On Specification and Correctness of OOD Frameworks in Computational Logic

Kung-Kiu Lau
Department of Computer Science
University of Manchester
Manchester M13 9PL, UK
kung-kiu@cs.man.ac.uk

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

Abstract

In current component-based software development (CBD), it is widely recognised that the distribution of tasks between objects and the contracts between them are key to effective design. In composing designs from reusable parts, increasingly the parts are Object-oriented Design (OOD) frameworks, namely descriptions of the interactive relationships between objects which participate in the interactions. Designs are then built by composing these frameworks, and any object in the final design will play (various) roles from several frameworks. In this paper, we discuss our preliminary efforts to define a formal semantics in computational logic for the specification and correctness of OOD frameworks, and briefly illustrate it with frameworks in the CBD methodology *Catalysis*. The novelty of our approach is *a priori* correctness for OOD frameworks (and components in general) and their composition, in contrast to current development methods which are mainly in the style of *posit-and-prove*, whereby proof of correctness is done by *a posteriori* verification. For component-based software development, we argue that *a priori* correctness is a better approach than *a posteriori* correctness.

1 Introduction

At present, component-based software development (CBD) exists in the form of object-oriented software development, based on Object-Oriented Design (OOD) methods. Most of the existing formal OOD methods such as Fusion [3, 5] and Syntropy [4] use classes or objects as the basic unit of design. However, it is increasingly recognised that classes are not the best focus for design (see e.g. [8, 6]), and frameworks (see e.g. [18]) are becoming widely used as the basic unit of reuse.

Typically, the work of creating a design has to be split up between the members of a team; and its artefacts — not just the final code, but design documents too — have to be stored, moved around, adapted, and possibly incorporated into more than one end-product. Typical design artefacts are rarely just about one object, but about groups of objects and the way they interact.

In OOD, the term *frameworks* is used for descriptions of groups of objects, their relationships, division of responsibilities, and interactions. Most of the design patterns discussed in books (e.g. [7]) and discussion groups are based around frameworks: for example, the 'observer' pattern which keeps many views up to date with one subject; or 'proxy', which provides a local representative of a remote object; or any of the more specialised design-ideas that are fitted together to make any system.

In this paper, we discuss our preliminary efforts to use our previous research on frameworks in computational logic as a background to define a formal semantics (in computational logic) for the specification and correctness of OOD frameworks, and briefly illustrate it with OOD frameworks in the CBD methodology *Catalysis* [6].

We will first explain our view of frameworks in computational logic, and then consider the possibility of applying their formalisation to OOD frameworks. Frameworks provide the basis for our approach to program development in computational logic, with the novelty of *a priori* correctness for components and their composition. This results from a more structured view of correctness, where we distinguish frameworks, specifications and programs. Frameworks give a first level of reuse, essentially by means of framework expansion and composition. Each framework may contain many correct programs, together with their specifications. The latter give a second level of reuse. Indeed, programs can be composed according to their specifications, while preserving correctness.

In OO terminology, frameworks correspond to (possibly partial and generic) object models, while programs in a framework correspond to methods. Specifications are similar to functional models, but this similarity is only partial, at the present stage of our research. We have introduced states in our work only recently, and we have yet to extend our notion of *a priori* correctness to methods that affect the states of objects. Nevertheless we believe that our preliminary results on frameworks are interesting, and should provide the basis for *a priori* correctness for OOD.

By contrast, current formal OO development methods, e.g. CARE [16], are mainly in the style of *posit-and-prove*. That is, proof of correctness is done by *a posteriori* verification. For component-based software development, we will argue that a better approach is to have *a priori* correctness, i.e. components should be correct in their own right, *prior to* their composition, in such a way that the correctness of their composition or reuse can be expressed, again *a priori*, in terms of the correctness of the components. Without *a priori* correctness, it will not be possible to define (and develop libraries of) correct components, and to use them to construct larger correct composites. *A posteriori* correctness is all right for developing single programs, but too cumbersome, perhaps impossible, for component-based software development.

2 Overview of Our Approach in Computational Logic

Our approach is based on a three-tier formalism (with model-theoretic semantics) illustrated in Figure 1. At the bottom level, we have *programs*, for computing (specified) relations. Programs are pure

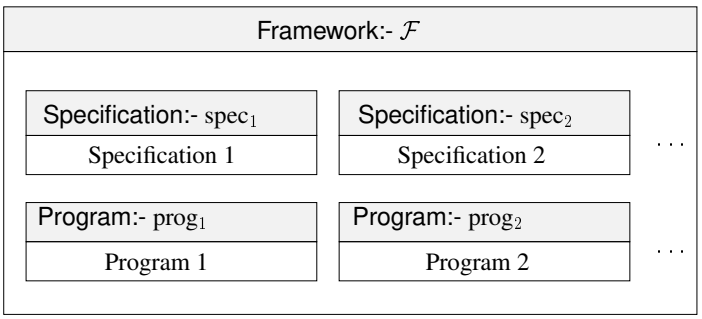


Figure 1: A three-tier formalism.

(standard or constraint) logic programs, and are therefore Horn clause theories. In the middle, we have *specifications* for defining or specifying new relations (and functions). At the top, we have a *specification framework*, or *framework* for short, that embodies an axiomatisation of (all the relevant knowledge of) the problem domain, that provides an unambiguous semantic underpinning for specifications and programs,

as well as the correctness relationship between them.

We will show that this approach yields a declarative semantics that allows us to define and reason about the (semantics and) correctness of open programs or modules, and of module composition or reuse, i.e. the notion of *a priori* correctness that we call *steadfastness*. This semantics also allows us to introduce and formalise *frameworks* as used in OOD.

3 Specification Frameworks

In this section, we briefly formalise specification frameworks.

Definition 3.1 A *specification framework* $\mathcal{F}(\Pi)$ with *parameters* Π is a pair $\langle \Sigma, Ax \rangle$, where Σ is a (many-sorted) signature, and Ax is a set of first-order axioms for the symbols of Σ. The parameters Π belong to Σ, and the symbols of $\Sigma \setminus \Pi$ are called *defined symbols*.

The axioms for the parameters are called p*-axioms*. The signature of p-axioms is a subsignature of Σ containing Π, and it will be denoted by Σ_p.

We say that a (specification) framework $\mathcal{F}(\Pi)$ is *open* if Π is not empty; otherwise, we say that it is *closed* and we indicate it by $\mathcal{F}$. Of course, a closed framework does not contain p-axioms, i.e. Σ_p is empty for such a framework.

A closed framework $\mathcal{F}$ axiomatises one problem domain, as an intended model (unique up to isomorphism). In our approach, intended models are *reachable isoinitial* models. A model I is *reachable* if its elements can be represented by ground terms; a reachable model of $\mathcal{F}$ is *isoinitial* iff ground quantifier-free formulas are true in it whenever they are true in every model of $\mathcal{F}$. Isoinitial models allow us to deal with negation properly.

In general, a framework may not have an isoinitial model. Hence the following adequacy condition:

Definition 3.2 A closed framework $\mathcal{F}$ is *adequate* if it has a reachable isoinitial model.

A typical closed framework is (first-order) Peano arithmetic $\mathcal{NAT}$, with the well-known axiomatisation, including the first-order induction schema. $\mathcal{NAT}$ has the standard structure of natural numbers as an intended (reachable isoinitial) model.

An open framework $\mathcal{F}(\Pi)$ has a non-empty set Π of parameters, which can be *instantiated* by a closed framework $\mathcal{G}$. The *instance*, denoted by $\mathcal{F}[\mathcal{G}]$, is the union of (the signatures and the axioms of) $\mathcal{F}(\Pi)$ and $\mathcal{G}$. It is defined only if Σ_p (i.e., the subsignature of Σ that is the signature of the p-axioms) is the intersection of the signatures of $\mathcal{F}(\Pi)$ and $\mathcal{G}$, and $\mathcal{G}$ proves the p-axioms.

Definition 3.3 An open framework $\mathcal{F}(\Pi)$ is *adequate* if, for every adequate closed framework $\mathcal{G}$, the instance $\mathcal{F}(\Pi)[\mathcal{G}]$ is an adequate closed framework.

A more general notion of instance could be given, involving renamings, and we will use renamings in the examples. However, our main interest is not in operations on frameworks, like composition or instantiation, where more general techniques could be imported from the algebraic tradition (see, e.g., the use of pushouts [20, 22]). The novelty of our approach is in the introduction of specifications and *a priori* correctness. To concentrate on the new aspects, without introducing further technicalities, we will use our simpler Definition 3.3. This can be done without loss of generality, because it can be shown that $\mathcal{F}(\Pi)$ is adequate according to Definition 3.3 iff it is adequate wrt any more general notion of instance.

Example 3.1 The following open framework axiomatises the (kernel of the) theory of lists with parametric element sort $Elem$ and parametric total ordering relation $\lhd$:

Specification Framework $\mathcal{LIST}(Elem, \lhd)$;

IMPORT: $\mathcal{NAT}$;

SORTS: $Nat, Elem, List$;

FUNCTIONS: $nil : \rightarrow List$;

$\cdot : (Elem, List) \rightarrow List$;

$nocc : (Elem, List) \rightarrow Nat$;

RELATIONS: $elemi : (List, Nat, Elem)$;

$\lhd : (Elem, Elem)$;

AXIOMS: C-AXS$(nil, \cdot)$;

$elemi(L, i, a) \leftrightarrow \exists h, T, j \,.\, L = h \cdot T \wedge$
$(i = 0 \wedge a = h \vee i = s(j) \wedge elemi(T, j, a))$;

$nocc(x, nil) = 0$;

$a = b \rightarrow nocc(a, b \cdot L) = nocc(a, L) + 1$;

$\neg a = b \rightarrow nocc(a, b \cdot L) = nocc(a, L)$;

P-AXIOMS: (total ordering axioms for $\lhd$)

$x \lhd y \wedge y \lhd x \leftrightarrow x = y$;

$x \lhd y \wedge y \lhd z \rightarrow x \lhd z$;

$x \lhd y \vee y \lhd x$.

where C-AXS$(nil, \cdot)$ contains Clark's Equality Theory (see [17]) for the list constructors $\cdot$ and nil, and the first-order induction schema $H(nil) \wedge (\forall a, J \,.\, H(J) \rightarrow H(a \cdot J)) \rightarrow \forall L \,.\, H(L)$; the function $nocc(a, L)$ gives the number of occurrences of a in L, and $elemi(L, i, a)$ means a occurs at position i in L.

If $\mathcal{INT}$ is a closed framework axiomatising integers Int with total ordering $\leq$, then $\mathcal{LIST}(Int, \leq)[\mathcal{INT}]$ is a closed framework that axiomatises finite lists of integers. Note the renaming of $Elem$ by Int and $\lhd$ by $\leq$.

It is worth pointing out that natural numbers are not necessary for a large class of programs on lists, like those for concatenation, sorting, reversing, and so on. Nevertheless, they are very useful for specifying in a declarative, non-recursive way, notions like: "a list A is a permutation of a list B", "a list A is ordered", and so on (see Example 4.4). That is, the signature of a specification framework is specification-oriented, rather than program-oriented: it contains the language and the axioms for speaking about specifications and programs.

Typically, a specification framework is a composite, i.e., it contains many parts, useful for speaking about various aspects of a problem domain. In this sense, it is a (declarative) counterpart of an OOD framework used in current OOD methods [14]. This will be discussed later in Section 6.

4 Specification and Correctness of Programs

Specification frameworks are similar to what are called specifications in algebraic ADTs (and therefore they correspond to *object models* in OMT [19]). The difference is that a specification framework typically has a richer, specification-oriented language. What we call specifications in a framework are program specifications, i.e. we maintain a strict distinction between specification frameworks and (program) specifications. In this section we define formally what we mean by program specifications, or just specifications, for short, and by correctness wrt such specifications.

Definition 4.1 In a specification framework $\mathcal{F}(\Pi)$, a (*program*) *specification* S_δ is a set of sentences that define new function or relation symbols δ in terms of the symbols Σ of $\mathcal{F}$.

S_δ can be interpreted as an *expansion operator* for (the signature Σ of) $\mathcal{F}$, since S_δ is added to $\mathcal{F}$ (thus expanding its signature to $\Sigma + \delta$). S_δ thus associates with every model m of $\mathcal{F}$ a (set of) model(s) m' such that in (each) m', δ is interpreted according to S_δ, whereas the old symbols are interpreted as in m.

The models m' are thus $(\Sigma + \delta)$-expansions of m, called S_δ*-expansions*. A specification S_δ is *strict*, if, for every model m of $\mathcal{F}$, there is one S_δ-expansion. It is *non-strict* otherwise.

We define program correctness for *closed* programs in the context of a *closed* framework, as follows:

Definition 4.2 Let $\mathcal{F}$ be a closed (instance of a) specification framework with an isoinitial model I. Let S_r be a specification of a (new) relation symbol r (in $\mathcal{F}$), and P_r be a program that computes r, with minimum Herbrand model H. P_r is *correct* wrt S_r in $\mathcal{F}$ iff the interpretation of r in H coincides with (or is isomorphic to) the interpretation of r in one of the S_r-expansions of I.

Let us see how this definition works for strict and non-strict specifications.

4.1 Strict Specifications

For a strict specification $\hat{S}_r$, the new symbol r defined in $\hat{S}_r$ has a unique interpretation with respect to I, and one in H. Correctness of P_r wrt $\hat{S}_r$ means that the two interpretations of r coincide, or, at least, are isomorphic. This is illustrated in Figure 2, where dotted arrows denote semantic mappings, and the full double-headed arrow represents an isomorphism.

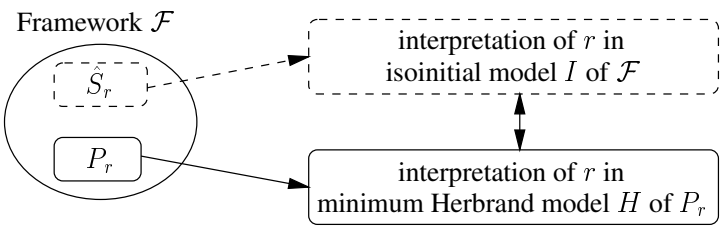


Figure 2: Strict specifications.

Example 4.1 An example of a strict specification is an *if-and-only-if specification* $\hat{S}_r$ of a new *relation* r in a framework $\mathcal{F}$:

$$r(x) \leftrightarrow R(x)$$

where $R(x)$ is any formula of the language of $\mathcal{F}$.

See Example 4.4 and Example 4.5 for examples of if-and-only-if specifications.

4.2 Non-Strict Specifications

If S_r is not strict, then r has many interpretations with respect to I. Correctness of P_r wrt S_r in this case means that the interpretation of r in H coincides with at least one of the interpretations of r wrt I. This is illustrated in Figure 3.

Example 4.2 An example of a non-strict specification is a *super-and-sub specification*:

$$\forall x\,.\,(R_{sub}(x) \rightarrow r(x)) \wedge (r(x) \rightarrow R_{super}(x))$$

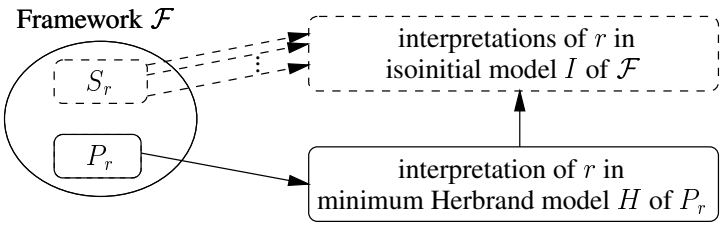


Figure 3: Non-strict specifications.

where $R_{sub}(x)$ and $R_{super}(x)$ are two formulas of the language of $\mathcal{F}$ such that $\mathcal{F} \vdash \forall x\,.\; R_{sub}(x) \rightarrow R_{super}(x)$.

The implication $\forall x\,.\; R_{sub}(x) \rightarrow R_{super}(x)$ is satisfied by the isoinitial model I of $\mathcal{F}$. Therefore the relation $R_{sub}(x)$ in I, i.e. the set of values $\mathbf{x}$ such that $I \models R_{sub}(\mathbf{x})$, is a sub-relation of the relation $R_{super}(x)$, and the specified relation r is any relation that is a super-relation of R_{sub} but is a sub-relation of R_{super}. This is illustrated in Figure 4.

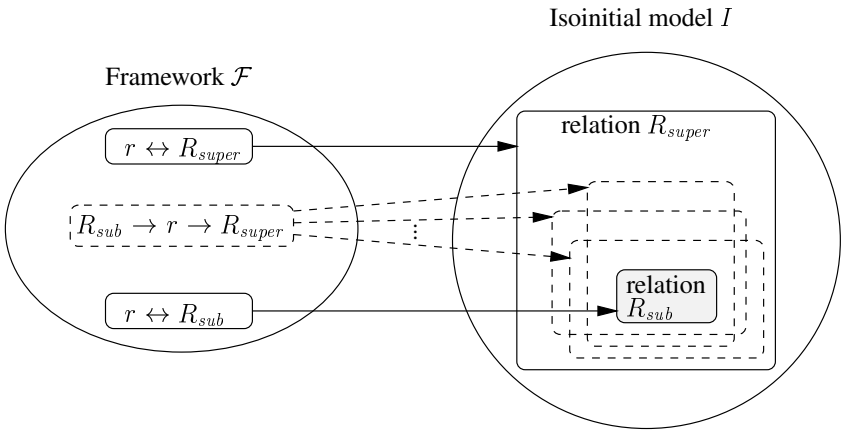


Figure 4: Super-and-sub specifications.

Example 4.3 An important form of a *super-and-sub specification* is a *conditional specification* of a new relation r in a framework $\mathcal{F}$:

$$\forall x, y\,.\; IC(x) \rightarrow (r(x,y) \leftrightarrow OC(y) \wedge R(x,y)) \tag{1}$$

where $IC(x)$ is a condition on x, $OC(y)$ is a condition on y, and $R(x,y)$ is a formula of the language of $\mathcal{F}$. (1) is equivalent to the following super-and-sub specification:

$$\forall x, y\,.\; (IC(x) \wedge OC(y) \wedge R(x,y) \rightarrow r(x,y)) \wedge (r(x,y) \rightarrow \neg IC(x) \vee OC(y) \wedge R(x,y))$$

See Example 4.5 for an example of a conditional specification.

A conditional specification is thus like a pre-post-condition style of specification as in VDM [9], Z [21], and B [1], except that it is declarative. Thus without local state, we can specify operations by pre-post-conditions in a declarative manner. This will be useful in the specification of OOD frameworks, as we will see later.

If we want to add state, we could do so by adding a sort, together with (updating) functions, see e.g. [2]. We actually prefer a more abstract approach involving (state) axioms that can evolve (see [12]).

In our approach, there is a clear distinction between frameworks and specifications. The latter introduce new symbols and assume their proper meaning only in the context of the framework.

Example 4.4 In $\mathcal{LIST}(Elem, \lhd)$, we can specify the usual length and concatenation functions l and $|$, and the usual 'membership', 'concatenation', 'permutation', 'ordered' and 'sort' relations mem, $append$, $perm$, ord and $sort$ as follows (we drop the universal quantifications at the beginning of specifications):

SPECS:
$$
\begin{array}{l}
mem(e, L) \leftrightarrow \exists i\,.\; elemi(L, i, e);\\
n = l(L) \leftrightarrow \forall i\,.\; i < n \leftrightarrow \exists a\,.\; elemi(L, i, a);\\
append(A, B, L) \leftrightarrow \forall i, a\,.\\
\qquad\qquad (elemi(A, i, a) \leftrightarrow elemi(L, i, a) \wedge i < l(A)) \wedge\\
\qquad\qquad (elemi(B, i, a) \leftrightarrow elemi(L, i + l(A), a));\\
perm(A, B) \leftrightarrow \forall e\,.\; nocc(e, A) = nocc(e, B);\\
C = A|B \leftrightarrow append(A, B, C);\\
ord(L) \leftrightarrow \forall i\,.\; elemi(L, i, e_1) \wedge elemi(L, s(i), e_2) \rightarrow e_1 \lhd e_2\\
sort(L, S) \leftrightarrow perm(L, S) \wedge ord(S).
\end{array}
$$

To distinguish the specified symbols from the signature of the framework, we will call them *s-symbols*. Also, specifications and axioms are clearly distinguished.

In general, an s-symbol can be used only in a single specification or in a fixed set of inter-dependent specifications. Indeed, since s-symbols may be defined by non-strict specifications, their meaning is implementation dependent: different correct programs implementing the same non-strict specification will give different meanings to the same s-symbol. However, a key feature of specifications in our approach is that they are implementation independent, as we will see in the next section, and we always want to preserve this property.

There are cases where some s-symbols δ with specifications S_δ can be used to expand the signature of the framework by δ and its axioms by S_δ. This is the case for strict specifications that preserve the adequacy of the framework. In this case, we can promote δ from the role of s-symbols to the role of framework symbols and S_δ from the role of specifications to the role of axioms, without losing framework adequacy and implementation independency. This may be very useful, since, in general, a richer framework language makes it easier to write specifications in a more compact and intuitive way.

In $\mathcal{LIST}(Elem, \lhd)$, the promotion of the symbols l, $|$, mem, $append$, $perm$, ord and $sort$ to framework symbols can be shown to be adequate. In the following, we will consider $\mathcal{LIST}$ thus expanded.

4.3 Steadfastness: Correctness of Open Programs

So far we have dealt with *closed* programs, i.e. programs without parameters. Now we discuss *open* programs, in particular the associated notion of correctness that we call *steadfastness*.

An *open program* may contain open relations,[1] or parameters. The parameters of a program P are relations to be computed by other programs. They are not defined by P.

A relation in P is *defined* (by 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). An open relation in P is also called a *parameter* of P.

A program is *closed* if it does not contain open relations. We consider closed programs a special case of open ones.

[1] We regard functions as functional relations, so by relations we mean functions *and* relations.

Open programs are always given in the context of an (open or closed) framework $\mathcal{F}(\Pi)$. In $\mathcal{F}(\Pi)$, we will distinguish program sorts, i.e. sorts that can be used by programs. A closed program sort must have constructors (see axioms C-AXS$(\ldots)$), and an open program sort may only be instantiated by program sorts. In programs, constant and function symbols may only be constructors. A program relation must be an s-symbol, i.e. it must have a specification.

Definition 4.3 Let $\mathcal{F}(\Pi)$ be a framework with a signature Σ. An *open program* $P_r(\pi)$ with specification (S_r, S_π) in $\mathcal{F}(\Pi)$ is a program with *defined predicate* r and *open predicates* (or *parameters*) π, where r and π are s-symbols, specified in the context of $\mathcal{F}(\Pi)$ by the specifications S_r and S_π.

Programs may be open independently from the framework, i.e., closed frameworks may contain open programs.

Example 4.5 In the closed framework $\mathcal{LIST}(Int, \leq)[\mathcal{INT}]$, we can have the following open program $P_{sort}(split, merge)$:[2]

$$\begin{array}{rcll} sort(nil, nil) & \leftarrow & & \\ sort(cons(x, nil), cons(x, nil)) & \leftarrow & & \\ sort(cons(x, cons(y, A)), W) & \leftarrow & split(cons(x, cons(y, A)), I, J) \wedge & (P_{sort}(split, merge)) \\ & & sort(I, U) \wedge sort(J, V) \wedge & \\ & & merge(U, V, W) & \end{array}$$

for sorting. The parameters $split$ and $merge$ of $P_{sort}(split, merge)$ may have many different interpretations wrt the isoinitial model of the framework because they may be defined by non-strict specifications (see below). We have the following (if-and-only-if) specification of $sort$:

$$sort(X, Y) \leftrightarrow perm(X, Y) \wedge ord(Y) \qquad (S_{sort})$$

(defined in Example 4.4), the following specification of $split$:[3]

$$\begin{array}{l} l(X) > 1 \wedge split(X, Y, Z) \rightarrow perm(X, Y|Z) \wedge l(Y) < l(X) \wedge l(Z) < l(X) \\ l(X) > 1 \rightarrow \exists Y, Z\,.\; split(X, Y, Z) \end{array} \qquad (S_{split})$$

and the following conditional specification of $merge$:

$$ord(X) \wedge ord(Y) \rightarrow (merge(X, Y, Z) \leftrightarrow ord(Z) \wedge perm(X|Y, Z)) \qquad (S_{merge})$$

The correctness relation between the specification (S_r, S_π) and the corresponding open program $P_r(\pi)$ can be defined (in $\mathcal{F}(\Pi)$) in a similar way to that in Definition 4.2. However, we cannot use minimum Herbrand models to define the correctness of open programs, because in Herbrand models, open relations are assumed to be empty, and therefore cannot play the role of parameters. So in [13] 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 informally, and then define correctness of open programs, and give its relevant properties. 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

[2] Without loss of generality, we choose the sorting example because it is concise (and familiar).

[3] This is an example of a *selector specification*, another form of non-strict specification (see [11]).

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. We can show that a minimum j-model (wrt $\subseteq_r$) exists. The minimum j-model of P will be indicated by j^P, and it represents the interpretation of r stated by the program P, in the pre-interpretation j.

Using minimum j-models, steadfastness in an interpretation can be defined as follows.

Definition 4.4 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.

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_1 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_1, 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_1, i.e., $P_r(p)$ is steadfast in i_1. Similarly, if we consider i_2 that interprets $p(z, x)$ as "$z * z = x$", to get steadfastness i_2 has to interpret $r(x)$ as "*x is a perfect square*".

Definition 4.5 Let $\mathcal{F}(\Pi)$ be a framework with signature Σ and axioms Ax, and $\mathcal{I}$ be a class of models of it. Let $P_r(\pi)$ be an open program with specifications (S_r, S_π) (in $\mathcal{F}(\Pi)$). $P_r(\pi)$ is *correct* in $\mathcal{F}(\Pi)$ with respect to $\mathcal{I}$ and (S_r, S_π) iff, for every interpretation $\mathsf{i} \in \mathcal{I}$ and 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 $\mathsf{i}_{\pi,r}$.

In this definiton, $\mathcal{I}$ is any class of models. In [13], we have introduced some proof-methods for correctness, which hold under suitable termination conditions. Since termination requires finite ground terms, we have to consider the class of Π-reachable[4] models. In practice, this is not a limitation, because we are interested only in the (unique up to isomorphism) reachable isoinitial model of $\mathcal{F}$, for a closed framework $\mathcal{F}$, and in the class of the reachable isoinitial models of the closed instances $\mathcal{F}[\mathcal{G}]$, for an open frameweork $\mathcal{F}(\Pi)$. Considering correctness in a framework as correctness in the class of its Π-reachable models, where our proof-methods apply, we have the following important properties of correct reusability:

Proposition 4.1 (Inheritance) *If $P_r(\pi)$ is correct in an open framework $\mathcal{F}(\Pi)$ wrt a specification (S_r, S_π), then it is correct wrt (S_r, S_π), in the isoinitial models of all the instances $\mathcal{F}[\mathcal{G}]$.*

Here, for simplicity, we have introduced inheritance only with respect to the closed instances of an open framework. Nevertheless a similar property holds if we introduce a subclass relation among open frameworks, where instances are the limiting cases, i.e. correctness is preserved by subclasses.

[4] I.e., reachable by adding constants to represent the elements of the possible open sorts of Π.

Inheritance yields a first level of correct reusability, namely reusability of correct programs through framework composition, subclassing and instantiation. This level of correct reusability would not be important, however, if we could not guarantee the correctness of the composition of the inherited open programs. This second, important level of correct reusability will be called *compositionality*. In compositionality, specifications play a central role, as shown by the following proposition:

Proposition 4.2 (Compositionality) *If $P_r(p)$ is correct in a closed framework $\mathcal{F}$ wrt a specification (S_r, S_p), and Q is a (closed) program for computing p, that is correct wrt S_p, then $P \cup Q$ is a (closed) program that is correct (in $\mathcal{F}$) wrt (S_r).*

As we can see, S_p is used to state when a program Q can be correctly composed with P, i.e., when we can correctly reuse P and Q. In order to ensure implementation indepencency of specifications, we must have one implementation independent specification for p, and then we can use any correct implementation of Q.

For the sake of simplicity, we have considered a composition yielding a closed program, but a similar reusability property holds when composition yields new open programs.

Example 4.6 We can show that the open program $P_{sort}(split, merge)$ in Example 4.5 for merge sort is correct in the open framework $\mathcal{LIST}(X, \lhd)$ with respect to the specifications $(S_{sort}, \{S_{split}, S_{merge}\})$ in Example 4.5.

This means that, in every instance of $\mathcal{LIST}(X, \lhd)$, e.g. $\mathcal{LIST}(X, \leq)[\mathcal{INT}]$ in Example 4.5, $P_{sort}(split, merge)$ 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 compositionality corresponds to *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.

5 Correct Classes

We can now extend our (model-theoretic) specifications to classes. In particular, the notion of steadfastness allows us to define the correctness of classes, and we can also deal with parametric classes.

Definition 5.1 A *class* K with abstract data type T and (possibly) open programs (methods) $P_{r_i}(\pi_i)$ $(1 \leq i \leq n)$ is a closed framework $\mathcal{F}$ with signature Σ and axioms Ax, which contains, for every program $P_{r_i}(\pi_i)$ $(1 \leq i \leq n)$, a corresponding specification (S_{r_i}, S_{π_i}).

The abstract data type T of S_K is the isoinitial model[5] of $\mathcal{F}$, while the meaning of the s-symbols r_i, π_i (for $1 \leq i \leq n$) is given by their (possibly non-strict) specifications.

It is easy to generalise this to parametric classes:

Definition 5.2 A *parametric class* $K(\Pi)$ with parametric abstract data type $T(\Pi)$ and (possibly) open programs (methods) $P_{r_i}(\pi_i)$ $(1 \leq i \leq n)$ is an open framework $\mathcal{F}(\Pi)$ with signature Σ and axioms Ax, which contains, for every program $P_{r_i}(\pi_i)$ $(1 \leq i \leq n)$, the corresponding specification (S_{r_i}, S_{π_i}).

For every closed instance $\mathcal{F}[\mathcal{G}]$, we have a corresponding instance $T[\mathcal{G}]$ of the parametric abstract data type $T(\Pi)$. In every instance, the meaning of the s-symbols r_i, π_i (for $1 \leq i \leq n$) is given by their (possibly non-strict) specifications.

Correctness of classes is defined as follows:

[5] More precisely, it is an isomorphism class.

Definition 5.3 A class is *correct* if its framework is adequate and its programs are correct (in the framework) with respect to their specifications.

A first study of the formal development of correct classes can be found in [10].

Pictorially, a generic class can be depicted as in Figure 5, using the three-tier formalism in Figure 1 in Section 2.

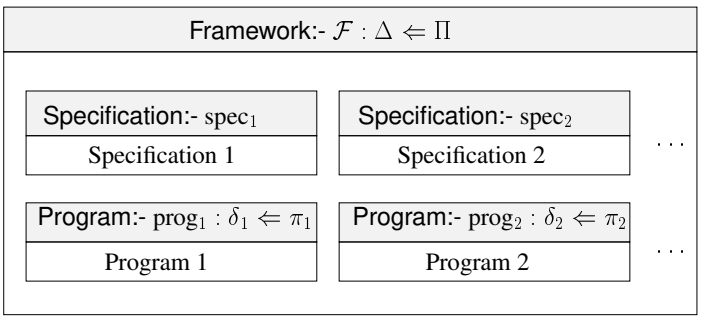
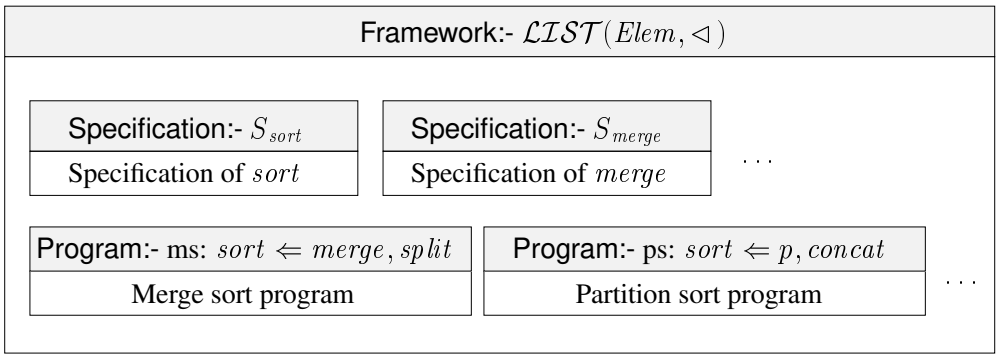


Figure 5: A generic class.

In the framework $\mathcal{F}$, Δ are *defined symbols* and Π are *framework parameters*; the specifications spec_i introduce the s-symbols; in the programs prog_i, the s-symbols δ_i are program-defined predicates and the s-symbols π_i are *program parameters*. Of course, for every program prog_i, its specification is $(S_{\delta_i}, S_{\pi_i})$.

Roughly speaking, compared to OMT, in our generic class, the (specification) framework corresponds to an *object model*. However, we distinguish three levels, that are all present in a framework. A *conceptual level*, corresponding to the signature and the axioms of the framework, a *specification level*, corresponding to program specifications, and an *implementation level*, corresponding to programs. The distinction between the three levels enhances reusability. Indeed, *reuse* and *correctness* of a generic class C occur at two levels: *framework* and *program* levels. C can be *reused* with different instances of Π and, in every instance, its open programs can be correctly reused, by composing the programs inherited from the frameworks used to build the instance. The correctness of C guarantees that all the instances are adequate, i.e., have an isoinitial model (this follows from the adequacy of the open framework), and that all the programs that can be built by composing the inherited open programs are correct with respect to their specifications (this follows from the correctness of the programs contained in correct frameworks and on the properties of steadfast programs.)

Example 5.1 The following class is correct:



where the specifications S_{sort}, S_{merge} and S_{split} are as given in Example 4.5, and the relations p and $concat$ can be specified in similar ways.

The program ms for merge sort is:

Program:- ms: $sort \Leftarrow merge, split$

$$
\begin{array}{rcl}
sort(nil, nil) & \leftarrow & \\
sort(x.nil, x.nil) & \leftarrow & \\
sort(x.y.A, W) & \leftarrow & split(x.y.A, I, J) \wedge sort(I, U) \wedge sort(J, V) \wedge \\
 & & merge(U, V, W)
\end{array}
$$

the program ps for partition sort is:

Program:- ps: $sort \Leftarrow p, concat$

$$
\begin{array}{rcl}
sort(nil, nil) & \leftarrow & \\
sort(x.nil, x.nil) & \leftarrow & \\
sort(a.A, Z) & \leftarrow & p(a.A, I, J, K) \wedge sort(I, I') \wedge sort(J, J') \wedge sort(K, K') \wedge \\
 & & concat(I', J', V) \wedge concat(V, K', Z)
\end{array}
$$

This class is correct because we can show that

- the framework $\mathcal{LIST}(Elem, \lhd)$ is *adequate*, and
- the programs ms and ps are *steadfast*.

In summary, our classes are declarative counterparts of frameworks used in current OOD methods [14], but with the advantage of having an associated notion of correctness.

$$
\begin{array}{ccccc}
\text{Correct class} & = & \text{Adequate framework} & + & \text{Steadfast programs} \\
 & & \downarrow & & \downarrow \\
 & & \text{ADT (+ class invariants)} & + & \text{Methods (+ pre-post-conditions)}
\end{array}
$$

An adequate framework is equivalent to a class in which the class invariants on the ADT are always satisfied because of the way the framework has been constructed; and steadfast programs are like methods that always satisfy their pre-post-condition specifications because of the *a priori* nature of correctness that steadfastness embodies.

Moreover, as we pointed out in Section 3, since our specification frameworks are composite frameworks, our generic class does not correspond to a type diagram for a *single* class. Rather it corresponds to a set of (interacting) classes in OMT. Therefore, our classes are declarative counterparts of OOD frameworks. Furthermore, their instances can be made into OOD frameworks if we introduce objects with states [12]. In this case, open specifications in a class are interfaces for partial or open objects, which become closed objects in closed instances of (i.e. compositions involving) the class.

In this paper, we have not considered objects and states (for a discussion see [12]), so our frameworks can (partially) model only the static level (i.e. the class diagram level) of OOD frameworks (see Section 6 later). However, they have better interfaces than current OOD frameworks, since they have proper specifications, and contain specifications for its programs.

6 Correct OOD Frameworks in *Catalysis*

In the rest of the paper we will try to demonstrate that the formal semantics introduced in the previous sections is suitable for specifying OOD frameworks used in the CBD methodology *Catalysis* [6]. However, in our work we have only introduced states recently (see [12]), and we do not yet have a notion of steadfastness for methods that modify the state of an object. Therefore, we can only analyse the static aspects of class diagrams and their instances. Only when we have introduced an appropriate notion of time, will we be able to address the dynamic aspects and hence fully extend our approach to OOD frameworks. When we do so, the advantage would be that our generic classes are more complete and therefore better components, since they also contain (steadfast) programs, with better interfaces (specifications). Moreover we can reason about their correctness and the correctness of their reuse, via steadfastness of their programs.

So, here we present only a first step in the direction of OOD frameworks. Since OOD frameworks are typically complex combinations of classes (objects), their description is more intuitive via a diagrammatic notation. We shall therefore use such a notation [14] and we will give some insights into the way diagrams representing OOD frameworks can be interpreted as specification frameworks (see also [14]).

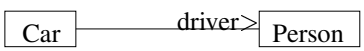
The double use of the term *framework* for both *specification framework* and *OOD modelling framework* is deliberate, since we use the former as a formalisation of the latter. To distinguish them, we shall use *framework* for the former and *OOD framework* for the latter.

We will consider the main categories of OOD frameworks in *Catalysis*, and for each category we will contend that it corresponds to a generic class, as defined in Section 5.

6.1 Closed *Catalysis* Frameworks

The simplest kind of OOD framework in *Catalysis* is a closed framework that represents a single type, whoose instances are directly objects.

Example 6.1 Consider the *Car* type represented in UML by the following type diagram:



The *Car* type corresponds to a simple class, represented by the following open (specification) framework:

Framework $\mathcal{CAR}$;

IMPORT: $\mathcal{PERSON}$;

SORTS: $Car, Person$;

FUNCTIONS: $self : \rightarrow Car$;
$driver : Car \rightarrow Person$;

AXIOMS: $\forall x, y : Car \,.\, x \neq y \rightarrow driver(x) \neq driver(y)$

The constant *self* has the usual meaning (i.e., it stands for the current object). The one-to-one association *driver* is interpreted as a function. Both *self* and *driver* are open symbols of the framework, since we do not have axioms for them. However, they have not been declared as parameters, since they are dynamic symbols. Their meaning will be fixed dynamically, in the various instances of the framework. To fix the meaning of a dynamic symbol, we do not pass it as a parameter, but we *explicitly define* it by state-axioms. For example, an object of type $\mathcal{CAR}$ is:

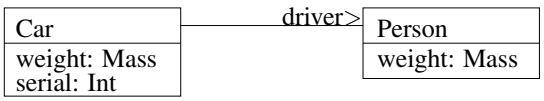
Object $spider : \mathcal{CAR}$;

STATE-AXIOMS: $self = spider$
$driver(spider) = John$

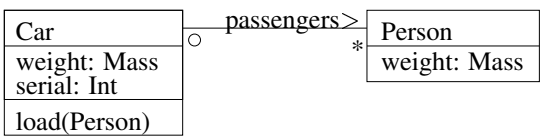
the type framework $\mathcal{CAR}$ is implicitly imported, and *spider* and *John* are implicitly added as new constants. The object *spider* is a closed framework. In its isoinitial model, *Car* is interpreted as the set containing one car *spider*, *Person* is interpreted as the set containing one person *John*, and *self* and *driver* are interpreted according to their explicit definitions.

Note that, in a system of cars and persons, the sorts *Car* and *Person* may contain many individuals, corresponding to the objects that have been created. However, in a single object we may have no knowledge of the other ones. This corresponds, in our example, to the fact that our *spider* object only knows (i.e., has constants for) itself and its driver. Using isoinitial semantics, we will require that the local isoinitial model (corresponding to the local knowledge) is isomorphically embedded into the global system-model (for more details, see [12]).

In general, *attributes* in type diagrams correspond to *constants* in frameworks, *associations* correspond to *functions* mapping (sets of) individuals into (sets of) individuals according to their multiplicity, *operations* to *relations*, and *invariants* to *axioms*. For example, the previous diagram may form part of a larger picture in which other attributes (constants), operations (relations) and invariants (axioms) may be added for *Car*:



If an association is decorated with a star $*$, then the result of the query is a set; while a $\circ$ means that the result of the query might be NIL, i.e., the empty set. Also, in general, the box representing a type may have three parts. For example, consider the diagram:



The third part lists operations that may be applied to its members. Normally, this is just a summary, and it is necessary to describe the operations in more detail separately. We may have static or dynamic

operations. The latter change the states of the objects involved, while the former do not. For static operations, we can apply our previous definitions of correctness, and we can guarantee *a priori* correctness. To specify the behaviour of dynamic operations at the system level, and therefore to be able to speak about their correctness, we need to introduce time. What we can do now, without time, is only to prove that the locally observable invariants are preserved.

In *Catalysis* we use *postconditions* to specify dynamic operations. For example, the operation load might be specified as follows:

op Car::load(boarder:Person)
post passengers = old(passengers) + boarder ∧
weight = old(weight) + boarder.weight

(Here + is overloaded to also mean set intersection.)

The postcondition is a relation between two states, before and after any occurrence of the operation. Any subexpression may therefore refer to its value beforehand using old(...).

As we can see, the *old* operator used in postconditions implicitly refers to time, i.e., we need a time-based refinement of the model-theoretic semantics of frameworks given in the previous sections. We do not yet have time in our formalisation, but nevertheless in the sequel we will use postconditions in our pictorial notation, and we will talk about objects. In particular, we will also use postconditions to describe operations not yet localised to any one object, i.e. where the effects are decided, but not yet who will take the responsibility for achieving them.

6.2 Open *Catalysis* Frameworks

Example 6.1 shows an example of a closed framework. Such a framework corresponds to the 'closed' traditional view of objects depicted in Figure 6, where an object has only one role. In *Catalysis*, we

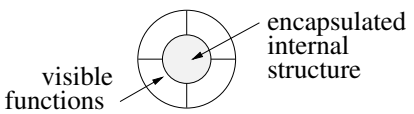


Figure 6: Traditional view of an object.

use open frameworks, in order to allow objects to have multiple roles in different frameworks, and to compose them by composing frameworks, as depicted in Figure 7.

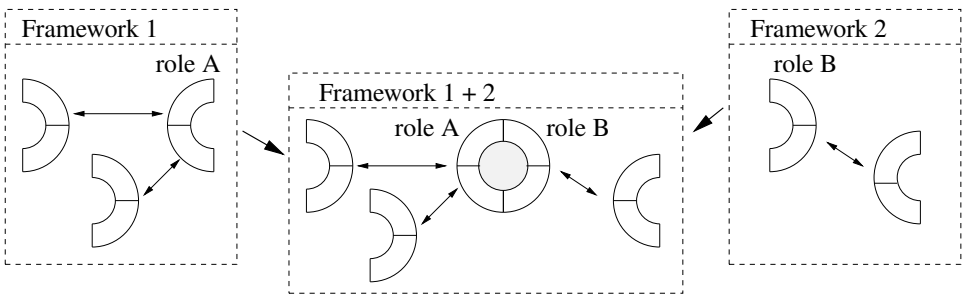


Figure 7: Objects by composing frameworks.

Example 6.2 Consider a domestic trades agency, which sends people to do plumbing or electrical work and the like. The agency handles several types of JobCategory (electrical, plumbing, central heating, ...) and each JobOccurrence (scheduled for a particular date at a given address) is for a particular JobCategory. Each JobCategory requires a set of Skills (e.g. heating requires electrical and plumbing) and each WorkPerson has a set of Skills. There are two essential invariants: that WorkPersons are not double booked (i.e. JobOccurrences they are scheduled for do not have overlapping dates); and that for every JobOccurrence, the skills of the scheduled WorkPerson must include at least the skills required for the JobCategory.

This kind of scheduling problem is a general resource allocation problem, that can be represented by the general open OOD framework shown in Figure 8. The angle brackets ⟨ ⟩ denote types that are parameters of the framework.

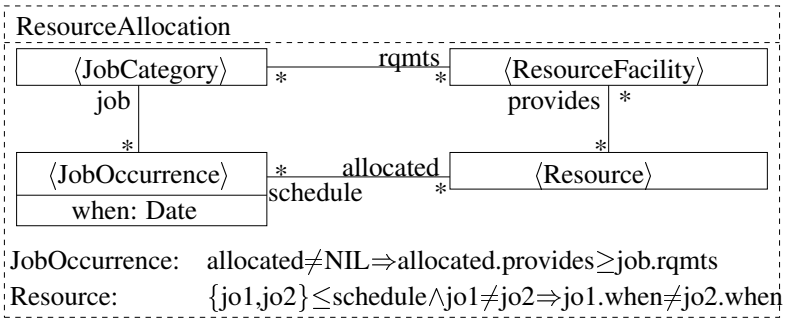


Figure 8: ResourceAllocation framework.

Thus an open OOD framework in *Catalysis* consists of (type diagrams of) interacting objects that correspond to (instances of) our generic classes, and invariants that correspond to axioms in our (specification) frameworks.

6.3 Composite *Catalysis* Frameworks

An open OOD framework can be instantiated by a renaming of its signature (see [14]), and its instances can be composed. Such a composite framework would look like its constituent frameworks, with invariants that are simply the conjunction of those in the latter. For example, from two instances of the ResourceAllocation framework in Figure 8, say the RoomAllocation framework and the InstructorAllocation framework, we could get the composite TeachingResourceAllocation framework as shown in Figure 9. Such a composite framework in *Catalysis* is of course also a generic class, like its constituents.

6.4 Interacting *Catalysis* Frameworks

A more interesting and powerful case of composite framework in *Catalysis* involves frameworks that interact with one another via parameters specified as *external* (*abstract*) *operations*. These parameters are relations governed by p-axioms in our generic classes. However, for this kind of OOD frameworks, we need time to treat correctness. So here we first look at an example in Catalysis, and then consider the kind of extension to our existing approach that would be needed in order to deal with correctness.

Example 6.3 Figure 10 shows an OOD framework in *Catalysis* for RetailDistribution with three abstract operations: A Retailer is anyone who sells Products to the public; Distributors sell only to Retailers. An

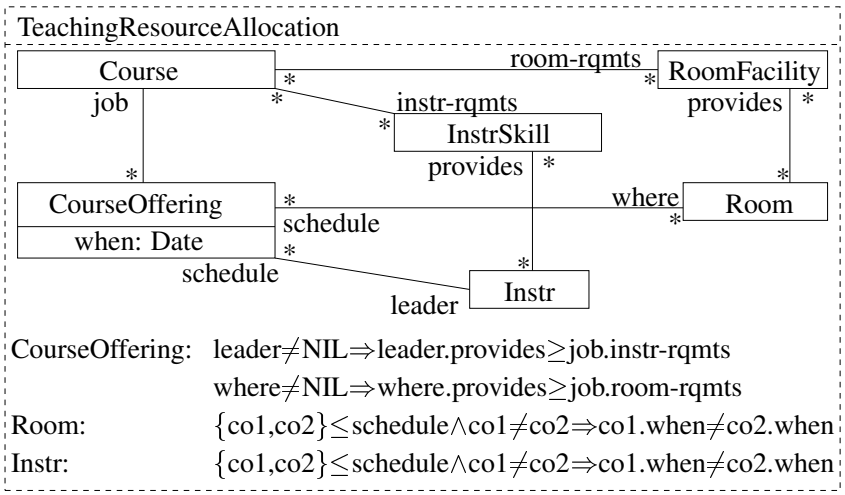


Figure 9: TeachingResourceAllocation framework.

abstract operation make-order is provided whereby the Retailer can create Orders; the Distributor can deliver Products, fulfilling an Order; and the Retailer can pay for them.

The middle section of the framework diagram is the *internal* structure: it defines what goes on between the participants. There are three internal abstract operations: make-order, deliver and pay. These are interactions that take place only between members of the types (or their subtypes). Their postconditions (for brevity we write them informally, and also omit the postcondition for pay) show the effects on the participants.

The bottom section is the external section: it is what can be seen from outside. External operations are always abstract. This is because we do not know in advance what other frameworks each participant may take part in. What we can do is to constrain the other actions in which the object is involved.

Retailer has the essential constraint that an Order is always present for Products whose stock levels are low. It is an external invariant: it applies only to operations belonging to other frameworks in which the objects take part. The operations of this framework do not themselves have to observe it (though they might be used to help maintain it).

Clearly, this OOD framework corresponds to a generic class. The pre-post-conditions in the middle (internal) section are just (conditional) specifications in a (specification) framework. The invariant in the bottom (external) section is a p-axiom, and the pre-post-conditions of the external operations are open or parametric specifications.

In this example, the correctness of the whole OOD framework would entail that, for example, the following informal specification is satisfied: “whenever a make-order operation has been executed (in the *past*), the corresponding delivery operation will be executed in some *future* time”.

This fact could be in some way implicit in the pre and post conditions, but we cannot yet write a corresponding formal specification (and therefore we cannot deal with this kind of sub-system correctness), because we cannot refer to past and future in our language. Our next step will be to study the possibility of introducing time in our approach.

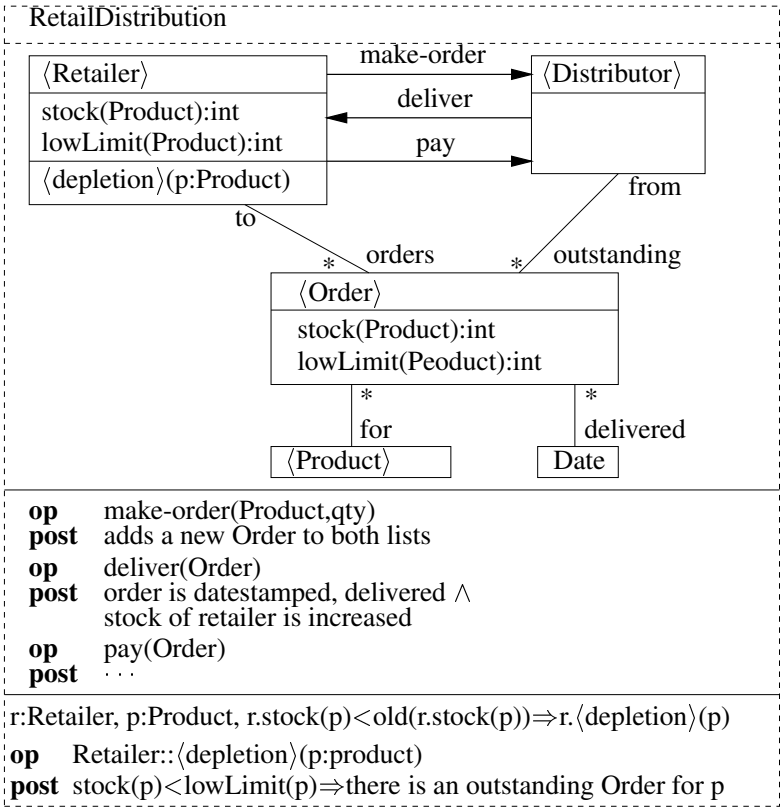


Figure 10: RetailDistribution framework.

7 Discussion and Concluding Remarks

We have presented the semantic foundations in computational logic for OOD frameworks. The novelty of our approach lies mainly in two aspects: (i) the use of model-theoretic semantics to characterise all the artefacts involved in program development, from ADTs (specification frameworks) to specifications to programs to generic classes; (ii) and more importantly, a (model-theoretic) notion of program correctness, called *steadfastness*, that we believe provides a suitable (new) criterion for component-based software development.

Our notion of a (correct) class as defined in Section 5 corresponds to OOD frameworks. However, there is an important difference from current OOD methodologies: we have specifications for both the (specification) framework *and* the programs (or methods), as well as (steadfast) methods in our class, all with formal semantics; whereas in OOD, they only specify the types (sets of objects) informally.

This difference is significant from the point of view of component-based software development. This is firstly because the key issue here is the precise specifications of interfaces between components, such as OOD frameworks. Our frameworks thus have the advantage of having precise specifications of their interfaces (with formal semantics).

Secondly this difference is significant because our frameworks contain (steadfast) methods and are therefore more complete components than OOD frameworks, which do not have methods. This is particularly significant because steadfastness provides a suitable correctness criterion for component-based software development. For a chosen problem domain, we can imagine a library of components that

have been developed separately each according to their specifications. If all these components are steadfast, then their compositions, if defined, would also be steadfast. Thus steadfastness extends to program composition. In current OOD, by contrast, methods have to be developed separately for each OOD framework, composite or not.

More importantly, if the problem domain or the corresponding library of components are themselves parametric, then steadfastness would guarantee the preservation of correctness through inheritance hierarchies. Therefore steadfastness would provide a suitable correctness criterion for component-based software development.

Steadfastness would also provide invaluable guidance for choosing the right components to construct a specified composite, because it defines *a priori* correct reuse. Therefore, provided components have well-designed interfaces, composing correct (indeed steadfast) programs from steadfast components would be more straightforward than from arbitrary components.

Our semantics provides a sound formalisation of OOD frameworks used in the CBD methodology *Catalysis*. However, our frameworks have better interfaces, and are more complete because they contain (steadfast) programs.

Finally, as we have already remarked, at present we can only deal with the static aspects of correctness, like the correctness of static methods (those that do not affect the state of a system, but only compute results) and the invariant preservation for the dynamic methods (those that can change the state). Consequently, we are not yet able to specify the global behaviour of systems of interacting objects. So our future work is to study the possibility of introducing a suitable temporal model into our approach.

References

[1] J.R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[2] R.H. Bourdeau and B.H.C. Cheng. A formal semantics for object model diagrams. *IEEE Trans. Soft. Eng.*, 21(10):799-821, 1995.

[3] D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, and P. Jeremaes. *Object-Oriented Development: The Fusion Method*. Prentice-Hall, 1994.

[4] S. Cook and J. Daniels. *Designing Object Systems*. Prentice-Hall, 1994.

[5] D.F. D'Souza and A.C. Wills. Extending Fusion: practical rigor and refinement. In R. Malan *et al*, editors, *Object-Oriented Development at Work*. Prentice-Hall 1996.

[6] D.F. D'Souza and A.C. Wills. *Catalysis: Components, Objects and Frameworks in UML*. Addison-Wesley, October 1998. Draft available at http://www.trireme.com/catalysis/book.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissades. *Design Patterns – Elements of Reusable Object-Oriented Design*. Addison-Wesley, 1994.

[8] R. Helm, I.M. Holland, and D. Gangopadhay. Contracts — Specifying behavioural compositions in OO systems. *Sigplan Notices* 25(10) (*Proc. ECOOP/OOPSLA 90*).

[9] C.B. Jones. *Systematic Software Development Using VDM*. Prentice Hall, 2nd edition, 1990.

[10] K.-K. Lau, C.D.M. Moss and M. Ornaghi. Formal development of correct classes in computational logic. In D.J. Duke and A.S. Evans, editors, *Proc. BCS-FACS Northern Formal Methods Workshop*, Electronic Workshops in Computing Series. Springer-Verlag, 1997.

[11] K.-K. Lau and M. Ornaghi. Forms of logic specifications: A preliminary study. In J. Gallagher, editor, *Proc. LOPSTR'96*, pages 295–312, *LNCS* 1207, Springer-Verlag, 1997.

[12] K.-K. Lau and M. Ornaghi. OOD Frameworks in Component-based Software Development in Computational Logic. In P. Flener and K.-K. Lau, editors, *Pre-Proceedings of LOPSTR'98*, Technical Report UMCS-98-6-1, Dept of Computer Science, University of Manchester, pages 172–179, June 1998.

[13] K.-K. Lau, M. Ornaghi, and S.-Å. Tärnlund. Steadfast logic programs. *J. Logic Programming* 38(3):259-294, March 1999.

[14] K.-K. Lau, M. Ornaghi, and A. Wills. Frameworks in *Catalysis*: Pictorial Notation and Formal Semantics. In M. Hinchey and S. Liu, editors, *Proc. 1st IEEE Int. Conf. on Formal Engineering Methods*, pages 213-220, IEEE Computer Society Press, 1997.

[15] K.-K. Lau, S. Liu, M. Ornaghi, and A. Wills. Interacting Frameworks in *Catalysis*. In J. Staples, M. Hinchey and S. Liu, editors, *Proc. 2nd IEEE Int. Conf. on Formal Engineering Methods*, pages 110-119, IEEE Computer Society Press, 1998.

[16] P. Lindsay and D. Hemer. Using CARE to construct verified software. In M. Hinchey and S. Liu, editors, *Proc. 1st IEEE Int. Conf. on Formal Engineering Methods*, pages 122-131, IEEE Computer Society, 1997.

[17] J.W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 2nd edition, 1987.

[18] R. Mauth. A better foundation: development frameworks let you build an application with reusable objects. *BYTE* 21(9):40IS 10-13, September 1996.

[19] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Sorenson. *Object-Oriented Modeling and Design*. Prentice-Hall, 1991.

[20] D. Sannella and A. Tarlecki. Essential concepts of algebraic specification and program development. *Formal Aspects of Computing*, 9:229-269, 1997.

[21] J.M. Spivey. *The Z Notation*. Prentice Hall, 2nd edition, 1992.

[22] M. Wirsing. Algebraic specification. In J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 675–788. Elsevier, 1990.