Snapshots Generation via Constructive Logic
- Extended Abstract -

Mario Ornaghi, Camillo Fiorentini and Alberto Momigliano

Dipartimento di Scienze dell’Informazione
Università degli Studi di Milano
{ornaghi,fiorenti,momiglia}@dsi.unimi.it

1 Introduction

A software information system $S$ allows users to store, retrieve and process information about the external world, typically a data base. We can differentiate two separate aspects in the data elaborated by $S$: the first concerns data types, while the second is related to the information on the external “real world” carried by the data. More precisely, a data type is a set of data together with the associated manipulations where the focus is on operations. In contrast, the other information carried by the data stored in $S$ is strongly related to their meaning in the real world. The need for properly treating data according to their meaning is becoming increasingly important, due to the wide quantity of information that is exchanged on the Internet [10].

In this context, we are developing COOML [9] (Constructive Object Oriented Modeling Language), an OO specification language where the focus is on the information carried by data. The main features of COOML are: a data model based on a predicative extension of the constructive intermediate logic $E^*$ [8], based on the BHK interpretation of the logical connectives [11]; a multi-layered structure, namely with one for the COOML logic layer, others for the problem logic and the computations; the possibility of choosing different problem logics and implementation languages. The data model is based on the notion of ”pieces of information” $i : S$, where $i$ is a structured information (“information value”) giving constructive evidence for the truth of a specification $S$. Pieces of information support both interoperability in heterogeneous systems and multiple views on data in heterogeneous knowledge domains.

We believe that COOML may have a role as an OO specification framework, and we are developing different prototypical tools. In this abstract, we focus on automatic snapshots generation. In modeling languages, a snapshots represents a system state. In particular, in the context of the UML, snapshots are object diagrams and snapshots generation in the presence of OCL constraints has proved to be useful both for understanding a specification and verifying its consistency in the problem domain [4]. In our approach, our snapshots generation algorithm is driven by the constructive content derived from the the computation of information values.

2 COOML Overview

Here we briefly discuss the COOML logic and the problem domain layers. More details can be found in [9]. The link between the data stored in a software system and their meaning in the “real world” is the result of the abstractions performed in the analysis phase. Typically (see e.g., [6]), the analysis has to produce a dictionary containing the abstract concepts used
in specifications and choose the data types. The analysis phase should result in a language
to talk about the world and its states. We call problem formulas the sentences of this lan-
guage. World-states can be formalized as classical interpretations. However, other problem
logics can be used, and even informal interpretations are allowed. We only need to assume
that problem formulas can be understood by the final user as properties that may or may
not hold in a world-state w. In problem formulas we may have expressions related to the
implementation language. Since we are interested in an OO approach, COOML introduces
a specific syntax to deal with objects and classes. There is a special predefined data type Obj
for object-identities. Class predicates are of the form o.C(), holding in a world-state w if o
is a live object of w. Since objects are rarely isolated entities, we also use class predicates of
the form o.C(e1, . . . , en), linking an object o of class C to its external environment by means
of the environment variables e1, . . . , en.

Example 1. We consider a simple “cash-register problem domain”. The dictionary provides
class predicates such as c.CashRegister() (c is a cash-register), r.Receipt(c) (r is the receipt
of the cash-register c), i.Item(r) (i is an item of the receipt r), and introduces terms such as
cost, code, price of an item, and grand total of a receipt (with self-explanatory meaning). If
that is the case, we can formalize some properties of the problem domain. For example, we
can give the following axioms, where grandtotal and price are partial functions:

(C1) r1 = r2 ← it.Item(r1) ∧ it.Item(r2)
(C2) grandtotal(r) = \sum i.Item(r) price(i) ← r.Receipt(c)
(C3) price(i) = cost(i) + cost(i)*VAT/100 ← i.Item(r)

The COOML logical layer takes the problem domain formulas as atoms, while reasoning
on them is delegated to the problem domain logic. The syntax of a COOML specification
P is shown below, where τ denotes a list of distinct variables of appropriate type τ and
similarly for constants ε : τ.

\[ AT ::= PF | \Box P \\
\[ P ::= AT | AND\{ P . . . P \} | OR\{ P . . . P \} | \exists\{ \tau \cdot \xi : P \} | FOR\{ \tau \cdot \xi : G(\xi) : P \} \]

COOML’s atoms (AT) consist of arbitrary problem formulas PF and \Box-formulas. The latter
serve the purpose of embedding classical truth in our constructive setting. In our language,
as in JML and OCL, universal quantification is bounded. The generator G(\xi) is a particular
problem formula, true for finitely many ground instances ε1, . . . , εm of closed terms; we call
them the terms generated by G(\xi).

We now address the semantics of atoms. The only information on a world-state w carried
by a ground problem formula Aσ is the elementary information value true. It simply means
that Aσ holds in w. The truth of a \Box-formula is defined interpreting the COOML
connectives as ordinary logical ones.

For a specification P, we define the information type IT(P) of P as follows, where informa-
tion values are lists built starting from the primitive data types of the problem domain:

\[ IT(A) = \{ \text{true} \}, \text{where } A \text{ is an AT} \]
\[ IT(\AND\{ P_1 . . . P_n \}) = \{ (i_1, . . . , i_n) | i_j \in IT(P_j), 1 \leq j \leq n \} \]
\[ IT(\OR\{ P_1 . . . P_n \}) = \{ (k, i) | 1 \leq k \leq n \text{ and } i \in IT(P_k) \} \]
\[ IT(\EXI\{ \tau \cdot \xi : P \}) = \{ (\xi, i) | \xi : \tau \text{ and } i \in IT(P) \} \]
\[ IT(\FOR\{ \tau \cdot \xi : G : P \}) = \{ ((\xi_1, i_1), . . . , (\xi_m, i_m)) | m \geq 0 \text{ and, for } 1 \leq j \leq m, \xi_j : \tau \text{ and } i_j \in IT(P) \} \]

A specification P gives meaning to the information values that belong to IT(P). A piece of
information is a pair i : P, with i \in IT(P). For every ground substitution σ, the meaning of
$i : P\sigma$ in a world-state $w$ is given by the relation $w \models i : P\sigma$ ($i : P\sigma$ is true in $w$):

\[
\begin{align*}
  w &\models true : A\sigma \quad \text{iff} \quad A\sigma \text{ holds in } w, \text{ where } A \text{ is an AT} \\
  w &\models (i_1, \ldots, i_n) : \text{AND} \{P_1 \ldots P_n\} \sigma \quad \text{iff} \quad w \models i_j : P_j\sigma, \text{ for all } j = 1, \ldots, n \\
  w &\models (k, i) : \text{OR} \{P_1 \ldots P_n\} \sigma \quad \text{iff} \quad w \models i : P_i\sigma \\
  w &\models (\zeta, i) : \text{EXI} \{\tau \zeta : P(\zeta)\} \sigma \quad \text{iff} \quad w \models i : P(\zeta)\sigma \\
  w &\models L : \text{FOR} \{\tau \xi \mid G(\xi) : P(\xi)\} \sigma \quad \text{iff} \quad \forall \eta \in \text{dom}(L) \text{ iff } G(\xi)\sigma \text{ holds in } w \quad \text{and } ((\zeta, i) \in L \text{ entails } w \models i : P(\zeta)\sigma)
\end{align*}
\]

For example, let us consider a piece of information in the cash-register system. The information contained is: the receipt $r7$ has two items, $\it1$ and $\it2$, having costs 10 and 5 respectively.

\[
((\it1(10 \text{ true})) \ (\it2(5 \text{ true}))) : \text{FOR} \{\text{Obj } \it{item}\mid \text{\it{Item}(r7)} : \text{EXI}\{\text{float } c : c = \text{cost}(i)\})
\]

2.1 Class Specifications in COOML

In COOML we introduce classes via class predicates for the problem domain. The specification of the objects of class $C$ is provided by a class definition of the following form, where $E_C$ is a problem formula, $S_C$ is a specification with name $\text{PtyName}$ and $M_C$ is a list of methods prototypes, possibly with pre and post-conditions:

\[
\text{Class } C \{ \\
\quad \text{ENV}\{\exists \varepsilon \mid \text{this}.C(\varepsilon) : E_C(\text{this}.\varepsilon)\}; \\
\quad \text{PtyName} : S_C(\text{this}.\varepsilon) \\
\quad M_C \\
\}
\]

The meaning of the above class specification in the COOML logic is given constraint axioms and class axioms:

\[
\begin{align*}
\text{ConstrAx}(C) & : \Box \{\text{FOR} \{\text{Obj } \text{this}, \exists \varepsilon \mid \text{this}.C(\varepsilon) : E_C(\text{this}.\varepsilon)\}\} \\
\text{ClassAx}(C) & : \quad \text{FOR} \{\text{Obj } \text{this}, \exists \varepsilon \mid \text{this}.C(\varepsilon) : S_C(\text{this}.\varepsilon)\} \\
\end{align*}
\]

Constraint axioms do not represent any information on the current world-state $w$; what $w \models true : \text{ConstrAx}(C)$ requires is only that $\text{ConstrAx}(C)$ holds in $w$. In contrast, $\text{ClassAx}(C)$ allows us to represent information on the current state by means of the corresponding information values. An OO system is (specified by) a set $S$ of class definitions. We associate with it the set of first order axioms $\text{Ax}(S)$ containing all axioms for all the classes in $S$.

Example 2. The axioms of Ex. 1 express the general properties of the problem domain, while the following COOML classes specify the cash OO system regarding its information content.

\[
\begin{align*}
\text{Class } \text{CashRegister} & \{ \\
\text{CashRegisterPty} & : \text{OR} \{\text{EXI}\{\text{Obj } !\text{receipt} : \text{receipt.Receipt}(\text{this})\}; \\
\quad \text{The receipt is empty;} \} \\
\}
\end{align*}
\]

\[
\begin{align*}
\text{Class } \text{Receipt} & \{ \\
\text{ENV}\{\text{Obj } \text{cash} \mid \text{this}.\text{Receipt}(\text{cash}) : true;\} \\
\text{ReceiptPty} & : \text{AND} \{\text{FOR} \{\text{Obj } \text{item} \mid \text{item.Item}(\text{this}) : \text{item.inCatalog}()\}; \\
\quad \text{EXI}\{\text{float } \text{total} : \text{total} = \text{grandtotal}(\text{this})\} \} \\
\}
\end{align*}
\]
**Class** Item {
  ENV{
    Obj receipt | this.Item(receipt) : this.inCatalog();
  }
  ItemPty: AND{ EXI{
    float cost : cost = cost(this) 
    EXI{
      float price : price = price(this);
    }
  } /* ensures \result = cost + cost*VAT/100 */
  float price();
}

The corresponding constraints and class axioms are defined as above; for instance, the ones for Item are:

- ConstrAx(Item) : □(FOR{Obj receipt this | this.Item(receipt) : this.inCatalog();})
- ClassAx(Item) : FOR{ Obj receipt this | this.Item(receipt) : ItemPty(this, receipt) }

Assume that $P_C : \text{ClassAx}(C)$ is the piece of information of a class axiom $\text{ClassAx}(C)$. In fact, $P_C$ is a (possibly empty) list of information values of the form $((o, t))$, where $o$ instantiates this and $t$ is a tuple of terms instantiating the environment variables $e$. We call $P_C$ a population of class $C$ and treat it as a set. The population $P$ of an OO system is the union of the populations of its classes. We say that an object $o$ belongs to the population $P$ iff there is an information value $((o, t))$ in $P$. A population $P$ is finite (an OO system has a finite set of objects) and each object $o$ of $P$ occurs in a unique information value $((o, t))$ in $P$ (an object belongs to an OO system in a unique copy). We identify system states with populations, and we define the semantics of system states as follows:

**Definition 1.** Let $P$ be a population for a system $S$ and $w$ a world-state. Then $w \models P : S$ iff:

1. $w \models P_C : \text{ClassAx}(C)$ for every class $C$ of $S$, where $P_C$ is the population of class $C$;
2. Every constraint axiom $A$ of $\text{Ax}(S)$ holds in $w$.

In our approach, an OO system $S$ is consistent iff it has a consistent population $P$, and $P$ is consistent iff there is at least a world-state $w$ such that $w \models P : S$.

### 2.2 Snapshots Generation and Consistency

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{uml_snapshots.png}
\caption{UML snapshots}
\end{figure}

Populations are similar to UML object diagrams, also called system snapshots [6]. This correspondence is illustrated in Fig. 1. In general, a population contains more information
than the corresponding UML snapshots and, more importantly, the pieces of information contained in it are related in a precise logical way to the problem domain. Formally:

**Theorem 1.** Let $P \sigma$ be an instance of a COOML specification and $w$ be a reachable world-state of the problem domain. Then: (a) $P \sigma$ holds in $w$ iff there is an information value $i \in \text{IT}(P)$ such that $w \models i : P \sigma$; (b) for every $j \in \text{IT}(P)$, we can extract a set $\mathcal{A}(j, P \sigma)$ of ground atoms from $j$ such that $w \models j : P \sigma$ iff $\mathcal{A}(j, P \sigma)$ holds in $w$.

This theorem allows us to use populations for the following purposes:

**Run time visualization and verbalization tools.** In an implementation of COOML specifications, we can gather at run time information values for the system or a portion of it. This allows us to extract the information contained in the memory state and visualize it in a way similar to UML snapshots; moreover, we can also verbalize it in a human readable way, using the signature and the knowledge of the problem domain. This is loosely connected to the issue of **proof verbalization** from proof-terms [3].

**Run time checking of problem domain properties.** For that, we use the set $\mathcal{A}(i, P \sigma)$ of atoms corresponding to the extracted information values. We assume that the property to be checked is formalized in an appropriate language. For example, let us assume that the current population is as in Fig. 1. To check the constraint (C2) of Ex. 1 using logic programming for the problem logic, we extract the set $\mathcal{A}(i, P_{\text{Receipt}} \sigma)$ containing the atoms $\text{Receipt}(c1), \text{it1.Item}(\text{rcpt1}), \text{it2.Item}(\text{rcpt1}), 11 = \text{price}(\text{it1}), 6.05 = \text{price}(\text{it2}), 17.05 = \text{grandtotal}(\text{rcpt1})$. Translating the above atoms into the required relational form, we can check the constraint (C2) using the following clause (with $\text{sum}$ suitably defined):

$$false(R,T) :- R.\text{receipt}(C), \text{sum}(X, (\text{I.item}(R), \text{cost}(I, X)), \text{Sum}), \text{total}(R,T), \text{T} \neq \text{Sum}.$$  

If the goal $false(R,T)$ fails, then the constraint is validated. Otherwise, we obtain the receipts where the total is wrong.

**Snapshots generation and consistency checking** Using the recursive definition of $\text{IT}(P)$, we can generate information values for the purpose of automatic snapshots generation and property checking. Snapshots generation occurs in two stages. In the first one, we generate a partially defined information value $p(\chi) \in \text{IT}(P)$, where object names are instantiated and the other data are indicated by suitable variable symbols $\chi$ related to the existential and universal quantifiers of $P$. In the second stage, the atoms $\mathcal{A}(p(\chi), P \sigma)$ are extracted and their consistency is checked. Of course, in general consistency is not decidable. However we can check certain properties using one of the existing tools or formalizing the former (if possible) in a suitable Horn theory.

It is worthwhile to give some insights on the algorithm for extracting the partial information values, because this algorithm already checks some constraints on the objects of a population. The user is required to name possible objects of a class, values of an attribute, environments, and the maximum number of snapshots (information values). This could be accomplished by means of a dedicated language such as ASSL [4,5]. We represent the relationships between objects stemming from the class specification and the user inputs as certain $\text{admit}$-clauses. For example, choosing at most two cash-registers and at most two items for each receipt, with $c1,c2,\text{it1},\text{it2}$ constants, we get:

$$\text{admit}(x.\text{CashRegister}()) \leftarrow \text{member}(x,[c1,c2]).$$
$$\text{admit}(\text{Receipt.Receipt}(x)) \leftarrow \text{admit}(x.\text{CashRegister}()).$$
$$\text{admit}(x.\text{Item}(y)) \leftarrow y = \text{Receipt.Receipt}(x) \land \text{admit}(y) \land \text{member}(x,[\text{it1},\text{it2}]).$$

\[5\]
The second admit-clause is fixed at the level of problem domain (each cash-register has at most one receipt). The algorithm tries to generate the information values for a class \(C\), choosing the live objects and the attribute values according to the admit-clauses. The instantiation of the object names proceeds while the information value is being constructed. To this aim, algorithm maintains two lists: the list \(\text{Done}\) of the current completely generated objects and a list \(\text{ToDo}\), containing a partially processed population. For example, if we have the lists \(\text{Done} = [c1.\text{CashRegister}]\) and \(\text{ToDo} = [\text{it1.Item}(x), \text{it2.Item}(x)]\), a new partially defined object \(x.\text{Receipt}(y)\) is generated, and then the solution \(y = c1, x = \text{receipt}(c1)\) is obtained. The list \(\text{Done}\) becomes

\[
[\text{c1.CashRegister()}, \text{receipt.Receipt(c1.CashRegister())}, \text{it1.Item(receipt.Receipt(c1.CashRegister()))}, \text{it2.Item(receipt.Receipt(c1.CashRegister()))}]
\]

and \(\text{ToDo}\) the empty list. We give an idea of the algorithm for generating the information values with Prolog-like pseudo-code. We only show the definition of the main predicate \(\text{info}(+\text{Formula}, –\text{Info}, +\text{Done}, +\text{ToDo}, –\text{Done}, –\text{ToDo})\)

\[
i\text{fo}(\text{A, true, L, C, L, C}) \leftarrow \text{isAtom}(\text{A}).
\]

\[
i\text{fo}(\text{and}[F_1 \ldots F_n], [l_1, \ldots, l_n], L_0, C_0, C_n) \leftarrow \bigwedge_{i=1}^n i\text{fo}(F_i, l_i, L_{i-1}, C_{i-1}, N_j, C_j).
\]

\[
i\text{fo}(\text{or}[F_1 \ldots F_n], [k, l], L, C, N, N, N) \leftarrow \text{fo}(F_k, l, L, C, N, N) \land k = 1, \ldots, n.
\]

\[
i\text{fo}(\text{ex}(T, F), [v, l], L, C, N, N) \leftarrow \text{choose}(V, T, L, C, L, C) \land \text{fo}(F, l, L, C, N, N).
\]

\[
i\text{fo}(\text{for}(T X, G, F), F_1, L, C, N, N) \leftarrow \text{generate}(X, T, G, D, o, L, C, L, C) \land \text{map}(F, D, o, L, C, N, N).
\]

The predicate \(\text{choose}(V, T, L, C, L, C)\) chooses a value for \(V\) according to the type \(T\)\(^1\) and the admit-clauses, and updates (if needed) the lists. Similarly, the predicate \(\text{generate}(X, T, G, D, o, L, C, L, C)\) generates a domain \(D\) for the generator formula \(G\). Finally, \(\text{map}(F, D, o, L, C, N, N)\) computes \([v_1, \ldots, v_n]\), where \(v_1, \ldots, v_n\) are the elements of \(D\). The generation succeeds provided that \(\text{ToDo}\) is empty, and builds an \(\text{Info}\) of \(\text{IT}(\text{Formula})\) that has objects \(\text{Done}\).

### 3 Conclusion

Various logically based modeling languages of OO systems have been proposed, using different formal contexts (e.g., [2,12,1]). The setup of \(\text{COOML}\) is the constructive semantics of evaluation forms. This semantics is related to Medvedev’s logic of finite problems [7] and it has been studied in [8]. Our general aim is to design a logically based framework for the specification of OO information systems, intended as software systems to store and manipulate information with an external meaning. \(\text{COOML}\) has been outlined in [9]. In this abstract we have considered the use of the \(\text{COOML}\) data model, based on pieces of information, for the purposes of run time visualisation and verbalisation of the information stored in the system state, run time checking of system properties, and snapshot generation.

In particular, one of the advantages of using a constructive approach to OO modeling is that the semantics itself, namely the notion of piece of information, directly supports the declarative generation and validation of system snapshots. Moreover, in contrast with model based approaches, information values can be shaped at different levels of granularity. For example, we can use \(\text{EXI}\{\{\varphi \in \exists y \exists x \forall y \exists z : x \neq y \land y \neq z \land \varphi(x, y, z)\} \land \varphi(x, y)\}\) to indicate that the information value has to contain a witness for \(x\), but not for \(y\). This opens the door to applying constructive logical methods to express and formally reason about more intensional system properties.

\(^1\) In the internal representation, \(T\) may contain environment informations.
References