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 define a formal semantics in computational logic for the specification and correctness of OOD frameworks, abd briefly illustrate it with frameworks in the CBD methodology *Catalysis* [6]. The novelty of our approach is *a priori* correctness for components and their composition. By contrast, current formal 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

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

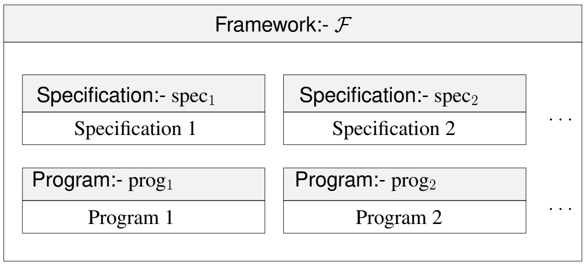


Figure 1: A three-tier formalism.

(standard or constraint) logic programs, and are therefore Horn clause theories (with initial models). 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. The axioms for the parameters are called *p-axioms*. 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 $\mathcal{F}$.

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 $\mathcal{NAT}$, using 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 $\mathcal{F}(\Pi)[\mathcal{G}]$, is the union of (the signatures and the axioms of) $\mathcal{F}(\Pi)$ and $\mathcal{G}$. It is defined only if Π 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 can be given, involving renamings (see also the pushout approach in algebraic ADTs [20, 22]. However, it can be shown that $\mathcal{F}(\Pi)$ is adequate according to Definition 3.3 iff it is adequate considering the more general notion of instance. Therefore we can use our simpler definition, without loss of generality.

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

$$
\begin{array}{rl}
\text{IMPORT:} & \mathcal{NAT}; \\
\text{SORTS:} & Nat, Elem, List; \\
\text{FUNCTIONS:} & nil \ : \ \to List; \\
& \cdot \ : \ (Elem, List) \to List; \\
& nocc \ : \ (Elem, List) \to Nat; \\
\text{RELATIONS:} & elemi \ : \ (List, Nat, Elem); \\
& \lhd \ : \ (Elem, Elem); \\
\text{AXIOMS:} & \text{C-AXS}(nil, \cdot); \\
& elemi(L, i, a) \leftrightarrow \exists h, T, j \,.\, L = h \cdot T \wedge \\
& (i = 0 \wedge a = h \vee i = s(j) \wedge elemi(T, j, a)); \\
& nocc(x, nil) = 0; \\
& a = b \to nocc(a, b \cdot L) = nocc(a, L) + 1; \\
& \neg a = b \to nocc(a, b \cdot L) = nocc(a, L); \\
\text{P-AXIOMS:} & \text{(total ordering axioms for } \lhd) \\
& x \lhd y \wedge y \lhd x \leftrightarrow x = y; \\
& x \lhd y \wedge y \lhd z \to x \lhd z; \\
& x \lhd y \vee y \lhd x.
\end{array}
$$

where $\text{C-AXS}(nil, \cdot)$ contains Clark's Equality Theory (see [17]) for the list constructors $\cdot$ and nil, and the first-order induction schema $H(nil) \wedge (\forall a, J \,.\, H(J) \to H(a \cdot J)) \to \forall L \,.\, H(L)$; 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 $\leq$.

Finally, it is worth pointing out that typically a specification framework is constructed incrementally from previously defined smaller frameworks. Hence the use of IMPORT in Example 3.1. Moreover, because it is a composite, a specification framework 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 the same as what are called specifications in algebraic ADTs (and therefore they correspond to *object models* in OMT [19]). 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.

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 $\mathcal{F}$. If S_δ contains symbols of Π, then it is called a *p-specification*.

S_δ can be interpreted as an *expansion operator* for (the signature Σ of) $\mathcal{F}$, since $S_d d$ is added to $\mathcal{F}$ (thus expanding its signature to $\Sigma + \delta$). $S_d d$ 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 $S_d d$, 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.[1]

We define program correctness for closed programs 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 (in $\mathcal{F}$) with an initial model (i.e. a minimum Herbrand model) H. P_r is *correct* wrt S_r in $\mathcal{F}$ iff H is isomorphic to one of the expansions of I determined by S_r, when both are restricted to the relation(s) r defined by S_r.

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.

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 4.6 for examples of if-and-only-if specifications.

[1] Non-strict specifications have been called *loose* specifications in the literature, but they have not been formalised before.

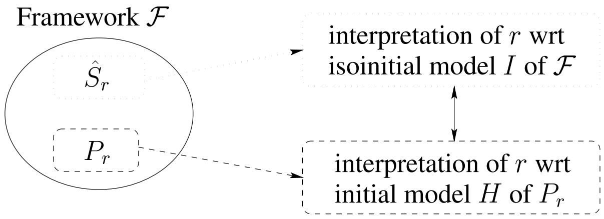


Figure 2: Strict specifications.

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.

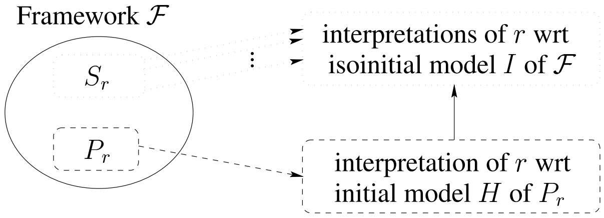


Figure 3: Non-strict specifications.

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

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

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

The implication $\forall x \,.\, R_{sub}(x) \to R_{super}(x)$ 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.

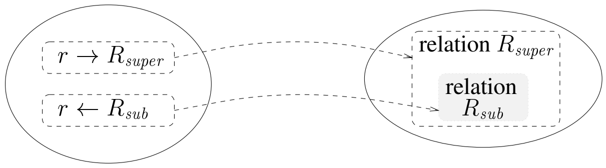


Figure 4: Super-and-sub specifications.

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

$$\forall x, y \,.\, IC(x) \rightarrow (r(x,y) \leftrightarrow OC(y) \wedge R(x,y))$$

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 $\mathcal{F}$.

See Example 4.6 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 *evolving axioms* (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):

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

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

An s-symbol δ with specification S_δ can be used to expand the signature of the framework by δ and its axioms by S_δ. An expansion is *adequate* iff framework adequacy is preserved.

The expansions of $\mathcal{LIST}(Elem, \lhd)$ by l, $|$, mem, $append$, $perm$, ord and $sort$ can be shown to be adequate. In the following, we will consider $\mathcal{LIST}$ thus expanded. Note that in the expanded framework these symbols can be used both as s-symbols and as symbols of the language.

4.3 Steadfast 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,[2] or parameters. The parameters of a program P are relations to be computed by other programs. They are not defined by P.

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

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.

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.

Even in a closed framework, we can have open programs, i.e. programs with parameters.

Example 4.5 In the closed framework $\mathcal{LIST}(Int, \leq)[\mathcal{INT}]$, we can have the following open program P_{sort}:[3]

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

for sorting. The parameters $split$ and $merge$ of P_{sort} may have many different interpretations wrt the isoinitial model of the framework because they may be defined by non-strict specifications (see Example 4.6).

Of course in an open framework, an open program may also inherit the parameters of the framework.

Definition 4.3 Let $\mathcal{F}(\Pi)$ be a framework with a signature Σ. A Σ-*program* is a program whose signature is a subsignature of Σ.

An *open program* $P_r(\pi_r)$ in $\mathcal{F}(\Pi)$ is a Σ-program with *defined predicates* r (specified by S_r), and *open predicates* (or *parameters*) $\pi_r \subseteq \Pi_r$, where Π_r is the subsignature of Σ that does not contain r.

Definition 4.4 The *specification* S_r of an *open program* $P_r(\pi_r)$ in a framework $\mathcal{F}(\Pi)$ is a specification of the relation r together with specifications of P_r's parameters $\pi_r \subseteq \Pi_r$, where Π_r is the subsignature of Σ that does not contain r.

The intended interpretation of S_r is the class of interpretations of r in terms of the class of interpretations of Π_r, or Π_r-interpretations, in $\mathcal{F}(\Pi)$.

Example 4.6 We can specify the open program P_{sort} in Example 4.5 in the closed framework $\mathcal{LIST}(Int, \leq)[\mathcal{INT}]$ by 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 $split$:[4]

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

and the following conditional specification of $merge$:

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

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

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

The correctness relation between the specification S_r and the corresponding open program $P_r(\pi_r)$ can be defined (in $\mathcal{F}(\Pi)$) in a similar way to Section 4.

For open programs (in open frameworks), we call our notion of correctness *steadfastness*. For an open program in a closed framework, this is defined as follows.

Definition 4.5 In a closed framework $\mathcal{F}$, let P_r be an open program for r, with parameters $\pi_1, \ldots, \pi_n$, and specifications $S_r, S_1, \ldots, S_n$, P_r is *steadfast* in $\mathcal{F}$ if, for any closed programs $P_1, \ldots, P_n$ that compute $\pi_1, \ldots, \pi_n$ such that P_i is correct wrt S_i, the (closed) program $P_r \cup P_1 \cup \ldots \cup P_n$ is correct wrt S_r.

Steadfastness in an open framework is defined as follows:

Definition 4.6 P_r is *steadfast* in an open framework $\mathcal{F}(\Pi)$ if it is steadfast in every instance $\mathcal{F}[\mathcal{G}]$.

Example 4.7 We can show that in the closed framework $\mathcal{LIST}(Int, \leq)[\mathcal{INT}]$, the open program P_{sort} in Example 4.5 for merge sort is steadfast.

The fact that P_{sort} is steadfast means that P_{sort} is always correct wrt its specification S_{sort} given in Example 4.6, provided that the programs for its parameters $merge$ and $split$ are correct wrt to their respective specifications S_{split} and S_{merge}. Furthermore, if we have steadfast programs for $merge$ and $split$, then their composition with P_{sort} will also be steadfast.

This example shows that steadfastness defines *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.

A more formal characterisation of steadfastness can be found in [13], in terms of both model and proof theory.

5 Specification and Correctness of 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 generic classes.

Definition 5.1 The *specification* S_K of a *class* K with abstract data type T and (possibly) open programs (methods) $P_{r_1}(\pi_1), \ldots, P_{r_n}(\pi_n)$ is a framework $\mathcal{F}(\Pi)$ together with specifications $S_{r_1}, \ldots, S_{r_n}, S_{\pi_1}, \ldots, S_{\pi_n}$, where $\pi_i \subseteq \Pi_i$ are the parameters in P_{r_i}, in $\mathcal{F}$, Π_i being the subsignature of Σ that does not contain r_i.

The intended interpretation of S_K is the isoinitial model of $\mathcal{F}$ in which $r_1, \ldots, r_n$ are interpreted in terms of the interpretations of Π_i.

It is easy to generalise this to generic classes:

Definition 5.2 The *specification* S_G of a *generic class* G with *polymorphic* abstract data type T and (possibly) open methods $P_{r_1}(\pi_1), \ldots, P_{r_n}(\pi_n)$ is an *open* framework $\mathcal{F}(\Pi)$ together with *open* specifications $S_{r_1}, \ldots, S_{r_n}$, $S_{\pi_1}, \ldots, S_{\pi_n}$, where $\pi_i \subseteq \Pi_i$ are the parameters in P_{r_i} in $\mathcal{F}(\Pi)$, Π_i being the subsignature of Σ that does not contain r_i.

The intended interpretation of S_G is the set of interpretations of $\mathcal{F}(\Pi)$ in which $r_1, \ldots, r_n$ are interpreted in terms of the interpretations of Π_i.

Correctness of classes is defined as follows:

Definition 5.3 A class K is *correct* wrt its specification $S_K = (\mathcal{F}, S_{r_1}, \ldots, S_{r_n}, S_{\pi_1}, \ldots, S_{\pi_n})$ iff $\mathcal{F}$ has an isoinitial model I and its (possibly) open methods $P_{r_1}, \ldots, P_{r_n}$ are correct in I.

A generic class G is *correct* wrt its specification $S_G = (\mathcal{F}, S_{r_1}, \ldots, S_{r_n}, S_{\pi_1}, \ldots, S_{\pi_n})$ iff $\mathcal{F}$ has a class of models, and its (possibly open) methods $P_{r_1}, \ldots, P_{r_n}$ are *steadfast* in $\mathcal{F}$.

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. In the framework $\mathcal{F}$, Δ are *defined symbols* and Π are *framework parameters*; the

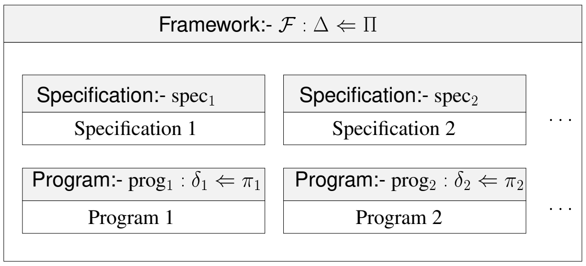


Figure 5: A generic class.

specifications spec_i may possibly contain parameters from Π; in the programs prog_i, δ_i are defined symbols and π_i are parameters.

Roughly speaking, compared to OMT, in our generic class, the (specification) framework corresponds to an *object model*, the specifications correspond to a *functional model*, while the programs correspond to a *dynamic model*, in a declarative manner. Therefore, our generic class specifies all three OMT models.[5] 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. Thus, if we add *state*, and hence *objects*, then an instance of a generic class contains an OOD framework (the latter does not contain programs, and usually does not contain any specifications either). In this paper, we will not deal with objects and states (for a discussion see [12]), so we will have declarative versions 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.

Now *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 π, i.e. C can be *composed* with other classes that define Π and π.

The *correctness* of (each instance of) C consists of

- the *consistency* of (each instance of) $\mathcal{F}$, and
- the *correctness* of (each instance of) each prog_i wrt its spec in (each instance of) $\mathcal{F}$.

Thus a correct (and hence correctly reusable) class consists of:

- an *adequate* specification framework, $\mathcal{F}$ with parameters Π,
- specifications $\mathrm{spec}_1, \mathrm{spec}_2, \ldots$

[5] In contrast to most diagrammatic notations, e.g. [2], which usually define only object models.

- *steadfast* programs $\mathrm{prog}_1, \mathrm{prog}_2$, with parameters $\pi_1, \pi_2, \ldots$

Example 5.1 The following class is correct:

Framework:- $\mathcal{LIST}(Elem, \lhd)$

Specification:- spec_1	Specification:- spec_2	
Specification of $sort$	Specification of $merge$	$\ldots$

Program:- ms	Program:- ps	
Merge sort program	Partition sort program	$\ldots$

where the specification spec_1 of $sort$ is S_{sort} (in Example 4.6); 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: $sort \Leftarrow p, concat$

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

the relations $split$ and $merge$ are as defined (by S_{split} and S_{merge}) in Example 4.6, and the relations p and $concat$ can be specified in similar ways.

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.

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.

6 Specification (and Correctness) of 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]. Furthermore, we will show 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.

However, since OOD frameworks are typically complex combinations of classes (objects), their description necessitates a diagrammatic notation. We shall therefore use such a notation [14]. Although we do not describe it here, this notation has a formal semantics based on the formal semantics we presented in the previous sections (see [14]).

The double use of the term *framework* for both *specification framework* and *OOD modelling framework* is deliberate, since the former is a formalisation of the latter, as we showed in earlier sections and in [14]. To distinguish them, we shall use *framework* for the former and *OOD framework* for the latter.

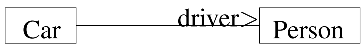
Now we show the specification (and correctness) of *Catalysis* frameworks. However, as we already mentioned, this requires the introduction of objects with states. In this paper, for lack of space, we shall not address objects with states (a discussion can be found in [12]). Instead we shall concentrate on a declarative approach.

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.

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



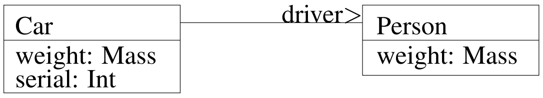
Clearly the *Car* type is the intended (isoinitial) model of the following closed (specification) framework:

Framework $\mathcal{CAR}$;
IMPORT: $\mathcal{PERSON}$;
SORTS: $Car, Person$;
FUNCTIONS: $driver \;:\; Car \rightarrow Person$;
RELATIONS: $\ldots$;
AXIOMS: $\ldots$

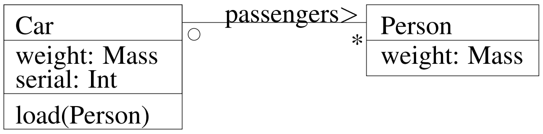
The attribute *driver* is interpreted as a function in the isoinitial model of $\mathcal{CAR}$.

Note that this model is an expansion of the *Person* type, which is the model of the framework $\mathcal{PERSON}$.

In general, *attributes* in type diagrams correspond to *functions* in frameworks, *operations* to *relations*, and *invariants* to *axioms*. For example, the previous diagram may form part of a larger picture in which other attributes (functions), 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. 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. In *Catalysis* we use *postconditions* to specify operations. For example, the operation load might be specified as follows:

op	Car::load(boarder:Person)
post	passengers = old(passengers) + boarder $\wedge$
	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(...).

Although we can interpret postconditions declaratively as a conditional specification (see Example 4.3), the use of postconditions normally entails that we have a state-based refinement of the model-theoretic semantics of frameworks given in the previous sections. Again, as we have pointed out, such a semantics can be obtained from the semantics we have presented, but for lack of space we will not describe the state-based semantics in this paper. 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

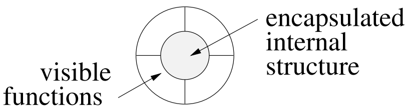


Figure 6: Traditional view of an object.

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.

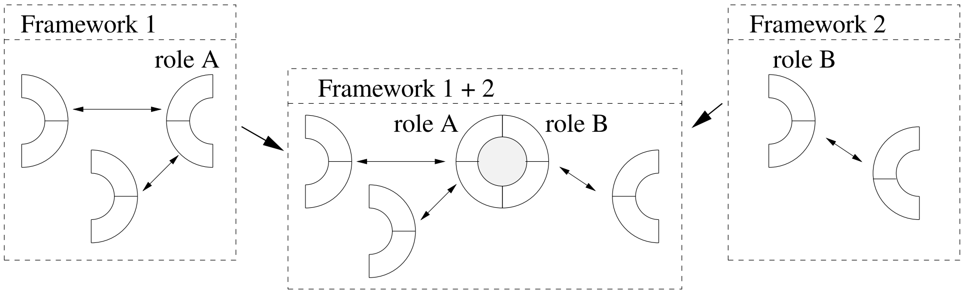


Figure 7: Objects by composing frameworks.

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.

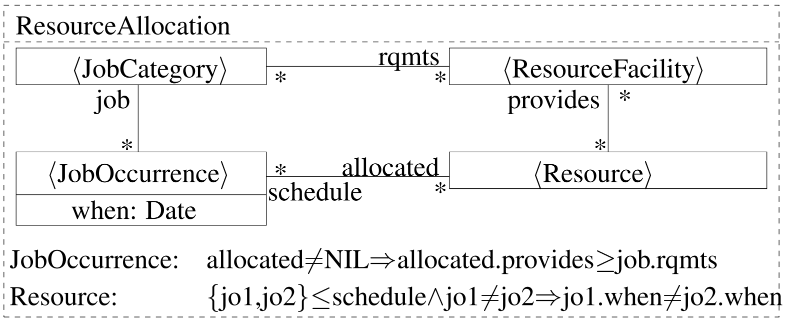


Figure 8: ResourceAllocation framework.

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.

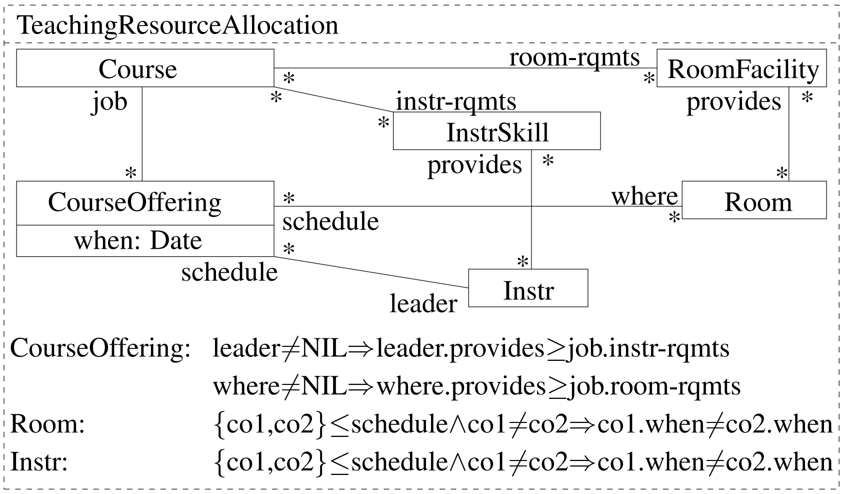


Figure 9: TeachingResourceAllocation framework.

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. A discussion of these frameworks can be found in [15].

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 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

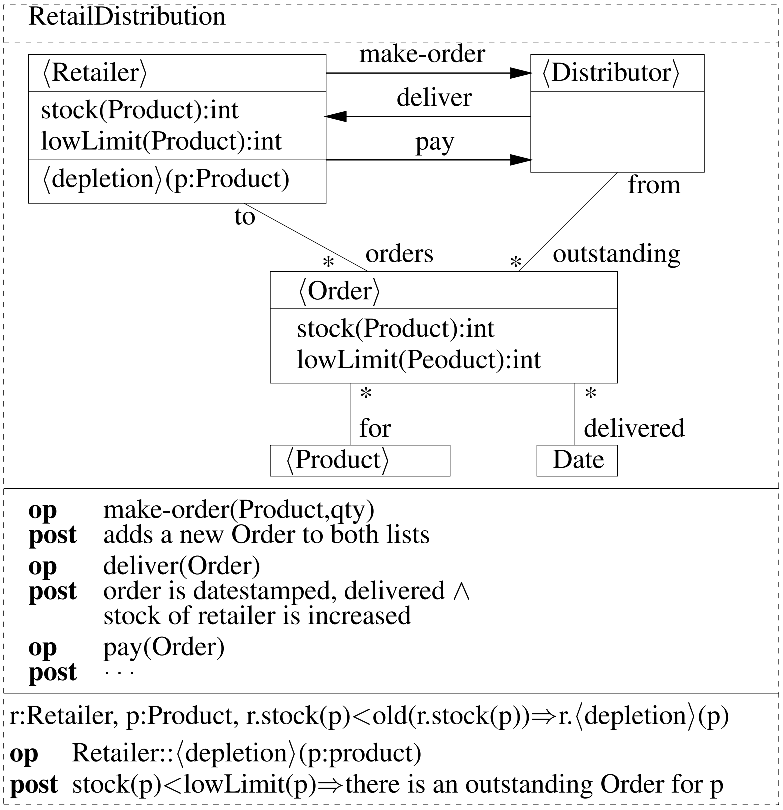


Figure 10: RetailDistribution framework.

the bottom (external) section is a p-axiom, and the pre-post-conditions of the external operations are p-specifications (see Definition 4.1).

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 component-based development methodology *Catalysis*. However, our frameworks have better interfaces, and are more complete because they contain (steadfast) programs. For future work, we plan to integrate our semantics with *Catalysis* or its successor, and then apply this technique to real-life case studies. To this end, we plan to develop appropriate tools for generating formal specification frameworks (and specifications) from diagrammatic descriptions of OOD frameworks in *Catalysis*.

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*, Tech-

nical 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. To appear in *J. Logic Programming*.

[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, 1997.

[15] K.-K. Lau, S. Liu, M. Ornaghi, and A. Wills. Interacting Frameworks in *Catalysis*. To appear in *Proc. 2nd IEEE Int. Conf. on Formal Engineering Methods*, 9-11 December 1998, Brisbane, Australia.

[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.