Towards 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, we have defined *steadfastness*, a notion of correctness that captures at once modularity, correctness and reusability. This paper attempts to extend 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

In our previous work, we have defined *steadfastness* [12], a notion of correctness that captures at once modularity, correctness and reusability. Correct reusability is one of the main features of what we called *specification frameworks* [11]. These are modules that specify and implement collections of steadfast (logic) programs.

In this abstract, we present work in progress for the first stage of a formalisation of Object-Oriented Design (OOD) frameworks. OOD frameworks are increasingly recognised as more reusable than single objects, and promise to be useful components in next-generation Component-based Software Development [14, 5]. Our approach is model-theoretic. In this, it is similar to algebraic approaches like [6, 4].

In [9, 7], we used a special version of specification frameworks to formalise OOD frameworks. However, we concentrated on the specification level, and disregarded correctness.

To address correctness, we are faced with two main difficulties:

(i) the proof-theoretic results of [12] do not apply directly, due to the distributed nature of OO systems;
(ii) our previous approach to correctness does not deal with dynamic methods, i.e., methods that change the current state of objects.

This paper represents only a first step towards correct OOD frameworks, since we do not yet address dynamic methods. Therefore o avoid confusion, we use *Object-Oriented Systems* to refer to the static version of OOD frameworks that we consider here.

In Section 2 we recall our previous results, by introducing specification frameworks and steadfastness. In Section 3 we formalise OO Systems and their correctness. Finally, in the conclusion, we briefly discuss related approaches and the future extension of our results to (dynamic) OOD frameworks.

2 Specification Frameworks and Correctness

Specification frameworks are the basis of our formalisation of OO Systems, and we recall them in this section. They are based on a three-tier formalism (with model-theoretic semantics) illustrated in Figure 1. At the bottom level, we have *programs*, for

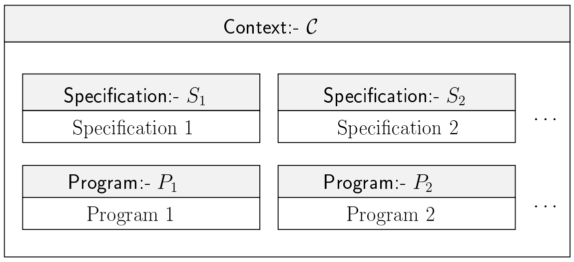

Fig. 1. A specification framework.

computing (specified) relations. Programs are standard or constraint logic programs. In the middle, we have *specifications* for defining or specifying new relations (and functions). At the top, we have the *meta-level* or *context*, that embodies the relevant knowledge of the problem domain.

We formalise a context as a first-order theory $\mathcal{T} = \langle \Sigma, Ax \rangle$. Σ is the *context signature* and Ax are the*context axioms* and axiomatise the needed data types and the problem domain. We distinguish between *closed* and *open* contexts.

A context is *closed* iff it has a reachable isoinitial model (see [9] for a justification of this choice and [3, 10] for a discussion of isoinitial theories).

Example 1. An example of a closed context is first-order arithmetic $\mathcal{NAT} = \langle \Sigma_{PA}, PA \rangle$. Σ_{PA} contains the unary function s (successor) and the binary functions $+$ (sum) and $*$ (product). PA contains the usual Peano's axioms for $s, +, *$ and all the instances of the first order induction schema. The standard structure $\mathcal{N}$ of natural numbers is a reachable isoinitial model of $\mathcal{NAT}$.

An open context $\mathcal{T}(\Pi) = \langle \Sigma, Ax \rangle$ has a set Π of *open symbols*. It can be *closed*, i.e. made into a closed context, by *instantiating* the symbols of Π by new axioms, that fix their meaning. The new axioms are called *closure axioms*, and a set of closure axioms is called a *closure*. An open context may contain suitable *constraints* (see

[9]), to be satisfied by its closures. We require that constrained contexts are *adequate*, where adequacy entails that any closure that satisfies the constraints gives rise to a *closed* context.

Example 2. The following open context axiomatises lists with generic elements X and a generic total ordering $\lhd$ on X.

$$
\begin{array}{lrl}
\multicolumn{3}{l}{\textbf{Context } \mathcal{LIST}(X, \lhd : [X, X]);} \\
\text{IMPORT:} & \multicolumn{2}{l}{\mathcal{NAT};} \\
\text{DECLS:} & \multicolumn{2}{l}{X, ListX : sort;} \\
& nil : & [\,] \rightarrow ListX; \\
& . : & [X, ListX] \rightarrow ListX; \\
& \_@(\_,\_) : & [X, Nat, ListX]; \\
& nocc : & [X, ListX] \rightarrow Nat; \\
\text{AXIOMS:} & CA : & constr(nil, .); \\
& @ : & x@(0, L) \leftrightarrow \exists y, J(L = y.J \wedge x = y); \\
& & x@(si, L) \leftrightarrow \exists y, J(L = y.J \wedge x@(i, J)); \\
& nocc : & nocc(x, nil) = 0; \\
& & x = y \rightarrow nocc(x, y.L) = nocc(x, L) + 1; \\
& & \neg x = y \rightarrow nocc(x, y.L) = nocc(x, L); \\
\text{CONSTRS:} & \multicolumn{2}{l}{TO(\lhd).}
\end{array}
$$

Fig. 2. An open context.

The signature Σ_{PA} and the axioms PA are imported. nil and . are the *list constructors.* The corresponding *constructor axioms* $constr(nil, .)$ contain Clark's equality theory as well as structural induction for . and nil. $x@(i, L)$ means that the element x occurs at position i in the list L, where positions start from 0. $nocc(x, L)$ is the number of occurrences of the element x in the list L.

By $TO(\lhd)$ we indicate the total ordering axioms for $\lhd$. The parameter $\lhd$ and its constraints $TO(\lhd)$ allow us to extend the framework by adding commonly used functions and predicates on ordered lists, and to reason about them; for example, we can define lexicographical ordering on lists and prove that it is a total ordering.

A closure can be obtained by closing X by Nat and $\lhd$ by the explicit definition $\forall x, y(x \lhd y \leftrightarrow x \leq y)$. This closure satisfies constraints, because $TO(\lhd)$ can be proved in $PA \cup \forall x, y\,.\ (x \lhd y \leftrightarrow x \leq y)$. We have closed X and $\lhd$ within $\mathcal{LIST}$ itself, for conciseness. Of course, they could also be closed within another context, after importing (reusing) $\mathcal{LIST}$ in it.

In a specification framework, the main purpose of the context is to axiomatise the problem domain and to reason about it, and to provide an unambiguous semantic underpinning for specifications and programs, as well as for the correctness relationship between them. Specifications and correctness are defined as follows.

Definition 1 (Specification). In a framework $\mathcal{F}$ with context $\mathcal{T}$, a *specification* S_δ is a set of sentences that define new function or relation symbols δ in terms of the symbols Σ of the context $\mathcal{T}$.

S_δ is interpreted as an expansion operator. It associates every model m of the context $\mathcal{T}$ with the set of its S_δ-*expansions*, namely the set of the expansions of m[1] that are models of S_δ. A specification S_δ is *strict* in a context $\mathcal{T}$, if, for every model m of $\mathcal{T}$, there is one S_δ-expansion. It is *non-strict* otherwise. For examples of strict and non-strict specifications, see Example 3.

Now we provide a model-theoretic characterisation of *program correctness* in the simplified case of pure logic programs. The extension to constraint logic programs is omitted for conciseness.

A relation is *defined* by a program P if and only if it occurs in the head of at least one clause of P. It is *open* if it is not defined (by P). A program P with defined symbols δ and open symbols π will be indicated by $P_\delta(\pi)$. $P_\delta(\pi)$ is *open* if π is not empty, otherwise it is *closed*, and is indicated by P_δ. A framework with context $\mathcal{T}(\Pi)$, specifications $\mathbf{S}$ and programs $\mathbf{P}$ is indicated by $\mathcal{F}(\Pi) = \langle \mathcal{T}(\Pi), \mathbf{S}, \mathbf{P} \rangle$. We will assume that each program relation r occurring in $\mathbf{P}$ has a specification $S_r \in \mathbf{S}$. Program specifications are defined as follows.

Definition 2 (Program Specification). The specification of a program $P_\delta(\pi)$ in a framework $\mathcal{F}(\Pi) = \langle \mathcal{T}(\Pi), \mathbf{S}, \mathbf{P} \rangle$ is the pair (S_δ, S_π), such that S_δ are the specifications for the *defined predicates* δ and S_π are the specifications for the *open predicates* π.

In [12] we introduced *minimum* j-*models*, together with the notion of *steadfastness*, to serve as the basis for a model and proof theory of the correctness of open programs. Here we first recall the definition of steadfastness, and then define correctness of open programs.

For simplicity, we consider open programs $P_r(\pi)$, with one defined predicate r and a set π of open predicates. An open program $P_r(\pi)$ has a signature $\Pi + r$, where Π contains the open predicates π and the (possible) function and constant symbols of $P_r(\pi)$. A pre-interpretation j is a Π-interpretation, i.e., symbols of Π are considered to be open, and a pre-interpretation j fixes their meaning arbitrarily, while the intended meaning of r is stated by the clauses of P, in terms of j. To define the intended meaning of r in a pre-interpretation j, we introduce j-models. A j-model of $P_r(\pi)$ is a model m of $P_r(\pi)$, such that $\mathsf{m} \mid \Pi = \mathsf{j}$, i.e., m coincides with j over Π. Since two distinct j-models m and n differ only for the interpretation of r, we can compare them by looking at r: we say that m is contained in n, written $\mathsf{m} \subseteq_r \mathsf{n}$, iff the interpretation of r in m is contained in that of r in n. The minimum j-model (wrt $\subseteq_r$) always exists and we can define steadfastness in an interpretation as follows.

Definition 3 (Steadfastness). Let $P_r(\pi)$ be an open program with signature $\Pi + r$. $P_r(\pi)$ is *steadfast* in a $(\Pi + r)$-interpretation i if and only if its minimum $\mathsf{i} \mid \Pi$-model coincides with i.

[1] In an expansion of a Σ-interpretation m, Σ is interpreted as in m.

More intuitively, steadfastness in i means that the interpretation of r in i coincides with the interpretation of r stated by $P_r(\pi)$, when the open symbols Π are interpreted as in i (i.e., when the pre-interpretation is $\mathsf{i} \mid \Pi$). Consider for example the open program $P_r(p): \quad r(x) \leftarrow p(z,x)$, where x and z are of sort Nat, and consider the interpretation i where Nat is the set of natural numbers, $r(x)$ means "*x is even*", and $p(z,x)$ means "$z+z=x$". The open subsignature Π of this program contains the sort Nat and the predicate p. If we interpret Nat and p as in i, we can easily see that the interpretation of $r(x)$ in the corresponding minimum model of $P_r(p)$ coincides with the interpretation of $r(x)$ in i, i.e., $P_r(p)$ is steadfast in i.

Definition 4 (Correctness of Open Programs). Let $\mathcal{F}(\Pi) = \langle \mathcal{T}(\Pi), \mathbf{S}, \mathbf{P} \rangle$ be a framework and i be a model of $\mathcal{T}(\Pi)$. Let $P_r(\pi)$ be an open program with specifications (S_r, S_π). $P_r(\pi)$ is *correct* in $\mathcal{F}$ with respect to i and (S_r, S_π) iff, for every every S_π-expansion i_π of i, there is a S_r-expansion $\mathsf{i}_{\pi,r}$ of i_π, such that $P_r(\pi)$ is steadfast in $\mathsf{i}_{\pi,r}$.

Definition 4 readily extends to programs with many defined predicates.

In [12], we have introduced some proof methods for correctness, which hold in the class of reachable isoinitial models of the closures $\hat{\mathcal{F}}$ of (the context of) an open framework $\mathcal{F}$. We have the following important properties of correct reusability:

Proposition 1 (Inheritance). If the correctness of a program $P_\delta(\pi)$ (wrt (S_δ, S_π)) can be proved in an adequate open framework $\mathcal{F}(\Pi)$, then $P_\delta(\pi)$ is correct (wrt (S_δ, S_π)) in the isoinitial models of all the closures $\hat{\mathcal{F}}$.

Proposition 2 (Compositionality). If $P_{\delta_1}(\pi_1 \cup \delta_2)$ is correct in a framework $\mathcal{F}$ wrt a specification $(S_{\delta_1}, S_{\pi_1 \cup \delta_2})$, and $Q_{\delta_2}(\pi_2)$ is is correct in $\mathcal{F}$ wrt $(S_{\delta_2}, S_{\pi_2})$, then $(P \cup Q)_{\delta_1 \cup \delta_2}(\pi_1 \cup \pi_2)$ is correct in $\mathcal{F}$ wrt $(S_{\delta_1 \cup \delta_2}, S_{\pi_1})$.

Example 3. In a framework with context $\mathcal{LIST}(X, \lhd)$, we can have the following open program $P_{sort}(split, merge)$:

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

for sorting, with the following specifications:[2]

$$\begin{array}{lll} S_{sort} & : & sort(X,Y) \leftrightarrow perm(X,Y) \wedge ord(Y) \\ S_{merge} & : & ord(X) \wedge ord(Y) \rightarrow (merge(X,Y,Z) \leftrightarrow ord(Z) \wedge perm(X|Y,Z)) \\ S_{split} & : & l(X) > 1 \wedge split(X,Y,Z) \rightarrow perm(X,Y|Z) \wedge l(Y) < l(X) \wedge l(Z) < l(X) \\ & & l(X) > 1 \rightarrow \exists Y,Z \,.\, split(X,Y,Z) \end{array}$$

S_{sort} is a strict (*if-and-only-if*) specification, while S_{merge} S_{split} are non-strict. S_{merge} is a *conditional* specification, and S_{split} is a *selector* specification (see [8]).

[2] We assume that commonly used symbols like the *concatenation* operation $|$, the *permutation* predicate $perm(X,Y)$ and the *ordered list* predicate $ord(X)$ have already been defined.

We can prove that $P_{sort}(split, merge)$ is correct with respect to the specifications $(S_{sort}, \{S_{split}, S_{merge}\})$. By inheritance and compositionality, in every instance of $\mathcal{LIST}(X, \lhd)$, it is always correct wrt the specification S_{sort}, provided that the programs for its parameters *merge* and *split* are correct wrt to their respective specifications S_{split} and S_{merge}.

This example shows that inheritance and compositionality provide *a priori* correctness of open programs in a framework. It thus corresponds to correctness of open modules in a library. It is to be contrasted with *a posteriori* correctness, i.e. correctness established by verification after program composition.

In the next section, we will show how correctness works in the context of OO Systems.

3 Correct OO Systems

In this section we introduce *OO Systems*, a static version of OOD frameworks (see the Introduction), and we show how specification frameworks can be used to deal with their correctness.

An OO system with classes $C_1, \ldots, C_n$ is indicated by $\mathcal{S}[C_1, \ldots, C_n]$. For simplicity, we assume that all the classes are based on a common context $\mathcal{P}$. We assume that $\mathcal{P}$ always contains the predefined sort Obj, generated by a denumerable set of predefined constants, to be used as *object identities*, an open *class predicate* $C_j : [Obj]$ for every class C_j of $\mathcal{S}$, and the predicate $Live : [Obj]$ defined by $Live(x) \leftrightarrow C_1(x) \vee \ldots \vee C_n(x)$.

Classes are formalised as in [9] (where they are called OBJ-frameworks). A class C implicitly includes the context signature, and introduces a set of dotted *open* symbols, that we call *attributes*. Through the dot notation, C works as a blueprint for generating objects of class C: for each attribute $.a$ of C and each object of class C with identity o, $o.a$ is an *object attribute* of the *object signature* Σ_o of o. Thus each object signature includes the (same) context signature, and signatures of distinct objects have disjoint attributes. A class C also contains axioms, specifications and programs. Context axioms are implicitly included, and C may add specific class axioms, mainly constraints for the attributes. As in [9], object states can be finitely represented by *state axioms*. Here we are not interested in concrete representations of the states of an object o, and we can identify states with the Σ_o-interpretations: for a state (i.e., interpretation) i, the value of an attribute $o.a$ is its interpretation in i.

OO systems are blueprints for systems of cooperating objects, and the states of such a system can be represented by the system interpretations. A system signature depends on the set O of the current live objects, where each object of O belongs to a class C of the OO system.

Definition 5 (System Signature). Let O be a set of live objects for an OO system $\mathcal{S}[C_1, \ldots, C_n]$. The corresponding *system signature* $\Sigma(O)$ contains the context signature and, for every class C_j, all the object attributes $o.a$ such that o is a live object of class C_j and $.a$ is an attribute of C_j.

The *state* of a set O of cooperating objects is represented by a system interpretation, defined as follows:

Definition 6 (System Interpretation). Let O be a set of live objects for an OO system $\mathcal{S}[C_1, \ldots, C_n]$. A *system interpretation* with live objects O is a $\Sigma(O)$-interpretation $\mathfrak{i}_S$ such that $\mathfrak{i}_S \models C_j(o)$ if and only if o is a live object of O with class C_j.

Example 4. Let us consider a set of towns connected by links. A link may be a route, or a flight connection, and so on. Every link has one source and one target town. We are interested in a framework for computing paths, i.e., sequences of links such that the target of a link coincides with the source of the subsequent one. We assume that each town t directly knows only the links with source t, and that the computation of paths is obtained by a possibly distributed collaboration of the various towns. A corresponding OO system can be defined as follows:

IMPORT $\mathcal{LIST}(Obj), \ldots$

Class *Link*;

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

CONSTRS: $Town(this.source) \land Town(this.target)$

SPECS: $this.path([L]) \leftrightarrow this@(0, L) \land \forall o(o \in L \to Link(o)) \land$
$[\forall a : Link(a)][\forall b : Link(b)]\forall i($
$a@(i, L) \land b@(s(i), L) \to a.target = b.source)$

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

Class *Town*;

ATTRIBUTES: $.link : [Obj];$

CONSTRS: $[\forall o : Live(o)](this.link(o) \to Link(o) \land o.origin = this);$

SPECS: $this.pathT(l) \leftrightarrow l = [\,] \lor [\exists x : Link(x)](this.link(x) \land x.path(l))$

PROGS: $this.pathT([\,]) \quad \leftarrow$
$this.pathT([X|L]) \leftarrow Link(X), this.link(X), X.path(L))$

The first statements define the Context. It contains the data types required, and, implicitly, the sort Obj, its constants and the class predicates $Link, Town : [Obj]$.

Link introduces the attributes *.source* and *.target*, representing the source and the target of a link, which must be of class *Town* by the constraints.

Town introduces the attributes *.link*; for a town object t, $t.link(l)$ means that l is a link with source t, as stated by the constraints. This corresponds to our assumption that each town knows its outgoing links.

Let O be the set of live objects containing two towns mi, mn and two links $l1$,$l2$. $\Sigma(O)$ contains the object attributes:

$$l1.source, l1.target, l2.source, l2.target : [\,] \to Obj;$$
$$mi.link, mn.link : [Obj]$$

A system interpretation with live objects O may be, for example, the interpretation $\mathfrak{i}_{e4}$ where $l1.source = mi$, $l1.target = mn$, $l2.source = mi$, $l2.target = mn$, mi has links $l1, l2$ (i.e., $mi.link(l)$ is true for $l = l1$ or $l = l2$), and mn has no links (i.e., $mn.link(l)$ is false for every link l). Clearly this corresponds to a state of the live objects.

Now we come to the problem of program correctness. There are two main difficulties. Firstly, we do not have a fixed signature, but as many signatures as there are sets of living objects. Secondly, we do not have a pure first-order syntax. For example, in the specification of *.path* in the class *Link* the prefixes a, b of $a.target = b.source$ are in the scope of the *bounded quantification* $[\forall a : Link(a)][\forall b : Link(b)]$. Quantified prefixes are needed to get a sufficiently expressive specification language, but they extend the syntax of first-order logic.

To overcome these difficulties, we translate extended formulas into first-order formulas of the current system signature, depending on the current system interpretation i_S.

For every class C_j, every formula F occurring in it, and every live object o of class C_j, F translates into $F \downarrow \mathsf{i}_S$, which we call the i_S*-projection* of F for o. The translation consists of three steps:

(i) In the first step, self-reference *this* is replaced by o.
(ii) In the second step, we recursively eliminate bounded quantifiers as follows:
  - $[\forall x : B(x)]F(x) \downarrow \mathsf{i}_S = F(o_1) \downarrow \mathsf{i}_S \wedge \ldots \wedge F(o_k) \downarrow \mathsf{i}_S$, where $o_1, \ldots, o_k$ are the objects such that $\mathsf{i}_S \models B(o_k)$;
  - $[\exists x : B(x)]F(x) \downarrow \mathsf{i}_S = F(o_1) \downarrow \mathsf{i}_S \vee \ldots \vee F(o_k) \downarrow \mathsf{i}_S$, where $o_1, \ldots, o_k$ are the objects such that $\mathsf{i}_S \models B(o_k)$;
  - For the propositional connectives the translation is recursive, in the obvious way.
(iii) In the third step, we evaluate the (possible) compound navigations, as follows. If a prefixed attribute $N.a$ contains a compound prefix N, we replace N by the object o_N it denotes in i_S (details in the full paper). If o_N is not a live object of a class C with an attribute $.a$, evaluation results into a *class error*.
i_S is said to be *class sound* if no type error occurs. If i_S is class sound, the projection $F \downarrow \mathsf{i}_S$ turns out to be a formula of the current system language.

A similar projection can be given for programs (details in the full paper).

Example 5. Let us consider the interpretation i_4 of Example 4. The projection $\downarrow \mathsf{i}_4$ of the specification of *.pathT* for the object mi of class $Town$ is:

$$S_{mi.pathT} : mi.pathT(l) \leftrightarrow l = [\,] \vee (mi.link(l1) \wedge l1.path(l)) \vee (mi.link(l2) \wedge l2.path(l))$$

Similarly, the program for $mi.pathT$ is the projection:

$$\begin{aligned} P_{mi.pathT} : mi.pathT([\,]) &\leftarrow \\ mi.pathT([X|L]) &\leftarrow mi.link(l1), l1.path(L) \\ mi.pathT([X|L]) &\leftarrow mi.link(l2), l2.path(L) \end{aligned}$$

Now we can define system models and program correctness in a system.

Definition 7 (System Model). A system interpretation i_S is a *system model* of an OO system $\mathcal{S} = [C_1, \ldots, C_n]$, if it is class sound, satisfies the context axioms and, for every class C_j, every axiom A of C_j, and every live object o of class C_j, $A \downarrow \mathsf{i}_S$ is true in i_S.

Definition 8 (Program Correctness in a System). Let $\mathcal{S} = [C_1, \ldots, C_n]$ be an OO system. A program P of a class C_j is *correct* in a system model i_S of $\mathcal{S}$ if, for every live object o of class C_j, the projection $P \downarrow \mathsf{i}_s$ for the object o is correct with respect to the i_S-projection for o of the specification of P.

P is *correct* in the OO system if it is correct in all its system models.

Example 6. The program $P_{mi.pathT}$ of Example 5 is correct with respect to $(\{S_{mi.pathT}\}, \{S_{l1.path}, S_{l2.path}\})$. The projection $S_{mi.pathT}$ has been given in Example 5. The projections $S_{l1.path}$, $S_{l2.path}$, for the Link objects $l1$, $l2$, are omitted for conciseness.

Here we use Definition 4 of correctness. It works, but to prove correctness we should give a separate proof in each system interpretation. To overcome this problem we introduce *global* (*specification*) *frameworks.* They have a fixed signature and serve as a metalevel where we can state and study properties of the system level.

To define the global framework associated with an OO system $\mathcal{S}$, we introduce the global signature Σ_G associated with $\mathcal{S}$.

Σ_G contains the problem signature Σ_P.

For every class C_i and every relation declaration $.r : a$ of C_i, Σ_G contains the declaration $r : [Obj|a]$.

For every class C_i and every function declaration $.f : a \to s$ of C_i, Σ_G contains the declaration $f : [Obj|a] \to s$.

Each formula F in a class C_j translates into a formula $F \uparrow$ of the global signature as follows.

If F contains $this$, it is replaced by $[\forall x : C_j(x)]F(this/x)$, where $this/x$ denotes the substitution of $this$ by a new fresh variable x. Then the following recursive translation $\uparrow$ is applied to atomic (sub)formulas and terms:

- $a(t_1, \ldots, t_n) \uparrow = a(t_1 \uparrow, \ldots, t_n \uparrow)$, for a non-dotted relation or function symbol a;
- $N.a(t_1, \ldots, t_n) \uparrow = a(N \uparrow, t_1 \uparrow, \ldots, t_n \uparrow)$, for a dotted relation or function symbol $N.a$.

A similar $\uparrow$ translation can be defined for programs (details in the full paper).

We associate each OO system $\mathcal{S}[C_1, \ldots, C_n]$ with a global framework $\mathcal{G}(\mathcal{S})$. It is a specification framework, where context axioms are those of the context of $\mathcal{S}$ and the $\uparrow$-translations of those of $C_1, \ldots, C_n$, specifications are the $\uparrow$-translations of those of $C_1, \ldots, C_n$, and programs are the $\uparrow$-translations of those of $C_1, \ldots, C_n$.

The interpretations of the global signature of $\mathcal{G}$ will be called *global interpretations*, and the *global models* are the models of the global context (i.e., the context of $\mathcal{G}$).

Each global interpretation i_G determines one system interpretation, which we will indicate by $\mathsf{i}_G \downarrow$, defined as follows:

- $\mathsf{i}_G \downarrow$ and i_G coincide over the problem signature, including the class predicates that determine the same set O of live objects.
- For every class C_j, every relation declaration $.r : [a]$ of C_j and every live object o of class C_j, $o.r : [a]$ is interpreted by $\mathsf{i}_G \downarrow$ according to i_G and the definition $o.r(x) \leftrightarrow r(o, x)$.

- For every class C_j, every function declaration $f : [a] \to s$ of C_j and every live object o of class C_j, $o.f : [a]$ is interpreted by $\mathsf{i}_G \downarrow$ according to i_G and the definition $o.f(x) = f(o, x)$.

  $\mathsf{i}_G \downarrow$ will be called the *system projection* of i_G.

Example 7. The global framework corresponding to the OO system of Example 4 is:

IMPORT $\mathcal{LIST}(Obj), \ldots$

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

CONSTRS: $[\forall x : Link(x)](Town(source(x)) \wedge Town(target(x));$
$[\forall o : Live(o)][\forall x : Town(x)](link(x, o) \to Link(o) \wedge origin(o) = x);$

SPECS: $[\forall x : Link(x)](path(x, [L]) \leftrightarrow x@(0, L) \wedge \forall o(o \in L \to Link(o)) \wedge$
$[\forall a : Link(a)][\forall b : Link(b)]\forall i($
$a@(i, L) \wedge b@(s(i), L) \to target(a) = source(b))$
$[\forall t : Town(t)](pathT(t, l) \leftrightarrow l = [\,] \vee [\exists x : Link(x)](link(t, x) \wedge path(x, l)));$

PROGS: $path(X, [X|L]) \leftarrow Link(X), T = target(X), pathT(T, L)$
$pathT(X, [\,]) \leftarrow Town(X)$
$pathT(T, [X|L]) \leftarrow Town(T), Link(X), link(T, X), path(X, L))$

A global interpretation i_G such that $inte_G \downarrow = \mathsf{i}_{e4}$ (see Example 4) is an interpretation where $source(l1) = mi$, $target(l1) = mn$, $source(l2) = mi$, $target(l2) = mn$ and $link(t, x)$ is true if $t = mi$ and $x = l1$, or $t = mi$ and $x = l2$, and is false if $t = mn$. One can see that there are many global interpretations giving rise to the same system interpretation; indeed it suffices that they coincide over the live objects.

The global framework corresponding to an OO system allows us to reason about the system framework and its system models, via the system projections of the global models. For example:

- *Class safeness.* We say that the constraints are *class safe* if, for every global model i_G of the global framework, its system projection is class sound. We can study class safeness at the global level.
- *Correctness.* We can prove the following main theorem:

  Theorem 1 (Correctness). *Correctness in a class $\mathcal{I}$ of global interpretations entails system correctness in their system projections.*

  It allows us to prove correctness at the system level by proving it at the global level, where the results of Section 2 apply. This is interesting, because the system level models correctness of distributed systems.
- *Compositionality.* We can define open OO modules as OO systems that assume cooperation with objects of external classes. An external class is a generic class that does not implement its methods and has generic specifications. We can have external classes because we can deal with open programs. An open OO module represents a reusable unit like OOD frameworks. For open OO modules we have correct reusability.

4 Discussion

We have defined correctness for OO systems, namely for (possibly) distributed sets of cooperating objects. We have introduced global frameworks as a metalevel 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. This allows us to introduce reusable OO modules.

We have considered only static programs, namely programs that do not change the current state of an OO system. This is restrictive, but nonetheless interesting, since it allows us to deal with, for example, correctness of queries in distributed OO systems.

Our approach is model-theoretic, so it is different from proof-theoretic or type-theoretic approaches [1, 13]. It is closer to other model-theoretic approaches, like the algebraic approach [2]. However, it extends algebraic methods since, by the three-tier architecture of specification frameworks, it allows us to introduce pre-proved correctness, even at the level of distributed systems of objects.

Our formalisation of static OO systems is an intermediate step towards the formalisation of (dynamic) OOD frameworks. The next step is the introduction of timed objects, namely objects with an object identity and a current time. We believe that this way we can treat dynamic aspects, while mantaining a model-theoretic notion of correctness, based on steadfastness.

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. A. Bertoni, G. Mauri, and P. Miglioli. On the power of model theory in specifying abstract data types and in capturing their recursiveness. *Fundamenta Informaticae*, VI(2):127–170, 1983.
4. R.H. Bourdeau and B. H.C. Cheng. A formal semantics for object model diagrams. *IEEE Trans. Soft. Eng.*, 21(10):799–821, 1995.
5. D.F. D'Souza and A.C. Wills. *Objects, Components, and Frameworks with UML: The Catalysis Approach.* Addison-Wesley, 1999.
6. H.-D. Ehrich. Object specification. In E. Astesiano, H.-J. Kreowski, and B. Krieg-Brückner, editors, *Algebraic Foundations of Systems Specifications*, chapter 12, pages 435–465. Springer, 1999.
7. J. Küster Filipe, K.-K. Lau, M. Ornaghi, and H. Yatsu. On dynamic aspects of OOD frameworks in component-based software development in computational logic. In A. Bossi, editor, *Proc. LOPSTR 99, Lecture Notes in Computer Science*, volume 1817, pages 43–62. Springer-Verlag, 2000.
8. K.-K. Lau and M. Ornaghi. Forms of logic specifications: A preliminary study. In J. Gallagher, editor, *Proc. LOPSTR 96, Lecture Notes in Computer Science 1207*, pages 295–312. Springer-Verlag, 1997.
9. 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.

10. 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.
11. K.-K. Lau and M. Ornaghi. Logic for component-based software development. In A. Kakas and F. Sadri, editors, *Computational Logic: From Logic Programming into the Future*. Springer-Verlag, to appear.
12. K.-K. Lau, M. Ornaghi, and S.-Å. Tärnlund. Steadfast logic programs. *J. Logic Programming*, 38(3):259–294, March 1999.
13. D. Miller. Forum: A multiple-conclusion specification logic. *TCS*, 165(1):201–231, 1996.
14. C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1998.