A Constructive Modeling Language for Object Oriented Information Systems

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

[1] Dipartimento di Scienze dell'Informazione, Università degli Studi di Milano
Via Comelico 39/41, 20135 Milano, Italy
{ornaghi,fiorenti,momiglia}@dsi.unimi.it
[2] Dipartimento di Informatica e Comunicazione, Università degli Studi dell'Insubria
Via Mazzini 5, 21100 Varese, Italy
{marco.benini,mauro.ferrari}@uninsubria.it

Abstract. The central aspect in an Information System is the *meaning* of data in the *external world* and the *information* carried by them. We propose a Modeling Language for Object Oriented Information Systems based on a *constructive logic of the pieces of information*, where the focus is on the meaning of data and on the correct way of storing, exchanging and elaborating information. Although the research work presented in this paper is still preliminary, we believe that its potential applications are of interest for the community.

1 Introduction

A software information system *S* allows 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 former concerns *data types*, while the latter is related to the *information on the external "real world"* carried by the data. Precisely, a data type is a set of data together with the associated manipulations where the focus is on *operations*. In contrast, the information carried by the data stored in *S* is strongly related to their *meaning in the real world*. The need of properly treating data according to their meaning is becoming increasingly important, due to the wide quantity of information that is exchanged in the Internet [5, 12]. Quoting [12]:

> "One of the recent unifying visions is that of Semantic Web, which proposed semantic annotation of data, so that programs can understand it, and help in making decisions [. . .] The scope of semantics-based solutions has also moved from data and information to services and processes".

Specification and correct processing of semantically annotated data is the basic motivation of our work: we propose a *Constructive Modeling Language* (CML, in short), where semantical annotations of data are formalized by a constructive *semantics of the pieces of information*. Hence, the focus of the paper is on the structure of the data stored in an OOIS (by OOIS, standing for OO Information system, we refer to a system modelled by the CML) and on the correct way of exchanging and manipulating pieces of information. Our semantics of the pieces of information is based on the *valuation form semantics* [9, 10], which is inspired by the BHK explanation [13] of constructive connectives, but it preserves the notion of truth of classical model theory. In fact, classical truth allows us to model the *meaning in the external world* *w* [10].

We present CML using a Java-like (JL) syntax. In Section 2, we explain the semantics of the pieces of information and we introduce the CML. In Section 3, we show how an OOIS translates into a set of Java classes. Each Java class has methods to correctly extract and transform pieces of information. The semantics of the pieces of information defines a constructive logic E^*, and correct transformations are derived using a fragment of a calculus for this logic. For lack of space, we will only briefly comment on the logic E^* in the conclusions, where future work is discussed and some references to related approaches are given.

2 The logical model of OOIS

We distinguish among *data types*, *information types*, and *object types*. Data are considered as special, immutable objects, without life-time and state. Their properties are general laws that hold independently of the external world. Thus data do not carry any information by themselves. In this paper we assume (using a Java-like syntax) int, boolean, String, ... as predefined data types. We introduce the special data type Obj of *object identities*. Each constant o : Obj uniquely identifies an object. We denote the signature of the predefined data types (including Obj) by Σ_D. We do not further discuss data types, and we focus on *information and object types*. Information types allow us to organize data into suitable structured ”information values”. Objects are the core: they contain the information values, the properties to interpret them in terms of the external world, and the methods to correctly manipulate them. As usual, *classes* group objects with common properties and methods. We distinguish *implementation classes* from *object types*. An object o has a unique implementation class C (the one used to create it), but may participate to many object types. The latter include C, the super-classes of C and the implemented interfaces.

2.1 System Signatures and Meaning

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., [7]), the analysis has to produce a dictionary containing the abstract concepts used in specifications, to choose the needed data types, and to devise general properties of the world that are of interest for the application at hand. We assume that the dictionary includes a first order *system signature* Σ_S that contains the data signature Σ_D and the problem signature Σ_P. Σ_S is designed in such a way that each state of the ”real world” is represented by a Σ_S-interpretation through the abstractions stated in the analysis phase. Σ_P may contain *predicate-declarations* of the form $p : [\mathrm{Obj}, s_1, \ldots, s_n] \rightarrow$ boolean, where $s_1, \ldots, s_n$ are sorts of Σ_D. Objects are abstractions of physical entities of the world, such as planets in the solar system, or correspond to conceptual entities, such as orbits (the example is from [1]). A ground atom $o.p(t_1, \ldots, t_n)$ (in OO dot-notation) represents a property of the entity o that may be true or false in a world–state w. In an OO approach, objects are classified. A *class–predicate* is a special predicate C, where C is a *class name*. The truth of $o.C(t_1, \ldots, t_k)$ in a world–state w means that o is a *live object* of w, with *class* C and *environment* $t_1, \ldots, t_k$. The environment is needed because an object is rarely an isolated entity. In general, it collaborates with other objects and may depend on them.

Σ_S-formulas and Σ_S-interpretations are defined as usual in classical logic, while $w \models F$ denotes the truth of a closed formula F in a Σ_S-interpretation w. World–states are represented at an abstract level by Σ_S-interpretations. We define the *class of the (abstract) world–states* as the subclass of the Σ_S-interpretations w such that: (i) the set of live objects is finite, and (ii) data types are interpreted as predefined.

Finally, the knowledge of the world is represented by a set of axioms and theorems that we denote by WKB (*World Knowledge Base*). The WKB includes a set of axioms Ax_D for reasoning on predefined data types.

Example 1. We consider the well known *eighth queens problem*. The physical objects of the real world are a chessboard and eight queens on it. A world–state is determined by the positions of the queens. We look for the states where no queen is attacked by an other one. At this analysis level, we have the class–predicates $\text{ChessBoard}[\text{Obj}] \rightarrow \text{boolean}$ and $\text{Queen} : [\text{Obj}, \text{Obj}] \rightarrow \text{boolean}$. $cb.\text{ChessBoard}()$ means that cb is a chessboard and $q.\text{Queen}(cb)$ that q is a queen on cb. The *environment* is the chessboard cb. To represent states, we introduce the predicate $inPosition : [\text{Obj}, \text{Obj}, \text{int}, \text{int}] \rightarrow \text{boolean}$. In terms of the real world, $q.inPosition(cb, r, c)$ means that queen q is on row r and column c of the chessboard cb. We may introduce in the WKB new predicates, useful for specification purposes, by *explicit definition*, and prove *classical lemmas* such as ($\vdash_{cl}$ being provability in classical logic):

$$\begin{aligned} D_{up} :\ & q.upAtt(i,j) \leftrightarrow \exists\ \text{Obj}\ cb, \text{int}\, r, c : q.inPosition(cb,r,c) \wedge 0 \le r \wedge r < i \\ & \qquad\qquad\qquad\qquad\qquad \wedge\ (j = c \vee abs(i-r) = abs(j-c)) \\ cl(1) :\ & this.inPosition(cb,i,j) \quad \vdash_{cl} \quad \neg this.upAtt(i,j); \end{aligned}$$

2.2 Properties, Information Values and Pieces of Information

Objects of an OOIS contain *information values* that are structured to represent *pieces of information* about the external world according to the object *properties*. Each Σ_S-formula is an *atomic property* (or atom). Atoms are interpreted as usual in classical logic, i.e., the only information associated with them is their truth. To introduce *structured* properties we use the following separated JL syntax (where B, F denote Σ_S-formulas, $\underline{\tau}\ \underline{x}$ a sequence $\underline{x}$ of variables with types $\underline{\tau}$ and $\underline{c} : \underline{\tau}$ a list of constants with types $\underline{\tau}$):

Atoms	$AT ::= F;$
Basic Properties	$BP := AT \mid \text{OR}\{AT \ldots AT\}$
Structures Properties	$SP := BP \mid BUP \mid \text{AND}\{SP \ldots SP\} \mid \text{EXI}\{\underline{\tau}\ \underline{x} : SP\}$
Bounded Universal Prop.	$BUP := BP \mid \text{FOR}\{\underline{\tau}\ \underline{x} \mid B : SP\}$

The *binding formula B* is a special atom, true for finitely many ground instances of $\underline{x}$. Class predicates $x.C(\ldots)$ are binding formulas for x. We use the abbreviation $\text{EXI}\{\underline{\tau}\ \underline{x} : P_1\ \ldots\ P_n\}$ for $\text{EXI}\{\underline{\tau}\ \underline{x} : \text{AND}\{P_1\ \ldots\ P_n\}\}$.

A property P represents both an information type and a formula (in the latter, $\text{OR}\{\ldots\}$ is a disjunction, $\text{AND}\{\ldots\}$ a conjunction, $\text{EXI}\{\underline{\tau}\ \underline{x} : \ldots\}$ is $\exists\ \underline{\tau}\ \underline{x} : (\ldots)$, $\text{FOR}\{\underline{\tau}\ \underline{x} \mid B : \ldots\}$ the bounded quantification $\forall\ \underline{\tau}\ \underline{x} : (B \rightarrow (\ldots))$. An information type is a set of information values, namely, a constant of the predefined data types or (recursively) a finite list of information values such as $((\text{"}John\text{"},1),(\text{"}Pluto\text{"},2))$. A property P gives meaning to the information values that belong to the *information type* $\textsc{it}(P)$ of P, defined as follows:

$$\begin{aligned}
\textsc{it}(\textsc{or}\{A_1\ \ldots\ A_n\}) &= 1..n;\\
\textsc{it}(\textsc{and}\{P_1\ \ldots\ P_n\}) &= \{(i_1,\ldots,i_n) \mid i_k \in \textsc{it}(P_k), 1 \leq k \leq n\}\\
\textsc{it}(\textsc{exi}\{\underline{\tau}\ \underline{x} : P\}) &= \{(\underline{c},i) \mid \underline{c} : \underline{\tau} \text{ and } i \in \textsc{it}(P)\};\\
\textsc{it}(\textsc{for}\{\underline{\tau}\ \underline{x} \,|\, B : P\} &= \{((\underline{c}_1,i_1),\ldots,(\underline{c}_m,i_m)) \mid\\
&\qquad m \geq 0 \text{ and for } 1 \leq k \leq m,\ \underline{c}_k : \underline{\tau} \text{ and } i_k \in \textsc{it}(P)\}
\end{aligned}$$

An information value for a BUP is an association list $L = ((\underline{c}_1,i_1),\ldots,(\underline{c}_m,i_m))$. We denote by $dom(L) = \{\underline{c}_1,\ldots,\underline{c}_m\}$ the domain of L. $\textsc{it}(P)$ does not depend on the free variables of P, i.e., $\textsc{it}(P) = \textsc{it}(P\sigma)$ for every substitution σ.

A *piece of information* is a pair $i : P$, where P is a property and $i \in \textsc{it}(P)$. For every substitution σ grounding P, the *meaning of* $\ i : P\sigma\ $ in a world–state w is given by the relation $w \,|\!\models i : P\sigma$ (to read $i : P\sigma$ *is true in* w) defined as follows:

$$\begin{aligned}
w \,|\!\models i : \textsc{or}\{A_1\ \ldots\ A_n\}\sigma &\text{ iff } w \models A_i\sigma\\
w \,|\!\models (i_1,\ldots,i_n) : \textsc{and}\{P_1\ \ldots\ P_n\}\sigma &\text{ iff } w \,|\!\models i_k : P_k\sigma, \text{for all } k = 1,\ldots,n\\
w \,|\!\models (\underline{c},i) : \textsc{exi}\{\underline{\tau}\ \underline{x} : P(\underline{x})\}\sigma &\text{ iff } w \,|\!\models i : P(\underline{c})\sigma\\
w \,|\!\models L : \textsc{for}\{\underline{\tau}\ \underline{x} \,|\, A(\underline{x}) : P(\underline{x})\}\sigma &\text{ iff } (\underline{c} \in dom(L) \text{ iff } w \models A(\underline{c})\sigma) \text{ and}\\
&\qquad ((\underline{c},i) \in L \text{ entails } w \,|\!\models i : P(\underline{c})\sigma)
\end{aligned}$$

In a piece of information $i : P$, the information value i is separated from its meaning. We can associate it with a semantically equivalent property P' with the same information type of P, without changing the involved information values or methods.

Example 2. The piece of information

$$((\text{"}John\text{"},1),(\text{"}Pluto\text{"},2)) :\ \textsc{for}\{\text{Obj}\ x \,|\, Occ(x,room5) : \textsc{or}\{Person(x); Dog(x);\}\}$$

means that in the current world-state "*John*" and "*Pluto*" are the occupants of *room*5, "*John*" is a person, and "*Pluto*" a dog. If the WKB contains $room5 = bigroom$ and $Person(x) \leftrightarrow Man(x) \vee Woman(x)$, we can replace the above property by $\textsc{for}\{\text{Obj}\ x \,|\, Occ(x,bigroom) : \textsc{or}\{Man(x) \vee Woman(x); Dog(x);\}\}$. Since the information type is the same ($\textsc{it}(Person(x)) = \textsc{it}(Man(x) \vee Woman(x)) = 1..1$), we can keep the same pieces of information and the same methods. Now the information is that "*John*" and "*Pluto*" are the occupants of *bigroom* and that "*John*" is a man or a woman. In contrast, we cannot replace $Person(x)$ by $\textsc{or}\{Man(x); Woman(x);\}$, because the information type of the latter is 1..2.

A piece of information $i : Ax$ for a set Ax of closed axioms is a set of pieces of information $i_A : A$, one for each axiom A of Ax. We say that $w \,|\!\models i : Ax$ iff $w \,|\!\models i_A : A$, for every $A \in Ax$. In the next subsection we model the states of an IOOS S by the pieces of information for the axioms defined by S.

2.3 OOIS Specifications

The axioms of an OOIS S are BUPs introduced by *class definitions* of the form:

Class C **extends** $C_1, \ldots, C_k$ {
  ENV$\{\underline{\tau}\ \underline{e} : F_C(\underline{t}_0); x_1.C_1(\underline{t}_1); \ldots ; x_k.C_k(\underline{t}_k);\}$ IT $\mathit{ptyName}\{S_C(\mathit{this}, \underline{e});\}\ M_C$
}
where:

– The *environment variables* of C are $\underline{e} = \{x_1, \ldots, x_k\} \cup \mathrm{vars}(\underline{t}_0, \underline{t}_1, \ldots, \underline{t}_k)$, and the *class-predicate* for C is $\mathit{this}.C(\underline{e})$. To the environment declaration we associate the *environment constraint*, which relates $\underline{e}$ to (a possible) $F_C(\underline{e})$ and states a link to the environments of the (possible) super-classes:

$$\mathbf{Env}(C) : \forall\ \underline{\tau}\ \underline{e} : \mathit{this}.C(\underline{e}) \to E_C(\underline{e}) \wedge x_1.C_1(\underline{t}_1) \wedge \cdots \wedge x_k.C_k(\underline{t}_k)$$

– In the IT declaration, $S_C(\mathit{this}, \underline{e})$ is a SP and *ptyName* is a name for it. After the IT-declaration there is a list M_C of method specifications. Methods are briefly discussed in the conclusion.
– We define the *class property* $P_C(\mathit{this}, \underline{e})$ of C as follows: if C has no super-classes, then $P_C(\mathit{this}, \underline{e}) = S_C(\mathit{this}, \underline{e})$, otherwise the properties of the super-classes are inherited according to $\mathbf{Env}(C)$, that is:

$$P_C(\mathit{this}, \underline{e}) = \text{AND}\{S_C(\mathit{this}, \underline{e})\ P_{C_1}(x_1, \underline{t}_1)\ \ldots\ P_{C_k}(x_k, \underline{t}_k)\}$$

where $P_{C_j}(\mathit{this}, \underline{e}_j)$ are (recursively) the properties of the super-classes C_j $(1 \leq j \leq k)$.
– For each *implementation class* C we introduce the *class axiom*:

$$\mathbf{Ax}(C) : \text{FOR}\{\underline{\tau}\ \underline{e} \,|\, C(\mathit{this}, \underline{e}) : P_C(\mathit{this}, \underline{e})\}$$

where P_C is the class property for C. No class axiom is associated with abstract classes and interfaces, because no object can be created by them.

We associate with a system specification S the set of first order axioms $\mathbf{Ax}(S)$, containing the environment constraints of all the classes and the class axioms of all the *implementation classes* of S.

Example 3. Below, we show the classes for the EightQueens system. EXI$\{\tau\ !x : P(x)\}$ abbreviates EXI$\{\tau\, x : \text{AND}\{P(x); \forall\, \tau\, y : P(y) \to y = x;\}\}$.
Class ChessBoard {
IT **chbPty**{AND{ EXI$\{$Obj $!\mathit{firstq} : \mathit{firstq}.$FirstQueen$(\mathit{this});\}$
  IntRows{FOR$\{\mathit{row} \,|\, 1..6 :$ EXI$\{$Obj $!q : q.$InQueen$(\mathit{this}, \mathit{row});\}\}\}$
  EXI$\{$Obj $!\mathit{lastq} : \mathit{lastq}.$LastQueen$(\mathit{this});\}\}\}$
}
abstract Class Queen {
ENV{ Obj chb; int $\mathit{row} : \mathit{chb}.$ChessBoard$(); \mathit{row} \in 0..7;\}$
IT **qPty**{EXI$\{$int$!\mathit{col} : \mathit{this}.\mathit{inPosition}(\mathit{chb}, \mathit{row}, \mathit{col}) \wedge \mathit{col} \in 0..7;\}$
}
abstract Class UpQueen **extends** Queen {
ENV{ Obj chb; int $\mathit{row} : \mathit{this}.$Queen$(\mathit{chb}, \mathit{row});\}$
IT **upqPty**{EXI$\{$Obj $!\mathit{dwn} : \mathit{dwn}.$Queen$(\mathit{chb}, \mathit{row}+1);\}\}$
}

```
abstract Class  DownQueen extends Queen {
ENV{ Obj chb; int row : this.Queen(chb, row); }
IT     dwqPty{EXI{Obj !up : up.Queen(chb, row − 1); }}
}
Class  FirstQueen extends UpQueen { ENV{Obj chb : this.UpQueen(chb, 0); } }
Class  LastQueen extends DownQueen { ENV{Obj chb : this.DownQueen(chb, 7); } }

Class  InQueen extends DownQueen, UpQueen {
ENV{Obj chb; int row : this.DownQueen(chb, row); this.UpQueen(chb, row); }
}
```

According to **chbPty**, each row contains one queen. A queen always stands on its row and can change its column. The first queen has only a lower queen, the last one has only an upper queen, and the intermediate queens have both. This in view of a search where each queen collaborates with its nearest queens to get the next chessboard configuration so that no queen is attacked. The axiomatisation corresponding to the EightQueens system is:

Axioms for the Environment Constraints

$\forall(\mathit{this}.\mathrm{FirstQueen}(\mathit{chb}) \to \mathit{this}.\mathrm{UpQueen}(\mathit{chb},0));$
$\forall(\mathit{this}.\mathrm{LastQueen}(\mathit{chb}) \to \mathit{this}.\mathrm{DownQueen}(\mathit{chb},7));$
$\forall(\mathit{this}.\mathrm{InQueen}(\mathit{chb},\mathit{row}) \to \mathit{this}.\mathrm{UpQueen}(\mathit{chb},\mathit{row}) \wedge \mathit{this}.\mathrm{DownQueen}(\mathit{chb},\mathit{row}));$
$\forall(\mathit{this}.\mathrm{UpQueen}(\mathit{chb},\mathit{row}) \to \mathit{this}.\mathrm{Queen}(\mathit{chb},\mathit{row}));$
$\forall(\mathit{this}.\mathrm{DownQueen}(\mathit{chb},\mathit{row}) \to \mathit{this}.\mathrm{Queen}(\mathit{chb},\mathit{row}));$
$\forall(\mathit{this}.\mathrm{Queen}(\mathit{chb},\mathit{row}) \to \mathit{chb}.\mathrm{ChessBoard}() \wedge \mathit{row} = 0,\ldots,7))$

Class Axioms [3]:

$\mathrm{FOR}\{\,|\,\mathit{this}.\mathrm{ChessBoard}() : \mathbf{chbPty}(\mathit{this})\}$
$\mathrm{FOR}\{\mathrm{Obj}\ \mathit{chb}\,|\,\mathit{this}.\mathrm{FirstQueen}(\mathit{chb}) : \mathrm{AND}\{\mathbf{upqPty}(\mathit{chb},0)\ \mathbf{qPty}(\mathit{this},\mathit{chb},0)\}\}$
$\mathrm{FOR}\{\mathrm{Obj}\ \mathit{chb}\,|\,\mathit{this}.\mathrm{LastQueen}(\mathit{chb}) : \mathrm{AND}\{\mathbf{dwqPty}(\mathit{chb},7)\ \mathbf{qPty}(\mathit{this},\mathit{chb},7)\}\}$
$\mathrm{FOR}\{\mathrm{Obj}\ \mathit{chb}, \mathrm{int}\ \mathit{row}\,|\,\mathit{this}.\mathrm{InQueen}(\mathit{chb},\mathit{row}) :$
$\quad \mathrm{AND}\{\mathbf{dwqPty}(\mathit{chb},\mathit{row})\ \mathbf{upqPty}(\mathit{chb},\mathit{row})\ \mathbf{qPty}(\mathit{this},\mathit{chb},\mathit{row})\}\}$

System States Let $\mathcal{P}_C : \mathbf{Ax}(C)$ be the piece of information for a class axiom $\mathbf{Ax}(C) = \forall\,\underline{\tau}\,\underline{x} : \mathit{this}.C(\underline{x}) \to P_C(\mathit{this},\underline{x})$. Then $\mathcal{P}_C$ is a (possibly empty) list of pieces of information of the form $((o,\underline{t}),i)$ (where o instantiates *this*). We call $\mathcal{P}_C$ a *population of class C*. We treat a population as a set. The population $\mathcal{P}$ of an OO system is the union of the populations of its classes. We say that an object o belongs to the population $\mathcal{P}$ iff there is an information value $((o,\underline{t}),i) \in \mathcal{P}$. A population $\mathcal{P}$ is finite (an OO system has a finite set of objects) and each object o of $\mathcal{P}$ occurs in a unique information–value $((o,\underline{t}),i) \in \mathcal{P}$ (an object belongs to an OO system in a unique copy). The environment constraints of $\mathbf{Ax}(S)$ do not contain information on the current state because they are closed atoms (the only information carried by $w \mid\models 1 : K$ is $w \models K$, and it must hold in every state). Thus, we leave them understood, identify system states with populations, and define truth of system states as follows:

[3] The self-reference *this* is implicitly universally quantified and we use **PtyName**(...) for the corresponding formula, for conciseness.

Definition 1. *Let $\mathcal{P}$ be a population for an OO system S, and w be a world–state. Then $w \mid\models \mathcal{P} : S$ iff:*

- *$w \mid\models \mathcal{P}_C : \mathbf{Ax}(C)$ for every class C of S, where $\mathcal{P}_C$ is the population of class C, and*
- *$w \models K$ for every environment and extension constraint K of $\mathbf{Ax}(S)$.*

Example 4. We generate a population for the EightQueen system, and a world–state w for it. We start with a single ChessBoard-object *chb*:

$$\mathcal{P}_{\text{ChessBoard}} = (((chb),(f,i,l))) \in \textsc{it}(\mathit{this}.\text{ChessBoard}() \to \mathbf{chbPty}(\mathit{this}))$$

We choose

$$f = (q_0,1), i = (((1),(q_1,1)),\ldots,((6),(q_6,1))),$$

and $l = (q_7,1)$. Thus, w has to satisfy [4]:

$$\begin{array}{l}
w \models \text{ChessBoard}(o) \text{ iff } o = chb,\\
w \mid\models 1 : \text{FirstQueen}(q_0,chb) \wedge \forall\ \text{Obj } x : \text{FirstQueen}(x,chb) \to x = q_0,\\
w \mid\models 1 : \text{InQueen}(q_r,chb,r) \wedge \forall\ \text{Obj } x : \text{InQueen}(x,chb,r) \to x = q_r \quad \text{for } r \in 1..6,\\
w \mid\models 1 : \text{LastQueen}(q_7,chb) \wedge \forall\ \text{Obj } x : \text{LastQueen}(x,chb) \to x = q_7.
\end{array}$$

A possible population for FirstQueen, InQueen, LastQueen containing $q_0,\ldots,q_7$ is:

$$\begin{array}{ll}
\mathcal{P}_{\text{FirstQueen}} & = ((q_0,chb),((0,1),(q_1,1))) \in \textsc{it}(\mathbf{Ax}(\text{FirstQueen}))\\
\mathcal{P}_{\text{InQueen}} & = ((q_1,chb,1),((0,1),(q_0,1),(q_1,1)),\ldots,\\
& \quad (q_6,chb,6),((0,1),(q_5,1),(q_7,1))) \in \textsc{it}(\mathbf{Ax}(\text{InQueen})),\\
\mathcal{P}_{\text{LastQueen}} & = ((q_7,chb),((0,1),(q_6,1))) \in \textsc{it}(\mathbf{Ax}(\text{LastQueen})).
\end{array}$$

requiring that:

$$\begin{array}{ll}
w \mid\models (0,1) : \mathbf{qPty}(q_r,chb) & \text{for } 0 \le r \le 7,\\
w \mid\models (q_{r+1},1) : \mathbf{dwqPty}(q_r,chb) & \text{for } 0 \le r \le 6,\\
w \mid\models (q_{r-1},1) : \mathbf{upqPty}(q_r,chb) & \text{for } 1 \le r \le 7.
\end{array}$$

Our population is

$$\mathcal{P} = \mathcal{P}_{\text{ChessBoard}} \cup \mathcal{P}_{\text{FirstQueen}} \cup \mathcal{P}_{\text{InQueen}} \cup \mathcal{P}_{\text{LastQueen}}$$

It is a *consistent population*, because a world–state w such that $w \mid\models \mathcal{P}$: EightQueens exists. It represents a chessboard *chb* where each row r contains the queen q_r in column 0. Other consistent populations with the same objects can be obtained, by changing the columns of the queens.

[4] $\textsc{and}\{A;B\}$ is equivalent to $A \wedge B$ if A, B are atoms, because $w \mid\models (1,1) : \textsc{and}\{A;B\}$ iff $w \mid\models 1 : A \wedge B$.

3 Deriving Java Programs

It is possible to extract a Java program from an OOIS, as follows: every class C of S is translated into a Java class J_C, which represents the environment and the pieces of information of C and which has the methods `pty()` and `info()` to wrap information values and properties into other suitable Java classes. Java does not allow multiple inheritance. However, if we can eliminate it by dropping some intermediate abstract classes, the translation equally works (in Example 5, we drop UpQueen and DownQueen). Properties are transformed in the standard form $\mathrm{EXI}\{\underline{x}\ \underline{\tau} : C_1; \ldots ; C_n\}$, where each C_j is either a BP or a BUP. The variables $\underline{x}\ \underline{\tau}$ become attributes of J_C. If C_j is an atom, a comment is generated; if it is BP, an `int` attribute is generated; if it is a BUP, an auxiliary class is generated, having the name shown in the OOIS-model (in Example 5, **IntRows**).

Example 5. The CML class Chessboard becomes the following Java class.

```
import info.*;
public class ChessBoard {
//ChbPty: exi:
         FirstQueen first;
         LastQueen last;
         //and:
         //true{first.FirstQueen(this);unique(first);}
         IntRows intRows = new IntRows(this);
         //true{last.LastQueen(this);unique(last);}
//endPty

/************************ WRAPPERS *************************/
public ExiInfo info(){
   ExiInfo info = new ExiInfo();
   ...//automatically generated }

public ExiPty info(){
   ExiPty pty = new AndPty();
   ...//automatically generated }
```

The auxiliary class IntRows is omitted for the sake of space. `ExiInfo` and `ExiPty` are classes of the package `info`, which implements pieces of information. The Java class `ChessBoard`, together with its wrapper methods, can be automatically generated starting from the corresponding CML class.

Classes generated in this way are regular Java classes that can be easily understood by a Java programmer; methods can be implemented in the usual way as well. The class `ExiPty` is a subclass of a class `Pty`. A `Pty`-object p represents a property and it has a method `implies` such that p.`implies`(q) returns an object m with a `map` method from $\mathrm{IT}(p)$ into $\mathrm{IT}(q)$. The map is correct, i.e., $w \,||\!\!= i : p$ entails $w \,||\!\!= m.\mathtt{map}(i) : q$. The algorithm for extracting m is based on a fragment of a calculus C^* for the constructive logic of the pieces of information. For lack of space, we cannot discuss this issue here, but we briefly comment on it in the conclusions. The map method supports correct exchange of semantically annotated data. For example, the current state of an object can be wrapped into a piece of information $i : M$ and sent to different (possibly remote) interfaces $R_1, \ldots, R_n$. Each R_j can use $i : M$ in a different way, mapping it according to its local knowledge.

4 Conclusion

Various logically based modeling languages of OO systems have been proposed, using different formal contexts (e.g., [1, 2, 11, 14]). Our setup is the constructive semantics of *valuation forms*. This semantics is related to Medvedev's logic of finite problems [8] and it has been studied in [9, 10]. Our aim is to design a logically based OO modeling language for information systems, intended as software systems to store and manipulate information with an *external meaning*. So far, we have concentrated our analysis on the way of organizing data and meaning in terms of populations of an *Object Oriented Information System* (OOIS). Actually, it is possible to translate an UML [4] class diagram D into an OOIS S_D, and to represent the populations of S_D as object diagrams instantiating D. Thus, we have an adequate expressive power. Although the work presented here is still a preliminary study, we believe that the approach is promising. In fact below we list some possible developments, which can turn it into useful applications. We give also some references to related approaches.

Snapshots and Consistency. In Example 4 we have generated a population for the EightQueens system. Populations correspond to UML object diagrams, also called system snapshots [4]. Showing snapshots is useful to understand an OO model and there are systems enabling snapshot generation (e.g., [6], based on OCL [14]). One of the problems with OO specification is consistency. For example, it is easy to build UML class diagrams with inconsistent multiplicities. In our approach, an OOIS S is consistent iff it has a consistent population $\mathcal{P}$, and $\mathcal{P}$ is consistent iff there is at least an abstract world w such that $w \mid\models \mathcal{P} : S$. In general, the consistency of a population is not decidable. We are studying a partial solution, requiring a restricted syntax for atoms.

Correct Information Exchange. Information values and their meaning are distinct aspects. Pieces of information $i : P$ combine them according to multiple meanings. A similar idea has been developed in XML technology, where XML documents can be interpreted according to different schemas [5]. It is possible to use this technology to wrap information values in XML documents and to define a suitable XML formalism (similar to XML schemas) to represent properties. This would support correct exchange of semantically annotated data, following the trend of Semantic Web [12].

Logical Issues. Our properties can be translated into a fragment of the predicative language of logic E^* introduced in [10]. In E^*, atoms are represented by **T**-formulas $\mathbf{T}(F)$ having information type 1..1, while the logical connectives introduce structured information types. If we represent each atom F by $\mathbf{T}(F)$ and we replace **Or**, **And**, **Exi**, **For** by the corresponding logical connectives, each property becomes an E^*-formula and we obtain a fragment of the predicative language of E^*. E^* is a maximal intermediate constructive propositional logic with a valid and complete calculus [10]. The full predicative extension of E^* has not been studied yet. For our fragment, there is a valid and complete calculus C^* (if we abandon requirements (i), (ii) for the world–states).

Methods and Proof as Programs. A method specification in the class Queen is, e.g.:

$\text{EXI}\{\text{Obj } q : \text{OR}\{q.upAtt(row, n); \neg\exists \text{ Obj } q : q.upAtt(row, n);\}\}$
$q.upAtt(n)$ where *upAtt* is a method declaration and *row* is an attribute

For every $n \in 0..7$, $upAtt(row, n)$ returns $(q, 1)$ (there is an upper queen q that attacks the position (row, n), where *row* is an environment attribute) or $(any, 2)$ (no such queen exists; *any* stands for any object). A Java implementation returns an `ExiInfo`-object. Since E^* is constructive, it is possible to use the calculus C^* to extract an implementation of $upAtt$. C^* has been used to define the method `includes` (see Section 3). It is possible to use C^* to derive the implementation of methods, but we have not developed this idea yet, although this is closely related to the well known idea of proofs as programs [3].

Implementation Issues. Our reference language is Java, but other OO languages may be employed as well. So far, we only have a partial prototypical implementation. The translation from CML classes into corresponding Java classes has been defined but not implemented yet (we do it manually), and our JL syntax is still unstable. We have implemented a hierarchy of classes to wrap information values and properties (see Example 5). The method `includes` provides a basic information transformation. To adapt it to different knowledge contexts, different WKB-packages can be imported, containing (a representation of) pre-proved *classical lemmas* of the form $\Gamma \vdash_{cl} F$, where F is an atom (atoms have a C^*-proof iff they have a classical proof). Classical lemmas can be formally proven or informally stated. An example is $Person(x) \vdash_{cl} Man(x) \vee Woman(x)$ (see Example 2).

References

1. M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag, New York, 1996.
2. A.L. Baker, C. Ruby, and G.T. Leavens. Preliminary design of JML: A behavioural interface specification language for Java. Technical report 98-06, Department of Computer Science, Iowa State University", 1998.
3. J.L. Bates and R.L. Constable. Proofs as programs. *ACM Transactions on Programming Languages and Systems*, 7(1):113–136, January 1985.
4. G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, Reading, Massachusetts, USA, first edition, 1999.
5. D. C. Fallside (Eds). "XML Schema Part 0: Primer". W3C Recommendation, May 2001. http://www.w3.org/TR/xmlschema-0.
6. M. Gogolla, M.Richters, and J. Bohling. Tool support for validating UML and OCL models through automatic snapshot generation. In *SAICSIT '03*, pages 248–257, 2003.
7. C. Larman. *Applying UML and Patterns*. Prentice Hall, Upper Saddle River, NJ, 1998.
8. Ju.T. Medvedev. Finite problems. *Soviet Mathematics Doklady*, 3:227–230, 1962.
9. P. Miglioli, U. Moscato, M. Ornaghi, S. Quazza, and G. Usberti. Some results on intermediate constructive logics. *Notre Dame Journal of Formal Logic*, 30(4):543–562, 1989.
10. 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.
11. D. Rémy. Using, understanding, and unravelling the OCaml language. From practice to theory and vice versa. *Applied Semantics. Advanced Lectures*. Lecture Notes in Computer Science, 2395:413–537, 2002.
12. A. Sheth. DB-IS research for Semantic Web and enterprises. Brief history and agenda. LSDIS Lab, Computer Science, University of Georgia, 2002. http://lsdis.cs.uga.edu/SemNSF/Sheth-Position.doc.
13. A.S. Troelstra. Aspects of constructive mathematics. In J. Barwise, editor, *Handbook of Mathematical Logic*. North-Holland, 1977.
14. J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modelling with UML*. Object Technology Series. Addison-Wesley, Reading/MA, 1999.