# Correct Object-Oriented Systems in Computational Logic

Kung-Kiu Lau[1] and Mario Ornaghi[2]

[1] Department of Computer Science, University of Manchester
Manchester M13 9PL, United Kingdom
kung-kiu@cs.man.ac.uk

[2] Dipartimento di Scienze dell'Informazione, Universita' degli studi di Milano
Via Comelico 39/41, 20135 Milano, Italy
ornaghi@dsi.unimi.it

**Abstract.** In our previous work in program development, we have defined *steadfastness*, a notion of correctness that captures at once modularity, correctness and reusability. This paper extends it to Object-Oriented Systems, namely systems of cooperating objects, and is a first step towards the introduction of correctness in Object-Oriented Design Frameworks. Such frameworks are increasingly recognised as more reusable than single objects, and promise to be useful components in next-generation Component-based Software Development.

## 1 Introduction

It is increasingly recognised that classes are not the best focus for design. Typical design artefacts are rarely just about one object, but about groups of objects and the way they interact. Object-Oriented Design (OOD) Frameworks, or just *frameworks* for short, are such groups of interacting objects. They have widely been accepted as more useful components than single objects, and are the focus of much attention in the evolution of current OO technology into Component-based Software Development (CBD). The goal of CBD is to provide the engineering science and tools for constructing software products by plugging components together, and frameworks would be suitable as such components.

To support CBD, a framework should have a good component *interface* providing all the information we need for its *correct reuse*. Conditions for correct reuse can be detailed as follows:

- We need to know what each component does (correctly): framework operations should have *declarative specifications* and composition of frameworks should yield the specification of the operations of the composite.
- We need an *assembly guide* to correctly reusing frameworks: the interface should provide *context dependencies*, i.e. conditions to be met in order to preserve correctness of framework operations (with respect to their specifications).

– We need to know what framework composition yields: the specifications of a composite framework should indicate when and how we can *correctly compose* the operations inherited from its components.

These conditions imply that a semantics for frameworks and their interfaces should incorporate a notion of *a priori correctness*, namely *pre*-proved correctness of any given component operation with respect to its own specifications, as well as *pre*-stated conditions (context dependencies) that will guarantee that component and operation compositions will preserve correctness.

Moreover, reusability requires open frameworks (to be composed with open, i.e. non-fixed, external parts) and open programs (programs using open operations of the open external parts); thus *a priori* correctness should consider *open programs* in *open contexts*.

In current OO technology, it is supposed to be possible to model frameworks [7, 5] in UML [13] (see e.g. [4]). However, it is not yet clear how frameworks are defined in UML, and a formal semantics is currently lacking. Such a semantics is necessary for a universally accepted and well understood definition of frameworks and interfaces, and a deeper understanding of correct reuse.

We have studied correct reuse and *a priori* correctness in the context of non-OO frameworks which we called *specification frameworks* [8]. We believe that they are a good starting point for formalising correct reuse in OOD frameworks. This paper represents a first step in this direction. In Section 2, before considering OOD frameworks, we briefly recall specification frameworks and *steadfastness*, namely our a priori correctness for logic programs [10]. Then we introduce our formalisation of OO Systems. It is based on a three-level architecture, containing an object level, a system level and a global level. Section 3 introduces the basic definitions and the syntax of the system level, while Section 4 contains the three-level formalisation and the main results. Correct OO Systems represent our first step towards a formalisation of correct reuse in OOD frameworks. In the conclusions we briefly discuss the perspectives of our future work, and we briefly comment on related approaches.

## 2 Specification Frameworks: non-OO Frameworks

Specification frameworks are based on a three-tier formalism, as shown in the example of Fig. 1.

At the bottom level, we have *programs*, for computing (specified) relations. Programs are standard or constraint logic programs and may be *open*. An open program contains at least one *open program predicate*, that is, a predicate that occurs in the body of some clause but in none of the heads. For example, the program $P_{sort}(split, merge)$ in Fig. 1 is open in the predicates *split* and *merge*.

In the middle, we have *specifications*. A specification $S_p$ declaratively defines the meaning of a program predicate $p$, in terms of the problem context formalised at the top level. We may have *strict* and *non-strict* specifications. For every interpretation of the problem context, a strict specification $S_p$ defines

Specification Framework $\mathcal{LIST}(X, \lhd; split, merge)$

Context $\mathcal{LIST}(X, \lhd : [X, X])$;

EXTENDS: $\mathcal{NAT}$;

DECLS:
$$\begin{aligned} X, ListX &: sort; \\ nil &: [\,] \to ListX; \\ . &: [X, ListX] \to ListX; \\ @ &: [X, Nat, ListX]; \\ nocc &: [X, ListX] \to Nat; \end{aligned}$$

CONSTRS: $TotalOrder(\lhd)$;

AXIOMS: ...

Specifications:
$$\begin{aligned} S_{sort} &: sort(X,Y) \leftrightarrow perm(X,Y) \wedge ord(Y); \\ S_{merge} &: ord(X) \wedge ord(Y) \to (merge(X,Y,Z) \leftrightarrow ord(Z) \wedge perm(X|Y, Z)); \\ S_{split} &: l(X) > 1 \wedge split(X,Y,Z) \to perm(X, Y|Z) \wedge l(Y) < l(X) \wedge l(Z) < l(X); \\ & \phantom{:} \ l(X) > 1 \to \exists Y, Z \,.\, split(X,Y,Z); \end{aligned}$$

Programs:

$P_{sort}(split, merge)$ :
$$\begin{aligned} sort(nil, nil) &\leftarrow \\ sort([X],[X]) &\leftarrow \\ sort([X,Y|A], W) &\leftarrow split([X,Y|A], I, J), sort(I,U), \\ &\phantom{\leftarrow}\ sort(J,V), merge(U,V,W) \end{aligned}$$

**Fig. 1.** The open specification framework $\mathcal{LIST}$.

one interpretation of $p$; for example, $S_{sort}$ is strict. A non-strict specification allows a family of interpretations of $p$, i.e., it allows different correct implementations; for example, $S_{merge}$ and $S_{split}$ are non-strict. Non-strict specifications are fundamental to improving reusability.

At the top, we have the *meta-level* or *context*, that embodies the relevant knowledge of the problem domain. The purpose of the context is to give a precise meaning to specifications and to support reasoning about program correctness. We formalise a context as a first-order theory axiomatising the necessary data types and the problem domain. We distinguish between *closed* and *open* contexts.

A context is *closed* iff it has a *reachable isoinitial model*; isoinitial semantics is similar to initial semantics of algebraic ADT's, from which our contexts are inspired (see [2] for algebraic ADT's; see [9] for a discussion of initial and isoinitial theories). For example, $\mathcal{LIST}$ includes the closed context $\mathcal{NAT}$; $\mathcal{NAT}$ contains the usual first-order axioms for arithmetic, and has the standard structure of natural numbers as a reachable isoinitial model.

We have introduced $\mathcal{NAT}$ and the symbols @ and *nocc* in a formalisation of lists for specification purposes. The intended meaning of $x@(i,l)$ is that $x$ occurs at position $i$ in the list $l$, while $nocc(x,l)$ denotes the number of occurrences of $x$ in $l$. In this way, we can define, for example, ordered lists by

$ord(L) \leftrightarrow \forall x, y : X; i : Nat \,.\ x@(i, L) \wedge y@(s(i), L) \rightarrow x \lhd y$, where $s$ is the successor function. The context for lists is open.

An open context contains a set $\Pi$ of *open symbols*. By suitable *closure* operations, it can be closed, i.e. made into a closed context, in many different ways, giving rise to many different *closed instances*. For example, in Fig. 1 the open context symbols are $X$ and $\lhd$, and a closure is shown in Fig. 2. Closures must satisfy the *constraints* CONSTR (see [8]), to guarantee that programs are inherited while preserving *a priori* correctness.

| Specification Framework $\mathcal{NL}$ extends $\mathcal{LIST}(X, \lhd; split, merge)$ |
|---|
| CLOSE: $X = Nat$;<br>$x \lhd y \leftrightarrow x \leq y$;<br>CONSTRSAT: $Theorem : TotalOrder(\lhd)$; |
| Specifications:<br>$S'_{merge} : ord(x) \rightarrow (merge([x], y, z) \leftrightarrow ord(z) \wedge perm([x]\|y, z))$<br>$S'_{split} : split([x, y\|z], [x], [y\|z])$ |
| Programs ... (correct programs for merge and split) |

**Fig. 2.** A closure of $\mathcal{LIST}$.

In a framework $\mathcal{F}$, we distinguish between an *import interface* and an *export interface*: the former indicates which information is to be provided for reusing $\mathcal{F}$, and the latter indicates what can be obtained by reusing $\mathcal{F}$. The import interface has *context dependencies*, namely conditions to be satisfied in order to guarantee *correct reuse*. The import interface corresponds to the *open context symbols* and the *open program predicates*.

At the context level, importing is implemented by *union* and *closure* operations. Union allows us to amalgamate different contexts, and can be formalised by pushouts, as in algebraic ADT's [2]. Closure allows us to fix the open symbols, and is independent from union. For example, in Fig. 2 union is not needed because natural numbers are already included and closure can be performed within the framework itself.

*Constraints* codify the *context dependencies* for the closures of context symbols. *Explicit closures* of open relations and functions will be needed in the next section. They look like explicit definitions, where the defined symbol $\sigma$ is not a new one, but is the open symbol to be closed. If $\sigma$ has constraints, they must become theorems after the closure. In Fig. 2, $x \lhd y$ is closed by the explicit closure

$$x \lhd y \leftrightarrow x \leq y$$

and the CONSTRSAT (constraint satisfaction) section proves $TotalOrder(\lhd)$ as a theorem.

The closure of an open program predicate $p$ with specification $S_p$ has to provide a program for $p$, which is to be correct with respect to $S_p$. That is, $S_p$ is the *context dependency* for $p$. Specifications work as *polymorphic* context dependencies through the operation of *specification reduction*. We have different kinds of specifications, with different reducibility conditions to guarantee that, if a specification $S'$ reduces to a specification $S$, then correctness with respect to $S'$ entails correctness with respect to $S$ (we say that $S'$ correctly overrides $S$). For example, $S_{merge}$ is a conditional specification (CS) and, by CS reduction, $S'_{merge}$ *correctly overrides* it. Similarly, $S'_{split}$ correctly overrides $S_{split}$. A correct implementation of $S'_{merge}$ and $S'_{split}$ gives rise to the insertion sort algorithm. Different overridings of $S_{merge}$ and $S_{split}$ allow us to specify and implement different correct algorithms, like merge sort or quick sort.

The export interface contains the non-open context symbols and the corresponding theorems, as well as the correct programs and their specifications. The exported context can be reused for reasoning and specification purposes, e.g., for specification reduction. By *a priori correctness*, the exported programs can be correctly reused according to their specifications. For example, $S_{sort}$ and $P_{sort}(split, merge)$ are exported. Thus $P_{sort}(split, merge)$ *correctly composes* with the local $P_{merge}$ and $P_{split}$ programs, and can be *correctly composed* with any program $P'$ open in a predicate *sort* with a specification $S'_{sort}$ such that $S_{sort}$ reduces to $S'_{sort}$.

In OO terminology, we could consider $\mathcal{LIST}$ as an *abstract class* with parameter $X$, *attribute* $\lhd$, *implemented method* $P_{sort}$ (with specification $S_{sort}$) and *abstract methods* specified by $S_{merge}$ and $S_{split}$. We can give different non-abstract subclasses, with different implementations of the abstract methods, correct with respect to different specifications $S_{merge}$ and $S_{split}$ that correctly override those of the superclass.

After closing $X$ over $Nat$ (or over other sorts included by union) we can use the non-abstract subclasses to create instances by closing $\lhd$ in different ways. Like objects in OO terminology, different instances have different signatures, which may be obtained by prefixing the instance identifiers. For example, we may have an instance $a$ with closure $x\ a.\lhd\ y \leftrightarrow x \leq y$, and an instance $d$ with closure $x\ d.\lhd\ y \leftrightarrow y \leq x$. Both $a$ and $d$ will *correctly reuse* the same $P_{sort}$ program of $\mathcal{LIST}$, together with the $S_{sort}$ specification, that is, $a.P_{sort}$, $a.S_{sort}$ and $d.P_{sort}$, $d.S_{sort}$ are the same. The difference is in the local closure of $\lhd$ and in the corresponding implementation of the ordering predicate: $a.P_{sort}$ will sort lists in ascending order, and $d.P_{sort}$ in descending order. Moreover, if $a$ and $d$ are created by different subclasses, they polymorphically (but correctly, by specification reduction) implement the operations specified by $S_{merge}$ and $S_{split}$ in the superclass $\mathcal{LIST}$.

This discussion shows that typical OO features can be modelled by adapting specification frameworks, and we already used them to model OOD frameworks

[8]. However, in our previous work *a priori* correctness was disregarded. We now consider it in the next section .

## 3 OO Systems: Towards OOD Frameworks

A formalisation of OOD frameworks should consider at least the following features:

(a) An OOD framework should have good component *interface*, providing all the information we need for the *correct reuse* of it; moreover, for correct method reuse (a kind of) *a priori* correctness is needed.
(b) An OOD framework defines a collection of *instances*, where each instance is a possibly distributed system of cooperating objects.
c) The instances of an OOD framework have states and methods that can update states.

Concerning (a), open specification frameworks containing steadfast programs meet this requirement. So, specification frameworks will provide the basis for our treatment. With respect to (b), we could adapt specification frameworks in a direct way, but this would give rise to a formalisation hiding the distributed nature of systems of cooperating objects. Instead, here we give a three-level architecture, which allows us to introduce more expressive specifications, and to deal with *a priori* correctness for methods that do not change the states of the instances of an OOD framework. We call such static OOD frameworks *OO systems*.

We introduce some preliminary notions in Section 3.1. Then, in Section 3.2, we discuss some problems related to the constraint and specification language, and we introduce OO systems. In Section 4, we consider our three-level architecture.

### 3.1 Classes, Objects and Systems

An OO system $\mathcal{S}$ is a set of related classes, where each class contains *attributes*, *constraints*, *specifications* and *methods*. Like in many OO approaches, we distinguish between *data* and *objects*, and we specify data and operations (programs) on them by a specification framework $\mathcal{D}$ (see Section 2), called the *data* (framework). $\mathcal{D}$ also codifies the problem domain, so that it can give meaning to specifications.

For simplicity, we will assume that $\mathcal{D}$ is closed, and we will indicate by $\mathfrak{i}_D$ its isoinitial model, and call it the *data model*.

We introduce OO systems gradually, starting from classes and objects.

**Objects and Classes.** We start from the following features of objects (see e.g. [1]): (a) objects have identities; (b) each object $o$ has one implementation class $C$, which fixes its attributes and methods. Moreover, in class-based approaches, we have inheritance and overriding polymorphism, related to subclassing and subtyping. We will not consider these aspects here, since our main concern is the characterisation of frameworks as systems of cooperating objects.

Concerning (a), we introduce a special sort $Obj$ together with a denumerable set of constants $o_1, o_2, \ldots : [\,] \to Obj$, used to name objects, and we give the *unique name axioms* $\neg o_i = o_j$ (for $i \neq j$). Constants of sort $Obj$ will be called *object identities.* We will assume that the above axiomatisation of $Obj$ always belongs to the data $\mathcal{D}$.

Concerning (b), we introduce (implementation) classes as follows: a *class* $C$ implicitly includes the data $\mathcal{D}$, and introduces a set of *attributes*, declared as new open dotted constant, function or relation symbols. $C$ also introduces *constraints, specifications and methods*, but we delay their treatment to Section 3.2.

Through the dot notation, a class $C$ works as a template for generating *objects of class* $C$. If $o$ is (the identifier of) an object of class $C$, $o.a$ is an *attribute of* $o$. The *object signature* of $o$ contains the attributes of $o$ and the signature of the data $\mathcal{D}$.

For example, the class $Town$ in Fig. 3 specifies *towns* as objects with roads, where each road is of sort $Obj$, i.e., it is in turn an object. The signature of the object $\underline{mi}$ of class $Town$ contains the data signature and the object attribute $\underline{mi}.road : [Obj]$.[1]

**Class** $Town$;
ATTRS:
$.road : [Obj]$;
CONSTRS: ...; SPECS: ...; PROGS: ...;
**Object** $\underline{mi} : Town$;
STATE: $\underline{mi}.road(x) \leftrightarrow x = \underline{l1} \vee x = \underline{l2}$

**Fig. 3.** An object of class $Town$

Object attributes are *open*; they can be closed by explicit closures (see Section 2), that we call *state axioms.*

By the properties of explicit closures, state axioms fix a unique expansion of the data model $\mathfrak{i}_D$, i.e., a unique intended interpretation of the attributes, representing the *current state* of the object. Different state axioms represent different states, and state transitions can be codified as updates of the state axioms.

For example, the state axiom of the object $\underline{mi}$ in Fig 3 is

$$\underline{mi}.road(x) \leftrightarrow x = \underline{l1} \vee x = \underline{l2}$$

[1] Object identifiers are underlined, in accordance with UML.

that is, $\underline{mi}$ has two roads, $\underline{l1}$ and $\underline{l2}$. If a new road $\underline{l3}$ is built, then the state axiom is updated into

$$\underline{mi}.road(x) \leftrightarrow x = \underline{l1} \vee x = \underline{l2} \vee x = \underline{l3}.$$

**Systems.** An OO system with classes $C_1, \ldots, C_n$ and data $\mathcal{D}$ is indicated by $\mathcal{S}[\mathcal{D}, C_1, \ldots, C_n]$. It is a blueprint for systems of cooperating objects, that we call *system instances.*

Each instance has a *population* $O$ of objects. We introduce the *class predicates* $C_1, \ldots, C_n : [Obj]$, with the following intended meaning: $C_i(o)$ means that $o$ is an object of the current population, with class $C_i$. We do not consider subclassing here, so each object of the population has one class;[2] this is stated by the *population constraints*

$$\neg\exists x : Obj\,\textbf{.}\;\; C_i(x) \wedge C_j(x) \quad (\text{for } i \neq j).$$

Class predicates are *open.* A *closure* of a class predicate $C_i$ of the form

$$C_i(x) \leftrightarrow x = o_1 \vee \cdots \vee x = o_n$$

is called a *class axiom*, and a finite population $O$ can be represented by a set of class axioms $CAx(O)$, one for each class, such that:

- $CAx(O)$ is consistent with the population constraints;
- $o$ belongs to $O$ with class $C_i$ iff $CAx(O) \vdash C_i(o)$.

That is, the class axioms $CAx(O)$ fix the current population $O$ and the classes of objects of $O$. $O$ underlies a population signature $\Sigma(O)$, according to the class axioms:

**Definition 1 (Population Signature).** *Let $\mathcal{S}[\mathcal{D}, C_1, \ldots, C_n]$ be an OO system, and $O$ be a population represented by a set of class axioms $CAx(O)$. The corresponding* population signature *$\Sigma(O)$ contains the data signature of $\mathcal{D}$, the class predicates $C_1, \ldots, C_n$ and all the object attributes $o.a$ such that $CAx(O) \vdash C_j(o)$ (i.e., $o$ belongs to $O$ with class $C_j$) and $.a$ is an attribute of $C_j$.*

For example, the OO system in Fig. 4 has data $\mathcal{LIST}(Obj)$ and two classes *Road* and *Town*. The figure also shows an instance, whose $O$ population is fixed by the class axioms. The corresponding signature $\Sigma(O)$ contains the object attributes $\underline{l1}.source : [\,] \rightarrow Obj$, $\underline{l1}.target : [\,] \rightarrow Obj$, $\underline{l2}.source : [\,] \rightarrow Obj$, $\underline{l2}.target : [\,] \rightarrow Obj$ for the objects of class *Road*, and $\underline{mi}.road : [Obj]$, $\underline{mn}.road : [Obj]$, for those of class *Town*.

Once we have fixed a population $O = \{o_1, \ldots, o_n\}$ and its signature $\Sigma(O)$ by $CAx(O)$, we have to fix the states of the objects of $O$, by suitable state axioms. A set of state axioms $SAx(o_1)$, …, $SAx(o_n)$ fixing the states of all the objects

[2] Subclassing and subtyping could be treated as well, but with more complex constraints.

**System** *Roads and Towns*;
DATA: $\mathcal{LIST}(Obj)$
**Class** *Road*;
ATTRS: $.source, .target : [\,] \to Obj$;
CONSTRS: ...; SPECS: ...; PROGS: ...;

**Class** *Town*;
ATTRS: $.road : [Obj]$;
CONSTRS: ...; SPECS: ...; PROGS: ...;
**Instance Of** *Roads and Towns*;
CLASS AX: $Town(x) \leftrightarrow x = \underline{mi} \vee x = \underline{mn}$; $Road(x) \leftrightarrow x = \underline{l1} \vee x = \underline{l2}$;
**Object** $\underline{l1}$;
STATE: $\underline{l1}.source = \underline{mi}$; $\underline{l1}.target = \underline{mn}$;
**Object** $\underline{l2}$;
STATE: $\underline{l2}.source = \underline{mn}$; $\underline{l2}.target = \underline{mi}$;
**Object** $\underline{mi}$;
STATE: $\underline{mi}.road(x) \leftrightarrow x = \underline{l1}$;
**Object** $\underline{mn}$;
STATE: $\underline{mn}.road(x) \leftrightarrow x = \underline{l2}$.

**Fig. 4.** The Roads and Towns System and an instance of it.

of $O$ will be called a set of *system state axioms for the population* $O$ and will be indicated, collectively, by $SAx(O)$.

Each triple $\langle O, CAx(O), SAx(O)\rangle$ defined as before is an *instance definition*. For example, Fig. 4 shows an instance definition with population $O = \{\underline{l1}, \underline{l2}, \underline{mi}, \underline{mn}\}$, $CAx(O)$ as declared in the CLASS AX (class axioms) section, and $SAx(O)$ as given in the STATE sections of the objects $\underline{l1}, \underline{l2}, \underline{mi}, \underline{mn}$.

The model-theoretic semantics of instance definitions is given by the following proposition:

**Proposition 1 (System States).** *Let $\langle O, CAx(O), SAx(O)\rangle$ be an instance definition for an OO system $\mathcal{S}[\mathcal{D}, C_1, \ldots, C_n]$. There is one $\Sigma(O)$-expansion $i_O$ of the data model $i_D$ such that $i_O \models CAx(O) \cup SAx(O)$.*

That is, each instance definition fixes a *system state*:

**Definition 2 (System States).** *A* system state *of a system $\mathcal{S}[\mathcal{D}, C_1, \ldots, C_n]$ is a $\Sigma(O)$-interpretation $i_O$ such that $i_O \models CAx(O) \cup SAx(O)$, for some instance definition $\langle O, CAx(O), SAx(O)\rangle$.*

### 3.2 Class Constraints, Specifications and Programs

To fully model UML class diagrams by our OO systems, we need to introduce constraints. For example, the class diagram in Fig. 5, corresponding to our OO system of Fig. 4, contains *multiplicity constraints*, that is, every *road* object has one target and one source *town*, and each *town* has a set of 0 or more roads of class *Road*. Multiplicity constraints are just an example. Constraints are an

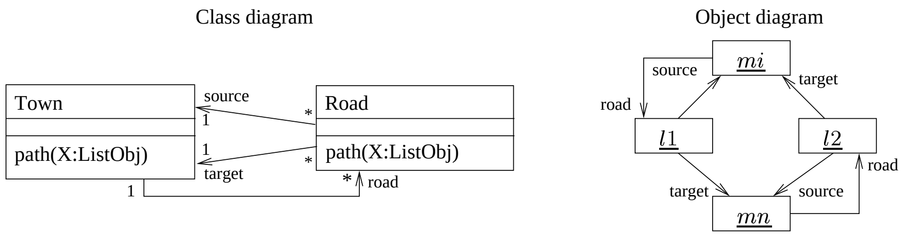


**Fig. 5.** UML diagrams for *Roads and Towns*.

essential modelling feature of UML class diagrams, and OCL (Object Constraint Language, see e.g. [14]) has been introduced for expressing them. We need a similar constraint language, using first-order logic. For example, the multiplicity constraint 1 on the $Town$ side of *road* can be expressed in the class *Road* by a constraint like:

$$\exists! t\,.\ t.road(this)$$

We can consider it as an abbreviation of a first-order formula like $\forall x\,.\ Road(x) \rightarrow \exists! t\,.\ road(x,t)$, by considering $.road$ as a binary relation symbol and the expression $x.road(y)$ as a syntactic variant of $road(x,y)$. This solution corresponds to the *global level* that will be introduced in Section 4 and will allow us to directly employ our theory of steadfastness to prove *a priori* correctness.

However the global level hides the distributed nature of OO systems, where each object $o$ has its own attribute signature and state, and computations are performed in a distributed way, through a collaboration among objects. To model systems of cooperating objects, we need to also consider this distributed view, and show that reasoning at the global level allows us to prove properties of the distributed *object level*.

In Section 4, we will consider a three-level formalisation, with a global level, an object level, and a system modelling level, that we will call the OO system level. The latter level, considered here, is the bridge between the object and the global levels, through a global and an object semantics.

In this section we introduce the OO system syntax for constraints, specifications and programs. It completes the syntax introduced in Section 3.1.

Firstly we introduce the $C(x)$-*quantifiers* (or *class quantifiers*) $[\forall x : C(x)]\,.$ and $[\exists x : C(x)]\,.$ , where $C$ is a class predicate.

Now, let $\mathcal{S} = [\mathcal{D}, C_1, \ldots, C_n]$ be an OO system. Each class $C_j$ of $\mathcal{S}$ has its own constraint language, where $C_j$-*constraints* are built from the usual logical symbols and the class quantifiers, starting from atomic formulas which may contain all the function and relation symbols of $\mathcal{D}$, and function and relation symbols of the form $this.a_j$, where $.a_j$ is an attribute of the class $C_j$, or $x.a_i$, where $.a_i$ is an attribute of class $C_i$ and $x$ is bounded by a $C_i(x)$-quantifier.

$C_j$*-specifications* have the same general syntax of $C_j$-constraints, with the restriction that they must be program-predicate specifications of the kind mentioned in Section 2.

**OO System** *Roads and Towns*

DATA: $\mathcal{LIST}(Obj)$

**Class** *Road*;

ATTRS: $.source, .target : [\,] \to Obj$;

CONSTRS: $Town(this.source) \wedge Town(this.target)$;
$[\exists! t : Town(t)].\ t.road(this)$;

SPECS: $this.path(L) \leftrightarrow this@(0, L) \wedge (\forall o : Obj.\ o \in L \to Road(o)) \wedge$
$([\forall a : Road(a), b : Road(b)].\ \forall i : Nat.$
$a@(i, L) \wedge b@(s(i), L) \to a.target = b.source)$;

PROGS: $this.path([this|L]) \leftarrow Town(T), T = this.target, T.path(L)$;

**Class** *Town*;

ATTRIBUTES: $.road : [Obj]$;

CONSTRS: $[\forall o : Road(o)].\ (this.road(o) \leftrightarrow o.source = this)$;

SPECS: $this.path(l) \leftrightarrow l = [\,] \vee [\exists x : Road(x)].\ this.road(x) \wedge x.path(l)$;

PROGS: $this.path([\,]) \quad \leftarrow$
$this.path([X|L]) \leftarrow this.road(X), Road(X), X.path(L)$;

**Fig. 6.** The complete *Roads and Towns* OO system

Concerning (logic) programs, we assume that $\mathcal{D}$ contains Clark's equality axioms for the signature to be used in program terms. Thus unification works as usual. We say that *a program predicate* $.p$ *is defined in a class* $C_j$ if $C_j$ contains a program $P_{this.p}$ that has clauses with head predicate $this.p$, and bodies of the form $L, K, B$, where $L$, $K$, $B$ are possibly empty lists of atoms such that:

- $L$ contains *local constraints* of the form $X = this.c$, where $.c : [\,] \to Obj$ is an attribute of $C_j$; or of the form $this.a(\tau_1, \ldots, \tau_n)$, where $.a$ is an attribute of $C_j$ and $\tau_1, \ldots, \tau_n$ are terms;
- $K$ contains *class constraints* of the form $C_i(X)$ where $C_i$ is a class predicate;
- $B$ contains predicates of the form $this.q$, where $.q$ (possibly coinciding with $.p$ in the case of recursion) is defined in $C_j$; or of the form $X.r$, where $K$ contains a class constraint $C_i(X)$ and $.r$ is defined in the class $C_i$ (with $i \neq j$).

Using the above syntax, the *Roads and Towns* system can be completed as in Fig. 6.

## 4 A Three-level Formalisation of OO Systems

Our formalisation of OO systems is based on a three-level architecture, containing an *object level*, a *system level* and a *global level*. The object level takes into account the distributed nature of systems of collaborating objects, and corresponds to the *object level* of UML. The system level corresponds to the class-based view of class diagrams, and corresponds to the *model level* of UML. Finally, the global level is similar to the *meta-model level* of UML, and considers the system $\mathcal{S}$ as a whole; it allows us to reason about properties that cannot be confined to single classes, and to formally treat *a priori* correctness of the (static) methods of $\mathcal{S}$.

In Sections 4.1 and 4.2 we will define the object and global levels, and state precise links between them and the system level by syntactic maps, which we call *projections*: $\downarrow$ projections will map the system level into the "lower" object level, and the $\uparrow$ projections will map it into the "upper" global level.

The $\downarrow$ projections define the *object-level semantics* of class constraints, specifications and programs. This semantics can be seen as a formal counterpart of the *object level* semantics of UML class diagrams.

The $\uparrow$ projections define a *global semantics*, where the OO system is considered as a whole; the organisation in classes disappears and we get a single non-OO open specification framework of the kind considered in Section 2. Thus *a priori* correctness in the context of specification frameworks works at the global level in OO systems.

The global level works as a meta-level for the object one. This is shown in Section 4.3, where we introduce syntactic and semantic maps between the global and the object level, discuss the links between the two levels and prove that global-level correctness entails object-level correctness. A general picture will be given (in Fig. 11) in Section 4.3.

### 4.1 The Object Level

Here we define the *object-level semantics* of OO system constraints, specifications and programs.

At this level, we are interested in populations of cooperating objects. Each population $O$ has its own signature $\Sigma(O)$, and system states with population $O$ are represented by $\Sigma_O$-interpretations (see Definition 2). Thus, object-level semantics has to interpret constraints and specifications of an OO system $\mathcal{S}[\mathcal{D}, C_1, \ldots, C_n]$ in terms of $\Sigma(O)$-interpretations. To this end, for every class $C_j$ and every object $o \in O$ with class $C_j$, we define a map of $C_j$-constraints and $C_j$-specifications into $\Sigma(O)$-formulas, which we call *o-constraints and o-specifications*. The interpretation of $o$-constraints and $o$-specifications is the usual one of first-order $\Sigma(O)$-formulas, and can be taken as the *intended* object-level semantics of $C_j$-constraints and $C_j$-specifications. We get a distributed semantics, where each object $o \in O$ has its own $o$-constraints and $o$-specifications.

For a class $C_j$ and an object $o \in O$ of class $C_j$, the map of $C_j$-constraints and $C_j$-specifications into the corresponding $o$-constraints and $o$-specifications is defined by the following $\downarrow_o O$ projection:

**Definition 3 ( $\downarrow_o O$ Projections).** *Let $O$ be a population for an OO system $\mathcal{S}$ with class axioms $CAx(O)$, $K$ be a $C_j$ constraint, and $o$ be an object of class $C_j$. The $\downarrow_o O$* projection *for $o$ is the recursive translation:*

- *For an atomic formula $\sigma(t_1, \ldots, t_n)$, the projection is:*

$$\sigma(t_1, \ldots, t_n) \downarrow_o O = \sigma^*(t_1 \downarrow_o O, \ldots, t_n \downarrow_o O)$$

  *where, if $\sigma$ is of the form this.a, then $\sigma^*$ is o.a; otherwise $\sigma$ and $\sigma^*$ coincide.*
- *For a class $C$ with class axiom $C(x) \leftrightarrow x = o_1 \vee \cdots \vee x = o_k$, $C(x)$-quantifiers are translated as follows:*

$$\begin{array}{l} ([\forall x : C(x)] \bullet\ F) \downarrow_o O = F[x/o_1] \downarrow_o O \wedge \ldots \wedge F[x/o_k] \downarrow_o O \\ ([\exists x : C(x)] \bullet\ F) \downarrow_o O = F[x/o_1] \downarrow_o O \vee \ldots \vee F[x/o_k] \downarrow_o O \end{array}$$

  *where the replacements $[x/\ldots]$ are applied to all the variable and prefix occurrences of $x$ in $F$.*
  *If the class axiom is $C(x) \leftrightarrow false$, i.e. no object of class $C$ exists, then*

$$\begin{array}{l} ([\forall x : C(x)] \bullet\ F) \downarrow_o O = true \\ ([\exists x : C(x)] \bullet\ F) \downarrow_o O = false. \end{array}$$

- *For the usual logical symbols, we have:*

$$\begin{array}{l} (\forall x : s \bullet\ F) \downarrow_o O = \forall x : s \bullet\ F \downarrow_o O \\ (\exists x : s \bullet\ F) \downarrow_o O = \exists x : s \bullet\ F \downarrow_o O \\ (\neg F) \downarrow_o O = \neg F \downarrow_o O \\ (A \wedge B) \downarrow_o O = A \downarrow_o O \wedge B \downarrow_o O \\ (A \vee B) \downarrow_o O = A \downarrow_o O \vee B \downarrow_o O \\ (A \rightarrow B) \downarrow_o O = A \downarrow_o O \rightarrow B \downarrow_o O. \end{array}$$

As an example, Fig. 7 shows the $\underline{mi}$-constraints and the $\underline{mi}$-specifications of the object $\underline{mi}$ belonging to the population considered in Fig. 4. Fig. 7 also shows the $\underline{mi}$-program.

Concerning programs, the purpose of $\downarrow_o O$-projections is to give their *object-level semantics*. We now define $\downarrow_o O$-projection and then discuss it.

Program clause projection replaces *this* by $o$, and then eliminates the possible class constraints as follows:

$$(o.P(\tau) \leftarrow L, C(X), K, B) \downarrow_o O = \{(o.P(\tau) \leftarrow N, K, B)[X/o_i] | 1 \le i \le k\}$$

where the class axiom for $C$ is $C(x) \leftrightarrow x = o_1 \vee \cdots \vee x = o_k$. If the class axiom is $C(x) \leftrightarrow false$, then the clause is deleted. Such class constraint elimination is iterated, until the set $K$ of class constraints becomes empty.

$$
\begin{array}{ll}
\textbf{Population } O\text{: } Town(x) \leftrightarrow x = \underline{mi} \vee x = \underline{mn}; & Road(x) \leftrightarrow x = \underline{l1} \vee x = \underline{l2}; \\
\textbf{Object } \underline{mi} : Town; & \\
\text{CONSTRS:} & (\underline{mi}.road(\underline{l1}) \leftrightarrow \underline{l1}.source = \underline{mi}) \wedge (\underline{mi}.road(\underline{l2}) \leftrightarrow \underline{l2}.source = \underline{mi}); \\
\text{SPECS:} & \underline{mi}.path(l) \leftrightarrow l = [\,] \vee (\underline{mi}.road(\underline{l1}) \wedge \underline{l1}.path(l)) \vee (\underline{mi}.road(\underline{l2}) \wedge \underline{l2}.path(l)); \\
\text{PROGS:} & \underline{mi}.path([\,]) \quad \leftarrow \\
 & \underline{mi}.path([\underline{l1}|L]) \leftarrow \underline{mi}.road(\underline{l1}), \underline{l1}.path(L); \\
 & \underline{mi}.path([\underline{l2}|L]) \leftarrow \underline{mi}.road(\underline{l2}), \underline{l2}.path(L);
\end{array}
$$

**Fig. 7.** $\underline{\mathrm{mi}}$-constraints, specifications, and program.

For a $C_j$-program $P$ and an object $o \in O$ with class $C_j$, the projection $P \downarrow_o O$ is obtained by translating all the clauses of $P$ in the above way. $P \downarrow_o O$ will be called the *o-program* corresponding to $P$.

Fig. 7 shows the $\underline{mi}$-program corresponding to the *path* program of the class *Town* of Fig. 6.

The purpose of the $\downarrow_o O$-projection is not to model the distributed computations of a population of cooperating objects, but to give an *object-level semantics* for such computations, while abstracting away from implementation details. With regard to the computation model, we follow the general OO approach, where programs of an object $o$ are just *o-copies* of class programs. An $o$-copy simply replaces *this* by $o$, as shown in Fig. 8.

$$
\begin{array}{ll}
\textbf{Object } \underline{mi} : Town; & \\
\text{STATE:} & \underline{mi}.road(\underline{l1}) \leftarrow; \\
\text{PROGS:} & \underline{mi}.path([\,]) \quad \leftarrow \\
 & \underline{mi}.path([X|L]) \leftarrow \underline{mi}.road(X), Road(X), X.path(L);
\end{array}
$$

**Fig. 8.** The object $\underline{mi}$ and its method.

To model the OO view of computations, we have to model them as collaborations, where each object uses its current state axioms to solve local constraints and (possibly) evaluate (local or remote) addresses of other objects, in order to send them messages. We will consider the simpler case, where state axioms can be represented as sets of Prolog facts, or more precisely, the Clark's completion of these facts. In Fig. 8, the state axiom is represented by the fact $\underline{mi}.road(\underline{l1})$, which implicitly stands for its completion $\forall x : Obj\,\textbf{.}\; \underline{mi}.road(x) \leftrightarrow x = \underline{l1}$. The $\underline{mi}.path([X|L])$-call locally solves $\underline{mi}.road(X)$ and gets the address $X = \underline{l1}$; by the call $Road(\underline{l1})$, the object $\underline{l1}$ is asked for its class; if $\underline{l1}$ answers *Road*, then $Road(\underline{l1})$ succeeds and the message $path(L)$ is sent to it; otherwise $Road(\underline{l1})$ fails. As we can see, this behaviour is equivalent to that of the $\downarrow_o O$-projection of Fig. 7, when the population is the one considered there, independently of the current state of the objects. The object-level semantics of the collaboration is given collectively by the $o$-programs of all the objects $o$ of the population, shown in Fig. 9.

$$
\begin{array}{ll}
\underline{mi}.path([\,]) & \leftarrow \\
\underline{mi}.path([\underline{l1}|L]) & \leftarrow \underline{mi}.road(\underline{l1}), \underline{l1}.path(L); \\
\underline{mi}.path([\underline{l2}|L]) & \leftarrow \underline{mi}.road(\underline{l2}), \underline{l2}.path(L); \\
\underline{mn}.path([\,]) & \leftarrow \\
\underline{mn}.path([\underline{l1}|L]) & \leftarrow \underline{mn}.road(\underline{l1}), \underline{l1}.path(L); \\
\underline{mn}.path([\underline{l2}|L]) & \leftarrow \underline{mn}.road(\underline{l2}), \underline{l2}.path(L); \\
\underline{l1}.path([\underline{l1}|L]) & \leftarrow \underline{mi} = \underline{l1}.target, \underline{mi}.path(L); \\
\underline{l1}.path([\underline{l1}|L]) & \leftarrow \underline{mn} = \underline{l1}.target, \underline{mn}.path(L); \\
\underline{l2}.path([\underline{l2}|L]) & \leftarrow \underline{mi} = \underline{l2}.target, \underline{mi}.path(L); \\
\underline{l2}.path([\underline{l2}|L]) & \leftarrow \underline{mn} = \underline{l2}.target, \underline{mn}.path(L);
\end{array}
$$

**Fig. 9.** The object-level semantics of the collaboration $\{\underline{mi}, \underline{mn}, \underline{l1}, \underline{l2}\}$.

This example explains our choice of considering $\downarrow_o O$-projections as the intended object-level semantics of programs and justifies Definition 4 below.

**Definition 4 (Collaboration Program).** *Let $\mathcal{S}[\mathcal{D}, C_1, \ldots, C_n]$ be an OO system and $O$ be a population for it. The collaboration program of $O$ is the union of the o-programs, for the objects o of the population.*

To conclude, we show how the $\downarrow_o O$-projections allow us to precisely define a model-theoretic semantics for OO systems and their correctness.

Let $\mathcal{S}[\mathcal{D}, C_1, \ldots, C_n]$ be an OO system, and $O$ be a population for it with class axioms $CAx(O)$ and population signature $\Sigma_O$.

We do not bother to represent states by state axioms, but instead we define system states in a more abstract, purely model-theoretic way:[3]

**Definition 5 ((Model-theoretic) System States).** *A $\Sigma(O)$-interpretation $i_O$ is a* system state *with population $O$ iff $i_O \models CAx(O)$.*

We define constraint satisfaction as follows:

**Definition 6 (Constraint Satisfaction).** *A $C_j$-constraint $K$ is satisfied by a system state $i_O$ with population $O$ if, for every object $o \in O$ with class $C_j$, $i_O \models K \downarrow_o O$.*

We define the object models as the system states that satisfy the constraints:

**Definition 7 (Object Models).** *$i_O$ is an object model of $\mathcal{S}[\mathcal{D}, C_1, \ldots, C_n]$ with population $O$ iff it satisfies all the class constraints of $\mathcal{S}$.*

With respect to UML, OO systems formalise class diagrams, and their object models formalise the object diagrams that are instances of the class diagrams which satisfy the constraints.

[3] System states defined by state axioms become particular cases.

Now we can define program correctness. The specifications of the collaboration program of $O$ are the $o$-specifications of the various classes of the system. They are standard first-order $\Sigma(O)$-formulas, and we can use the definition of correctness in a class of interpretations, as given in [10].

**Definition 8 (Program Correctness).** *The programs of $\mathcal{S}[\mathcal{D}, C_1, \ldots, C_n]$ are* correct *in a population $O$ if the collaboration program of $O$ is correct with respect to its specifications in the class of object models of $\mathcal{S}$ with population $O$.*

We remark that correctness is a collective property of the population $O$, i.e., it is a property of the collaboration among the objects of $O$.

We define correct OO systems as follows:

**Definition 9 (Correct OO systems).** *An OO system $\mathcal{S}[\mathcal{D}, C_1, \ldots, C_n]$ is* correct *if its programs are correct in every population for it.*

### 4.2 The Global Level

The proof methods for program correctness given in [10] apply directly, if we consider a fixed population. However, we cannot reason thus in general, for *any* population, because different populations generate different signatures and projections. Thus we cannot reason about the correctness of OO systems and about other properties like specification reduction. The global level allows us to overcome this difficulty. It is defined as follows.

We associate with an OO system $\mathcal{S}[\mathcal{D}, C_1, \ldots, C_n]$ a global signature $\Sigma_G$, where the dot notation is replaced by an extra argument of sort $Obj$, namely:

(a) for every class $C_i$ and every relation declaration $.r : a$ of $C_i$, $\Sigma_G$ contains the declaration $r : [Obj|a]$;
(b) for every class $C_i$ and every function declaration $.f : a \to s$ of $C_i$, $\Sigma_G$ contains the declaration $f : [Obj|a] \to s$.

We assume that the *same* attribute or program predicate does *not* occur in two different classes. Since in our example of *Roads and Towns*, the program predicate *path* occurs in both *Town* and in *Road*, we assume the following class renaming: *path* is implicitly renamed into $path_{Town}$, in class *Town*, and into $path_{Road}$, in class *Road*. We remark that class renaming is not needed at the object level, because dot renaming suffices there.

Under the above assumption, the global signature is the union of all the class signatures, transformed by (a) and (b) above.

The same transformation maps the program signature of the various classes into the global program signature: a program predicate declaration $.p : [a]$ is mapped into $p : [Obj|a]$.

In global formulas, class bound quantifiers are interpreted as the following abbreviations:

$$[\forall x : C(x)].\ F(x) \quad \text{abbreviates} \quad \forall x : Obj.\ C(x) \to F(x)$$
$$[\exists x : C(x)].\ F(x) \quad \text{abbreviates} \quad \exists x : Obj.\ C(x) \land F(x)$$

Now we can translate constraints, specifications and programs at the system level into global formulas. This translation will be called the *global projection*, and will be indicated by $\uparrow$.

For a class $C_j$ and a constraint $K$ of $C_j$, the projection $F \uparrow$ is defined by the following steps.

(i) If $K$ contains $this$, then $K$ is replaced by $[\forall x : C_j(x)]\,.\ F[this/x]$, where $x$ is a new variable of sort $Obj$.
(ii) In the formula resulting from (i), the dot notation $\tau.a(\tau_1, \ldots, \tau_n)$ is treated as a syntactic variant of the standard notation $a(\tau, \tau_1, \ldots, \tau_n)$. It should be clear how to recursively translate from the dot notation into the standard one, so we omit the details.

In this way, the projection $K \uparrow$ is a $\Sigma_G$-formula. The same $\uparrow$-projection also works for specifications.

**Definition 10 (Global Constraints and Specifications).** *Let* $\mathcal{S}[\mathcal{D}, C_1, \ldots, C_n]$ *be an OO system.*

*The* global constraints *for* $\mathcal{S}$ *are all the* $\Sigma_G$*-formulas obtained as* $\uparrow$*-projections of the* $C_j$*-constraints, for* $1 \leq j \leq n$*.*

*The* global specifications *for* $\mathcal{S}$ *are all the specifications obtained as* $\uparrow$*-projections of the* $C_j$*-specifications, for* $1 \leq j \leq n$*.*

At the global level, the organisation in classes disappears, but classes survive in class axioms and in class bounded quantifiers. This situation is shown in Fig. 10, which shows the global constraints and the global specifications obtained by the $\uparrow$-projection from those of the *Roads and Towns* OO system. Fig. 10 also contains one global program. For global programs, we proceed as follows.

**Global** $(Roads\ and\ Towns) \uparrow$;

DATA: $\mathcal{LIST}(Obj)$;

DECLS: $Road, Town : [Obj]$;
$source, target : [Obj] \to Obj$;
$link : [Obj, Obj]$;

CONSTRS: $[\forall x : Road(x)]\,.\ Town(source(x)) \land Town(target(x)$;
$[\forall x : Road(x)]\,.\ [\exists! t : Town(t)]\,.\ link(t, x)$;
$[\forall x : Town(x)]\,.\ [\forall r : Road(r)]\,.\ link(x, r) \leftrightarrow source(r) = x$;

SPECS: $[\forall x : Road(x)]\,.\ path_{Road}(x, L) \leftrightarrow x@(0, L) \land (\forall o : Obj\,.\ o \in L \to Link(o)) \land$
$([\forall a : Road(a), b : Road(b)]\,.\ \forall i : Nat\,.$
$a@(i, L) \land b@(s(i), L) \to target(a) = source(b))$
$[\forall t : Town(t)]\,.\ path_{Town}(t, l) \leftrightarrow l = [\,] \lor [\exists x : Road(x)]\,.\ link(t, x) \land path(x, l)$;

PROGS: $path_{Town}(X, [\,]) \leftarrow Town(X)$
$path_{Town}(X, [X|L]) \leftarrow T = target(X), Road(X), path_{Road}(T, L)$
$path_{Road}(T, [X|L]) \leftarrow link(T, X), Town(T), Road(X), path_{Town}(X, L))$

**Fig. 10.** The global *Roads and Town* system.

A program clause of a class $C_j$ of the form $this.p(\tau) \leftarrow L, K, B$ is translated into $p(X, \tau) \leftarrow L', C_j(X), K', B'$, where $L', K', B'$ are obtained by replacing $this$ by a new variable $X$, and by passing from the dot to the standard notation. By this $\uparrow$-projection we build the global program:

**Definition 11 (The Global Program).** *Let $\mathcal{S}[\mathcal{D}, C_1, \ldots, C_n]$ be an OO system. The* global program *for $\mathcal{S}$ is the set of all clauses obtained as $\uparrow$-projections of the $C_j$-programs, for $1 \leq j \leq n$.*

Now we can define global models and global correctness. Let $\Sigma_G$ be the global signature of an OO system $\mathcal{S}[\mathcal{D}, C_1, \ldots, C_n]$. We say that a $\Sigma_G$-interpretation $\mathsf{i}_G$ has population $O$ if, for every class predicate $C_j$ and object identifier $o$, $\mathsf{i}_G \models C_j(o)$ iff $o \in O$ with class $C_j$. Global models are assumed to have finite populations.

**Definition 12 (Global Models).** *Let $\mathcal{S}[\mathcal{D}, C_1, \ldots, C_n]$ be an OO system, and $\Sigma_G$ be the corresponding global signature. A $\Sigma_G$-interpretation $\mathsf{i}_G$ is a* global model *of $\mathcal{S}$ if $\mathsf{i}_G$ has a finite population $O$, is an expansion of the data model $\mathsf{i}_D$ and satisfies the population constraints and the global constraints.*

**Definition 13 (Global Correctness).** *Let $\mathcal{S}[\mathcal{D}, C_1, \ldots, C_n]$ be an OO system, and $P_G$ be its global program. $\mathcal{S}$ is* globally correct *if $P_G$ is correct with respect to its global specifications in the class of global models.*

To conclude, we define the global framework as follows. It is just a non-OO specification framework of the kind considered in Section 2. Thus, we can apply the proof methods of [10] and we can reason about specification reduction. In the next section we will show that reasoning at the global level correctly reflects onto the object level.

**Definition 14 (Global Framework).** *The* global framework *associated with an OO system $\mathcal{S}$ is the specification framework which has the global signature of $\mathcal{S}$, and contains the data, class constraints, global constraints, global specifications and the global program of $\mathcal{S}$.*

### 4.3 Main Results

Now we introduce a semantic map $\Downarrow$ from global models to the object level models. With this map, we show that we can reason on the object level using the global level as a meta-level. The general picture is shown in Fig. 11.

The projections $\uparrow$ and $\downarrow_o O$ have already been explained. The map $\Downarrow$ is a semantic map from $\Sigma_G$- interpretations with population $O$ into $\Sigma(O)$-interpretations. As the picture shows, $\Downarrow$ is not injective.

For an OO system $\mathcal{S}[\mathcal{D}, C_1, \ldots, C_n]$, a global interpretation $\mathsf{i}_G$ with population $O$, and a $C_j$-constraint $F$, the following reflection property can be proved:

$$\mathsf{i}_G \models F \uparrow \quad \text{iff} \quad \mathsf{i}_G {\Downarrow} \models (F \downarrow_{o_1} O \wedge \cdots \wedge F \downarrow_{o_k} O)$$

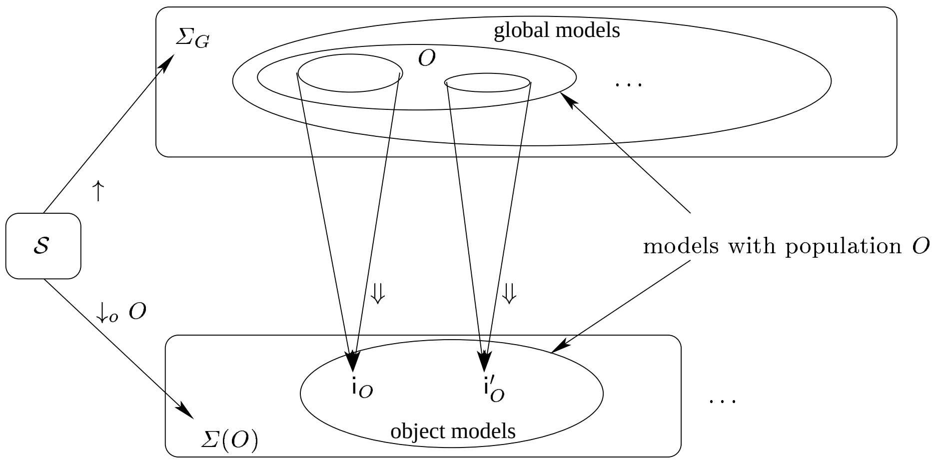


**Fig. 11.** The links among the three levels.

where $o_1, \ldots, o_k$ are the objects of $O$ with class $C_j$. That is, constraints that hold at the global level reflect onto constraints that hold at the object level. Similar reflection properties hold for the specifications and the intended models of programs. In particular, this allows us to show that we can prove correctness at the global level, so as to guarantee it at the object level.

Now we define $\Downarrow$. We will adopt the following notation:

- $\mathsf{i}$, $\mathsf{j}$, ... will denote interpretations;
- $\sigma^{\mathsf{i}}$, $\sigma^{\mathsf{j}}$, ..., will indicate the interpretations of $\sigma$ in $\mathsf{i}$, $\mathsf{j}$, ...;
- $\mathcal{S}[\mathcal{D}, C_1, \ldots, C_n]$ will be an OO system with global signature $\Sigma_G$, and $O$ a population for it with class axioms $CAx(O)$ and signature $\Sigma(O)$.

**Definition 15 (The $\Downarrow$ map).** *Let $\mathsf{i}_G$ be a $\Sigma_G$-interpretation such that $\mathsf{i}_G \models CAx(O)$. Then $\mathsf{i}_G\Downarrow$ is the $\Sigma(O)$ interpretation such that, for every object $o \in O$:*

- *for every relation attribute $o.a$ of $o$, $\langle \alpha_1, \ldots, \alpha_n \rangle \in o.a^{\mathsf{i}_G\Downarrow}$ iff $\langle o, \alpha_1, \ldots, \alpha_n \rangle \in a^{\mathsf{i}_G}$, and*
- *for every function attribute $o.f$ of $o$, $o.f^{\mathsf{i}_G\Downarrow}(\alpha_1, \ldots, \alpha_n) = f^{\mathsf{i}_G}(o, \alpha_1, \ldots, \alpha_n)$.*

**Theorem 1 (Constraint Reflection).** *Let $F$ be a $C_j$-constraint of $\mathcal{S}$, $\mathsf{i}_G$ be a $\Sigma_G$-interpretation such that $\mathsf{i}_G \models CAx(O)$, and $C_j(x) \leftrightarrow x = o_1 \vee \cdots \vee x = o_k$ be the class axiom for $C_j$. Then*

$$\mathsf{i}_G \models F\uparrow \quad \textit{iff} \quad \mathsf{i}_G\Downarrow \models (F \downarrow_{o_1} O \wedge \cdots \wedge F \downarrow_{o_k} O).$$

*Proof.* (Sketch) $F\uparrow$ is $[\forall x : C_j(x)]$. $F[this/x]$. Since $\mathsf{i}_G \models C_j(x) \leftrightarrow x = o_1 \vee \ldots \vee x = o_k$, $\mathsf{i}_G \models F\uparrow$ iff $\mathsf{i}_G \models F[this/o_1]\uparrow \wedge \cdots \wedge F[this/o_k]\uparrow$ (we will say that we have *$O$-eliminated* the class bounded quantifier). Thus it suffices to prove that $\mathsf{i}_G \models F[this/o_i]\uparrow$ iff $\mathsf{i}_G\Downarrow \models F\downarrow_{o_i} O$, where $1 \leq i \leq k$. To prove this, we $O$-eliminate the class bounded quantifiers of $F[this/o_i]\uparrow$. We get a formula that is structurally equivalent to $F\downarrow_{o_i} O$, and the the proof can be concluded by an induction on the structure of formulas. The base case follows from the definition of $\Downarrow$.

A reflection result also holds for interpretations involving program predicates and specifications. The definition of the $\Downarrow$-map readily extends to the interpretations of program predicates and the proof of Theorem 1 readily extends to a proof of the following one.

**Theorem 2 (Specification Reflection).** *Let $S_{this.p}$ be a $C_j$-specification, $j_G$ be a $(\Sigma_G + p)$-interpretation such that $j_G \models CAx(O)$, and $C_j(x) \leftrightarrow x = o_1 \vee \cdots \vee x = o_k$ be the class axiom for $C_j$. Then $j_G\Downarrow$ is a $(\Sigma(O) + \{o_1.p, \ldots, o_k.p\})$-interpretation and*

$$j_G \models S_{this.p}\uparrow \quad \textit{iff} \quad j_G\Downarrow \models (S_{this.p}\downarrow_{o_1} O \wedge \cdots \wedge S_{this.p}\downarrow_{o_k} O).$$

We can express system properties using the syntax introduced in Section 3.2. Since each class $C_j$ has its syntax, we will call $C_j$-formulas the constraint and specification formulas that can be expressed in $C_j$. The following theorem follows from Theorems 1 and 2.

**Theorem 3 (Reasoning Reflection).** *Let $F$ be a $C_j$-formula of $\mathcal{S}$. If $F\uparrow$ is a theorem of the global framework, then, for every population $O$, every system model $i_O$ with population $O$, and every $o \in O$ with class $C_j$, we have that $i_O \models F\downarrow_o O$.*

This theorem allows us, in particular, to reason about constraints, specifications and specification reduction, when subclassing is considered.

Concerning the reflection properties of programs, we show that the models of the global program are mapped into those of the collaboration program.

We use the j-models semantics given in [10], which we briefly recall here, but taking constraints into account in order to deal with local and class constraints.

We say that an open program $P$ has type $\delta \Leftarrow (\gamma)(\pi)$ if $\delta$ are the predicates defined by $P$, $\gamma$ the constraints occurring in it, and $\pi$ its open predicates. Constraints $\gamma$ are assumed to be computed by a constraint solver, while open predicates $\pi$ have specifications, but no programs; $\pi$ allow reuse through specification reduction, as explained in Section 2. By $P : \delta \Leftarrow (\gamma)(\pi)$ we indicate that $P$ has type $\delta \Leftarrow (\gamma)(\pi)$.

Let $\Gamma$ be a signature that contains the signature of $P : \delta \Leftarrow (\gamma)(\pi)$, except the defined predicates $\delta$. We call a $\Gamma$-interpretation a *pre-interpretation* for $P : \delta \Leftarrow (\gamma)(\pi)$.[4] For a pre-interpretation $\mathsf{j}$, we define the *j-models* as the $(\Gamma + \delta)$-expansions of $\mathsf{j}$ that are models of $P$.

We can show (see [10]) that there is the *minimum j-model of $P$*, indicated by $= \mathsf{j}^P$. Intuitively, $\mathsf{j}^P$ corresponds to the success set of $P$ when computations are

[4] This definition of pre-interpretation extends that in [11].

performed in accordance with j. We can consider j as an interpretation of the constraints according to a specific constraint solver, and of the open predicates according to a specific reuse. Thus, j-models can be assumed as the semantics of open programs with constraints.

Now, let $\mathcal{S}$ be an OO system with global signature $\Sigma_G$, and $O$ be a population for it. The global and the collaboration programs have the following features.

The global program $GP$ has type:

$$GP : \delta_G \Leftarrow (\lambda_G, \gamma), (\pi_G)$$

where the defined predicates $\delta_G$ coincide with the (global-level) predicates that are defined in at least one class, the open predicates $\pi_G$ are the (global-level) program predicates that are not defined by any class, $\lambda_G$ are the the local constraint symbols and $\gamma$ the class constraint symbols used in $GP$.

We say that the set $\{o_1.a, \ldots, o_k.a\}$ is the $O$-expansion of the $C_j$-attribute $.a$ declared in $C_j$, if $o_1, \ldots, o_k$ are the objects of $O$ with class $C_j$. The collaboration program $CP_O$ of the population $O$ has type:

$$CP_O : \delta_O \Leftarrow (\lambda_O), (\pi_O)$$

where the defined predicates $\delta_O$ coincide with the (union of) the $O$-expansions of the predicates defined in at least one class $C_j$, the local constraint symbols $\lambda_O$ are the $O$-expansions of the local constraint symbols used in at least one class program, and the open predicates $\pi_O$ coincide with the $O$-expansions of the program predicates not defined by the classes of $\mathcal{S}$. There are no class constraints in $CP$, because they are eliminated by the $\downarrow_o O$ projections.

As an example, the global program of the *Road and Towns* system has type

$$GP : path_{Road}, path_{Town} \Leftarrow (target, link, Road, Town)();$$

it has no open predicates. The collaboration program with population $\underline{mi}, \underline{mn}, \underline{l1}, \underline{l2}$ has type

$$CP : \underline{l1}.path, \underline{l2}.path, \underline{mi}.path, \underline{mn}.path \Leftarrow (\underline{l1}.target, \underline{l2}.target, \underline{mi}.link, \underline{mn}.link)().$$

We can see that the $\Downarrow$-map, extended to program predicates, maps $(\Sigma_G+\pi_G)$-interpretations into $(\Sigma(O)+\pi_O)$-interpretations and $(\Sigma_G+\pi_G+\delta_G)$-interpretations into $(\Sigma(O) + \pi_O + \delta_O)$-interpretations.

The following program reflection theorem holds:

**Theorem 4 (Program Models Reflection).** . *Let* $GP : \delta_G \Leftarrow (\lambda_G, \gamma), (\pi_G)$ *be the global program of an OO system* $\mathcal{S}$*, and* $CP_O : \delta_O \Leftarrow (\lambda_O), (\pi_O)$ *be the collaboration program with population* $O$*. Let* $j$ *be a* $(\Sigma + \pi_G)$*-interpretation such that* $j \models CAx(O)$*. Then*

$$(j^{GP})\Downarrow = (j\Downarrow)^{CP_O}.$$

*Proof.* A detailed proof would require much material from [10]. Here we say only that the ideas behind the proof are very similar to those used in partial evaluation. Indeed, by the definitions of the $\uparrow$ and $\downarrow_o O$ projections, $CP_O$ can be obtained from $GP$ by a kind of partial evaluation of the class constraints of $GP$ using the class axioms $CAx(O)$.

The following theorem can be proved as a consequence of Theorems 2 and 4. It allows us to reason about correctness.

**Theorem 5 (Correctness Reflection).** *$\mathcal{S}$ is correct iff it is globally correct.*

## 5 Conclusion

We have defined system correctness for OO systems, namely for (possibly) distributed sets of cooperating objects. We have introduced global frameworks as a meta-level to deal with properties of OO systems, in particular to deal with their *a priori correctness*, namely pre-proved correctness that holds in all the system states. We have also shown how we can give a semantics for UML object and class diagrams with constraints.

This is a sound basis for introducing correctly reusable OO modules. They can be formalised as open OO systems, namely OO systems where the collaboration and global programs are open, because the collaboration is assumed to be with generic classes, that we call templates. A template $T$, like a class, has constraints, and specifies program predicates $\pi$ by non-strict specifications $S_\pi$, but it does not contain programs for them. $T$ works as a placeholder for classes which contain the programs for $\pi$. A class $C$ correctly instantiates $T$ if the signature of $C$ contains (modulo renaming) that of $T$, the constraints of $C$ allow us to prove (at the global level, see Theorem 3) those of $T$, and the specifications $S'_\pi$ of $C$ correctly override $S_\pi$ (also at the global level). Different correct instantiations correspond to different correct reuses.

We can deal with open OO systems without any modification of our formalisation, because we can deal with open programs. As the global framework is an open specification framework, we have correct reusability with interfaces and context dependencies, as discussed in Section 2. Thus, an open OO system represents a reusable unit like an OOD framework.

Open OO systems allow us to formalise OOD frameworks, such as *Catalysis* frameworks [5], with one proviso: we have considered only static programs, namely programs that do not change the current state of an OO system. That is we can only deal with correctness of query methods, in any given state of the system. Methods (programs) that change the current state are allowed, but we do not yet have a formalisation of their *a priori correctness*. Our next step will be to introduce timed objects, and to deal with the correctness of dynamic programs.

Our approach is model-theoretic, so it is different from proof-theoretic or type theoretic approaches [1, 12]. It is closer to other model-theoretic approaches, like the algebraic ones [2]. In particular our treatment is similar to that of [6, 3]. However, our work extends the algebraic methods since, by the architecture of specification frameworks, it allows us to introduce pre-proved correctness, even at the level of distributed systems of objects.

## References


1. M. Abadi and L. Cardelli. *A Theory of Objects.* MIT Press, 1985.
2. E. Astesiano, H.-J. Kreowski, and B. Krieg-Brückner, editors. *Algebraic Foundations of Systems Specifications.* Springer, 1999.
3. R.H. Bourdeau and B. H.C. Cheng. A formal semantics for object model diagrams. *IEEE Trans. Soft. Eng.*, 21(10):799–821, 1995.
4. J. Cheesman and J. Daniels. *UML Components: A Simple Process for Specifying Component-based Software.* Addison-Wesley, 2001.
5. D.F. D'Souza and A.C. Wills. *Objects, Components, and Frameworks with UML: The Catalysis Approach.* Addison-Wesley, 1999.
6. H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 2.* Springer-Verlag, 1989.
7. G. Larsen. Designing component-based frameworks using patterns in the UML. *Comms. ACM*, 42(10):38–45, October 1999.
8. K.-K. Lau and M. Ornaghi. OOD frameworks in component-based software development in computational logic. In P. Flener, editor, *Proc. LOPSTR 98, Lecture Notes in Computer Science 1559*, pages 101–123. Springer-Verlag, 1999.
9. K.-K. Lau and M. Ornaghi. Isoinitial semantics for logic programs. In J.W. Lloyd *et al*, editor, *Proceedings of the First Int. Conf. on Computational Logic, Lecture Notes in Artificial Intelligence 1861*, pages 223–238. Springer-Verlag, 2000.
10. K.-K. Lau, M. Ornaghi, and S.-Å. Tärnlund. Steadfast logic programs. *J. Logic Programming*, 38(3):259–294, March 1999.
11. J.W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, second edition, 1987.
12. D. Miller. Forum: A multiple-conclusion specification logic. *TCS*, 165(1):201–231, 1996.
13. J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual.* Addison-Wesley, 1999.
14. J. Warmer and A. Kleppe. *The Object Constraint Language.* Addison-Wesley, 1999.