of information needed to justify P according to I. In fact, we call *simple property* an atomic formula of the form `T`$\{P^{ext}\}$. A simple property S represents a basic information unit, i.e., it has an unique information term `tt` where `tt` is a constant. This means that the only information we have is the *truth* of S, and 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` (`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 which makes $P(x)$ true.

Universal. $((\mathbf{x}_1, I_1), \ldots, (\mathbf{x}_n, I_n))$: `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 $I_1 : P(\mathbf{x}_1)$, $\ldots$, $I_n : P(\mathbf{x}_n)$ 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 are

```
((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. $(\mathtt{k}, I_k)$: `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 predicate `john is Passenger(t1)` contains the information term `(1,tt)`, then

[2] But one can change this. We do not discuss this issue here for lack of space.

[3] In this paper the precise syntax of generators is omitted.

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

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 `C(`$\underline{e}$`)` 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 $I : \texttt{thAx}$ 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:

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

and possible information terms I_1, I_2, I_3 are

$$I_1 = \texttt{((mini,CoachInfo)),}\quad I_2 = \texttt{(([john,t1],(1,tt)))}$$
$$I_3 = \texttt{(([t1,mini],(2,((1,tt), (2,(john,tt)),(3,tt),(4,tt))))}$$
$$\texttt{([t2,mini],(1,(1,tt))),}$$
$$\texttt{([t3,mini],(1,(2,tt))))}$$

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

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

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 shown, while for `guides` we have only partial knowledge, expressed by the `T`-properties, and nothing is said about `Person`. The relationship with classical models can be better explained by comparing the constructive and classical reading of CooML properties. 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 $\texttt{IC}(I : \texttt{thAx})$ has a reachable model, for at least one piece of information $I : \texttt{thAx}$. Furthermore, one can prove that $\texttt{IC}(I : \texttt{thAx})$ 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 $\mathscr{T}$, 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 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 and `isOf(t1,'Trip',[mini])` follows from the third domain predicate in Fig 2. In Section 3 we will introduce consistent snapshot generation and we will show how it is a way of understanding and validating CooML specifications.

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 $\underline{x}$:

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

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

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

The information content $\textsc{ic}(I : P)$ represents the minimum amount of information needed to get evidence for P according to I. This can be formalized as follows.

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:*

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

The information content $\textsc{ic}(I : P)$ represents an information about the current world state. We equiparate the information content of $\mathscr{C}$ to its closure's 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 the following properties to hold:

$$\mathscr{C} \subseteq \mathscr{C}^* \tag{1}$$
$$\mathscr{C}_1 \subseteq \mathscr{C}_2^* \text{ implies } \mathscr{C}_1 \sqsubseteq \mathscr{C}_2 \tag{2}$$

Using the above properties, we can prove:

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

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

This establishes the minimality of $\textsc{ic}(I : P)$ with respect to $\sqsubseteq$.

Now we can 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 $I : \mathtt{thAx}$.

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 consistent with respect to the constraints $\mathscr{T}$ ($\mathscr{T}$-*consistent*) iff there exists a reachable model of* $\textsc{ic}(I : \mathtt{thAx}) \cup \mathscr{T}$.

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

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

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

3 A snapshots generation algorithm and its theory

A Snapshot Generation algorithm (SG) for a CooML theory $T = \langle \mathtt{thAx}, \mathscr{T} \rangle$ 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, 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 based on which sound and complete implementations can be developed.

3.1 Consistency check

To recognize inconsistent attempts, *SG* uses an internal representation of the information content of the current generation state *S*, denoted by INFO_S.

Here we briefly discuss a simplified version of consistency check in our Prolog implementation, called SnaC. 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 of ground facts, or of clauses of the form `H :- eq(t1,t2)`, or `false :-` *Body*, where:

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

Clauses with head `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, we interpret `eq` as equality and `false` as falsehood).

As an example, let us consider the SnaC representation P_{cComp} (Fig. 3) of the information content of the `coachCompany` package (Fig. 2). The facts and the constraints in the first two rows come from the translation of domain properties. For example, the first row contains the translation of `dom{x; x is Coach(), [mini])}`. The other facts come from the translation of atoms. The clause `guides(X,t2):- eq(X,p0)` translates `T{exi{Person p; guides(p,t2)}}`, where `p0` is a fresh Skolem constant. Finally, `false :- guides(P,t3)` translates `T{not exi{Person p; guides(p,t3)}}`.

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

Fig. 3: The SnaC representation P_{cComp}.

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

(a) `false` finitely fails for 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 `coachCompany`, by the properties of the translation.

(b) If we add the constraint

```
c1)   false :- vacant(S,C,T),booked(_P,S,C,T).
```

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

(c) If we add the constraint

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

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

The above discussion is reflected in the following theorem:

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

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

In the first case, SnaC accepts $I : \texttt{thAx}$ as a $\mathscr{T}$-consistent snapshot. In the second one, if $\mathscr{U}$ is empty we get inconsistency. If $\mathscr{U}$ is not empty, it is given as an answer. We omit the proof, since it is somewhat implicit in the above discussion.

A result similar to the above theorem can be obtained admitting a larger class of simple properties and PD constraints, using techniques similar to those used in CLP, when one sees a CLP system as constituted by a *constraint 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.

3.2 Validating specifications via the 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*, to reduce the number of the 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 how an *answer set* program is constructed to direct grounding.

The number of the 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 axiomatization. For example:

```
choice(isOf(C,'Coach',[])) :- member(C,[c1,c2]).
choice(isOf(P,'Passenger',[])) :- 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 $A \rightarrow \mathtt{choice}(A)$, which, together with the user's definition of `choice`, sets up to 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's generation requirements, it can be queried with generation goals. A sample goal is:

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

where Prolog lists represent information terms. 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])`

Although SG fully generates the snapshot, building information terms for all the classes in the package, we omit it for conciseness. If the user asks for more solutions, all the possible snapshots will be shown. In the above example, there are two more solutions, where `c1` has no assigned trip and `c1` has two.

We now sketch some ways in which the SG 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 requirements and check if they match the user expectations. To this aim, it's useful to tune the generation requirements 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 [20].

Partial and Full Model Checking As traditional in software model checking, here the goal is to show that, under the assumption of the generation requirements, no snapshot satisfies an undesired property. This is obtained if the SG 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 prototype algorithm

Now we describe a general schema for the Snapshot Generation Algorithm, whereas SnaC is just a first rough implementation. Let $T = \langle \texttt{thAx}, \mathscr{T} \rangle$ be a CooML theory, where $\texttt{thAx} = \texttt{and}\{\texttt{clAx}(C_1), \ldots, \texttt{clAx}(C_n)\}$. 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 grounded. SG gradually instantiates P_0, possibly generating new goals. It divides the population in two separate sets: TODO, containing the G-goals not solved yet, DONE, containing the solved ones. A *generation state* has the form $S = \langle \text{DONE}, \text{TODO}, \text{CLOSED}, \text{INFO} \rangle$, 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 $I : \texttt{isOf}(o, C, \underline{e}) \in \text{DONE}$, $\text{IC}(I : Pty_C(o, \underline{e})) \subseteq \text{INFO}$.

The following definitions are in order:

- A state S is *in solved form* if $\text{TODO} = \emptyset$.
- A state S has *domain* $\text{Dom}(S) = \{\texttt{isOf}(o, C, \underline{e}) \mid I : \texttt{isOf}(o, C, \underline{e}) \in \text{DONE} \cup \text{TODO}\}$.
- $\langle \text{DONE}_1, \text{TODO}_1, \text{CLOSED}_1, \text{INFO}_1 \rangle \preceq \langle \text{DONE}_2, \text{TODO}_2, \text{CLOSED}_2, \text{INFO}_2 \rangle$ iff:
  1. $\text{DONE}_1 \subseteq \text{DONE}_2$, $\text{Dom}(S_1) \subseteq \text{Dom}(S_2)$, $\text{INFO}_1 \subseteq \text{INFO}_2$;
  2. If $closed(C, \underline{e}) \in \text{CLOSED}_1$, then $\texttt{isOf}(o, C, \underline{e}) \in \text{Dom}(S_1)$ iff $\texttt{isOf}(o, C, \underline{e}) \in \text{Dom}(S_2)$.

The SGA starts from initial state $S_0 = \langle \emptyset, \text{TODO}_0, \emptyset, \emptyset \rangle$ and yields a *solution* $S = \langle \text{DONE}, \emptyset, \text{CLOSED}, \text{INFO} \rangle$ such that $S_0 \preceq S$; since $\text{TODO} = \emptyset$, for every $I : \texttt{isOf}(o, C, \underline{e}) \in \text{TODO}_0$, DONE contains a ground information term $(I : \texttt{isOf}(o, C, \underline{e}))\sigma$ 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 $\langle S,\ I : \texttt{isOf}(o, C, \underline{e}),\ S' \rangle$ s.t.:

p1. $I : \texttt{isOf}(o, C, \underline{e}) \in \text{TODO}$ (the selected goal);
p2. $(I : \texttt{isOf}(o, C, \underline{e}))\sigma \in \text{DONE}'$ and $I : \texttt{isOf}(o, C, \underline{e}) \notin \text{TODO}'$ (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 listing for a non deterministic SGA based on expansion steps is next.

```
SG ($\langle \texttt{thAx}, \mathscr{T} \rangle$, $\mathscr{G}$, $ToDo_0$)
1  $Thy = \texttt{thAx}$; $PDAx = \mathscr{T} \cup \mathscr{G}$; $S = \langle \emptyset, ToDo_0, \emptyset, \emptyset \rangle$;  $UC = \emptyset$;
2  **while** $ToDo \neq \emptyset$ **do**
3      **if** error(S) **fail**;
4      **else** % *Generation Step:*
5          Choose $I : \texttt{isOf}(o, C, \underline{e}) \in ToDo$ and compute $\langle S,\ I : \texttt{isOf}(o, C, \underline{e}),\ S' \rangle$;
6          $S = S'$;
7  **if** globalError(S) **fail**;
8  **else return** $S$, $UC$
```

$\textsc{ToDo}_0$ are the G-goals to be solved under theory $\langle \texttt{thAx}, \mathscr{T} \rangle$ and generation requirements $\mathscr{G}$. The variable UC stores the "unsolved constraints" generated by the "error tests" $\texttt{error}(S)$ and $\texttt{globalError}(S)$. They check consistency against "local" and "global" integrity constraints. The `error` must be *monotonic*, i.e., $\texttt{error}(S)$ and $S \preceq S'$ entails $\texttt{error}(S')$; `globalError` applies only to states in solved form.

The SGA is a general schema, whose core is the implementation of expansion steps and error predicates. The latter use the integrity constraints `false :- ...` to detect inconsistency and are based on a generalization of the ideas presented in Section 3.1. A call to an error predicate either returns "true" when inconsistency is detected, or updates UC and returns "false". When a state S in solved form is reached, SG returns UC as an answer; if it is empty, S is consistent. The current implementation could be improved, namely in detecting more than the trivial inconsistencies and no simplification is supported.

To state the adequacy results, we introduce some additional notation (ITP), which associates a class C_j and population P their information terms:

$$\begin{aligned} \textsc{itp}(P, C_j) &= [\,[[o_{j_1}\,|\,\underline{e}_{j_1}], T_{j_1}], \dots, [[o_{j_k}|\underline{e}_{j_k}], T_{j_k}] \mid \_Tail] \\ \textsc{itp}(P) &= [\,\textsc{itp}(P, C_1), \dots, \textsc{itp}(P, C_n)\,] \end{aligned}$$

where $\{T_{j_1} : \texttt{isOf}(o_{j_1}, C_j, \underline{e}_{j_1}), \dots, T_{j_k} : \texttt{isOf}(o_{j_k}, C_j, \underline{e}_{j_k})\}$ is the set of G-goals of P with class C_j; if no G-goal with class C_j belongs to P, then $\textsc{itp}(P, C_j) = [\_Tail]$.

Theorem 4 (Correctness). *Let* $S^* = \langle \textsc{Done}^*, \emptyset, \textsc{Closed}^*, \textsc{Info}^* \rangle$ *be a state computed by SG with theory* $T = \langle \texttt{thAx}, \mathscr{T} \rangle$ *and generation requirements* $\mathscr{G}$*, and let* $I^* = \textsc{itp}(\textsc{Done}^*)$ *be the information term of the generated population* $\textsc{Done}^*$*. 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 easily follows assuming that for every state S: (i) $\textsc{Info}_S$ satisfies $\mathscr{G}$, by the way SG performs grounding; (ii) when $\texttt{error}(S)$ or $\texttt{globalError}(S)$ returns "true", then $\textsc{Info}_S$ is inconsistent with respect to $\mathscr{T}$; (iii) when $\texttt{globalError}(S)$ returns "false", then UC is empty and $\textsc{Info}_S \cup \mathscr{T}$ is consistent, or $\textsc{Info}_S \cup \mathscr{T} \cup UC$ is inconsistent.

Theorem 5 (Completeness). *Let* $S_0 = \langle \emptyset, \textsc{ToDo}_0, \emptyset, \emptyset \rangle$ *be an initial state of SG with theory T and generation requirements* $\mathscr{G}$*. If there is a state* $S = \langle \textsc{Done}, \emptyset, \textsc{Closed}, \textsc{Info} \rangle$ *such that* $S_0 \preceq S$*, then there is a computation of SG reaching a state* S^* *in solved form such that* $S_0 \preceq S^* \preceq S$*.*

The proof of Theorem 5 follows from the above properties p1, p2, p3.

4 Related work and conclusion

We have presented the semantics of the object oriented modelling language CooML, a language in the spirit of the UML, but based on a constructive semantics, in particular the BHK explanation of logical connectives. 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 a goal in the setting of a CooML specification, and we have outlined a non-deterministic generation algorithm, 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 generation requirements. In an implementation of the 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 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] the specification of features models are translated into SAT problems; tentative solutions are then propagated with a Truth Maintenance System. Related work is also [17], where design space specs are seen as trees whose nodes are constrained by OCL statements and BDD's 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 [10,13]. Here we have not considered *methods*, although 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 current prototype, there is little emphasis on the simplification of unsolved constraints. This could be partially ameliorated 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 [8]. Another direction comes from the relation between CooML's approach to incomplete information and answer set programming [1, 19], in particular disjunctive LP [14,23]. A naive extension of the SGA to this case would lead to inefficient solutions, yet the literature offers several ways constraints and ASP can interact [6, 15, 18]. 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 [22], in the context of partial logics [3, 7].

More information about the project can be found at `http://cooml.dsi.unimi.it`, while `http://cooml.dsi.unimi.it/sgcooml.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. Partial models and logic programming. *Theor. Comput. Sci.*, 48(3):229–255, 1986.
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. C. Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*. Prentice Hall, Upper Saddle River, NJ, 2004.
13. 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.
14. 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.
15. 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.
16. 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.
17. 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.
18. I. Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Ann. Math. Artif. Intell.*, 25(3-4):241–273, 1999.
19. 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.
20. 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.
21. 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.
22. T. C. Przymusinski. Well-founded and stationary models of logic programs. *Ann. Math. Artif. Intell.*, 12(3-4):141–187, 1994.
23. 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.
24. A. S. Troelstra. From constructivism to computer science. *TCS*, 211(1-2):233–252, 1999.

25. J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modelling with UML.* Object Technology Series. Addison-Wesley, Reading/MA, 1999.

A Proofs of the theorems

In this appendix we prove the theorems of the paper *Snapshot generation in a constructive object-oriented modeling language*, M. Ferrari, C. Fiorentini, A. Momigliano and M. Ornaghi, submitted to LOPSTR 2007.

A.1 Proofs of the theorems in Section 2.1

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 $\mathrm{IC}(I : P) \sqsubseteq \mathscr{C}$.*

Proof. Point 1) can be easily proved by induction on P.
We prove Point 2) by using properties (1) and (2) of $\mathscr{C}^*$. By Property (2), it suffices to show that:

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

We prove (*) by induction on P. The base case $I : P = \mathtt{tt} : P$ follows by Property (1).
Let us assume $\mathscr{C} \rhd (I_1,\dots,I_n) : \mathtt{and}\{P_1 \dots P_n\}$. Since $\mathscr{C} \rhd I_j : P_j$ for all $j \in 1..n$, by induction hypothesis $\mathrm{IC}(I_j : P_j) \subseteq \mathscr{C}^*$ for all $j \in 1..n$, hence $\bigcup_{1\le j\le n} \mathrm{IC}(I_j : P_j) \subseteq \mathscr{C}^*$, that is $\mathrm{IC}(\,(I_1,\dots,I_n) : \mathtt{and}\{P_1 \dots P_n\}) \subseteq \mathscr{C}^*$.
If $\mathscr{C} \rhd (k,I) : \mathtt{case}\{P_1 \dots P_n\}$ then $\mathscr{C} \rhd I : P_k$ and, by induction hypothesis, we have $\mathrm{IC}(I : P_k) \subseteq \mathscr{C}^*$, hence $\mathrm{IC}(\,(k,I) : \mathtt{case}\{P_1 \dots P_n\}) \subseteq \mathscr{C}^*$.
The remaining cases are similar.

Lemma 1. *Let $\mathtt{w}$ be a reachable model state and P a CooML property. Then $\mathtt{w} \models P$ iff there is a piece of information $I : P$ such that $\mathtt{w} \models \mathrm{IC}(I : P)$.*

Proof. By induction on the structure of P. Here we prove the "only if" part (the "if" statement is similar). The where P is a simple property is trivial.
Assume that $\mathtt{w} \models \mathtt{and}\{P_1 \dots P_n\}$. Since $\mathtt{and}$ is interpreted as $\wedge$, we get $w \models P_j$, for all $j \in 1..n$. By induction hypothesis, for every $j \in 1..n$ there is a piece of information $I_j : P_j$ such that $w \models \mathrm{IC}(I_j : P_j)$. Thus, $w \models \bigcup_{1\le j\le n} \mathrm{IC}(I_j : P_j)$, namely $w \models \mathrm{IC}(\,(I_1,\dots,I_n) : \mathtt{and}\{P_1 \dots P_n\})$.
Suppose $\mathtt{w} \models \mathtt{case}\{P_1 \dots P_n\}$. Since $\mathtt{case}$ corresponds to $\vee$, there is $k \in 1..n$ such that $\mathtt{w} \models P_k$; by induction hypothesis, there is a piece of information $I : P_k$ such that $\mathtt{w} \models \mathrm{IC}(I : P_k)$, hence $\mathtt{w} \models \mathrm{IC}(\,(k,I) : \mathtt{case}\{P_1 \dots P_n\})$.
Let $\mathtt{w} \models \mathtt{exi}\{\underline{\tau}\ \underline{x};\ P(\underline{x})\}$. Since $\mathtt{exi}$ is interpreted as $\exists$ and $\mathtt{w}$ is reachable, there is a tuple of terms $\underline{\mathbf{x}}$ such that $\mathtt{w} \models P(\underline{\mathbf{x}})$. By the induction hypothesis, there exists a piece of information $I : P$ such that $\mathtt{w} \models \mathrm{IC}(I : P(\underline{\mathbf{x}}))$, and this implies $\mathtt{w} \models \mathrm{IC}(\,(\underline{\mathbf{x}},I) : \mathtt{exi}\{\underline{\tau}\ \underline{x};\ P(\underline{x})\})$.
Let $\mathtt{w} \models \mathtt{for}\{\underline{\tau}\ \underline{x}; G(\underline{x}) \rightarrow P(\underline{x})\}$. By definition of G, there is a list of terms $L = [\underline{\mathbf{x}}_1, \dots, \underline{\mathbf{x}}_n]$ such that $\mathtt{w} \models \underline{x}$ iff $\mathtt{w} \models \mathit{member}(\underline{x}, L)$, hence $\mathtt{w} \models \mathtt{dom}(\underline{x}; G(\underline{x}); L)$. Moreover, for all $j = 1..n$, since $\mathtt{w} \models G(\underline{\mathbf{x}}_j)$, we have $w \models P(\underline{\mathbf{x}}_j)$. By the induction hypothesis, for every $j = 1..n$ there exists a piece of information $I_j : P_j$ such that $w \models \mathrm{IC}(I_j : P(\mathbf{x}_j))$.
We set

$$I : P \;=\; (\,(\underline{\mathbf{x}}_1, I_1), \dots, (\underline{\mathbf{x}}_n, I_n))\ :\ \mathtt{for}\{\underline{\tau}\ \underline{x}; G(\underline{x}) \rightarrow P(\underline{x})\}$$

and the assertion is proved.

By the previous lemma we get:

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

A.2 Proof of Theorem 3 in Section 3.1

Let us indicate by M the meta-interpreter assumed in Section 3.1. In the simplified version considered in this paper, we have:

1. In a theory $T = \langle \mathtt{thAx}, \mathscr{T} \rangle$, the PD constraints $\mathscr{T}$ are directly expressed in the form of logic program clauses.
2. A reachable model of $\mathrm{IC}(I : \mathtt{thAx})$ is also a reachable model of the SnaC representation of $\mathrm{IC}(I : \mathtt{thAx})$, where reachability is with respect to the signature of the CooML theory $T = \langle \mathtt{thAx}, \mathscr{T} \rangle$ enriched by a denumerable set of Skolem constants.
3. A model of the SnaC representation that interprets `false` as falsehood and `eq` as the identity [4] is a model of $\mathrm{IC}(I : \mathtt{thAx})$.
4. Skolem constants may occurr only within `eq`-atoms and the latter may occurr only in the body of clauses of the form H `:- eq(`t,t'`)`.
5. When M selects an `eq`-atom E, it "solves" E just by collecting it. For the other selected atoms, M behaves in the standard way.
6. By 4 and 5, a goal G succeeds for a SnaC program P with M and a (possibly empty) set of collected equations $\mathscr{U}$ if and only if it succeeds for the standard logic program $P \cup \{$`eq(_, _)`$\}$. Moreover, for every computed a.s. σ of G for P with M and collected equations $\mathscr{U}$, $G\sigma$ suceeds for the standard program $P \cup \mathscr{U}$.

We can prove our theorem.

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

1. *If the goal `false` finitely fails from P with meta-interpreter M, then $I : \mathtt{thAx}$ is $\mathscr{T}$-consistent.*
2. *If the goal `false` succeeds from P with M collecting a set of constraints $\mathscr{U}$, then $I : \mathtt{thAx}$ is not consistent with respect to $\mathscr{T} \cup \mathscr{U}$.*

Proof. 1. Since `false` fails for P with M, then it fails for $P \cup \{$`eq(X,X)`$\}$. Thus the minimum Herbrand model H_P of $P \cup \{$`eq(X,X)`$\}$ is a reachable model of P that interprets `eq` as the equality and `false` as falsehood. By the porperties of the translation, H_P is a model of $\mathrm{IC}(I : \mathtt{thAx})$, i.e., it is a reachable model of $\mathscr{T}$ that satisfies $\mathrm{IC}(I : \mathtt{thAx})$, and we get $\mathscr{T}$-consistency.
2. In this case, `false` belongs to the minimum Herbrand model of $P \cup \mathscr{U}$. Thus it belongs to all the reachable models of $P \cup \mathscr{U}$. Since the models of $\mathscr{T} \cup \mathrm{IC}(I : \mathtt{thAx})$ inerprets `false` as falsehood, by 2 no reachable model of $\mathscr{T} \cup \mathrm{IC}(I : \mathtt{thAx})$ exists.

In the non-simplified version, Skolem constants are treated under the control of a suitable meta-predicate `holds`, which stores in the generation state of SnaC the "unsolved" equations. Furthermore, some immediate simplifications are applied when possible.

[4] The standard first order logic identity, we do not assume Clark's Equality Theory for Skolem constants.

A.3 Proofs of the theorems in Section 3.3

The proof of Theorem 4 is almost immediate, assuming (i), (ii), (iii) in Section 3.3. Here we outline an implementation of the error predicates which satisfies (ii), (iii) and uses the simplified version of consistency checking considered in Section 3.1. Following the ideas of Section 3.1, we can implement a `check_false(S,U)` predicate invoking the `false` head of the integrity constraints occurring in $\textsc{Info}_S \cup \mathscr{T}$ [5] and returning the collected equations as answer substitutions. More precisely, `U = `$\mathscr{U}$ is an answer substitution of `check_false(S,U)` iff the goal `false` succeeds for $\textsc{Info}_S \cup \mathscr{T}$ with M and collected $\mathscr{U}$ (see Section A.2). The `globalError(`S`)` predicate calls `check_false(`S`,U)` (considering the integrity constraints in the PD or in $\textsc{Info}_S$) to compute the answers `U` $= \mathscr{U}_1$, ..., `U` $= \mathscr{U}_n$. If one of them is empty, `globalError(`S`)` exits returning "true", otherwise it stores the answers in UC and exits returning "false". The predicate `error(`S`)` behaves in the same way. The only difference is that it checks only the monotonic integrity constraints. The logical reading of a non-empty $UC = \{\mathscr{U}_1, \ldots, \mathscr{U}_n\}$ is the formula $(\exists \underline{c}_1\, \mathscr{U}_1) \vee \cdots \vee (\exists \underline{c}_n\, \mathscr{U}_n)$, where $\underline{c}_j$ are the Skolem constants occurring in $\mathscr{U}_j$. We will indicate by UC^L the logical reading of UC.

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

Proof. It sufficies to prove that the implementation outlined above satisfies (i), (ii), (iii). Concerning (i), we still assume that the grounding procedure of SG is implemented in such a way that the generation constraints are satisfied in every generated state. Concerning (ii), it can be proved as follows. If `globalError(`S`)` (`error(`S`)`) exits with "true", then there is an answer `U = `$\mathscr{U}$ with $\mathscr{U} = \emptyset$; by 2. of Theorem 3, we get that $\textsc{Info}_S$ is inconsistent with respect to $\mathscr{T}$ (for `globalError`) or the monotonic part of $\mathscr{T}$ (for `error`). Concerning (iii), we have two cases. Case 1: UC is empty. We have to prove that $\textsc{Info}_S$ is consistent with respect to $\mathscr{T}$; this follows from 1 of Theorem 3 and from the fact that UC is empty iff `check_false(`S`,U)` finitely fails. Case 2: $UC = \{\mathscr{U}_1, \ldots, \mathscr{U}_n\}$. We have to prove that $\textsc{Info}_S \cup \mathscr{T} \cup UC^L$ is inconsistent; this follows from the fact that $\textsc{Info}_S \cup \mathscr{T} \cup \mathscr{U}_j$ is inconsistent for $1 \leq j \leq n$ (by 2. of Theorem 3) and from the fact that we always choose a fresh name when a Skolem constant is introduced.

Theorem 5 (Completeness). *Let $S_0 = \langle \emptyset, \textsc{ToDo}_0, \emptyset, \emptyset \rangle$ be an initial state of SG with theory T and generation requirements $\mathscr{G}$. If there is a state $S = \langle \textsc{Done}, \emptyset, \textsc{Closed}, \textsc{Info} \rangle$ such that $S_0 \preceq S$, then there is a computation of SG reaching a state S^* in solved form such that $S_0 \preceq S^* \preceq S$.*

Proof. We show that, starting from the state S_0, we can compute a sequence of states $S_k = \langle \textsc{Done}_k, \textsc{ToDo}_k, \textsc{Closed}_k, \textsc{Info}_k \rangle$, with $0 \leq k \leq n$, such that:

(i). $S_0 \preceq S_k \preceq S$ for every $0 \leq k \leq n$;

[5] $\mathscr{T}$ are the (clauses in) the PD constraints.

(ii). $\textsc{Done}_k \subset \textsc{Done}_{k+1} \subseteq \textsc{Done}$ for every $0 \leq k \leq n-1$.

(iii). $\textsc{ToDo}_n = \emptyset$;

Note that (ii) guarantees the finiteness of the computation. Suppose that the states $S_0, \ldots, S_k$ have already been defined. If $\textsc{ToDo}_k = \emptyset$, we set $n = k$ and the computation halts. Otherwise, let $I : \texttt{isOf}(o, C, \underline{e}) \in \textsc{ToDo}_k$ and let S_{k+1} be the state such that $\langle S_k,\ I : \texttt{isOf}(o, C, \underline{e}),\ S_{k+1} \rangle$ is an expansion step of SG. By properties (2) and (3) in the definition of expansion step, (i) and (ii) follow. We take $S^* = S_n$ and the theorem is proved.