Frameworks in *Catalysis*: Pictorial Notation and Formal Semantics

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

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.co

Abstract

In OO Design, 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, the parts are therefore 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. Practitioners of OO Design use pictorial notations for design. However, in order to reason formally about design, we need a sound (formal) semantics for the diagrams. In this paper, we show that frameworks can be formalised as many-sorted theories, and then present a pictorial representation of such theories, developed in the Catalysis *project.*

1. Introduction

Formal methods for program or system development need to have a good pictorial notation if they are to be adopted by practitioners for real-world work. Such a notation must of course be soundly based on the underlying formal semantics. The marriage of a 'user-friendly' pictorial notation and a formal semantics is all the more critical for OO Design, where the task is made much more complex by the use of objects.

Most of the existing formal OO Design 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. [11, 18]). 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 hopefully incorporated into more than one end-product. If we look at typical design artefacts, they are rarely just about one object, but about groups of objects and the way they interact.

In the *Catalysis* project [6], we use the term *modelling frameworks*, or just *frameworks* for short, 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 shall first explain our (informal) view of frameworks in *Catalysis*. Then we define frameworks formally as (first-order) many-sorted theory with a model-theoretic semantics. Finally, we show how they can be represented using a pictorial notation for practical system development.

2. Frameworks in *Catalysis*

In this section, we briefly describe the *Catalysis* view of frameworks. A more detailed account can be found in [22].

The traditional view of an object, as shown in Figure 1, regards an object as a closed entity with one fixed role.

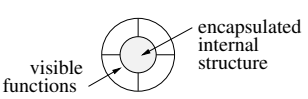


Figure 1. Traditional object picture.

This, however, does not reflect the nature of objects (and classes that describe them) in practical systems. In particular, objects tend to have more than one role in more than one context or framework.

In *Catalysis*, informally, a framework consists of a *goal*, a set of *typed roles*, a set of (*abstract*) *operations*, and a set of *trigger rules*. The goal is a description of purpose which this framework is intended to achieve, and often contains an invariant which we are seeking to preserve; it might be to maintain a constantly correct view of a subject. The typed roles represent the participating objects and their static relationships (how they are linked by pointers etc.) — for example, Subject and Observer. The operations represent their interactions — the messages they send to each other, such as update. The triggers characterise state changes caused by external operations which must prompt the interactions. External operations are those in which an object is involved by virtue of its participation in other frameworks.

In general, an object in a running system will not just play one role in one framework. Typically, it may be a participant in several frameworks. For example, it could play the role of the observed subject of several observer frameworks, whilst simultaneously being a proxy for others. Indeed frameworks would not be of much use unless they were designed with the assumption that their participant objects would play different roles in different frameworks. In a sense, the frameworks are the interesting pieces of design, and the objects are just where they are tied together in a particular system. This interesting perspective has been promoted by Helm *et al* [11], Nierstrasz and Tsichritzis [19], Harrison *et al* [12], Reenskaug *et al* [20], and others, and is illustrated in Figure 2.

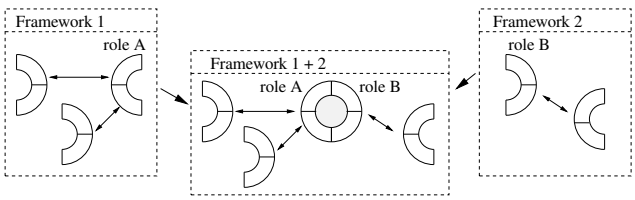


Figure 2. Objects by composing frameworks.

For each role it plays in every framework in which it participates, an object has an interface, namely the behaviour expected of it in that framework. So when we design a framework, we discuss an interface for each participant. Although we know that each will have other interfaces (which will have some effect on its state), we cannot anticipate what they will be.

Therefore, we think OO Design should not be about encapsulated objects, as illustrated in Figure 1. Rather, we focus on frameworks, each of which containing interfaces of several 'partial' objects. The objects merge when we compose frameworks (to make a system or a bigger framework). Once it is clear what the complete requirements on an object are, we can fix its internal structure.

Designing and developing frameworks, rather than classes, gives a much more flexible set of components to keep in a library. Ideally, to design a system, you select or build a set of frameworks, and fit them together. In practice, though, this is a bit of an exaggeration: you tend to begin with the basic classes, and then work out how they fit together in terms of frameworks.

When you incorporate a framework into a system, you are imposing the relationships and protocol of actions that it expresses onto some selected objects (belonging to selected classes) in the system. The idea is that this should then achieve the framework's stated goal for that group of objects.

3. Frameworks as Many-sorted Theories

In this section, we take a very abstract, declarative view of a framework $\mathcal{F}$, and define it formally as a (fully first-order) many-sorted logical theory (with equality) with model-theoretic semantics. The purpose of our abstract approach is to define formally (and completely) not only a specification language but also its semantics. We use a model-theoretic semantics because we want a completely declarative specification language. This does not mean that we cannot specify states and objects. Rather, our position is that at the most abstract level, states should only be represented logically. By subsequent refinement, however, we should be able to 'implement' states more concretely by operations on some pre-defined state space (and prove the correctness of any such implementation). Similarly, our frameworks define abstract generic classes, and objects (with states) can be obtained from them by suitable instantiations. Thus in this section we will not consider objects with states.

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.

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 [10]. '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. [9, 23, 21]). However, whilst an algebraic abstract data type is an *initial* model ([10]) 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. However, if a framework $\mathcal{F}$ has a reachable model i, then it can be shown that i is an isoinitial model of $\mathcal{F}$ if and only if for every closed atomic formula A (in $\mathcal{F}$), either $\mathcal{F} \vdash A$ or $\mathcal{F} \vdash \neg A$. This is a useful *existence condition* for isoinitial models. Furthermore, it can be shown that if $\mathcal{F}$ is a (possibly infinite) recursively enumerable axiomatisation, then there exists an isoinitial model in which relation symbols are interpreted by decidable relations, and function symbols by total computable functions. In practice, we only ever construct and use such frameworks (see [15, 17] for a related discussion).

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. For lack of space, we cannot give the full syntax of frameworks here, but it can be found in e.g. [15] along with examples of commonly uses frameworks.

3.1 Closed and Open Frameworks

We distinguish between *closed* and *open* frameworks, depending on the absence or presence of *parameters*. Whilst a closed framework has *one* intended (isoinitial) model, an open framework has a *class* of intended models. An interesting and useful kind of open framework is one that has relations as parameters. In such a framework, apart from the usual axioms for the functions and relations, there is an additional set of *parameter axioms*, or *p-axioms*, for the parameters. As we will briefly see later (in Section 5.3), *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.

4. Frameworks as Pictures

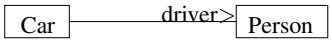
In the previous section, we have defined a formal semantics for frameworks. Typically, a framework is a composite one made up of several frameworks, and contains several abstract data types. These types are sets of objects, corresponding to different instantiations of the framework. Moreover, we can introduce states into these objects by introducing operations on a state-space that refines the model-theoretic semantics of the framework. However, for constructing more complex frameworks for practical use, we would need a pictorial notation (with formal semantics), which would be more concise and thus expressive. In this section, we will show how we represent frameworks pictorially in *Catalysis*, in particular frameworks that contain objects with states. For lack of space, we will not give full details of the pictorial notation, nor its formal semantics as a refinement of the abstract semantics of the previous section (a preliminary account can be found in [16]).

4.1 Closed Frameworks

Example 4.1 As a first example, consider the following closed framework:

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

The intended (isoinitial) model of $\mathcal{CAR}$ is the *Car* type represented in UML by the following type diagram:



This may form part of a larger picture in which other attributes (functions), operations (relations) and invariants (axioms) may be added for *Car*. For example, we might have:

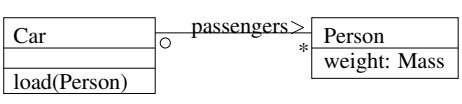
Such a diagram is a pictorially-presented set of assertions or axioms about the types involved. Since types are just sets of objects, it is a set of assertions about objects, and can be translated into that form. For example, the previous diagram can be translated into the following set of assertions:

$$\forall x \in \text{Car} \,.\, (x.\text{weight}) \in \text{Mass} \wedge (x.\text{serial}) \in \text{Int} \wedge (x.\text{driver}) \in \text{Person} \wedge (x.\text{driver}.\sim\text{driver}) = x \wedge (\forall p \in \text{Person} \,.\, (p.\sim\text{driver}) \in \text{Car})$$

In a framework, attributes are considered to be abstract. Actual members of type Car might or might not have parts labelled weight or serial. The purpose of an attribute is to help describe the type and its behaviour to its users: nothing is said about how the attribute should be implemented. For example, an attribute like *driver* is interpreted as a function in the isoinitial model of $\mathcal{CAR}$, i.e. nothing is said about how the attribute should be implemented. In a system built to deal with Cars and Persons, driver might be a pointer, or a relation in a database, or a function that derives the information from elsewhere. Or it might not be directly implemented at all: the internal structure can be very different from the specification in the framework, whose main purpose is to help explain to the clients the externally visible operations.

No distinction is made between objects and values. The integers Int, for example, are a set of objects which happen to be immutable; operations like + do not actually change anything: they are static relationships between the members of the type. No distinction is made between attributes and associations either. For instance, if it is not convenient to draw Person as a separate box, we could have made driver:Person an attribute of 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. For example, consider the diagram:



This represents the assertion:

$$(\forall x \in \text{Car} \,.\, (x.\text{passengers}) \in \text{Set}(\text{Person}) \wedge (\forall p \in \text{Person} \,.\, p \in x.\text{passengers}) \Rightarrow p.\sim\text{passengers} = x)) \wedge (\forall p \in \text{Person}, (p.\sim\text{passengers}) \in \{\text{Car} \cup \text{NIL}\}) \wedge (p \in p.\text{passengers}.\sim\text{passengers}))$$

Used properly, diagrams like the ones we have seen should be embedded within proper specification and design documents, with descriptive narrative. It is often convenient to deal with different aspects of a design in different parts of the text, e.g. to talk about drivers in one place and passengers in another. So we allow one type to be mentioned in many diagrams. The meaning of the whole document is just all the assertions conjoined together. For example, a Car has both passengers and a driver.

Finally, the box representing a type may have three parts, e.g. Car in the previous 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 will use *postconditions* to specify operations. For example, the operation load might be specified as follows:

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(...).

The use of postconditions entails that we have a state-based refinement of the model-theoretic semantics of frameworks given in Section 3. However, such a semantics is beyond the scope of this paper. Nevertheless, in the sequel we will use postconditions in our pictorial notation (in Section 5). 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.

It should be obvious that if we start from a type diagram, rather than a text-based definition of a framework, we can translate the former into the latter in a (reasonably) straightforward manner. For lack of space, we omit the details of such a translation algorithm here, and in the sequel, we shall also dispense with textual framework definitions and use pictorial ones only. Of course it should be noted that whilst a pictorial framework is good for practical design purposes, the corresponding text-based framework (and its formal semantics) is indispensable for formal reasoning purposes. Therefore, our objective in *Catalysis* is to have a tool that generates the latter automatically, so that we can perform any necessary proofs about pictorial frameworks.

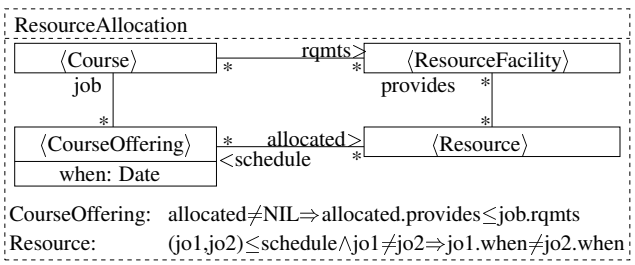
Again for lack of space, we will not present the pictorial notation formally. Instead we will rely on examples to illustrate it.

4.2 Open Frameworks

Example 4.1 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. To allow objects to have multiple roles in different frameworks, and to compose them by composing frameworks, as depicted in Figure 2, we need to use open frameworks. In this section, we show how to construct such frameworks in *Catalysis*.

Example 4.2 Suppose, as a professional OO designer, I am called to analyse a domestic trades agency, which sends people to do plumbing or electrical work and the like. I find that 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.

While drawing the framework, it occurs to me that this kind of scheduling problem must occur in many situations. So I generalise the type names a bit, so that the framework is:



Thus a pictorial open framework consists of (type diagrams of) interacting objects (or more accurately, generic classes) and *invariants* (or axioms).

The framework in the previous diagram is a simple open framework: it is a generic template, which can be put in a library of re-usable design pieces.

To the client, I deliver a copy of the ResourceAllocation framework, together with the assertion:

```
ResourceAllocation
   [ ResourceFacility  \ Skill,
     Resource          \ WorkPerson ]
```

which means 'make a copy of ResourceAllocation, substituting the names as in the bracketed list'. This provides the client with the framework they need.

5. Operations on Frameworks

In order to use open frameworks as templates for constructing frameworks, we need to define operations on them, together with their formal semantics. In this section, we do so for the model-theoretical semantics of frameworks introduced in Section 3.

5.1. Signature and Theory Morphisms

We recall that a framework is a (many-sorted) first-order theory, with a signature Σ consisting of a set S of sort symbols, a set F of function symbols, and a set R of relation symbols. Now first-order theories are an institution [8], so we need to introduce some properties of an institution, and some terminology.

Definition 5.1 A signature $\Sigma = \langle S, F, R\rangle$ is a *subsignature* of $\Delta = \langle S', F', R'\rangle$, written $\Sigma \preceq \Delta$, iff $S \subseteq S'$, $F \subseteq F'$ and $R \subseteq R'$.

Let ρ be a function that maps the symbols of a signature Σ into another signature Δ. Then ρ *generates* $\rho(\Sigma) = \langle \rho(S), \rho(F), \rho(R)\rangle$, where $\rho(S)$ is the image of S, $\rho(F)$ contains the declarations $\rho(f) : \rho(a) \Rightarrow \rho(s)$ such that $f : a \Rightarrow s \in F$, and $\rho(R)$ the declarations $\rho(r) : \rho(a)$ such that $r : a \in R$.

If $\rho(\Sigma) \preceq \Delta$, then ρ is a *signature morphism* from Σ to Δ. If it is an injective map, then ρ is a *signature extension*. If bijective, it is a *renaming*.

Let $\rho : \Sigma \Rightarrow \Delta$ be a signature morphism. The interpretations $int(\Delta)$ of Δ are related to the interpretations $int(\Sigma)$ of Σ by the *reduct* operation $|\rho : int(\Delta) \Rightarrow int(\Sigma)$ defined thus:

$$\begin{array}{lcl} s^{\mathsf{i}|\rho} & = & \rho(s)^{\mathsf{i}} \\ f^{\mathsf{i}|\rho} : a \Rightarrow s & = & \rho(f)^{\mathsf{i}} : \rho(a) \Rightarrow \rho(s) \\ r^{\mathsf{i}|\rho} : a & = & \rho(r)^{\mathsf{i}} : \rho(a) \end{array}$$

$\mathsf{i}|\rho$ is the *ρ-reduct* of i.

If $\rho(\sigma) = \sigma$ for every symbol σ, i.e. $\Sigma \preceq \Delta$, we get the usual notion of reduct to a subsignature (see e.g. [2]). In this case, the reduct will also be denoted by $\mathsf{i}|\Sigma$.

It is immediate to extend a morphism $\rho : \Sigma \Rightarrow \Delta$ to a map $\rho : L_\Sigma \Rightarrow L_\Delta$, where L_Σ and L_Δ are the languages generated by Σ and Δ respectively, and the following satisfaction condition (see [8]) can be easily proved:

Theorem 5.1 For every closed Σ-formula F and every Δ-interpretation i, $\mathsf{i} \models \rho(F) \Leftrightarrow \mathsf{i}|\rho \models F$.

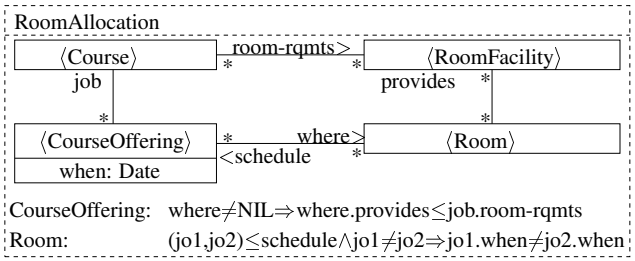
Finally, given a signature morphism $\rho : \Sigma \Rightarrow \Delta$ and a theory $\mathcal{T} \subseteq L_\Sigma$, we denote the *ρ-image* of $\mathcal{T}$ by $\rho(\mathcal{T})$.

5.2. Instantiating Frameworks

An open framework can be instantiated by a renaming of its signature. To see an example, we return to the pictorial notation.

Example 5.1 Continuing from Example 4.2, supposing the following week, I am asked to model the scheduling 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, ...).

I spot that this is a resource allocation problem — two such problems combined, actually, which can be modelled by two instances of the ResourceAllocation framework, one for RoomAllocation and one for InstructorAllocation. These instances can be obtained by a renaming of the signature of ResourceAllocation. For example, the framework for RoomAllocation is:



A similar framework can be constructed for InstructorAllocation.

5.3 Composing Frameworks

To discuss the composition of frameworks, 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 $\mathcal{G}(\Pi_1) = \langle \Sigma_1, \mathcal{T}_1, \mathcal{TH}_1, \Pi_1 \rangle$ 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 proof obligation $\rho(\mathcal{T}|\Pi) \subseteq \mathcal{T}_1 \cup \mathcal{TH}_1$ is satisfied. That is, we require that the p-axioms of $\mathcal{F}$ become theorems or axioms of $\mathcal{F}_1$.

Definition 5.2 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 $\mathcal{F}_1$. $\mathcal{TH}_c = \mathcal{TH}_1 \cup (\rho(\mathcal{TH}) - \mathcal{T}_1)$ If ρ is the identity, we indicate the corresponding composition by $\mathcal{F}[\mathcal{F}_1]$.

Composition is associative, i.e. we have the theorem:

Theorem 5.2 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.)

Moreover, it is related to framework extension, i.e. signature extension of a framework:

Definition 5.3 Let $\mathcal{F}(\Pi) = \langle \Sigma, \mathcal{T}, \mathcal{TH}, \Pi \rangle$ and $\mathcal{G}(\Pi') = \langle \Sigma', \mathcal{T}', \mathcal{TH}', \Pi' \rangle$ be two frameworks, and $\rho : \Sigma \Rightarrow \Sigma'$ be a signature extension. $\mathcal{G}(\Pi')$ is a ρ-*extension* of $\mathcal{F}$, written $\mathcal{F}(\Pi) \preceq_\rho \mathcal{G}(\Pi')$,[1] iff: $\rho(\mathcal{T}) \subseteq \mathcal{T}' \cup \mathcal{TH}'$; and for every σ not used in Π, $\rho(\sigma)$ does not occur in Π'.

We can easily prove the following theorem:

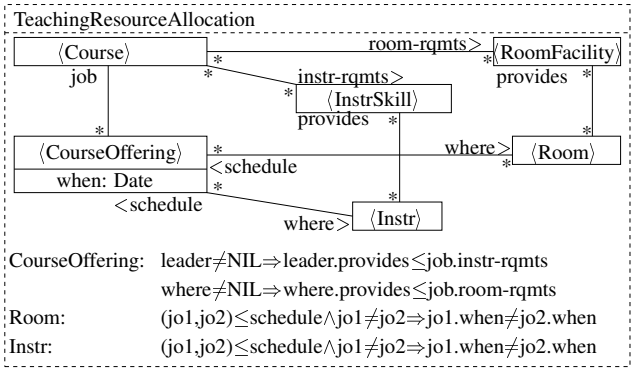
Theorem 5.3 Let $\mathcal{F}(\Pi)$ and $\mathcal{F}_1(\Pi_1)$ be two frameworks. If $\mathcal{F}[\rho, \mathcal{F}_1]$ is defined, then $\mathcal{F}(\Pi) \preceq_\rho \mathcal{F}[\rho, \mathcal{F}_1](\Pi_1)$ and $\mathcal{F}_1(\Pi_1) \preceq \mathcal{F}[\rho, \mathcal{F}_1](\Pi_1)$.

These theorems can be used to introduce sound *extension rules* as compositions of a particular kind (see [14] for a discussion).

Example 5.2 Continuing from Example 5.1, we can now complete the framework for Teaching Resource Allocation by composing the renamed instances RoomAllocation and InstructorAllocation of ResourceAllocation. This is shown as follows:[2]

[1] If ρ is the identity, we write $\mathcal{F}(\Pi) \preceq \mathcal{G}(\Pi')$.

[2] In the sequel, we shall abbreviate Instructor to Instr.



The invariants in TeachingResourceAllocation are just the conjunction (or union) of those in RoomAllocation and InstructorAllocation.

What does this mean? Just that the model for the teaching outfit is made by taking together all the types and assertions implied by both instantiations of the framework. If there are extra things I want to say about this particular model, then I only have to add them in. For example, we may need to record when an Instructor acquired a given skill, and who certified her. In practice, I would probably also add some operations to the types; although it is useful just as a model, providing a common pattern of relationships and invariants.

A more interesting and powerful case of framework composition involves frameworks that interact with one another via parameters specified as *external* (*abstract*) *operations*. These parameters are relations governed by p-axioms. However, to describe these frameworks and their composition formally, we need a formal semantics for objects with states, i.e. a state-based refinement of our model-theoretic semantics for frameworks. As we already pointed out, we will not give such a semantics in this paper. Nevertheless, we will briefly indicate how our pictorial notation can be used to describe such frameworks and their composition. We leave the corresponding formal semantics to another paper.

A framework that interacts with others via external operations can be represented in our pictorial notation, but with three sections. The first section would consist of the usual type diagram. The middle section would define the *internal* structure: it defines what goes on between the participants, i.e. it specifies the *internal* abstract operations, by means of postconditions (on a state space). For example, if the framework is a Retailer, then these operations would probably include make-order, deliver, pay, etc. These are interactions that take place only between members of the types (or their subtypes). Their postconditions show the effects on the participants.

The third 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. Thus in this section, we may have *invariants* which apply only to operations belonging to other frameworks in which the objects take part. The operations of this framework do not themselves have to observe such invariants (though they might be used to help maintain them).

An (internal or external) 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 [13]. Both participants are named in the headers and referred to in the postcondition.

The set of operations on the types involved need not be complete. For example, in a Retailer framework, we may have an operation that has the effect of increasing the stock, and we may have an invariant aimed at restoring it when it gets low, but there need not be operations that deplete the stock. This is the point of a 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.

6. Conclusion

Frameworks are increasingly recognised as the most useful unit of OO Design. In this paper, we have described frameworks at two levels. At a very abstract level, we have formally defined their model-theoretic semantics (and informally defined their syntax). The purpose of this is to define a completely declarative specification language. Such a language and its model-theoretic semantics are essential for reasoning formally about framework construction and operations on frameworks.

However, for practical system development using OO Design, we need a pictorial notation for specifying objects and their roles in frameworks. Nevertheless, such a notation should also have a formal semantics which should be a state-based refinement of the model-theoretic semantics of frameworks. Without detailing how this refinement works

(it is beyond the scope of this paper), we have described frameworks at another level, i.e. at the level of pictures with a state-based semantics for objects, frameworks and their interactions.

Within the *Catalysis* project, we are trying to formalise the semantics of our pictorial notation, and to design and implement a tool for generating formal semantics automatically from pictorial frameworks. Such a tool would bridge the gap between the two levels of semantics used in this paper. Moreover, it would provide an essential basis for practical OO Design.

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] C.C. Chang and H.J. Keisler. *Model Theory*. North-Holland, 1973.

[3] D. Coleman *et al*. *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. *Component-Based Development using Catalysis*. Draft book.

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

[8] J.A. Goguen and R.M. Burstall. Institutions: Abstract model theory for specification and programming. *J. ACM* 39(1):95–146, 1992.

[9] 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.

[10] 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.

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

[12] W. Harrison, H. Osher, and H. Mili. Subjectivity in object-oriented systems. In workshop reports, OOPSLA 94 & 95.

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

[14] 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.

[15] 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.

[16] K.-K. Lau and M. Ornaghi. Towards an object-oriented methodology for deductive synthesis of logic programs. In M. Proietti, editor, *Proc. LOPSTR 95*, *LNCS* 1048, pages 152–169, Springer-Verlag, 1996.

[17] 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.

[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] O. Nierstrasz and D. Tsichritzis, editors, *Object-Oriented Software Composition*. Prentice-Hall International, 1994.

[20] T. Reenskaug *et al*. *Working with Objects*. Manning/Prentice-Hall, 1995.

[21] D. Sannella and A. Tarlecki. Essential concepts of algebraic specification and program development. To appear in *Formal Aspects of Computer Science*.

[22] A. Wills. Frameworks. In Sun, Patel, and Sun, editors, *Proc. Int. Conf. OO Info. Syst. 1996*.

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