Interacting Frameworks in *Catalysis*

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

Shaoying Liu†
Dept. of Computer Science
Hiroshima City University
151-5, Ozuka, Numata-cho,Asaminami-ku
Hiroshima, 731-31, Japan
liu@white.sel.cs.hiroshima-cu.ac.jp

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

Alan Wills
Trireme International Ltd
24 Windsor Road, Manchester M19 2EB, UK
alan@trireme.com

Abstract

In current OO Design, most of the existing (semi-formal) methods use classes or objects as the basic unit of design. However, it is increasingly recognised that classes are not the best focus for design. Typical design artefacts are about groups of objects and the way they interact. In the Catalysis *project, we use the term* frameworks *for descriptions of groups of objects, their relationships, division of responsibilities, and interactions. The power of frameworks lies in the fact that interactions can also occur at the level of frameworks. In a previous paper, we defined a framework formally as a (first-order) many-sorted theory with a model-theoretic semantics and showed how it can be represented using a pictorial notation for practical system development. However, we considered only frameworks that do not interact with one another. In this paper, we discuss frameworks that interact with one another, and show that such frameworks are very powerful tools for formal OO Design.*

1 Introduction

In OO Design, it is widely recognised that the distribution of tasks between objects and the contracts between them are key to effective design. Most of the existing (semi-formal) 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. [10, 17]). Rather, in composing designs from reusable parts, the parts are preferably *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 [16], we discussed frameworks in the OO Design methodology *Catalysis* [6], and showed that these frameworks can be formalised as many-sorted theories. We also presented a pictorial representation of such theories, since practitioners of OO Design often prefer to use pictorial notations for design. Thus we could reason formally about design by using the sound (formal) semantics for the diagrams (cf. [2, 7]).

The power of frameworks lies in the fact that interactions can also occur at the level of frameworks. However, in [16], we considered only frameworks that do not interact with one another. In this paper, we shall first briefly summarise our treatment of *Catalysis* frameworks in [16], and then expand it to *interacting frameworks*, viz. frameworks that interact with one another via external operations. We will show that interacting frameworks are very powerful tools for formal OO Design.

*Sponsored by Hiroshima City University under Hiroshima City University Grant for Special Academic Research (International Studies) SCS-FM (A440).

†Supported in part by the Ministry of Education of Japan under Grant-in-Aid for Scientific Research on Priority Areas (No.90264960) and by Hiroshima City University under Hiroshima City University Grant for Special Academic Research (No. 9842).

2 Overview of Non-Interacting Frameworks in *Catalysis*

We begin with a brief overview of the different kinds of frameworks in *Catalysis* that do not interact with one another. In the sequel, we shall abbreviate OO Design to OOD.

2.1 Closed OOD Frameworks

The simplest kind of OOD framework in *Catalysis* is a *closed* framework, i.e. a framework with *no* parameters, that represents a single (composite) type.

Example 2.1 The following type diagram (in UML-like notation) in *Catalysis* defines the relationship between the *Car* type and the *Person* type:

A car has an attribute[1] *driver* which is a person.

2.2 Open OOD Frameworks

Example 2.1 shows an example of a closed framework. Such a framework corresponds to the 'closed' traditional view of objects depicted in Figure 1, where an object has only one role. In *Catalysis*, we use *open* frameworks, i.e. frameworks with *parameters*, in order to allow objects to have multiple roles in different frameworks, and to compose them by composing frameworks, as depicted in Figure 2.

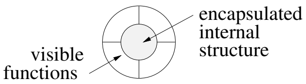

Figure 1. Traditional view of an object.

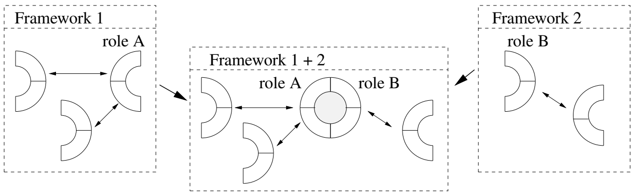

Figure 2. Objects by composing frameworks.

[1]In *Catalysis*, no distinction is made between attributes and associations.

Example 2.2 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 3. The angle brackets $\langle\ \rangle$ denote types that are parameters of the framework.

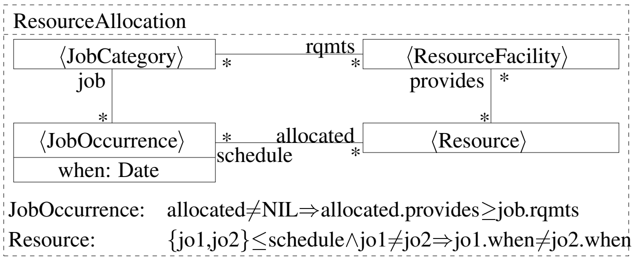

Figure 3. ResourceAllocation framework.

2.3 Composite OOD Frameworks

An open OOD framework can be instantiated by a renaming of its signature (see [16]), 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.

Example 2.3 Consider the scheduling problem for a teaching organisation. It has a portfolio of courses, offered on various dates throughout the year. Each instructor is qualified to teach some set of courses, and some courses require certain other skills such as knowledge of OS/2. Each course must be allocated a room, and there are various facilities that rooms must offer (equipped with PCs, has video, capacity>20, ...).

This is a resource allocation problem — two such problems combined, actually, which can be modelled by two instances of the ResourceAllocation framework (Figure 3), one for RoomAllocation and one for InstructorAllocation. These instances can be obtained by a renaming of the signature of ResourceAllocation. The framework for RoomAllocation is as shown in Figure 4. The framework for InstructorAllocation is as shown in Figure 5. By composing these two frameworks we can get the composite TeachingResourceAllocation framework as shown in Figure 6. The invariants in TeachingResourceAllocation are just the conjunction (or union) of those in RoomAllocation and InstructorAllocation.

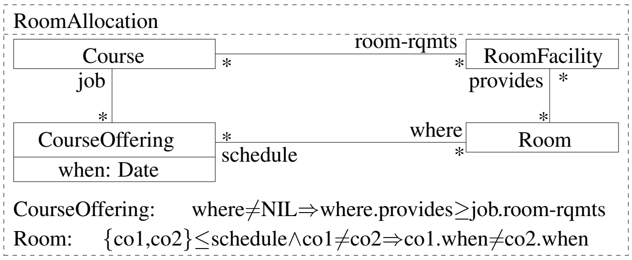

Figure 4. RoomAllocation framework.

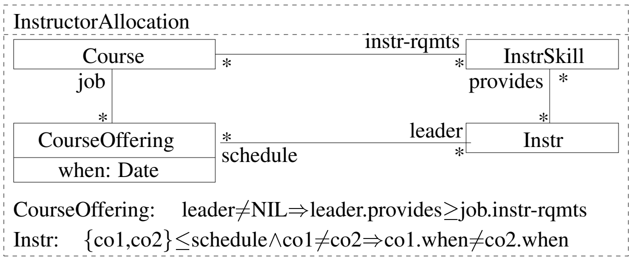

Figure 5. InstructorAllocation framework.

3 Interacting Frameworks in *Catalysis*

A more interesting and powerful case of framework composition involves frameworks that interact with one another via parameters specified as *external* (*abstract*) *operations*. To describe such interacting frameworks properly, we need first to formalise frameworks. We also need to include a formal semantics for objects with states; however for lack of space we will not do so in this paper, and refer the reader to [2] and [15] for two examples of possible formalisations.

3.1 Formalisation of Closed Frameworks

We distinguish between *closed* and *open* frameworks, depending on the absence or presence of *parameters*. First we deal with closed frameworks.

A framework $\mathcal{F}$ typically contains definitions of abstract data types, as well as theorems that hold, and relevant properties such as induction axioms, in a problem domain. For simplicity, we shall construct frameworks incrementally, with each framework defining just one abstract data type.

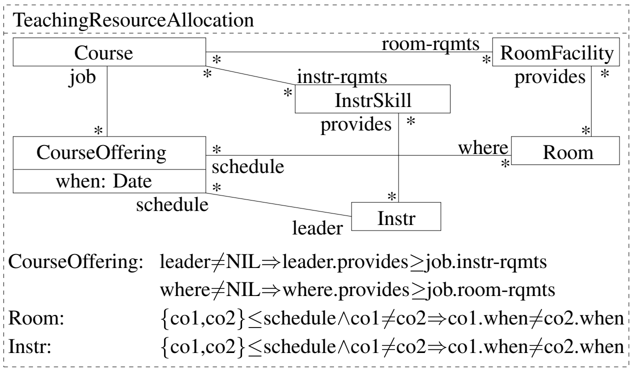

Figure 6. TeachingResourceAllocation framework.

The signature of $\mathcal{F}$ contains:

- a *sort symbol* T corresponding to the abstract data type, and possibly other *imported* sort symbols;
- *function symbols* that correspond to operations on T, in particular, these include the *constructors* for T;
- *relation symbols* that correspond to predicates on T.

The abstract data type axiomatised by $\mathcal{F}$ is the *intended model* of $\mathcal{F}$, viz. the *isoinitial model* of $\mathcal{F}$, defined as follows:

Definition 3.1 A model i^* is an *isoinitial model* of $\mathcal{F}$ iff, for every other model i of $\mathcal{F}$ there is a unique isomorphic embedding $h : \mathsf{i}^* \Rightarrow \mathsf{i}$.

An isomorphic embedding $h : \mathsf{i}^* \Rightarrow \mathsf{i}$ is a homomorphism with the additional property of preserving negation, i.e. for every relation symbol r, including identity, $(\alpha_1, \ldots, \alpha_n) \notin r^{\mathsf{i}^*}$ entails $(h(\alpha_1), \ldots, h(\alpha_n)) \notin r^{\mathsf{i}}$, where r^{i^*} and r^{i} are the interpretations of r in i^* and i respectively.

Less formally, if $\mathcal{F}$ has a *reachable* model, i.e. one where each element (of the domain) can be represented by a ground term, then:

> an *isoinitial model* i of $\mathcal{F}$ is a reachable model such that for any relation r defined in $\mathcal{F}$, ground instances $r(t)$ or $\neg r(t)$ are true in I iff they are true in all models of $\mathcal{F}$.

Such a (reachable) model is also an *initial* model, i.e. a reachable model j of $\mathcal{F}$ such that for any relation r defined in $\mathcal{F}$, ground instances $r(t)$ are true in j iff they are true in all models of $\mathcal{F}$.

Both initial and isoinitial theories enjoy the so-called 'no junk' and 'no confusion' properties [9]. 'No junk' means that the (initial or isoinitial) model is reachable (by ground terms), and 'no confusion' means that two ground terms of the domain of the model are identical iff they are equal according to the axioms. However, isoinitial theories handle negation properly, whereas initial theories can only do so via so-called 'final models'. Negation is an important property in reasoning about specifications and program correctness in general, e.g. in proving invariants. Therefore we have chosen isoinitial theories as frameworks (see [1] for a discussion of isoinitial theories).

The syntax we will use for $\mathcal{F}$ is similar to that used in algebraic abstract data types (e.g. [8, 19, 18]). However, whilst an algebraic abstract data type is an *initial* model ([9]) of its specification, the intended model of $\mathcal{F}$, i.e. the abstract data type it axiomatises, is an *isoinitial* model.

In general, the existence of an isoinitial model is of course not guaranteed. Therefore, in order to ensure consistency, we require frameworks to be *adequate*:

Definition 3.2 A closed framework is *adequate* iff it has a reachable isoinitial model.

Example 3.1 Natural numbers can be specified by the following framework:

Framework $\mathcal{NAT}$;

SORTS: Nat;

FUNCTIONS:

$0 \quad : \quad \to Nat$;

$s \quad : \quad Nat \to Nat$;

$+, * \quad : \quad (Nat, Nat) \to Nat$;

AXIOMS:

$\forall x.\,(\neg 0 = s(x)) \wedge \forall x, y.\,(s(x) = s(y) \Rightarrow x = y)$;

$\forall x.\,(x + 0 = x)$;

$\forall x, y.\,(x + s(y) = s(x + y))$;

$\forall x.\,(x * 0 = 0)$;

$\forall x, y.\,(x * s(y) = x + x * y)$;

$\forall (H(0) \wedge \forall i.\,(H(i) \Rightarrow H(s(i))) \Rightarrow \forall x.\,H(x))$.

It can be shown that the structure of natural numbers, i.e. the term model generated by the constructors 0 (zero) and s (*successor*), is an isoinitial model of $\mathcal{NAT}$. The framework $\mathcal{NAT}$ is thus adequate.

The last axiom in $\mathcal{NAT}$ is an induction schema. This illustrates the fact that a framework contains more than just abstract data type definitions. Typically, in a framework, we can find induction schema which allow us to prove many useful properties such as commutativity and associativity, and other theorems that have been established.

We can axiomatise other basic types such as the integer type Int, in a similar manner to $\mathcal{NAT}$.

Example 3.2 To illustrate our formalisation of basic types in *Catalysis*, we return to Example 2.1. Here, the *Car* type is the intended (isoinitial) model of the following closed framework:

Framework $\mathcal{CAR}$;

IMPORT: $\mathcal{PERSON}$;

SORTS: $Car, Person$;

FUNCTIONS: $driver \quad : \quad Car \to Person$;

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, a framework $\mathcal{F}$ typically defines a new abstract data type by using pre-defined types. Axioms for the old sorts are imported, and new ones are added to define new functions and relations on T.

3.2 Formalisation of Open Frameworks

Of course our incremental approach to framework construction can also be used for constructing *open* frameworks.

Example 3.3 For example, (finite) sequences of elements of a pre-defined type[2] X can be axiomatised as follows:[3]

Framework $\mathcal{SEQ}(X)$;

IMPORT: $\mathcal{NAT}, \mathcal{X}$;

SORTS: Seq;

FUNCTIONS:

$[\,] \quad : \quad \to Seq$;

$cons \quad : \quad (X, Seq) \to Seq$;

$nocc \quad : \quad (X, Seq) \to Nat$;

RELATIONS:

$elemi \quad : \quad (Seq, Nat, X)$;

AXIOMS:

$\neg[\,] = cons(a, B) \wedge$
$\quad cons(a_1, B_1) = cons(a_2, B_2) \Rightarrow a_1 = a_2 \wedge B_1 = B_2$;

$nocc(x, [\,]) = 0$;

$a = b \Rightarrow nocc(a, cons(b, L)) = nocc(a, L) + 1$;

$\neg a = b \Rightarrow nocc(a, cons(b, L)) = nocc(a, L)$;

$elemi(L, 0, a) \Leftrightarrow \exists B\,.\,(L = cons(a, B)$;

$elemi(L, s(i), a) \Leftrightarrow$
$\qquad \exists b, B\,.\,L = cons(b, B) \wedge elemi(B, i, a)$;

$H([\,]) \wedge (\forall a, J\,.\,H(J) \Rightarrow H(cons(a, J)) \Rightarrow \forall L\,.\,H(L)$.

[2] For convenience we will freely mix up 'type' and 'sort'.

[3] Here we will omit the outermost quantifiers, and use lower and upper case names for elements and sequences respectively.

where $\mathcal{X}$ is the (predefined) framework for the type X. Note that the imported signature is understood.

$\mathcal{SEQ}(X)$ is essentially the kernel of the standard axiomatisation of sequences. The first axiom is the freeness axiom for the sequence constructors [ ] and $cons$. The function $nocc(a, L)$ gives the number of occurrences of a in L. $elemi(L, i, a)$ means a occurs at position i in L. $nocc$ and $elemi$ enable us to define other useful list-based predicates. The last axiom is an induction schema for reasoning about all such predicates.

Similarly, we can axiomatise (finite) bags and sets in terms of (finite) sequences of elements of type X.

This illustrates the hierarchical nature of our approach to framework construction [13, 14], starting from the simplest type, e.g. natural numbers. The same approach is of course adopted by related work in algebraic abstract data types; however, our approach is different in the use of isoinitial semantics as the intended semantics of the specification language.

Unlike $\mathcal{NAT}$ which is a closed framework, $\mathcal{SEQ}(X)$ is an open framework with the parametric type X. Whilst a closed framework has *one* intended (isoinitial) model, an open framework has a *class* of intended models.

Example 3.4 Consider the open framework $\mathcal{SEQ}(X)$. For every interpretation of the open sort X, we get a corresponding intended model of $\mathcal{SEQ}(X)$. For example, suppose $\mathcal{INT}$ is a closed framework axiomatising the set Int of integers. Then $\mathcal{SEQ}(Int)$ automatically imports $\mathcal{INT}$ and becomes a closed framework with an isoinitial model where Int is the set of integers, Nat contains the natural numbers, and Seq finite sequences of integers.

3.3 Formalisation of Interacting Frameworks

A more interesting and useful kind of open framework is one that has relations as parameters. Such frameworks *interact* with one another through these parameters.

Example 3.5 For example, we can extend $\mathcal{SEQ}(X)$ to $\mathcal{SEQ}(X, \lhd)$, which is an open framework that axiomatises the (kernel of the) theory of sequences with parametric element type X and partial ordering relation $\lhd$:[4]

[4]We shall write $\lhd$ in infix notation for clarity.

Framework $\mathcal{SEQ}(X, \lhd)$;

IMPORT: $\mathcal{NAT}, \mathcal{X}$;

SORTS: Nat, X, Seq;

FUNCTIONS: (same as the functions in $\mathcal{SEQ}(X)$)

RELATIONS: $elemi \;:\; (Seq, Nat, X)$;
$\quad \lhd \;:\; (X, X)$;

AXIOMS: (same as the axioms in $\mathcal{SEQ}(X)$)

P-AXIOMS: $x \lhd y \wedge y \lhd x \Leftrightarrow x = y$;
$\quad x \lhd y \wedge y \lhd z \Rightarrow x \lhd z$

Apart from the usual axioms for the functions and relations, $\mathcal{SEQ}(X, \lhd)$ has an additional set of *parameter axioms*, or *p-axioms*, for the parameter $\lhd$. In general, *p-axioms* provide a very versatile channel for frameworks to interact with one another.

The class of intended models of an open framework can also be defined formally under isoinitial semantics, in a parametric manner. For lack of space, however, we shall not do so here.

Nevertheless, to ensure consistency, we also require open frameworks to be adequate. Here we briefly define adequacy for open frameworks.

As we saw in Example 3.4, an open framework $\mathcal{F}(\Pi)$ can be *instantiated* by a closed framework $\mathcal{G}$. The corresponding *instance*, denoted by $\mathcal{F}(\Pi)[\mathcal{G}]$, is the union of (the signatures and the axioms of) $\mathcal{F}(\Pi)$ and $\mathcal{G}$, and 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 of $\mathcal{F}(\Pi)$.

Now we can define adequacy for open frameworks:

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.

For example, we can show that the open framework $cseq(X, \lhd)$ in Example 3.5 is adequate.

3.4 Composing Interacting Frameworks

Interacting frameworks interact via their composition in the style of Figure 2. In order to discuss their interaction or composition properly, we need to define the formal semantics of framework composition in *Catalysis*.

To discuss the composition of interacting frameworks in *Catalysis*, we shall denote an open framework by

$$\mathcal{F}(\Pi) = \langle \Sigma, \mathcal{T}, \mathcal{TH}, \Pi \rangle$$

where Π is the set of parameters, Σ is the signature, $\mathcal{T}$ the set of axioms, and $\mathcal{TH}$ the set of theorems. In a closed framework, Π is empty. In an open one, any sort symbol s or relation declaration $r : a$ can be used as a parameter.

Identity $=\ :\ (s,s)$ is a parameter iff s is a parameter too. The set of p-axioms is simply the restriction $\mathcal{T}|\Pi$, namely the set of axioms that contain *only* symbols of Π. Note that in closed frameworks there are no p-axioms.

We say that a sort symbol is used in the parameters Π iff it belongs to Π or if it occurs in some declaration of Π, whilst a declaration is used iff it belongs to Π.

The composition of $\mathcal{F}(\Pi)$ with $\mathcal{F}_1(\Pi_1) = \langle \Sigma_1, \mathcal{T}_1, \mathcal{TH}_1, \Pi_1 \rangle$ is performed through a Π-renaming ρ defined thus: $\rho(\Sigma)$ is a Π-*renaming for* Σ and Σ_1 if for every σ used in Π, $\rho(\sigma)$ belongs to Σ_1, whilst the sort symbols and declarations not used in Π are mapped into names that do not occur in Σ_1. For a Π-renaming ρ, we can build ρ-*amalgamation* of the signatures

$$\rho(\Sigma) + \Sigma_1 = \langle \rho(S) \cup S_1, \rho(F) \cup F_1, \rho(R) \cup R_1 \rangle$$

where $\Sigma = \langle S, R, F \rangle$ and $\Sigma_1 = \langle S_1, R_1, F_1 \rangle$.

Let ρ be a Π-renaming. The composition operation $\mathcal{F}[\rho, \mathcal{F}_1]$ is defined if the following proof obligation is satisfied:

$$\rho(\mathcal{T}|\Pi) \subseteq \mathcal{T}_1 \cup \mathcal{TH}_1 \tag{1}$$

That is, we require that the p-axioms of $\mathcal{F}$ become theorems or axioms of $\mathcal{F}_1$.

Definition 3.4 If the composition operation $\mathcal{F}[\rho, \mathcal{F}_1]$ is defined, the resulting *composite framework* $\mathcal{F}[\rho, \mathcal{F}_1](\Pi_c) = \langle \Sigma_c, \mathcal{T}_c, \mathcal{TH}_c, \Pi_c \rangle$ is defined as follows:

- $\Pi_c = \Pi_1$, i.e. the parameters of the composite are the (possible) ones of $\mathcal{F}_1$.
- $\Sigma_c = \rho(\Sigma) \cup \Sigma_1$, i.e. the new signature enriches Σ_1 by $\rho(\Sigma)$.
- $\mathcal{T}_c = \mathcal{T}_1 \cup (\rho(\mathcal{T}) - \mathcal{TH}_1)$, i.e. we consider the union of the axioms, and we eliminate possible axioms of $\rho(\mathcal{T})$ that are theorems of $\mathcal{F}_1$.[5]
- $\mathcal{TH}_c = \mathcal{TH}_1 \cup (\rho(\mathcal{TH}) - \mathcal{T}_1)$

If ρ is the identity, we indicate the corresponding composition by $\mathcal{F}[\mathcal{F}_1]$.

Example 3.6 Now we show a simple example of framework composition. First consider the following closed framework:

Framework $\mathcal{K}_{a,b}$;
SORTS: $Elab$;
FUNCTIONS: $a, b :\Rightarrow Elab$;
AXIOMS: $\neg a = b$;

[5] This does not mean that axioms are independent, but simply that they remain disjoint from theorems, to avoid redundancies.

The interpretation such that $Elab$ is interpreted as $\{a, b\}$ is an isoinitial model of $\mathcal{K}_{a,b}$.

Then also consider the following open framework that axiomatises the (open) theory of stacks:

Framework $\mathcal{STACK}(Elem)$;
SORTS: $Elem, Stacks$;
FUNCTIONS:
$empty \ :\ \Rightarrow Stacks$;
$push \ :\ (Elem, Stacks) \Rightarrow Stacks$;
AXIOMS:
$\neg empty = push(x, S)$;
$push(x_1, S_1) = push(x_2, S_2) \Rightarrow x_1 = x_2 \wedge S_1 = S_2$;
P-AXIOMS:
$\exists x, y\, (\neg x = y)$;

where we have omitted the universal closure of the axioms, upper-case variables are of sort $Stacks$, and lower-case variables are of sort $Elem$.

The p-axioms require that the parameter $Elem$ contains at least two distinct elements. Via $Elem$ the framework $\mathcal{STACK}$ axiomatises a class of isoinitial models. That is, whenever we choose a particular set of elements (fixed by a closed framework), we get an axiomatisation of stacks with the chosen elements.

Now $\mathcal{STACK}(Elem)$ can be composed with $\mathcal{K}_{a,b}$ by the renaming $Elem \mapsto Elab$ ($\rho(\sigma) = \sigma$ for the other symbols). We will use the syntax:

Framework $\mathcal{STACK}_{a,b}$;
COMPOSES: $\mathcal{STACK}(Elem)[\mathcal{K}_{a,b}]$
WITH: $Elem \ \mapsto \ Elab$;
OBLIGATION: $\exists x, y\, (\neg x = y)$;

The composition is defined. Indeed, the proof obligation $\exists x, y\, (\neg x = y)$ immediately follows from the axiom $\neg a = b$ of $\mathcal{K}_{a,b}$.

We can prove that composition is associative, i.e. we can prove the following theorem:

Theorem 3.1 If $(\mathcal{F}_0[\rho_1, \mathcal{F}_1])[\rho_2, \mathcal{F}_2]$ is defined, then $\mathcal{F}_0[\rho_2\rho_1, \mathcal{F}_1[\rho_2, \mathcal{F}_2]]$ is defined and

$$(\mathcal{F}_0[\rho_1, \mathcal{F}_1])[\rho_2, \mathcal{F}_2] = \mathcal{F}_0[\rho_2\rho_1, \mathcal{F}_1[\rho_2, \mathcal{F}_2]]$$

where $\rho_2\rho_1$ is the composition of the two renamings.

(The proof follows from the definition of composition.)

For consistency of open framework composition, we have the following theorem which states that our framework composition preserves adequacy:

Theorem 3.2 If $\mathcal{F}(\Pi)$ and $\mathcal{F}_1(\Pi_1)$ are adequate, and $\mathcal{F}[\rho, \mathcal{F}_1]$ is defined, then $\mathcal{F}[\rho, \mathcal{F}_1]$ is adequate.

A proof is given in [12].

3.5 Interacting Frameworks in *Catalysis*: An Example

Having laid the formal foundations for interacting frameworks and their composition, we round off this paper with a look at an example of such frameworks and their composition in *Catalysis*. We shall use a diagrammatic notation introduced in [16], which we have also used in the previous examples in this paper. Although our example is described informally, it is worth pointing out that our diagrammatic notation has the formal semantics that we have outlined for frameworks and their composition (see [16]).

In *Catalysis*, interactions between frameworks take place via *abstract operations*, which are formalised as relation parameters with p-axioms.

Example 3.7 The diagram in Figure 7 shows a framework for RetailDistribution with three abstract operations.

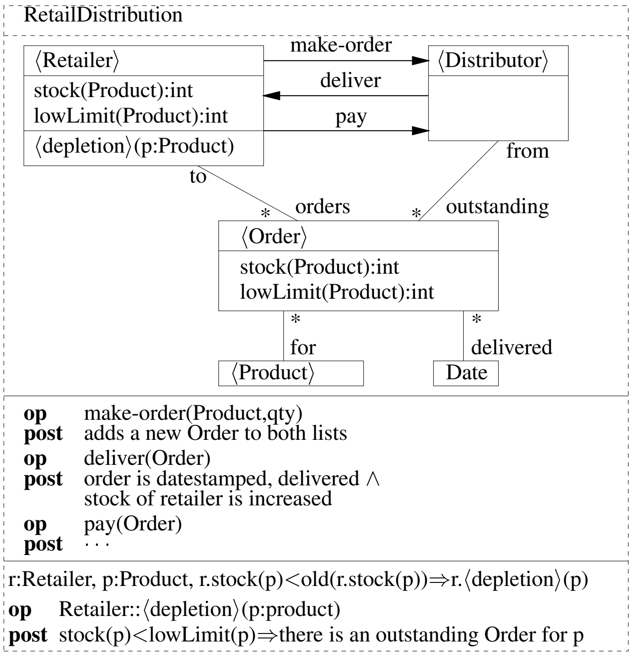

Figure 7. RetailDistribution framework.

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 (thus it corresponds to p-axioms). 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.

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.

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).

An abstract operation is used in analysis or high-level design to summarise the effects of what will finally turn out to be a sequence of smaller messages, possibly in both directions. Both participants will be affected by the more detailed design, since each will have to understand the more detailed protocol (just as, for example, someone may know that money can be got from a bank ATM, but not know the detailed operating instructions). The idea is similar to transactions and to use-cases [11]. Both participants are named in the headers and referred to in the postcondition.

The set of operations on either of the two types is obviously incomplete. For example, we have shown an operation that has the effect of increasing the stock, and there is an invariant aimed at restoring it when it gets low; but where are the operations that deplete the stock? And although we have shown the effect of a deliver operation, there is no indication of what prompts it to happen: does it occur as soon as an Order is made? Or in batches every week? Or when the Distributor's own stock is high enough?

This is the point of a(n open) framework. It focuses on the relationships between a given set of roles, but always with the assumption that each object will also play other roles in other frameworks; and that these relationships will include operations affecting its state. So if an object which is a Retailer is also a Shop, then sales to customers will deplete the stock; if it is a Restaurant, cooking food (whether we successfully sell it or not) depletes the stock. We find out the missing information when we tie this framework to others.

Now we can show an example of composition of interacting frameworks such as RetailDistribution, by considering a Shop, which is both a Retailer and a Vendor. First we define a Vendor in the framework PublicVending, which defines the relationship of Customers to Vendors as shown in Figure 8. In PublicVending, the sell operation, cash and Products are transferred in opposite directions.

A Shop is a type of object that plays the roles of both Retailer and Vendor. So now we compose the two frameworks into a single picture, with Customer, Shop, and Distributor

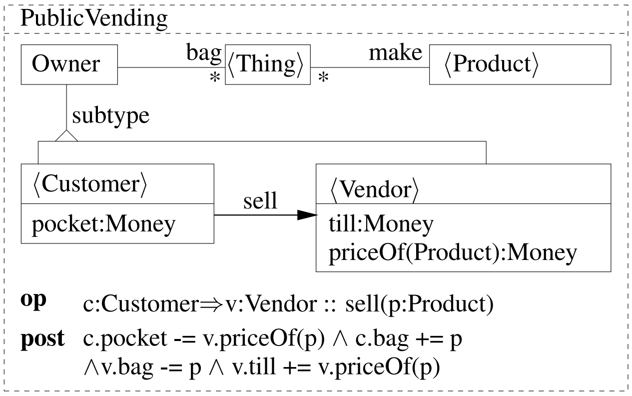

Figure 8. PublicVending framework.

as the key players. This is shown in Figure 9.

ShopAsRetailVendor

RetailDistribution	[	Retailer	/	Shop	]
Public Vending	[	Vendor	/	Shop	]

Shop :: (p:Product, Stock(p) = bag[make=p].size)

Figure 9. ShopAsRetailVendor framework – outline.

In PublicVending, the Vendor's stock was represented as a set of Things, each of which is an example of a Product; so this model must be tied up, using an invariant, with the Retailer's stock which had been modelled as an integer for any Product.

In more detail, the framework might be generated by a tool to look like the diagram in Figure 10. The final step is to implement the types with classes. Supposing that Shop:sell is not refined further, but implemented as a single message, the designer will have to observe the invariant from RetailDistribution whenever stocks get depleted. In this case, the only known way of creating an Order when necessary is using make-order — so a call to this will sometimes have to be part of executing sell.

4 Conclusion

We have presented a formalisation of interacting (open) frameworks in the OOD methodology *Catalysis*. The parameters of such frameworks are represented by relation symbols which are axiomatised by p-axioms. It is via p-axioms that frameworks in *Catalysis* interact.

Interacting frameworks are powerful tools for formal OOD. The traditional view of objects in existing OOD methodologies restricts an object to a fixed entity with just one role (in one framework). In contrast, open interacting frameworks in *Catalysis* allows objects to have multiple roles in different frameworks, as is the case with objects in practical systems.

Our interacting frameworks and their composition also have a formal semantics. Thus we believe that such frameworks provide very powerful tools for practical formal OOD.

The work that is most closely related to ours is that of Bourdeau and Cheng [2], and France *et al* [7]. Our work is different and novel mainly in the use of frameworks in general, and the use of interacting frameworks in particular.

Future work includes completing the formalisation of OOD frameworks by adding a formalisation of objects with states. Here we envisage a similar approach to that of [2], by adding states as sorts and operations on these sorts for updating states. Another important piece of future work will be the implementation of a tool for supporting the diagrammatic notation for frameworks, together with a tool for formal reasoning about frameworks.

Acknowledgements

We are grateful to the referees for their helpful comments and suggestions for improving this paper.

References

[1] A. Bertoni, G. Mauri and P. Miglioli. On the power of model theory in specifying abstract data types and in capturing their recursiveness. *Fundamenta Informaticae* VI(2):127–170, 1983.

[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] R.B. France, J.-M. Bruel, M. Larrondo-Petrie, E. Grant, and M. Saksena. Towards a rigorous object-oriented analysis and design method. *Proc. 1st IEEE*

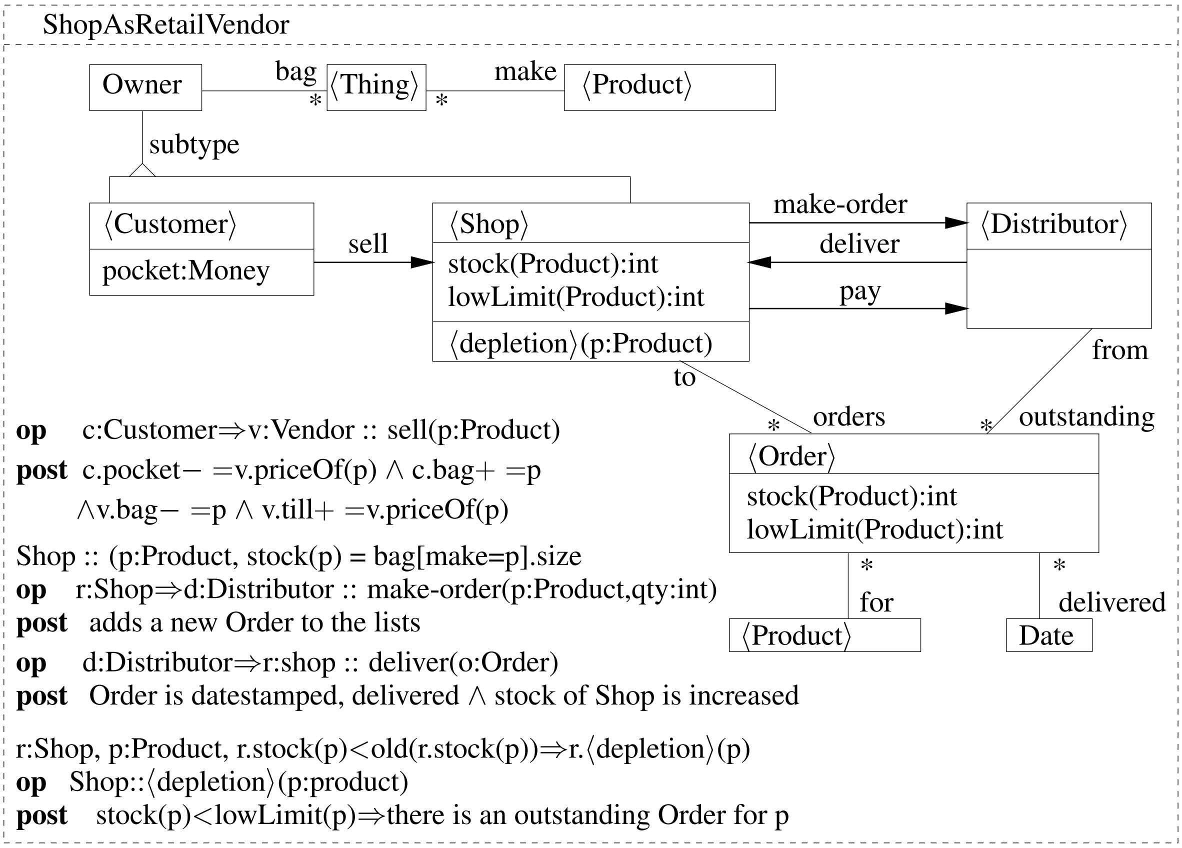

Figure 10. ShopAsRetailVendor framework.

Int. Conf. on Formal Engineering Methods, pages 7-16, IEEE Computer Society, 1997.

[8] J.A. Goguen and J. Meseguer. Unifying functional, object-oriented and relational programming with logical semantics. In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 417–477. MIT Press, 1987.

[9] J.A. Goguen, J.W. Thatcher, and E. Wagner. An initial algebra approach to specification, correctness and implementation. In R. Yeh, editor, *Current Trends in Programming Methodology, IV*, pages 80-149. Prentice-Hall, 1978.

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

[11] I. Jacobson *et al*. *Object-Oriented Software Engineering*. Addison-Wesley, 1992.

[12] C. Kreitz, K.-K. Lau, and M. Ornaghi. Formal reasoning about modules, reuse and their correctness. In D.M. Gabbay and H.J. Ohlbach, editors, *Proc. Int. Conf. on Formal and Applied Practical Reasoning*, *LNAI* 1085, pages 384–399, Springer-Verlag 1996.

[13] K.-K. Lau and M. Ornaghi. On specification frameworks and deductive synthesis of logic programs. In L. Fribourg and F. Turini, editors, *Proc. LOPSTR 94 and META 94*, *LNCS* 883, pages 104–121, Springer-Verlag, 1994.

[14] K.-K. Lau, C.D.M. Moss and M. Ornaghi. Formal development of correct classes in computational logic. 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.

[15] 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.

[16] 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.

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

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

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