

A Constructive Object Oriented Modeling Language for Information Systems

Mario Ornaghi^{a,1} Marco Benini^b Mauro Ferrari^b
Camillo Fiorentini^a Alberto Momigliano^a

^a *Dipartimento di Scienze dell'Informazione, Università degli Studi di Milano,
via Comelico 39, 20135 Milano, Italy*

^b *Dipartimento di Informatica e Comunicazione, Università degli Studi dell'Insubria,
via Mazzini 5, 21100 Varese, Italy*

Abstract

One of the central aspects in an Information System is the *meaning* of data in the *external world* and the *information* carried by them. We propose a Constructive Object Oriented Modeling Language (COOML) for information systems, based on a *constructive logic of pieces of information*. The focus is on the definition of a data model suitable for organizing the information stored in OO systems. The underlying constructive logic supports a correct way of storing, exchanging and elaborating information.

Keywords: Object Oriented Data Models, Constructive Logics.

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. 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

¹ Contacting author: ornaghi@dsi.unimi.it

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 in the Internet [6,15]. Quoting [15]: “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”.

The specification and correct processing of semantically annotated data is the basic motivation of our work: we propose a *Constructive Object Oriented Modeling Language* (COOML, in short), where semantical annotations of data are formalized by a *constructive semantics of pieces of information*. This provides us with a *data model* suitable for OOIS (standing for OO Information System) and a *constructive logic* formalizing correct manipulation and exchange of annotated pieces of information. We give here a brief explanation of the main features of COOML, its data model and general architecture.

We distinguish among *data types*, *information types* and *object types*. As usual, a *data type* introduces a set of data and of computable operations on them. In COOML specifications, the choice of the data types is open. They can be selected depending on the problem domain and on the programming language adopted for implementing COOML specifications. A datum, say the integer 2, does not carry any information by itself. To get an information, we have to associate it with a *problem domain sentence*, such as “John owns 2 cars”. Structured sentences refer to structured pieces of information. For example, the sentence “I know the number of cars of Ted and John” corresponds to a data structure associating one integer with Ted and another with John.

The *data model of COOML* is based on *information types*, which we define as a *type of structured pieces of information related to a type of structured sentences*. The elements of an information type are called *information values*. COOML structured sentences are logical sentences of the *constructive logic* E_{cooml}^* (for brevity denoted by E_c^*) explained in Section 4, a predicative extension of the logic E^* presented in [12].

We can formalize classes and objects by a special kind of E_c^* -formulas. As usual, *objects* have dynamic life time and state; *classes* group objects with common properties and methods. Our data model formalizes *object states as information values* that can be interpreted in terms of the external world, whose intended meaning (that is, of information values) is reflected in the COOML specification of the corresponding classes. This allows us to automatically extract pieces of information represented by the current state of an OO system in a human readable way.

Since the logic E_c^* is constructive, a proof of $S_1 \vdash_{E_c^*} S_2$ in the problem

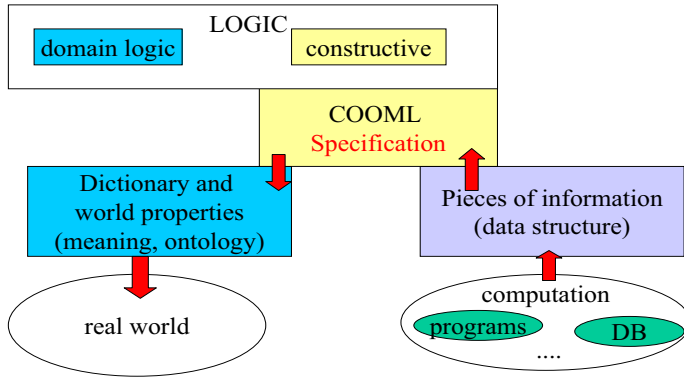


Fig. 1. COOML Architecture

domain implicitly contains an algorithm to correctly transform the information values for a specification S_1 into the information for another specification S_2 . Thus, the inference engine of E_c^* supports information extraction according to different information types (or views).

The possibility of dealing with multiple views and different meanings of information values is enhanced by the multi-layer architecture of COOML, illustrated in Figure 1. The computational layer includes the data type level, supporting basic data and operations, and the information type level, supporting computations on information values. A COOML specification level lays a bridge between the computation tower and the problem domain tower, i.e., it enables us to understand the information values in terms of the problem domain. On the top we have the COOML logical level, which allows us to reason on specifications. COOML is open with respect to the problem domain logic, i.e., reasoning on the problem domain can be based on different logics, or it can even be informal. We do not commit here to a fixed language for the problem domain, but, for the sake of concreteness, we sometimes use the JML specification style [3], denoted by the keyword `JML`.

We present COOML using a Java-like (JL) syntax. In Section 2, we explain the semantics of pieces of information and we introduce the language. In Section 3, we show how a COOML specification translates into a set of Java classes. In Section 4 we present a natural deduction calculus for E_c^* . We then conclude with some comments about directions of future work.

2 The Data Model of COOML

In this section, we explain the data model of COOML. We treat the various aspects separately. In Subsection 2.1, we explain the minimal requirements to be satisfied by the problem domain logic. Then, in Subsection 2.2, we introduce information types and the related semantics. Next (Subsection 2.3), we present class specifications in COOML and our approach to describe system snapshots (Subsection 2.4). Finally, Subsection 2.5 introduces inheritance.

2.1 Problem Logics 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., [8]), the analysis has to produce a *dictionary* containing the abstract concepts used in specifications and choose the *data types* (possibly depending on the implementation language). The analysis phase should result in a (formal or informal) language to *talk about the world and its states*. We call *problem formulas* the sentences of this language. A problem formula F may depend on variables with sorts from the chosen data types and it can be instantiated by grounding substitutions. 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 the instances $F\sigma$ can be understood by the final user as properties that may *hold or not 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*. Constants of sort *Obj* identify objects and *class predicates* of the form $o.C()$, with the following meaning: $o.C()$ holds in a world-state w iff o is a *live object* of w . Since objects are rarely isolated entities, we also use class predicates of the form $o.C(e_1, \dots, e_n)$, linking an object o of class C to its external environment by means of the *environment variables* e_1, \dots, e_n .

Example 2.1 As an example, we consider a simple “cash-register problem domain”. The dictionary provides terms such as:

- Cash Register, the class of the cash-register objects, with the class predicate $c.CashRegister()$.
- Receipt, the class of the receipt objects; since a receipt r is drawn by a cash-register c , we introduce the class predicate $r.Receipt(c)$.
- Item, the class of the items of a receipt, with the (self-explanatory) class predicate $i.Item(r)$ and the predicate $i.inCatalog()$ (the item i is available).

- Cost, the cost of an item excluding VAT. We introduce the informal expression “ $\backslash cost : float$ is the cost of $\backslash item : Obj$ ”, where “ $\backslash \dots$ ” indicates a variable in an informal sentence.
- Price, the price of an item including VAT. We introduce the informal expression “ $\backslash price : float$ is the price of $\backslash item : Obj$ ”.
- Grand total, the total amount of a receipt. We introduce the informal expression “ $\backslash total : float$ is the grand total of $\backslash receipt : Obj$ ”.

During the analysis, one also has to devise *general properties* of the world that are of interest for the application at hand. In our example, we state that the price p of an item is obtained from its cost c by the problem formula $p = c + c * VAT/100$ and that the grand total of a receipt is the sum of the prices of its items. Although informal, the previous statements are rigorous and enable us to reason about the problem domain. We call *problem domain logic* the overall (formal or informal) result of the analysis phase, including the language of the problem formulas and the general problem domain properties.

2.2 COOML Specifications

In COOML, objects contain *information values* that are structured according to *specifications* (SP). We use a Java-like notation; $\underline{\tau} \underline{x}$ denotes a list of distinct variables x_1, \dots, x_n of type τ_1, \dots, τ_n respectively, and similarly for $\underline{c} : \underline{\tau}$. The syntax of SP is defined as follows:

$$\begin{aligned}
 AT &::= PF \mid \Box SP \\
 BUP &::= \text{FOR}\{\underline{\tau} \underline{x} \mid G(\underline{x}) : SP\} \\
 SP &::= AT \mid BUP \mid \text{AND}\{SP \dots SP\} \mid \text{OR}\{SP \dots SP\} \mid \text{EXI}\{\underline{\tau} \underline{x} : SP\}
 \end{aligned}$$

Atoms (AT) consist of arbitrary problem formulas PF and \Box -formulas. The latter (corresponding to the **T**-formulas of [12]) serve the purpose of embedding classical truth in our constructive setting. In our language, as in JML, universal quantification is bounded (*bounded universal property* (BUP)). The *generator* $G(\underline{x})$ is a particular *problem formula*, true for finitely many ground instances $\underline{c}_1, \dots, \underline{c}_m$ of closed terms; we call them the *terms generated by* $G(\underline{x})$. We assume that terms generation is decidable, and this stems from the analysis phase. We write $\text{EXI}\{\underline{\tau} \underline{x} : P_1 \dots P_n\}$ instead of $\text{EXI}\{\underline{\tau} \underline{x} : \text{AND}\{P_1 \dots P_n\}\}$ and a similar abbreviation for a $\text{FOR}\{\dots : \text{AND}\{\dots\}\}$ specification; $\text{EXI}\{\underline{\tau} ! \underline{x} : P\}$ denotes unique existence.

We now address the informal semantics of atoms. The only information on a world-state w carried by a problem formula $A\sigma$ is the elementary information value *true*. It simply means that $A\sigma$ holds in w as explained in the previous section. For a \Box -formula, $\Box(\text{OR}\{P_1 \dots P_n\})\sigma$ holds in a world-state w iff at least one of the $P_j\sigma$ holds (in w), $\Box(\text{AND}\{P_1 \dots P_n\})\sigma$ holds iff all the $P_j\sigma$ hold,

$\Box(\text{EXI}\{\underline{\tau} \underline{x} : P(\underline{x})\})\sigma$ holds iff $P(\underline{x})\delta$ holds for a reassignment δ of \underline{x} in σ , and
 $\Box(\text{FOR}\{\underline{\tau} \underline{x} \mid G(\underline{x}) : P(\underline{x})\})\sigma$ holds in w iff $P(\underline{x})\delta$ holds for all the reassignments δ of \underline{x} such that $G(\underline{x})\delta$ holds.

For a specification P , the *information type* $\text{IT}(P)$ of P is defined as follows, where information values are lists built starting from the primitive data types of the problem domain:

$$\begin{aligned} \text{IT}(A) &= \{ \text{true} \}, \text{ where } A \text{ is an AT} \\ \text{IT}(\text{AND}\{P_1 \dots P_n\}) &= \{ (i_1, \dots, i_n) \mid i_j \in \text{IT}(P_j), 1 \leq j \leq n \} \\ \text{IT}(\text{OR}\{P_1 \dots P_n\}) &= \{ (k, i) \mid 1 \leq k \leq n \text{ and } i \in \text{IT}(P_k) \} \\ \text{IT}(\text{EXI}\{\underline{\tau} \underline{x} : P\}) &= \{ (\underline{c}, i) \mid \underline{c} : \underline{\tau} \text{ and } i \in \text{IT}(P) \} \\ \text{IT}(\text{FOR}\{\underline{\tau} \underline{x} \mid G : P\}) &= \{ ((\underline{c}_1, i_1), \dots, (\underline{c}_m, i_m)) \mid \\ &\quad m \geq 0 \text{ and, for } 1 \leq j \leq m, \underline{c}_j : \underline{\tau} \text{ and } i_j \in \text{IT}(P) \} \end{aligned}$$

That is, information types shape the information values according to the logical structure of the corresponding specification. In particular, an information value for a BUP is an association list $L = ((\underline{c}_1, i_1), \dots, (\underline{c}_m, i_m))$, where $\underline{c}_1, \dots, \underline{c}_m$ are the tuples of terms generated by $G(\underline{x})$. We denote by $\text{dom}(L) = \{\underline{c}_1, \dots, \underline{c}_m\}$ the domain of L . We point out that $\text{IT}(P)$ does not depend on the free variables of P , i.e., $\text{IT}(P) = \text{IT}(P\sigma)$ for every substitution σ .

A specification P gives meaning to the information values that belong to $\text{IT}(P)$. A *piece of information* is a pair $i : P$, with $i \in \text{IT}(P)$. 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) defined as follows:

$$\begin{aligned} w \models \text{true} : A\sigma &\text{ IFF } A\sigma \text{ holds in } w, \text{ where } A \text{ is an AT} \\ w \models (i_1, \dots, i_n) : \text{AND}\{P_1 \dots P_n\}\sigma &\text{ IFF } w \models i_j : P_j\sigma, \text{ for all } j = 1, \dots, n \\ w \models (k, i) : \text{OR}\{P_1 \dots P_n\}\sigma &\text{ IFF } w \models i : P_k\sigma \\ w \models (\underline{c}, i) : \text{EXI}\{\underline{\tau} \underline{x} : P(\underline{x})\}\sigma &\text{ IFF } w \models i : P(\underline{c})\sigma \\ w \models L : \text{FOR}\{\underline{\tau} \underline{x} \mid G(\underline{x}) : P(\underline{x})\}\sigma &\text{ IFF } (\underline{c} \in \text{dom}(L) \text{ iff } G(\underline{c})\sigma \text{ holds in } w \\ &\quad \text{and } ((\underline{c}, i) \in L \text{ entails } w \models i : P(\underline{c})\sigma) \end{aligned}$$

One can easily check that $w \models i : P\sigma$ implies that $\Box P\sigma$ holds in w . A structured specification explains the truth of the specification in terms of the truth of its parts, as shown by the following example.

Example 2.2 In the problem domain of Example 2.1, we can specify a generic receipt by the following specification $S(\text{receipt})$.

Specification $S(\text{receipt})$:

```
AND{ EXI{float total : \total is the grand total of \receipt; }
    FOR{Obj item | item.Item(receipt): EXI{float cost : \cost is the cost of \item; } } }
```

Let us assume that, for $\text{receipt} = r31$, we have the piece of information

```
((17.05 true) (((it1)(10 true)) ((it2)(5.5 true)))) : S(r31)
```

and that the VAT is 10%. We can automatically extract the following human readable information: *17.05 is the grand total of r31, the items of r31 are*

it1 and *it2*, the cost of *it1* is 10 and the cost of *it2* is 5.5 (we point out that the prices of the items are 11 and 6.05 respectively).

To ensure human readability, the specification dictionary is crucial. It should also be clear that we can use distinct equivalent dictionaries. This follows from the fact that, 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.

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 OO system S by the pieces of information for the axioms defined by S .

2.3 Class Specifications in COOML

In COOML we introduce classes via a $o.C(\underline{e})$ class predicate 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 *PtyName* and M_C is a list of methods prototypes, possibly with pre and post-conditions:

```

Class  $C$  {
  ENV{  $\underline{\tau} \underline{e} \mid this.C(\underline{e}) : E_C(this, \underline{e});$  }
  PtyName :  $S_C(this, \underline{e})$ 
   $M_C$ 
}

```

We call E_C the *environment constraint* of C , since it constrains the possible objects instantiating the class C , depending on the environment. Its logical meaning is expressed by the *constraint axiom*:

$$\mathbf{ConstrAx}(C) : \Box(\text{FOR}\{\text{Obj } this, \underline{\tau} \underline{e} \mid this.C(\underline{e}) : E_C(this, \underline{e})\})$$

We call $S_C(this, \underline{e})$ the *information structure* of C , since it describes the structure and the meaning of the information values carried by the instances of C , according to the *class axiom*:

$$\mathbf{ClassAx}(C) : \text{FOR}\{\text{Obj } this, \underline{\tau} \underline{e} \mid this.C(\underline{e}) : S_C(this, \underline{e})\}$$

In the sequel, the self-reference *this* is implicitly universally quantified and we use $PtyName(this, \underline{e})$ for the corresponding formula $S_C(this, \underline{e})$. An OO system is (specified by) a set S of class definitions. We associate with it the set of first order axioms $\mathbf{Ax}(S)$ containing the environment constraints of all classes and class axioms of all the classes of S .

Example 2.3 Let's go back to the cash-register system of Example 2.1. We introduce the class specifications corresponding to the UML diagram of Figure 2.

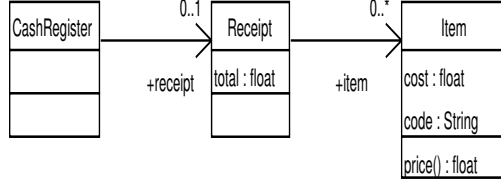


Fig. 2. The cash-register class diagram (A)

```

Class CashRegister {
CashRegisterPty: OR{ EXI{ Obj !receipt : receipt.Receipt(this); }
    The receipt is empty; }
}

Class Receipt {
ENV{ Obj cash | this.Receipt(cash) : true; }
ReceiptPty: AND{ FOR{ Obj item | item.Item(this) : item.inCatalog(); }
    EXI{ float total : \total is the grand total of \this; } }
}

Class Item {
ENV{ Obj receipt | this.Item(receipt) : this.inCatalog(); }
ItemPty: AND{ EXI{ float cost : \cost is the cost of \this; }
    EXI{ String code : \code is the code of \this; }
    EXI{ float price = price() : \price is the price of \this; } }

/* ensures \result = cost + cost*VAT/100 */
float price();
}
  
```

According to **CashRegisterPty**, a cash-register has just one receipt or an empty one. A receipt provides the list of its items (the information associated with the first sub-property of **ReceiptPty**) and the total, associated with the second sub-property of **ReceiptPty**. Finally, an item is described by its cost, its code and its price (the information associated with **ItemPty**). With *float price = price()*, we indicate that *price* is not a class attribute, but the value returned by the method *price()*. In contrast, *cost* and *code* are attributes. The axiomatisation corresponding to the cash-register system is:

Constraint Axioms

ConstrAx(Receipt) \square (FOR{ Obj cash | this.Receipt(cash) : true; })
ConstrAx(Item) \square (FOR{ Obj receipt | this.Item(receipt) : this.inCatalog(); })

Class Axioms

ClassAx(CashRegister) FOR{ | this.CashRegister() : **CashRegisterPty**(this) }
ClassAx(Receipt) FOR{ Obj cash | this.Receipt(cash) : **ReceiptPty**(this, cash) }
ClassAx(Item) FOR{ Obj receipt | this.Item(receipt) : **ItemPty**(this, receipt) }

2.4 System States

Assume that $\mathcal{P}_C : \mathbf{ClassAx}(C)$ is the piece of information of a class axiom $\mathbf{ClassAx}(C)$. In fact, \mathcal{P}_C is a (possibly empty) list of information values of the form $((o \ \underline{t}) \ i)$, where o instantiates *this* and \underline{t} is a tuple of terms instantiating the environment variables \underline{e} . We call \mathcal{P}_C a *population of class C* and treat it 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 and the only information carried by $w \models \text{true} : A$ is that A holds in w . Thus, we leave them as understood, we identify system states with populations, and we define the semantics of system states as follows:

Definition 2.4 Let \mathcal{P} be a population for an OO system S and w a world-state. Then $w \models \mathcal{P} : S$ iff:

- (i) $w \models \mathcal{P}_C : \mathbf{ClassAx}(C)$ for every class C of S , where \mathcal{P}_C is the population of class C ;
- (ii) A holds in w for every environment constraint A of $\mathbf{Ax}(S)$.

We now show how to generate a population \mathcal{P} for the cash-register system and a world-state w_{cash} for it. We start with a single CashRegister object **cr1**:

$$\mathcal{P}_{\text{CashRegister}} = (\text{cr1} \ (1 \ (\text{rcpt1} \ \text{true})))$$

The specification **CashRegisterPty**(**cr1**) requires **rcpt1.Receipt**(**cr1**) to hold in the world-state w_{cash} we are generating. By $\mathbf{ClassAx}(\text{Receipt})$, we have to build the information value for **rcpt1**:

$$\mathcal{P}_{\text{Receipt}} = (\text{rcpt1} \ (((\text{it1} \ (1 \ \text{true})) \ (\text{it2} \ (1 \ \text{true}))) \ (17.05 \ \text{true})))$$

This means that the predicates **it1.Item**(**rcpt1**) and **it2.Item**(**rcpt1**) hold in w_{cash} and that 17.05 is the grand total of **rcpt1**. We have to build the information values for the items (where VAT is 10%):

$$\begin{aligned} \mathcal{P}_{\text{Item}} = \\ ((\text{it1} \ ((10 \ \text{true}) \ ("a15" \ \text{true}) \ (11 \ \text{true})) \\ (\text{it2} \ ((5.5 \ \text{true}) \ ("b121" \ \text{true}) \ (6.05 \ \text{true})))) \end{aligned}$$

Namely, 10 is the cost of **it1**, "a15" is its code and 11 is its price, and so on (note that the grand total of **rcpt1** is actually the sum of the prices of **it1** and **it2**). The generated population is the union of the above populations. Populations correspond to UML object diagrams, also called system snapshots [5]. For example, a UML snapshot corresponding to the above

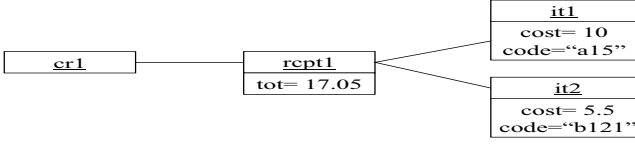


Fig. 3. UML snapshot

population is shown in Figure 3.

In COOML, snapshots can be printed out in a human readable format, by means of pieces of information that collate the information values to the corresponding specifications. For example, a possible print-out of the information contained in the previous population is shown in Figure 4. We remark

Receipt rcpt1:

```
env  rcpt1.Receipt(cr1);
for   it.Item(this): it1.inCatalog();
      it2.Item(this): it2.inCatalog();
exi   float 17.05: 17.05 is the grand total of rcpt1
```

Item it1:

```
env  it1.Item(rcpt1)
exi   float 10: 10 is the cost of it1
      string "a15": "a15" is the code of it1
      float 11: 11 is the price of it1
```

Item it2:

```
env  it2.Item(rcpt1)
exi   float 5.5: 5.5 is the cost of it2
      string "b121": "b121" is the code of it2
      float 6.05: 6.05 is the price of it2
```

Fig. 4. Print-out of the population

that the print-out contains more information than the corresponding UML diagram; indeed, the latter can be derived by the pieces of information, while the converse does not hold.

The ability to show snapshots is a useful tool in order to understand an OO model and there are systems enabling snapshot generation (e.g., [7], based on OCL [17]). In fact, 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 OO system S is consistent iff it has a consistent population \mathcal{P} , and \mathcal{P} is consistent iff there is at least a world-state w such that $w \models \mathcal{P} : S$. For example, the previous population is consistent, since there is a world-state w_{cash} such that the constraint axioms of S hold in w_{cash} and:

$$\begin{aligned}
 w_{cash} &\models \mathcal{P}_{\text{CashRegister}} : \mathbf{ClassAx}(\text{CashRegister}) \\
 w_{cash} &\models \mathcal{P}_{\text{Receipt}} : \mathbf{ClassAx}(\text{Receipt}) \\
 w_{cash} &\models \mathcal{P}_{\text{Item}} : \mathbf{ClassAx}(\text{Item})
 \end{aligned}$$

In general, the consistency of a population is not decidable. We are studying a partial solution based on extraction of atoms that are assumed to be true from pieces of information. Provided that the problem domain rules have a restricted syntax, we can exploit standard results such as consistency of Horn theories.

2.5 Class Hierarchy

In COOML we can formalize sub-classing with inheritance and (possibly) methods overriding:

```

Class  $C$  extends  $C_1, \dots, C_k$  {
  ENV{  $\tau \underline{e} \mid this.C(\underline{e}) : E_C(this, \underline{e}); x_1.C_1(\underline{t}_1); \dots; x_k.C_k(\underline{t}_k);$  }
  PtyName :  $S_{EC}(this, \underline{e})$ 
   $M_C$ 
}

```

where the variables of the terms $\underline{t}_1, \dots, \underline{t}_k$ belong to the environment variables \underline{e} . Note that now the environment constraint for C relates its environment \underline{e} to the environments of its superclasses C_1, \dots, C_k . The specification S_{EC} extends the specifications inherited from the superclasses. The whole information structure S_C of C is defined as follows:

$$S_C(this, \underline{e}) = \text{AND}\{ S_{EC}(this, \underline{e}) \ S_{C_1}(x_1, \underline{t}_1) \ \dots \ S_{C_k}(x_k, \underline{t}_k) \}$$

where the $S_{C_j}(x_j, \underline{t}_j)$ may recursively inherit from further superclasses.

Example 2.5 Let's add the class Discounted extending Item (Figure 5) to the UML diagram of Figure 2:

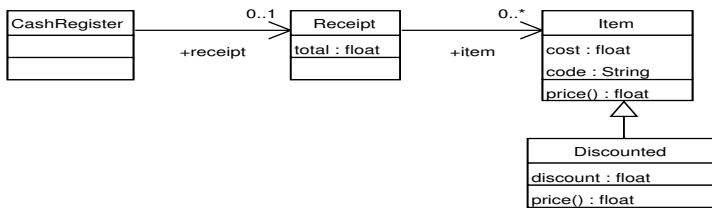


Fig. 5. The cash-register class diagram (B)

```

Class CashRegister {
CashRegisterPty: ...
}

```

```

Class Receipt {
ENV{ Obj cash | this.Receipt(cash) : true;}
ReceiptPty: ...
}

```

```

Class Item {
ENV{ Obj receipt | this.Item(receipt) : this.inCatalog();}

```

```
ItemPty: ...
}
```

```
Class Discounted extends Item {
ENV{ Obj receipt | this.Discounted(receipt) : this.Item(receipt); this.inCatalog(); }
DiscountedPty: EXI{ float discount : \discount is the discount of \this; }

/* ensures \result = (cost - discount) + (cost * VAT /100) */
float price();
}
```

The axioms for the classes CashRegister, Receipt and Item remain unchanged. The axioms for the new class are:

```
ConstrAx(Discounted)  □ ( FOR{ Obj receipt | this.Discounted(receipt) :
                        this.Item(receipt); this.inCatalog(); } )
```

```
ClassAx(Discounted)  FOR{ Obj receipt | this.Discounted(receipt) :
                        AND{ DiscountedPty(this, receipt); ItemPty(this, receipt); } }
```

3 Toward Deriving Java Programs

We outline a procedure to extract a skeleton Java program from a COOML specification. This is similar in spirit to the way UML class diagrams may generate Java code. To this aim, it is preferable to convert a specification into a *conjunctive normal form*

$$\text{EXI}\{ \underline{\tau} \underline{x} : \text{AND}\{ B_1 \dots B_n \} \}$$

where each B_j may be an atom, a BUP or a disjunction $\text{OR}\{\dots\}$ of atoms. This form can always be obtained, provided that suitable new classes are introduced. Every COOML class C in normal form is translated into a Java class J_C , which represents the environment and information values of C and which has both the methods **Info** `getInfo()` to wrap up information values and **Explanation** `explain(Pty p)` to extract human oriented explanations, as we detail next. The variables $\underline{\tau} \underline{x}$ become attributes of J_C . For every formula B_j , we produce an attribute as follows: if B_j is a problem formula, we insert a comment; if it is a BUP, an auxiliary class is generated, while if it is $\text{OR}\{\dots\}$, an `int` attribute is introduced. This is illustrated in Figure 6.

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. Each class **C** has two associated classes **CInfo** and **CPty** that are automatically generated. **CInfo** is a subclass of an abstract predefined class **Info** and allows us to extract the information value i corresponding to the current state of the Java program. The extracted i has information type defined by **CPty**, a subclass of the predefined abstract class **Pty**, used to represent COOML properties. More precisely, **Pty** and **Info** have several predefined subclasses representing the COOML properties and the corresponding information types. The Java programmer does not need to know about those

```

class CashRegister{
  // Exi:
  Obj receipt;
  // Or:
  int case;
  // 1:  receipt.Receipt(this);
  // 2:  The receipt is empty;

  /*****
  * information extraction, automatically generated
  *****/

  public Info getInfo(){
    return new CashRegisterInfo(this);
  }

  /* assumes: p implied by CashRegisterPty */
  public Explanation explain(Pty p){
    return getInfo().explain(p);
  }
}

class Receipt{
  //env:
  Obj cash; /* this.Receipt(cash) */
  //Exi:
  float total;
  //And
  // For{Obj item | item.Item(this): item.inCatalog();}
  ReceiptInfo for_item;
  // true: \total is the grand total of \this;

  .... ‘‘information extraction as before’’ ....
}

class Item{
  //env:
  Obj receipt; /* this.Item(receipt); this.inCatalog(); */
  //Exi:
  float cost;
  String code;
  // And
  // price = price();
  //true: \cost is the cost of \this;
  //true: \code is the code of \this;
  //true: \price is the price of \this;

  /*@ requires cost>0;
  ensures \result = cost + cost*VAT/100;  @*/
  float price(){
    // TO DO
    return 0;
  }
  .... ‘‘information extraction as before’’ ....
}

```

Fig. 6. Sample derived Java program

classes, except for the classes extending the predefined subclass **ForInfo** of **Info**, such as **ReceiptInfo** in the example. The programmer has to use the attribute **for_item** to wrap the information values associated up with the items in the expected way. To this aim, **ReceiptInfo** contains suitable methods to retrieve and update the items stored by **for_item**; in fact, this class essentially contains the Java collection methods specialized to items.

For each class **C**, a **CInfo** object i has information type **CPty** and a method $i.\text{explain}(p)$ that builds an explanation of the information content according to a property p ; the latter is, in general, a singleton object $cpty$ of class **CPty** representing the property defined in the COOML specification of C . To support multiple views on data, we can use properties constructively entailed by $cpty$. Thus, each **Pty** object p has a method **implies** such that $p.\text{implies}(q)$ returns *null* or an object m with a **map** method from $\text{IT}(p)$ into $\text{IT}(q)$. The algorithm for extracting m is based on a suitable logical calculus ND_{c^*} , and it uses a set of *problem domain rules* (**PR**). We explain those in the next section. Here we give a first example that shows a possible use of the method **implies** within the method **explain**. Let us assume that, in the context of the class **Item**, we supply the following rule:

$$\frac{\backslash price \text{ is the price of } \backslash this \quad \text{JML}\{price != cost + cost * VAT/100\}}{\text{Error: } \backslash price \text{ should be } \text{JML}\{cost + cost * VAT/100\}} \text{ PR}$$

The entailment algorithm assumes that in the **PR** rules the basic operations and predicates, such as in JML specifications, can be evaluated. In particular, using the above rule, the algorithm can prove the following “error property” in the class **Receipt**:

FOR{Obj $item$ | $item.\text{Item}(this)$:
OR{ok($this$); Error: $\backslash price$ should be $\text{JML}\{cost + cost * VAT/100\}$ } }

Invoking the **explain(p)** method of a receipt object $rcpt$, with **p** instantiated to the above error property, the entailment is recognized by a call to the **implies** method of the class **ReceiptPty**. The corresponding map is generated and transforms the input explanation into one containing the list of the (possible) wrong items. **Map** and **Explanation** are predefined classes. A **Map** object represents an algorithm to transform information values, while an **Explanation** object represents a piece of information, collating an information value to the corresponding property. This supports also correct exchange of semantically annotated data. For example, the current state of an object can be wrapped into a piece of information $i : P$ and sent to different (possibly remote) interfaces/contexts. Each one can use i in a different way, mapping it according to its local problem domain rules.

4 The Calculus

Our properties can be translated into a fragment of the predicative language of the logic E^* [12], a maximal intermediate constructive propositional logic with a valid and complete calculus, whose full predicative extension has not been studied yet. In E^* , atoms are represented by \Box -formulas having information type *true*, while the logical connectives introduce structured information types. If we represent each problem formula F by $\Box F$ and we replace AND, OR, EXI, FOR by the corresponding logical connectives, each property becomes an E^* -formula. For our fragment, E_c^* , we use a natural deduction calculus ND_{c^*} , whose rules are described in Table 1 and 2. For the sake of brevity, we only consider the binary version of propositional connectives, the n -ary extension being obvious. Similarly, we give quantifiers rules for “singleton” lists of variables, where you can read τx for $\underline{\tau} x$.

ND_{c^*} is basically an intuitionistic calculus with bounded universal quantification, a rule PR to introduce valid problem formulas entailments and a bounded version of Grzegorzczuk’s rule (GR), which can be used to perform exhaustive search. Valid problem domain entailments are introduced by the PR rule, while $\text{I-}\Box$ allows us to infer arbitrary valid formulas. The \Box can be eliminated only from the Harrop fragment of the calculus. The usual provisos about parameters apply.

We now show the soundness of ND_{c^*} with respect to the semantics of pieces of information given in Subsection 2.2. Let Γ be a set of formulas $\{H_1, \dots, H_n\}$. By $\pi : \Gamma \vdash C$ we denote a proof in the calculus ND_{c^*} having as undischarged assumptions H_1, \dots, H_n and as conclusion C ; $\gamma \in \text{IT}(\Gamma)$ denotes the set $\{i_1, \dots, i_n\}$, where $i_j \in \text{IT}(H_j)$ for all $1 \leq j \leq n$; $w \models \gamma : \Gamma\sigma$ means that $w \models i_j : H_j\sigma$ for all $1 \leq j \leq n$.

Theorem 4.1 *Let $\pi : \Gamma \vdash C$ be a proof in the calculus ND_{c^*} and let $\gamma \in \text{IT}(\Gamma)$. Then, there is $i \in \text{IT}(C)$ such that, for all world-state w and ground substitutions σ , $w \models \gamma : \Gamma\sigma$ entails $w \models i : C\sigma$.*

Proof. The proof is by induction on the structure of π . We remark that the proof is constructive, indeed we actually build an information value for the proved formula. Thus, we implicitly define an algorithm which, given a proof π and information values γ for its assumptions, outputs an information value for the conclusion C of π . We only analyze some significant cases.

$\frac{\begin{array}{c} \Gamma_1 \\ \vdots \\ \pi_1 \\ C_1 \end{array} \quad \begin{array}{c} \Gamma_2 \\ \vdots \\ \pi_2 \\ C_2 \end{array}}{\text{AND}\{C_1 \ C_2\}} \text{ I AND}$	$\frac{\begin{array}{c} \Gamma \\ \vdots \\ \pi \\ \text{AND}\{C_1 \ C_2\} \end{array}}{C_j} \text{ E AND} \quad j \in \{1,2\}$
$\frac{\begin{array}{c} \Gamma \\ \vdots \\ \pi \\ C_j \end{array}}{\text{OR}\{C_1 \ C_2\}} \text{ I OR} \quad j \in \{1,2\}$	$\frac{\begin{array}{c} \Gamma_0 \\ \vdots \\ \pi_0 \end{array} \quad \begin{array}{c} \Gamma_1, [C_1] \\ \vdots \\ \pi_1 \\ D \end{array} \quad \begin{array}{c} \Gamma_2, [C_2] \\ \vdots \\ \pi_2 \\ D \end{array}}{D} \text{ E OR}$
$\frac{\begin{array}{c} \Gamma \\ \vdots \\ \pi \\ C(t) \end{array}}{\text{EXI}\{\tau \ x : C(x)\}} \text{ I EXI}$	$\frac{\begin{array}{c} \Gamma_0 \\ \vdots \\ \pi_0 \end{array} \quad \begin{array}{c} \Gamma_1, [C(p)] \\ \vdots \\ \pi_1 \\ D \end{array}}{D} \text{ E EXI}$
$\frac{\begin{array}{c} \Gamma, [G(p)] \\ \vdots \\ \pi \\ C(p) \end{array}}{\text{FOR}\{\tau \ x \mid G(x) : C(x)\}} \text{ I FOR}$	$\frac{\begin{array}{c} \Gamma_0 \\ \vdots \\ \pi_0 \\ G(t) \end{array} \quad \begin{array}{c} \Gamma_1 \\ \vdots \\ \pi_1 \\ \text{FOR}\{\tau \ x \mid G(x) : C(x)\} \end{array}}{C(t)} \text{ E FOR}$
$\frac{\begin{array}{c} \Gamma, [G(p)] \\ \vdots \\ \pi \\ \text{OR}\{C(p) \ B\} \end{array}}{\text{OR}\{\text{FOR}\{\tau \ x \mid G(x) : C(x)\} \ B\}} \text{ GR}$	

Table 1
The calculus ND_{c^*}

The proof π is:

$$\frac{\begin{array}{c} \Gamma_1 \\ \vdots \\ \pi_1 \\ C_1 \end{array} \quad \begin{array}{c} \Gamma_2 \\ \vdots \\ \pi_2 \\ C_2 \end{array}}{\text{AND}\{C_1 \ C_2\}} \text{ I AND}$$

Suppose that $w \models \gamma : \Gamma\sigma$. By induction hypothesis on the subproofs π_1 and π_2 , there are $i_1 \in \text{IT}(C_1)$ and $i_2 \in \text{IT}(C_2)$ such that $w \models i_1 : C_1\sigma$ and $w \models i_2 : C_2\sigma$. Let us take $(i_1, i_2) \in \text{IT}(\text{AND}\{C_1 \ C_2\})$; we have $w \models (i_1, i_2) : \text{AND}\{C_1 \ C_2\}\sigma$ as required.

$\frac{H_1, \dots, H_n}{A} \text{ PR} \quad \text{if } A \text{ is an atom and } H_1 \wedge \dots \wedge H_n \rightarrow A \text{ is a valid PF}$	
$\frac{\begin{array}{c} \Gamma \\ \vdots \\ \pi \\ C \\ \hline \end{array} \text{ I } \Box}{\Box C}$	$\frac{\begin{array}{c} \Gamma \\ \vdots \\ \pi \\ \Box A \\ \hline \end{array} \text{ E } \Box \quad A \text{ an atom}}{A}$
$\frac{\begin{array}{c} \Gamma \\ \vdots \\ \pi \\ \Box(\text{AND}\{C_1 \ C_2\}) \\ \hline \end{array} \text{ E } \Box \text{ AND} \quad j \in \{1, 2\}}{\Box C_j}$	$\frac{\begin{array}{c} \Gamma \\ \vdots \\ \pi \\ \Box(\text{FOR}\{\tau \ x \mid G(x) : C(x)\}) \\ \hline \end{array} \text{ E } \Box \text{ FOR}}{\text{FOR}\{\tau \ x \mid G(x) : \Box C(x)\}}$

Table 2
The calculus ND_{c^*} (continued): PR and \Box rules

The proof π is:

$$\frac{\begin{array}{c} \Gamma_0 \\ \vdots \\ \pi_0 \\ \text{OR}\{C_1 \ C_2\} \end{array} \quad \begin{array}{c} \Gamma_1, [C_1] \\ \vdots \\ \pi_1 \\ D \end{array} \quad \begin{array}{c} \Gamma_2, [C_2] \\ \vdots \\ \pi_2 \\ D \end{array}}{D} \text{ E OR}$$

Suppose that $w \models \gamma : \Gamma\sigma$. By induction hypothesis on π_0 , there is $i \in \text{IT}(\text{OR}\{C_1 \ C_2\})$ such that $w \models i : \text{OR}\{C_1 \ C_2\}\sigma$. Let us assume that $i = (1, i_1)$, with $i_1 \in \text{IT}(C_1)$ (the case $i = (2, i_2)$ is similar). Then, by induction hypothesis on π_1 , where we associate i_1 with C_1 , there is $i_D \in \text{IT}(D)$ such that $w \models i_D : D\sigma$, and this concludes the case.

The proof π is:

$$\frac{\begin{array}{c} \Gamma, [G(p)] \\ \vdots \\ \pi \\ C(p) \end{array}}{\text{FOR}\{\tau \ x \mid G(x) : C(x)\}} \text{ I FOR}$$

Suppose that $w \models \gamma : \Gamma\sigma$. Let c_1, \dots, c_m be the terms generated by $G(x)$ and let us consider, for all $1 \leq j \leq m$, the proofs π_j obtained from π by instantiating the parameter p with c_j . By induction hypothesis, where we associate *true* with the atom $G(c_j)$, there is $i_j \in \text{IT}(C(x))$ such that $w \models i_j : C(c_j)\sigma$. Let us consider the list $L = ((c_1, i_1), \dots, (c_m, i_m))$. One can easily check that $w \models L : \text{FOR}\{\tau \ x \mid G(x) : C(x)\}\sigma$. \square

With respect to completeness, we conjecture this to hold for the version of ND_{c^*} with unbounded universal quantification, using the techniques of [12].

```

ENVASS: receipt.Receipt(cr){
  LOCAL 1: FOR{Obj item | item.Item(receipt) : item.inCatalog(); }

  ENVASS: [2] item.Item(receipt){
    PRULE: JAVA{check(item);} item.Item(receipt);  $\implies$  interesting \receipt;
    PROVE: OR{ JAVA{!check(item);} interesting \receipt; }
  }

  PROVE[GR 1,2]: OR{ FOR{Obj item | item.Item(receipt) : JAVA{!check(item);} }
    interesting \receipt; }

  PRULE: FOR{Obj item | item.Item(receipt) : JAVA{!check(item);} }
     $\implies$  non-interesting \receipt;

  PROVE: OR{non-interesting \receipt; interesting \receipt; }
}

```

Fig. 7. A contextual proof

The case we are considering here seems to be harder since finiteness cannot be captured in first-order logic.

We aim to use the calculus ND_{c^*} in a modular way with respect to the class hierarchy, e.g. proofs are developed in the context of the appropriate class. A useful contextual proof-pattern is:

$$\begin{array}{ccc}
 o.C(\underline{t}) & \mathbf{ClassAx}(C) & o.C(\underline{t}) \quad \mathbf{ClassAx}(C) \\
 \vdots \pi_1 & & \vdots \pi_n \\
 A_1 & \dots\dots\dots & A_n \\
 & \vdots \pi_{loc} & \\
 & H &
 \end{array}$$

We say that $o.C(\underline{t})$ is the *environment assumption* and the formulas A_1, \dots, A_n are *local assumptions*. The latter are obtained by repeated applications of elimination rules from the environment assumption and the axiom $\mathbf{ClassAx}(C)$. Then, in order to develop a proof in the context of C , one has only to choose the local assumptions and prove the final consequence H .

We are working on a *modular* calculus, dependent on COOML classes, which allows us to contextualise proofs and to realize the *proofs as programs paradigm* in our Java implementation, by translating proofs into regular Java methods. The proof in Figure 7 shows the general idea (the syntax is provisional); in particular, it illustrates the use of (run-time computable) Java expressions inside a proof and an application of the rule GR. Here, we assume that the class `Item` implements a method `check` to check whether an item is of interest for us. The proof constructively shows that we can exploit the stored information to state whether a receipt contains at least one interesting item. The proof can be translated into a Java method of the class `Receipt` performing the job.

The declaration “ENVASS: $o.C(\underline{t})$ ” introduces the environment assumption $o.C(\underline{t})$ and automatically proves (if possible) the used local assumptions using the axiom $\mathbf{ClassAx}(C)$. In the inner proof (the one with

nested environment $item.Item(receipt)$, $receipt.Receipt(cr)$), the declaration “ $JAVA\{check(item);\}$ ” indicates that the entailment algorithm assumes

$$OR\{ JAVA\{check(item);\} \quad JAVA\{!check(item);\} \}$$

(namely, the run-time computability of $check$) to constructively prove the consequence $OR\{JAVA\{!check(item);\} \quad interesting \setminus receipt;\}$. Then, we apply the rule GR to this proof in the upper context $receipt.Receipt(cr)$, discharging the assumption “ $ENVASS: [2] \quad item.Item(receipt);\}$ ”. We remark that the local assumption 1 is needed to guarantee that $item.Item(receipt)$ is a generator for the eigenvariable $item$, as required for a sound application of the rule GR.

To conclude, we briefly comment on generators. We implement BUP’s via collections and generators as Java iterators. This will allow us to implement the proofs as programs paradigm in our modular calculus, since iterators provide the iterative computation implicit in the validity proof for the rules I FOR and GR. Since the set of live objects of a system is always *finite*, every class predicate could be used as a generator without any restriction; however, the problem would be what to do with the garbage collector when accessing the currently live objects. This is a common problem in OO specification languages, see for example the unclear semantical status of annotations such as $\setminus allocated$ in JML [9].

5 Conclusion

Various logically based modeling languages of OO systems have been proposed, using different formal contexts (e.g., [1,3,14,17]). 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*.

Our setup is the semantics of pieces of information and it is based on the *evaluation form semantics* [11,12]. This is inspired by the BHK explanation [16] of constructive connectives and gives rise to a constructive logic E^* . More precisely, E^* is related to Medvedev’s logic of finite problems [10] and has been studied in [11,12]. It preserves the notion of truth of classical model theory, which is a natural *problem domain logic*, where models represent *world-states* and formulas express *world properties*. Although the logic E_c^* of COOML is based on E^* , it does not commit to any specific problem domain logic; concerning the latter, even informal reasoning is allowed. Moreover, E_c^* is constructive and its notion of entailment supports methods for the correct exchange and manipulation of pieces of information.

So far, we have concentrated our analysis on the way of organizing data and meaning in terms of populations of an OOIS. Actually, it is possible to

translate an UML [5] class diagram D with OCL constraints 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.

Methods and Proofs as Programs

So far, we have privileged the study of the data model and the use of the calculus ND_c^* to extract and manipulate the information stored in an OO system. It is possible to use it 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 [4]. This is in our future plans, together with a suitable modular calculus. As briefly discussed in Section 4, this would allow us to base reasoning on the modular structure of classes and to mix proofs and query methods (i.e., methods that do not change the current state). Finally, E_c^* is a “static” logic, i.e., it works as far as building system snapshots and reasoning on them, but it does not consider updates. Updates and side effects can be modelled in the problem domain logic. We may choose different logics, with the only restriction that a notion of truth in a world-state is given, such as methods based on specification by contract (see e.g. [2,9,13], the latter being based on constructive logics).

Correct Information Exchange

Information values and their meaning are distinct aspects. Pieces of information $i : P$ combine values according to multiple meanings. A similar idea has been developed in XML technology, where XML documents can be interpreted according to different schemas [6]. It is possible to use this technology to wrap up information values in XML documents and to define a suitable XML formalism (similar to XML schemas) to represent properties. The constructive logical system underlying COOML would support correct exchange of semantically annotated data, following the trend of Semantic Web [15]. This could be based on libraries of problem domain packages (see also next paragraph) defining and classifying concepts.

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 COOML 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 Section 3). The method `Pty.implies` provides a basic information transformation. To adapt it to different knowledge contexts, different problem domain packages can be imported, containing (a representation of) valid *problem rules*.

References

- [1] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag, New York, 1996.
- [2] W. Ahrendt, T. Baar, B. Beckert, R. Bubel, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, A. Roth, S. Schlager, and P. H. Schmitt. The KeY Tool. *Software and System Modeling*, 4:32–54, 2005.
- [3] 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.
- [4] J.L. Bates and R.L. Constable. Proofs as programs. *ACM Transactions on Programming Languages and Systems*, 7(1):113–136, January 1985.
- [5] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, Reading, Massachusetts, USA, first edition, 1999.
- [6] D. C. Fallside (Eds). XML Schema Part 0: Primer. W3C Recommendation, May 2001. <http://www.w3.org/TR/xmlschema-0>.
- [7] 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.
- [8] C. Larman. *Applying UML and Patterns*. Prentice Hall, Upper Saddle River, NJ, 1998.
- [9] G.T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, and J. Kiniry. JML Reference Manual (DRAFT), April 2005.
- [10] Ju.T. Medvedev. Finite problems. *Soviet Mathematics Doklady*, 3:227–230, 1962.
- [11] 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.
- [12] 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.
- [13] I. Poernomo. Proofs-as-Imperative-Programs: Application to Synthesis of Contracts. In *Manfred Broy, Alexander Zamulin (Eds.), Proceedings of the 5th Ershov Conference, 2003, Novosibirsk, Russia*, LNCS 2890, 112–119, 2003.
- [14] D. Rémy. Using, understanding, and unravelling the OCaml language. From practice to theory and vice versa. *Applied Semantics. Advanced Lectures*. LNCS, 2395:413–537, 2002.
- [15] 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>.
- [16] A.S. Troelstra. Aspects of constructive mathematics. In J. Barwise, editor, *Handbook of Mathematical Logic*. North-Holland, 1977.
- [17] J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modelling with UML*. Object Technology Series. Addison-Wesley, Reading/MA, 1999.