

Interconnection of Object Specifications

Grant Malcolm

We present a very simple account of interconnections of systems of distributed, concurrent, interacting objects. We give an abstract definition of object class specifications, and show how these may be composed into larger systems in a way that captures complex objects and parallel composition with synchronisation. The distributed autonomy of objects is one of the key concepts in object-orientation, and we hope that this simple treatment of parallel composition will clarify the most important aspects of the hierarchical structure of distributed systems of objects.

Of particular interest is our use of commutativity properties to give a *truly concurrent* model of parallelism in distributed systems, in contrast to *interleaving* models of parallelism. If a and b are atomic events that occur independently in distributed components of a system, then in an interleaving model of the parallel composition of a and b , either a happens before b , or b before a . In our approach, we simply deny the relevance of any temporal ordering to the execution of independent events a and b by asserting that ‘ a then b ’ is the same as ‘ b then a ’. In other words, independence of events is expressed by commutativity properties.

The use of commutativity properties to capture true concurrency dates back at least as far as the work of Mazurkiewicz (for example [27]). Monteiro and Pereira [28] explicitly use limits of monoids to obtain commutativity properties for a sheaf theoretic model of concurrency. Our approach to interconnections of objects is similar in many respects to Monteiro and Pereira’s sheaf theoretic approach, and also relies on work by Goguen on categorical systems theory (see [9, 11, 13]; applications of this work to object-orientation can be found in Goguen [16], Ehrich *et al.* [8], Wolfram and Goguen [31], and Cirstea [4]).

We consider objects to have a local, hidden state, aspects of which may be observed by means of *attributes*, and which may be updated by *methods*. Thus, an object class is specified by declaring a set of attributes, a set of methods, and stating how the methods affect the values of the attributes. This is the basis of the hidden sorted algebra approach to object specification [15, 18, 19, 26]. Hidden sorted algebra is a variety of many sorted algebraic specification which uses hidden sorts to model the hidden local states of objects, and visible sorts to model the data values that objects manipulate.

Goguen and Diaconescu [18] have proposed a construction called the ‘independent sum’ to capture the parallel connection with synchronisation of systems of interacting objects. A goal of the present paper is to investigate in a simple and abstract way some of the ideas underlying the independent sum construction. Let us illustrate this construction by means of an example (for technical details, see Goguen and Diaconescu, *loc. cit.*).

The following specification defines a class of cells which store natural numbers; the specification imports a module called **DATA**, which we assume defines the data types used in the object class specification, in this case at least the

sort `Nat` of natural numbers.

```
obj X is pr DATA .
  hsort h .
  op init : -> h .
  op getx : h -> Nat .
  op putx : h Nat -> h .
  var H : h .   var N : Nat .
  eq getx(init) = 0 .
  eq getx(putx(H,N)) = N .
endo
```

The local state of these cells is represented by the hidden sort `h`, and each cell has an assignment method `putx` to update its state and an attribute `getx` which returns the value currently held in the cell. The constant `init` of sort `h` represents the initialisation, or creation, of the cell.

Goguen and Diaconescu consider such a specification as describing the behaviour of a cell object, and introduce a construction called ‘independent sum’ for composing systems of interacting objects out of such specifications. For example, suppose the above specification defines the behaviour of a cell `X`, and suppose the cell `Y` is defined by a similar specification, with all `x`’s replaced by `y`’s. Then the behaviour of the system consisting of `X` and `Y` operating in parallel is defined by the following specification:

```
obj X||Y is pr DATA .
  hsort h .
  op init : -> h .
  op getx : h -> Nat .
  op putx : h Nat -> h .
  op gety : h -> Nat .
  op puty : h Nat -> h .
  var H : h .   vars M N : Nat .
  eq getx(init) = 0 .
  eq getx(putx(H,N)) = N .
  eq gety(init) = 0 .
  eq gety(puty(H,N)) = N .
  eq getx(puty(H,N)) = getx(H) .
  eq gety(putx(H,N)) = gety(H) .
  eq putx(puty(H,M),N) = puty(putx(H,N),M) .
endo
```

Note that while `h` represents the state of `X` in the first specification, in the second specification `h` represents the product of the states of `X` and `Y`, i.e., it represents the state of the composite system. The first four equations in the specification `X||Y` are taken directly from `X` and from `Y`; the final three equations state that `X` and `Y` operate independently: assignment to `X` doesn’t affect the value of `Y`’s attribute, and vice-versa. The final equation is particularly relevant to subsequent sections: the independence of `X` and `Y` is captured by stating that assignment to `X` *commutes* with assignment to `Y`.

The independent sum `X||Y` represents the parallel connection of `X` and `Y`, synchronised on `init` (i.e., `X||Y` has just one initialisation, which synchronously

initialises X and Y). The process can be repeated to give a hierarchical construction of large systems. For example, if we have a third cell, Z , then we could form $Y||Z$, which we could then conjoin with $X||Y$ to give $(X||Y)||_Y(Y||Z)$. The subscripted Y indicates that the common part Y is to be shared, that is, the independent sum will not make duplicate versions of the object Y . This last composite object might be thought of as the pushout of the inclusions of Y into $X||Y$ and $Y||Z$; in fact, it is not a pushout, though Goguen and Diaconescu give a universal characterisation for the independent sum construction.

In the following sections, we investigate other universal ways of capturing parallel connection with synchronisation. Section 1 gives some preliminary technical background; Section 2 discusses a basic approach to interconnections of simple classes of processes; Sections 3 and 4 discuss interconnections of object and systems of objects, respectively. A key idea in what follows is the use of commutativity properties to capture true concurrency, similar to the commutativity axioms in the independent sum.

Acknowledgements

The research reported in this paper has been supported in part by the CEC under ESPRIT-2 BRA Working Groups 6071, IS-CORE (Information Systems CORrectness and REusability) and 6112, COMPASS (COMPrehensive Algebraic Approach to System Specification and development), and a contract under the management of the Information Technology Promotion Agency (IPA), Japan, as part of the Industrial Science and Technology Frontier Program ‘New Models for Software Architectures’, sponsored by NEDO (New Energy and Industrial Technology Development Organization).

1 Some Preliminaries

In order to make our account reasonably self-contained, we begin with brief summaries of some technical notions used in the sequel. The main concepts we require are: categories, functors, limits, monoids, right actions of monoids, and sheaves. These are introduced in the following subsections; readers already familiar with the concepts can skip to Section 2.

1.1 Categories

Category theory allows an attractively abstract treatment of many constructions in mathematics and computer science. We give only a few basic definitions here; good introductions can be found in [3, 14, 23, 29].

A **category** C consists of the following:

- a class $|C|$ of **objects**;
- a class $||C||$ of **morphisms** (sometimes called ‘arrows’);
- two maps $\partial_0, \partial_1 : ||C|| \rightarrow |C|$, which give, respectively, the **source** and **target** of a morphism;
- for each $c \in |C|$, a distinguished morphism 1_c called the **identity**, with $\partial_0(1_c) = c = \partial_1(1_c)$;

- a partial operation $_ ; _ : \|\mathbf{C}\| \times \|\mathbf{C}\| \rightarrow \|\mathbf{C}\|$, called **composition**, which is defined on (f, g) when $\partial_1(f) = \partial_0(g)$, in which case $\partial_0(f ; g) = \partial_0(f)$ and $\partial_1(f ; g) = \partial_1(g)$.

Moreover, it is required that the following axioms be satisfied:

- Identities are neutral elements of composition, in that for each $c \in |\mathbf{C}|$ and $f, g \in \|\mathbf{C}\|$,

$$1_c ; f = f \quad \text{and} \quad g ; 1_c = g$$

whenever these compositions are defined;

- composition is associative, in that for all $f, g, h \in \|\mathbf{C}\|$,

$$f ; (g ; h) = (f ; g) ; h$$

whenever both sides are defined.

For $f \in \|\mathbf{C}\|$ we write $f : a \rightarrow b$ iff $\partial_0(f) = a$ and $\partial_1(f) = b$. The same information can be presented pictorially:

$$a \xrightarrow{f} b$$

and if $g : b \rightarrow c$, then the composite $f ; g : a \rightarrow c$ can be depicted:

$$a \xrightarrow{f} b \xrightarrow{g} c$$

Moreover, if $h : c \rightarrow d$, then the equation $f ; (g ; h) = (f ; g) ; h$ can be depicted thus:

$$\begin{array}{ccc}
 a & \xrightarrow{f ; g} & c \\
 f \downarrow & & \downarrow h \\
 b & \xrightarrow{g ; h} & d
 \end{array}$$

Such a picture is referred to as a **commuting diagram**: this means that any two paths in the diagram that start and end at the same points are equal.

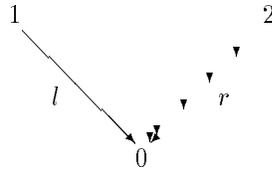
The ‘objects’ of a category are different from the ‘objects’ of object-orientation; if there is any risk of confusing the two, we will refer to the former as ‘category objects’, and the latter simply as ‘objects’.

An important example of a category is **Sets**, whose objects are sets, and whose morphisms are functions (or, more precisely, triples (A, f, B) where A and B are sets and f is a map from A to B ; then $\partial_0(A, f, B) = A$ and $\partial_1(A, f, B) = B$). Composition of morphisms is just composition of functions. (We write composition in ‘diagrammatic order’, i.e., in the same direction as the arrows. The more usual notation for $f ; g$ is $g \circ f$.)

Another example is the category that arises from a partial order. If (C, \leq) is a partial order, then we can construct a category, which we also call (C, \leq) , whose objects are the elements of C , and whose morphisms are pairs of elements (a, b) such that $a \leq b$. In particular, there is a morphism $(a, b) : a \rightarrow b$ iff $a \leq b$. If $a \leq b$ and $b \leq c$, then the composite $(a, b); (b, c)$ is $(a, c) : a \rightarrow c$.

Given any category C , its **opposite** category C^{op} is obtained by turning around the arrows of C . Specifically, the objects of C^{op} are the objects of C , and the morphisms are those of C , but $f : a \rightarrow b$ in C^{op} iff $f : b \rightarrow a$ in C . For example, if (C, \leq) is the category arising from a partial order, then $(C, \leq)^{\text{op}}$ is the category arising from the partial order (C, \geq) , where $c \geq c'$ iff $c \leq c'$.

An example of a very artificial category is the category whose objects are 0, 1, and 2, with identity morphisms for each of these objects, and with a morphism $l : 1 \rightarrow 0$ and another $r : 2 \rightarrow 0$. This category can be pictured as a diagram:



The identity morphisms have been omitted from this diagram. In following sections we discuss the relationship between this category and synchronisation: for future reference, let us call this category *Sync*. We might think of *Sync* as denoting a system consisting of two objects, 1 and 2, which are synchronised on a common subobject, 0. In fact, this rather strange example of a category can be thought of as a pictorial representation of such a system, and an instance of such a system would be obtained by instantiating the category objects 0, 1 and 2 with objects, and instantiating the arrows l and r with morphisms between those objects which express the relevant subobject relationship. Such instantiations may be achieved by *functors*.

1.2 Functors and Natural Transformations

Given two categories C and D , a **functor from C to D** is a pair $F = (|F|, \|F\|)$, where $|F|$ maps objects of C to objects of D , i.e., $|F| : |C| \rightarrow |D|$, and $\|F\|$ maps morphisms of C to morphisms of D in such a way that:

- for all $f \in \|C\|$, if $f : a \rightarrow b$ then $\|F\|(f) : |F|(a) \rightarrow |F|(b)$;
- $\|F\|(1_a) = 1_{|F|(a)}$; and
- if $f : a \rightarrow b$ and $g : b \rightarrow c$ are morphisms of C , then $\|F\|(f; g) = \|F\|(f); \|F\|(g)$.

We write $F : C \rightarrow D$ to denote that F is a functor from C to D .

From now on, if F is a functor we will write F for both $|F|$ and $\|F\|$, and Fx instead of $F(x)$, for x an object or an arrow, so that, for example, the main features of functors can be summarised concisely as follows.

- for all $f \in \|\mathbf{C}\|$, if $f : a \rightarrow b$ then $Ff : Fa \rightarrow Fb$;
- $F 1_a = 1_{Fa}$; and
- if $f : a \rightarrow b$ and $g : b \rightarrow c$ are morphisms of \mathbf{C} , then $F(f;g) = Ff;Fg$.

An example of a functor is the *identity* functor, $I : \mathbf{Sets} \rightarrow \mathbf{Sets}$, defined by $Ia = a$ for each set a , and $If = f$ for each function f . Another example is $List : \mathbf{Sets} \rightarrow \mathbf{Sets}$, defined by $List a = a^*$, the set of lists over the set a , and $List f = f^*$, the function that applies $f : a \rightarrow b$ componentwise to a list in a^* , giving a list in b^* as result.

Given two functors $F, G : \mathbf{C} \rightarrow \mathbf{D}$, a **natural transformation** η from F to G is a family $\eta_a : F(a) \rightarrow G(a)$ of morphisms of \mathbf{D} indexed by objects $a \in |\mathbf{C}|$, such that for all morphisms $f : a \rightarrow b$ of \mathbf{C} , the following diagram commutes.

$$\begin{array}{ccc}
 a & & Fa \xrightarrow{\eta_a} Ga \\
 f \downarrow & & Ff \downarrow \quad \quad \downarrow Gf \\
 b & & Fb \xrightarrow{\eta_b} Gb
 \end{array}$$

That is, $Ff; \eta_b = \eta_a; Gf$.

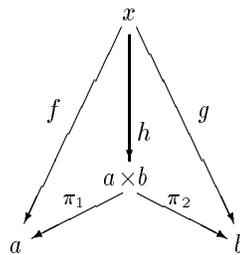
For example, the family of functions η_a that take $x \in a$ to the singleton list $\langle x \rangle \in a^*$ is a natural transformation from I to $List$.

1.3 Limits

Given two objects a and b in a category \mathbf{C} , a **product** of a and b is an object $a \times b$ of \mathbf{C} together with two morphisms $\pi_1 : a \times b \rightarrow a$ and $\pi_2 : a \times b \rightarrow b$ such that for any pair of morphisms $f : x \rightarrow a$ and $g : x \rightarrow b$ there exists a unique morphism $h : x \rightarrow a \times b$ such that

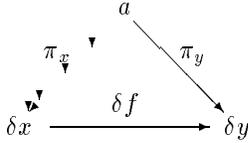
$$h; \pi_1 = f \quad \text{and} \quad h; \pi_2 = g .$$

The following diagram represents this situation.



For example, in the category \mathbf{Sets} products are given by Cartesian products with the standard projection functions. In the category arising from a partial order (C, \leq) , products correspond to greatest lower bounds.

More generally, limits are defined as follows. A **diagram** (in a category \mathbf{C}) is a functor $\delta : \mathbf{X} \rightarrow \mathbf{C}$, where the category \mathbf{X} is called the **shape** of the diagram. A **cone** for a diagram $\delta : \mathbf{X} \rightarrow \mathbf{C}$ is an object a of \mathbf{C} together with a family of morphisms $\pi_x : a \rightarrow \delta x$ for each $x \in |\mathbf{X}|$ such that $\pi_x ; \delta f = \pi_y$ for each $f : x \rightarrow y$ in \mathbf{X} , as in the following diagram.

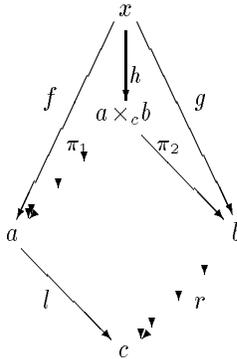


We write $\pi : a \Rightarrow \delta$ to indicate that a and the family of morphisms π_x are a cone for δ .

A **limit** of a diagram $\delta : \mathbf{X} \rightarrow \mathbf{C}$ is a cone $\pi : a \Rightarrow \delta$ such that for any other cone $\rho : b \Rightarrow \delta$ there is a unique morphism $h : b \rightarrow a$ such that $h ; \pi_x = \rho_x$ for all $x \in |\mathbf{X}|$.

As an example of this general situation, products in a category \mathbf{C} are limits of diagrams $\delta : \mathbf{X} \rightarrow \mathbf{C}$, where \mathbf{X} is the category with two objects 1 and 2 and no arrows (except identities). The product $a \times b$, together with the two projections π_1 and π_2 , is a limit of the diagram δ which maps 1 to the object a and 2 to b .

Another important example of limits is the following, which will be useful for talking about parallel composition with synchronisation. Suppose we are given two morphisms $l : a \rightarrow c$ and $r : b \rightarrow c$ in a category \mathbf{C} (this is the same as supposing there is a diagram $\delta : \mathbf{Sync} \rightarrow \mathbf{C}$, where \mathbf{Sync} is the category pictured at the end of Section 1.1). A **pullback** (of l and r) is an object $a \times_c b$ of \mathbf{C} together with two morphisms $\pi_1 : a \times_c b \rightarrow a$ and $\pi_2 : a \times_c b \rightarrow b$ such that $\pi_1 ; l = \pi_2 ; r$ (that is, $\pi : a \times_c b \Rightarrow \delta$), with the ‘universal property’ that for any $f : x \rightarrow a$ and $g : x \rightarrow b$ such that $f ; l = g ; r$ there is a unique $h : x \rightarrow a \times_c b$ such that $h ; \pi_1 = f$ and $h ; \pi_2 = g$.



We call $a \times_c b$ the **pullback object**. For example, in **Sets** the pullback object of l and r would be $\{(x, y) \in a \times b \mid l(x) = r(y)\}$. Pullbacks, and limits in general, can be thought of as combining the objects in a given diagram in a minimal way such as to make all the projections commute with the arrows of the diagram.

In following sections, we use limits to model parallel composition of systems of object specifications. It is useful to know when an arbitrary diagram of such specifications can be composed: we say that a category \mathbf{C} is **complete** iff there is a limit in \mathbf{C} for every diagram $\delta : X \rightarrow \mathbf{C}$.

1.4 Monoids

Objects have state, and the specification of a class of objects fixes the visible attributes of an object's state, as well as the methods by which that state may be changed. An object class is therefore an abstraction of objects in the real world, which may reveal unexpected facets, or be altered by all manner of unforeseen circumstances. Abstracting yet further from reality, we might proscribe the simultaneous effect of two or more methods on an object's state; doing so, we impose a monoid structure on the fixed set of methods proper to an object class. Applying methods one after the other corresponds to multiplication in the monoid, and applying no methods corresponds to the identity of the monoid.

A **monoid** is a set M with an associative binary operation $\bullet_{\mathcal{M}} : M \times M \rightarrow M$, usually referred to as 'multiplication', which has an identity element $e_{\mathcal{M}} \in M$. If $\mathcal{M} = (M, \bullet_{\mathcal{M}}, e_{\mathcal{M}})$ is a monoid, we often write just M for \mathcal{M} , and e for $e_{\mathcal{M}}$; moreover for $m, m' \in M$, we usually write $m m'$ instead of $m \bullet_{\mathcal{M}} m'$.

For example, A^* , the set of lists containing elements of A , together with concatenation $++ : A^* \times A^* \rightarrow A^*$ and the empty list $[] \in A^*$, is a monoid. This example is especially important for the material in later sections.

A **monoid homomorphism** is a structure preserving map between the carriers of two monoids. In other words, $f : (M, \bullet_M, e_M) \rightarrow (N, \bullet_N, e_N)$ means that f is a map $M \rightarrow N$ which distributes over multiplication and preserves identities. That is,

$$\begin{aligned} f(m \bullet_M m') &= f(m) \bullet_N f(m') \\ f(e_M) &= e_N \end{aligned}$$

for all $m, m' \in M$.

This gives us a category \mathbf{Mon} of monoids and monoid homomorphisms. Moreover, \mathbf{Mon} is complete, and there is an interesting relationship between limits in \mathbf{Mon} and independent sums. For example, consider the following diagram in \mathbf{Mon} .

$$\begin{array}{ccc} (M, \bullet_M, e_M) & & (N, \bullet_N, e_N) \\ & \searrow f & \downarrow g \\ & & (L, \bullet_L, e_L) \end{array}$$

The pullback object of this diagram in \mathbf{Mon} has

$$\{(m, n) \in M \times N \mid f(m) = g(n)\}$$

as carrier. Multiplication in the pullback object is defined componentwise

$$(m, n) \bullet (m', n') = (m \bullet_M m', n \bullet_N n')$$

and the identity is (e_M, e_N) .

We might think of M and N as being specifications of objects which share a common subobject L , and the morphisms f and g as restricting programs of M and N to programs of L . The pullback object represents the parallel composition of M and N synchronised at L , or their independent sum. An element $m \in M$ gives rise to an element $\hat{m} = (m, e_N)$ of the pullback object, and similarly for $n \in N$. These elements satisfy the commutativity axioms of independent sums, because

$$\hat{m} \hat{n} = (m, e_N) (e_M, n) = (m, n) = (e_M, n) (m, e_N) = \hat{n} \hat{m} .$$

The program $\hat{m} \hat{n}$ can be seen as representing the parallel composition of the programs m and n . As with independent sums, the commutativity of their composition expresses the irrelevance of any temporal ordering of their execution.

1.5 Right Actions of Monoids

In following sections we simply view the methods of an object class as a monoid; a natural extension of this analogy is to view an object's state as a *right action* of its methods.

A **right action** of a monoid $\mathcal{M} = (M, \bullet, e)$ consists of a set S and a binary operation $\oplus : S \times M \rightarrow S$ such that

$$\begin{aligned} s \oplus e &= s \\ s \oplus (m \bullet m') &= (s \oplus m) \oplus m' \end{aligned}$$

for all $s \in S$ and $m, m' \in M$.

If (S, \oplus) and (T, \otimes) are right actions of \mathcal{M} , a morphism of right actions $f : (S, \oplus) \rightarrow (T, \otimes)$ is a map $f : S \rightarrow T$ such that

$$f(s \oplus m) = f(s) \otimes m .$$

This gives us a category $\text{RA}(\mathcal{M})$ of right actions of \mathcal{M} . This category is also complete. For example, the pullback object of

$$(S, \oplus) \xrightarrow{f} (T, \otimes) \xleftarrow{g} (U, \ominus)$$

in $\text{RA}(\mathcal{M})$ is the right action (P, \odot) , where

$$P = \{(s, u) \in S \times U \mid f(s) = g(u)\}$$

and \odot is defined by

$$(s, u) \odot m = (s \oplus m, u \ominus m) .$$

If \mathcal{M} is the monoid of methods in the specification of an object class, and if we ignore the attributes in the specification, it is possible to think of a right action of \mathcal{M} as being a ‘model’ or ‘implementation’ of the specification. A useful property of such ‘models’ is that they move back along monoid morphisms (cf. the same property of models and signature morphisms in the institutions of Goguen and Burstall [17]).

Proposition 1 Every monoid homomorphism $f : \mathcal{M} \rightarrow \mathcal{N}$ induces a functor $\text{RA}(f) : \text{RA}(\mathcal{N}) \rightarrow \text{RA}(\mathcal{M})$.

Proof. Let (S, \oplus) be a right action of \mathcal{N} . We define $\text{RA}(f)(S, \oplus)$ to be (S, \oplus_f) , where

$$s \oplus_f m = s \oplus f(m) .$$

It is straightforward to check that this is a right action of \mathcal{M} . Moreover for $g : (S, \oplus) \rightarrow (T, \otimes)$ a morphism of right actions on \mathcal{N} , we define $\text{RA}(f)(g)$ to be just $g : (S, \oplus_f) \rightarrow (T, \otimes_f)$. That this is a morphism of right actions follows from the following equalities:

$$g(s \oplus_f m) = g(s \oplus f(m)) = g(s) \otimes f(m) = g(s) \otimes_f m .$$

□

1.6 Sheaves

Sheaf theory is used in many branches of mathematics, the underlying theme in its various applications being the passage from local to global properties [20]. It provides a formal notion of coherent systems of observations: a number of consistent observations of various aspects of an object can be uniquely pasted together to give an observation over all of those aspects. The passage from local to global properties, and the pasting together of local observations of behaviour allow sheaf theory to be usefully applied in computer science, to give models for concurrent processes [28, 7, 25] and objects [16, 8, 31, 4]. We give a basic definition of ‘sheaf’ below; fuller accounts can be found in [30, 24].

We may consider a sheaf as giving a set of observations of an object’s behaviour from a variety of ‘locations’. The notion of location is formalised by the following

Definition 2 A **complete Heyting algebra** is a partially ordered set (C, \leq) such that:

- for all $c, d \in C$, there is a greatest lower bound $c \wedge d$
- for all subsets $\{c_i \mid i \in I\}$ of C , there is a least upper bound $\bigvee_{i \in I} c_i$
- greatest lower bounds distribute through least upper bounds:

$$\left(\bigvee_{i \in I} c_i \right) \wedge d = \bigvee_{i \in I} (c_i \wedge d) .$$

□

For example, any topological space with the subset inclusion ordering is a complete Heyting algebra. Another important example, the subsystems of a given system, is described in Section 4 below.

Definition 3 Let C be a complete Heyting algebra; a **presheaf** F on C is a functor from C^{op} to **Sets**. That is, for each $c \in C$ there is a set $F(c)$, and for $c, d \in C$ such that $c \leq d$, there is a **restriction function** $F_{c \leq d} : F(d) \rightarrow F(c)$, subject to the following conditions:

- $F_{c \leq c} = id_{F(c)}$, the identity on the set $F(c)$; and
- if $c \leq d \leq e$, then $F_{d \leq e}; F_{c \leq d} = F_{c \leq e}$.

□

Notation 4 For a presheaf F on C , if $c \leq d$ in C and $x \in F(d)$, we often write $x|_c$ for $F_{c \leq d}(x)$. □

Note that the observations given by a presheaf can be *structured* in the sense that presheaves can be defined as functors to some concrete category of structures. For example, we consider presheaves of monoids below, which are functors $C^{\text{op}} \rightarrow \text{Mon}$. In this case the restriction functions are required to preserve that structure; thus, for example, the restriction functions of presheaves of monoids are monoid homomorphisms.

A sheaf is a presheaf which allows families of consistent local observations to be pasted together to give a global observation.

Definition 5 A presheaf F is a **sheaf** iff it satisfies the following **pasting condition**:

- if $c = \bigvee_{i \in I} c_i$ and $x_i \in F(c_i)$ is a family of elements for $i \in I$ such that $x_i|_{c_i \wedge c_j} = x_j|_{c_i \wedge c_j}$ for all $i, j \in I$, then there is a unique $x \in F(c)$ such that $x|_{c_i} = x_i$ for all $i \in I$.

□

For example, suppose that a system is just a functor $F : C^{\text{op}} \rightarrow \text{Sets}$, where $c \leq c'$ means that c is a subsystem of c' , and where $F(c)$ is the set of observations that can be made of the subsystem c . Then if F is a sheaf, we can paste together, in a unique way, *local* observations at a set of subsystems c_i , to give a *global* observation of the system at $c = \bigvee_{i \in I} c_i$, provided that the local observations do not contradict each other. This idea is further developed in the following sections.

2 Process Classes

A class of objects is specified by declaring the attributes of objects in the class, by declaring the methods that operate upon their states, and by defining the effect of the methods on the attributes. If we ignore, for the moment, the attributes of objects, and concentrate on just the methods, then we arrive at a rather simplistic notion of *process* classes. This notion is too simplistic to have any intrinsic interest, but we use it to illustrate the ideas behind our approach to interconnections of objects. Perhaps the simplest way of specifying a class of processes is to state the set of events that the processes of that class can engage in. Thus we might say that a process class specification is just a set. This would correspond to specifying the alphabet of a process in a formalism such as CSP [21]. More generally, we might instead say that a process class specification is a monoid, which can be thought of as specifying either the programs that a class of processes can carry out, or the history traces of the events that it engages in. Using monoids rather than alphabets also has the

advantage that, for the purposes of comparing classes of processes, we can use monoid morphisms, rather than renaming of events, to compare the actions of processes.

Consider, for example, a clock as a process that can engage in only one event, a ‘tick’. The monoid of programs for the class of clocks is the set of lists $\{\surd\}^*$, where \surd denotes a tick of a clock. A slightly more complex example might be a process which can engage in two distinct events, say called ‘dot’ and ‘dash’, so that the corresponding monoid is $\{\text{dot}, \text{dash}\}^*$. Suppose we want to conjoin such a process with a clock in such a way that a dot lasts for one tick of the clock, but a dash takes two ticks. This is expressed by the monoid morphism

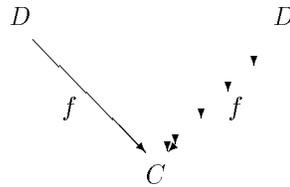
$$f : \{\text{dot}, \text{dash}\}^* \rightarrow \{\surd\}^*$$

where

$$\begin{aligned} f([\]) &= [\] \\ f(l \text{ dot}) &= f(l) \surd \\ f(l \text{ dash}) &= f(l) \surd \surd \end{aligned}$$

for all $l \in \{\text{dot}, \text{dash}\}^*$. The morphism f takes a list of dots and dashes and returns the amount of ticks of the clock required. For example, if a history of a ‘dot-dash’ process is the list ‘dot dash dot’, then the corresponding history of the clock process is ‘ $\surd \surd \surd \surd$ ’ — four ticks.

This morphism f can be used to synchronise two dot-dash processes which share the same clock process. The result of such a synchronisation should consist of histories of each of the dot-dash processes provided that these take the same number of ticks of the clock. For example, one process might go ‘dot dash dot’ while the other goes ‘dash dash’; this would be acceptable, as both of these histories require four ticks of the clock. Given two dot-dash processes with a common clock, the result of their parallel composition is the pullback object of the following diagram in \mathbf{Mon} , where $C = \{\surd\}^*$ and $D = \{\text{dot}, \text{dash}\}^*$.



The pullback object of this diagram is $\{(d_1, d_2) \in D \times D \mid f(d_1) = f(d_2)\}$. That is, given two lists d_1 and d_2 of dots and dashes, which represent histories of two processes, the pair (d_1, d_2) is a history of the parallel composition of the two processes provided that d_1 and d_2 both require the same number of ticks of the shared clock.

The point illustrated by this example is that systems can be thought of as diagrams, and that the behaviour of a system is its limit. In developing this for interconnections of systems of objects we are following ideas and results from categorical systems theory (cf. Goguen [9, 16]).

It is possible to contrive some rather curious examples of the behaviour of composite processes. Let

$$\begin{aligned} A &= \{a\}^* \\ B &= \{b\}^* \\ C &= \{a, b\}^* \\ D &= \{a, b\}^* \end{aligned}$$

be four processes, arranged as in Figure 1 where the morphisms between the

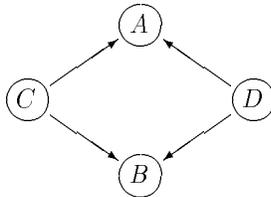


Figure 1: C and D observe A and B

processes are the obvious restriction functions, e.g., the arrow from C to A restricts a list over $\{a, b\}$ to a list over $\{a\}$. The idea is that A is a process which can participate only in the event a , B can participate only in the event b , and C and D are two processes which can either observe a happening in A or b happening in B . The behaviour of this system is the limit in \mathbf{Mon} , which has as carrier the set

$$\{(w, x, y, z) \in A \times B \times C \times D \mid y|_A = w = z|_A \wedge y|_B = w = z|_B\}.$$

An element of this set is (a, b, ab, ba) , because $ab|_A = a = ba|_A$ and $ab|_B = b = ba|_B$. This suggests that C observes a happening in A and then observes b happening in B , whereas D observes first b happening in B and *then* a happening in A . Monteiro and Pereira [28] refer to such a trace as an ‘impossible behaviour’, although we see nothing unacceptable with such *relativistic* behaviour, since none of the objects share a clock. Indeed, synchronising the system with a clock which is shared by all objects is the only reasonable way of avoiding such ‘impossible behaviours’.

3 Object Specification

In the previous section we outlined a relationship between limits in the category of monoids and the independent sum of process specifications. This relied on viewing the set of methods of a class of processes as a monoid. However, an object class specification also declares a number of attributes, which allow the observation of aspects of an object’s state. An object class specification should therefore also give a set D representing the data values that the attributes might take. If an object is to have more than one attribute, then the set D can be thought of as the set of tuples of all the attributes. Moreover, there should be some means of specifying the way in which methods affect an object’s

attributes. A simple way of achieving this is to provide a function that maps methods to attribute values: this function can be thought of as returning the attribute values after evaluating the given method in an object's initial state.

Definition 6 An **object class specification** is a triple (P, ε, D) , where P is a monoid, D is a set, and ε is a function from the carrier of P to D . We often write $P \xrightarrow{\varepsilon} D$ instead of (P, ε, D) . \square

For example, the specification of a stack object might be $P \xrightarrow{\varepsilon} D$, where:

- P is the set of lists over $\{\text{pop}\} \cup \{\text{push}(n) \mid n \in \omega\}$;
- D is the set $\{\text{error}\} \cup \omega$, representing the value on the top of the stack; and
- ε is defined by

$$\begin{aligned}\varepsilon([]) &= \text{error} \\ \varepsilon(p \text{ push}(n)) &= n \\ \varepsilon(p \text{ pop}) &= \varepsilon(\text{tail}(p))\end{aligned}$$

for all $p \in P$, where

$$\begin{aligned}\text{tail}([]) &= [] \\ \text{tail}(p \text{ push}(n)) &= p \\ \text{tail}(p \text{ pop}) &= \text{tail}(\text{tail}(p)).\end{aligned}$$

Consider two object class specifications (P, ε, D) and (P', ε', D') . A morphism from the one to the other should consist of a translation $f : P \rightarrow P'$ of programs and a translation $g : D \rightarrow D'$ of state attributes, such that for any program $p \in P$, if we translate p and then evaluate the result in (P', ε', D') , we should get the same result as evaluating p and then translating the resulting state attributes. In other words, we require that $\varepsilon'(f(p)) = g(\varepsilon(p))$, i.e., the following diagram commutes.

$$\begin{array}{ccc} P & \xrightarrow{\varepsilon} & D \\ f \downarrow & & \downarrow g \\ P' & \xrightarrow{\varepsilon'} & D' \end{array}$$

Note that f should be a monoid morphism, so, strictly speaking, we should have $U(f) : U(P) \rightarrow U(P')$ as the leftmost arrow in this diagram, where U is the forgetful functor from **Mon** to **Sets**; however, omitting U is in keeping with the notation of Section 1.4.

Definition 7 A **morphism of object class specifications**

$$(P, \varepsilon, D) \rightarrow (P', \varepsilon', D')$$

is a pair (f, g) , with $f : P \rightarrow P'$ a monoid morphism and $g : D \rightarrow D'$ in **Sets**, such that $\varepsilon ; g = f ; \varepsilon'$. This gives rise to a category **Obj** of object class specifications and morphisms. \square

Note that, because f is a monoid morphism rather than just a map between sets of methods, we allow methods to be translated into composite programs.

Inheritance is modelled by inclusions of object class specifications (cf. [8, 16, 5], for example). The intuition is that the inheriting specification specialises the inherited specification by adding more methods and more attributes. Thus if (P, ε, D) inherits from (P', ε', D') , there should be an inclusion $P' \hookrightarrow P$, and a restriction $g : D \rightarrow D'$ of state attributes.

Definition 8 A **containment of object class specifications** is a morphism $(f, g) : (P, \varepsilon, D) \rightarrow (P', \varepsilon', D')$ such that f has a pre-inverse $i_f : P' \rightarrow P$ (that is, $i_f ; f = id_{P'}$). This gives a subcategory **Obj_C** of **Obj** with object class specifications as objects and containments as morphisms. \square

This definition implies that $\varepsilon' = i_f ; \varepsilon ; g$; that is, the effect of inherited methods on state attributes is not altered.

In the previous section, we suggested that interconnection of objects, at least as far as methods are concerned, is captured by limits on the category of monoids. Moreover, it makes sense to use limits for the state attributes of aggregated objects: the state space of a composite object will be composed of the state spaces of its component parts in a coherent way. Therefore, the following proposition allows us to build specifications of classes of composite objects.

Proposition 9 The category **Obj** of object class specifications is complete. \square

In fact, limits in **Obj** are constructed by taking limits of methods and of state attributes. The following notation allows us to give an easy characterisation of limits in **Obj**.

Notation 10 Note that a diagram $X \rightarrow \mathbf{Obj}$ is the same thing as a functor $P : X \rightarrow \mathbf{Mon}$, a functor $D : X \rightarrow \mathbf{Sets}$, and a natural transformation $\varepsilon : P ; U \rightarrow D$, where U is the forgetful functor $\mathbf{Mon} \rightarrow \mathbf{Sets}$. Extending the notation of Definition 6, we write diagrams of object class specifications as $P \xrightarrow{\varepsilon} D$. \square

Proof of Proposition 9. Suppose $P \xrightarrow{\varepsilon} D$ is a diagram of object class specifications, and let $\pi : P' \Rightarrow P$ be the limit of P in **Mon**, and let $\varpi : D' \Rightarrow D$ be the limit of D in **Sets**. Then composing π with ε gives a cone $\pi ; \varepsilon : P' \Rightarrow D$ of D , which induces a morphism $\varepsilon' : P' \rightarrow D'$:

$$\begin{array}{ccc}
 P' & \xrightarrow{\varepsilon'} & D' \\
 \pi_x \downarrow & & \downarrow \varpi_x \\
 P x & \xrightarrow{\varepsilon_x} & D x
 \end{array}$$

It is clear from this diagram that $(\pi, \varpi) : (P' \xrightarrow{\varepsilon'} D') \Rightarrow (P \xrightarrow{\varepsilon} D)$ is a cone for $(P \xrightarrow{\varepsilon} D)$, and it is straightforward to check that it is a limiting cone. \square

This describes how to paste together an arbitrary diagram of object class specifications: parallel composition of objects is given by limits.

It is also possible to paste together *models* of object class specifications, where a model is something that evaluates methods in a way that agrees with the specification.

Definition 11 A **model** for an object class specification $P \xrightarrow{\varepsilon} D$ is a tuple (s, S, \oplus, α) , where (S, \oplus) is a right action of P , and $s \in S$ and $\alpha : S \rightarrow D$ are such that

$$\alpha(s \oplus p) = \varepsilon(p)$$

for all $p \in P$. \square

Given a diagram of object specifications and models for each of the specifications in the diagram, we can build a model for the limit of the diagram.

Proposition 12 Let $P \xrightarrow{\varepsilon} D : \mathbf{X} \rightarrow \mathbf{Obj}$ be a diagram of object class specifications, and suppose that for each $x \in |\mathbf{X}|$ we have a $(Px \xrightarrow{\varepsilon_x} Dx)$ -model $(s_x, Sx, \oplus_x, \alpha_x)$, where $S : \mathbf{X} \rightarrow \mathbf{Sets}$ is a functor, and $s : 1 \rightarrow S$, $\oplus : S \times P \rightarrow S$ and $\alpha : S \rightarrow D$ are natural transformations¹. Then these models give rise to a model of the limit of $P \xrightarrow{\varepsilon} D$. \square

This model is constructed by taking the limit of $S : \mathbf{X} \rightarrow \mathbf{Sets}$; the operations required to make this a model of the limit of $P \xrightarrow{\varepsilon} D$ can be constructed by using the universal properties of the limits of S , P and D . The details are left as an exercise for the reader.

4 System Specification

In the previous section we presented limits of diagrams of objects as a means of composing object class specifications into specifications of complex object classes. In this section we generalise this by considering systems of objects and interconnections of such systems. As in the previous section, our goal is to examine ways of composing systems of objects to form larger systems.

There are various ways of defining systems of objects. The most general way also seems to provide the richest structure for system specifications. The following definitions are due to Goguen [9, 11, 13].

Definition 13 A **system specification** is a diagram $\delta : \mathbf{X} \rightarrow \mathbf{Obj}$ of object class specifications. \square

The category \mathbf{X} can be thought of as defining the ‘shape’ or ‘topology’ of the system.

Morphisms of system specifications should specify how systems are to be composed into larger systems; in particular, such a morphism should identify

¹In saying that $s : 1 \rightarrow S$ is a natural transformation we mean that $s_x \in Sx$ for each $x \in |\mathbf{X}|$, and for each arrow $f : x \rightarrow y$ in \mathbf{X} we have $Sf(s_x) = s_y$.

the shared components of systems which are to be composed. Suppose $\delta : \mathbf{X} \rightarrow \mathbf{Obj}$ and $\zeta : \mathbf{Y} \rightarrow \mathbf{Obj}$ are system specifications; a morphism from δ to ζ should interpret δ in ζ . That is, there should be a functor $F : \mathbf{X} \rightarrow \mathbf{Y}$ which interprets the topology of δ in that of ζ , and also a means of comparing the object class specifications $\delta(x)$ and $\zeta(F(x))$ for any object $x \in |\mathbf{X}|$. In particular, we want a means of encoding the methods and state attributes of $\zeta(F(x))$ as methods and state attributes of $\delta(x)$; this may be achieved by a natural transformation $\eta : F ; \zeta \rightarrow \delta$, as in the following picture.

$$\begin{array}{ccc}
 \mathbf{X} & \xrightarrow{F} & \mathbf{Y} \\
 \delta \downarrow & & \downarrow \zeta \\
 \mathbf{Obj} & & \mathbf{Obj} \\
 & \xleftarrow{\eta} & F ; \zeta \\
 \delta & &
 \end{array}$$

Definition 14 Given system specifications $\delta : \mathbf{X} \rightarrow \mathbf{Obj}$ and $\zeta : \mathbf{Y} \rightarrow \mathbf{Obj}$, a **morphism of system specifications** $\delta \rightarrow \zeta$ is a pair (F, η) where $F : \mathbf{X} \rightarrow \mathbf{Y}$ is a functor and $\eta : F ; \zeta \rightarrow \delta$ is a natural transformation. This gives a category \mathbf{Sys} of system specifications. \square

As we argued in the previous section, a system can be thought of as an object, by taking the limit of that diagram. In accordance with the slogan of categorical systems theory ‘Behaviour is given by limits’ (cf. Goguen [16]), this is made explicit in the following

Definition 15 The functor $\mathbf{BEH} : \mathbf{Sys} \rightarrow \mathbf{Obj}$ takes a system specification to its limit in \mathbf{Obj} , called its **behaviour**. \square

In fact, objects can be viewed as systems composed of only one object. This gives a functor $\mathbf{Obj} \rightarrow \mathbf{Sys}$, which is right adjoint to \mathbf{BEH} (Goguen [11]).

Just as objects were composed into complex objects by taking limits of diagrams, so too are systems composed into larger systems.

Definition 16 An **interconnection of system specifications** is a diagram $\delta : \mathbf{X} \rightarrow \mathbf{Sys}$. The **result** of an interconnection is its colimit in \mathbf{Sys} . \square

The following proposition states that the category \mathbf{Sys} is cocomplete; arbitrary diagrams of systems can be composed into larger systems by taking colimits. The proposition is a special case of a theorem of Goguen’s [11], which asserts the result for any complete category of objects.

Proposition 17 Every interconnection of system specifications has a result. \square

Moreover, the behaviour of the result of an interconnection $\delta : \mathbf{X} \rightarrow \mathbf{Obj}$ can be computed by taking the limit in \mathbf{Obj} of the limits of each system $\delta(x)$, that is, the limit of $\delta ; \mathbf{BEH} : \mathbf{X} \rightarrow \mathbf{Obj}$.

Another way of structuring objects into systems is to consider only topologies that have the form of a partial order. In this way, we consider only sub-object relationships between components of a system.

Definition 18 A **PO-system** is a functor $S : C^{\text{op}} \rightarrow \mathbf{Obj}$ where C is a partially ordered set (C, \leq) . \square

PO-systems are special kinds of system specifications, and morphisms of PO-systems are just system specification morphisms. The idea is that a PO-system is one that is constructed hierarchically from smaller systems by means of parallel connection with synchronisation, like the cell $(X||Y)||_Y(Y||Z)$ that we considered at the beginning of this paper.

Analogously to the above, we can define interconnections of PO-systems as diagrams, whose results are given by colimits, and analogously to Proposition 17 we have

Proposition 19 The category of PO-systems is cocomplete. \square

In other words, PO-systems may be constructed hierarchically.

Finally, we can begin to relate these results and the sheaf theoretic model of Monteiro and Pereira [28], and also the categorical systems slogan ‘Objects are Sheaves’ [16], by showing that any PO-system specification gives rise in a canonical way to a sheaf.

Definition 20 A **sheaf of objects** is a PO-system $S = P \xrightarrow{\varepsilon} D : C^{\text{op}} \rightarrow \mathbf{Obj}$ where C is a complete Heyting algebra and P and D are sheaves. A morphism of sheaves of objects is a system specification morphism whose functor part distributes through greatest lower bounds. \square

In order to construct a sheaf of objects from a PO-system, we first show that every partial order gives rise to a complete Heyting algebra.

Definition 21 Given a partially ordered set $C = (C, \leq)$, its **topology** is the complete Heyting algebra $\Omega C = (\Omega C, \sqsubseteq)$, where ΩC is the collection of downwards-closed subsets of C (i.e, subsets $X \subseteq C$ such that if $x \in X$ and $c \leq x$ then $c \in X$). Moreover, define the functor $H : C^{\text{op}} \rightarrow \Omega C^{\text{op}}$ by $Hc = \{x \in C \mid x \leq c\}$. \square

In fact, Ω is a functor from the category of partial orders to the category of complete Heyting algebras, and is left adjoint to the forgetful functor which views a complete Heyting algebra as a partial order, and H is the unit of this adjunction.

Now we construct sheaves of objects as follows.

Definition 22 Given a PO-system $S = P \xrightarrow{\varepsilon} D : C^{\text{op}} \rightarrow \mathbf{Obj}$ let $S^\Omega : \Omega C^{\text{op}} \rightarrow \mathbf{Obj}$ be the functor that takes a downwards-closed subset $X \subseteq C$ to the limit of the diagram $X^{\text{op}} \hookrightarrow C^{\text{op}} \xrightarrow{S} \mathbf{Obj}$. \square

Proposition 23 S^Ω is a sheaf of objects. \square

The proof of this proposition uses basic properties of limits; we omit the details. Because S^Ω is defined by limits, for any $c \in C$ there is a projection $\eta_c : S^\Omega(Hc) \rightarrow Sc$. In fact, we have a natural transformation $\eta : (H; S^\Omega) \rightarrow S$ and therefore a morphism of system specifications $(H, \eta) : S \rightarrow S^\Omega$. This morphism satisfies a special property which expresses the canonicity of the construction of S^Ω .

Theorem 24 Let T be a sheaf of objects. For every system specification morphism $(F, \zeta) : S \rightarrow T$ there is a unique morphism of sheaves $(G, \theta) : S^\Omega \rightarrow T$ such that $(H, \eta); (G, \theta) = (F, \zeta)$. \square

This theorem states that there is an adjunction between PO-systems and sheaves of objects, so that we can view S^Ω as the best possible way of extending S to a sheaf of objects.

5 Some Speculations

We have given a simple formalisation of object classes and systems of objects, and studied ways in which systems of objects may be interconnected. Much of the motivation for this account comes from Goguen and Diaconescu’s independent sum construction [18], which captures parallel composition with synchronisation. As in Monteiro and Pereira’s sheaf theoretic model of concurrency [28], we use limits in the category of monoids to obtain the commutativity properties which express truly concurrent parallel composition. Following Goguen [16] we model both behaviour and parallel composition by means of limit constructions, and we take a step towards the ‘objects are sheaves’ paradigm by giving an adjunction between PO-systems and sheaves. We expect that, by using a more general notion of sheaf as a functor on a category with a Grothendieck topology [24], we can obtain an adjunction between system specifications and sheaves of objects.

Our definition of object class specification is similar to abstract definitions of automata in categories [1, 2, 10, 12]. Since our definition was motivated by Goguen and Diaconescu’s independent sum construction in hidden sorted algebra, it would be interesting to examine the relationship between Nerode equivalence in automata theory and behavioural equivalence in hidden sorted algebra. Mazurkiewicz’s ‘process monoids’ [27] suggests that there is a relation between our work and Nerode equivalence.

An important aspect of object orientation that we have not covered is object creation and deletion. It might be possible to treat this in a simple way by looking at restrictions of systems to subcategories. Since a system is a functor $S : \mathbf{X} \rightarrow \mathbf{Obj}$, deleting an object would correspond to restricting S to a subcategory of \mathbf{X} ; similarly, object creation would correspond to extending S to a supercategory. Such an approach to dynamic systems is a very interesting area for future research.

Another very interesting and exciting area we intend to explore is the relationship between sheaf theoretic models of objects and the logic of toposes. Every topos has an ‘internal logic’ which is a kind of intuitionistic type theory [22, 24]. Since the category of sheaves on a particular category form a topos, it might be possible to use that logic to express and constructively prove useful properties of systems of objects. Moreover, the category of right actions of a monoid is a topos, so it would be interesting to examine its internal logic as a means of reasoning about models of specifications. Our use of monoids to capture the structure of programs is perhaps somewhat arbitrary: the topos of representations of a group would be appropriate to the free group structure of programs presented in [6].

References

- [1] Jiri Adámek. Observability and Nerode equivalence in concrete categories. In Ferenc Gécseg, editor, *Fundamentals of Computation Theory*. Springer-Verlag Lecture Notes in Computer Science 117, 1981.
- [2] Michael Arbib and Ernest Manes. Machines in a category: an expository introduction. *SIAM Review*, 16:163–192, 1974.
- [3] Michael Barr and Charles Wells. *Category Theory for Computing Science*. Prentice Hall, 1990.
- [4] Corina Cirstea. A distributed semantics for FOOPS. Technical Report PRG-TR-20-95, Programming Research Group, University of Oxford, 1995.
- [5] José Felix Costa, Amílcar Sernadas, and Cristina Sernadas. Object inheritance beyond subtyping. *Acta Informatica*, 5:5–26, 1994.
- [6] Edsger W. Dijkstra. The unification of three calculi. In Manfred Broy, editor, *Program Design Calculi*, pages 197–231. Springer Verlag, 1993.
- [7] Rakesh Dubey. On a general definition of safety and liveness. Master’s thesis, School of Electrical Engineering and Comp. Sci., Washington State Univ., 1991.
- [8] Hans-Dieter Ehrich, Joseph Goguen, and Amílcar Sernadas. A categorial theory of objects as observed processes. In J.W. de Bakker, Willem de Roever, and Gregorz Rozenberg, editors, *Foundations of Object Oriented Languages*. Springer-Verlag Lecture Notes in Computer Science 489, 1991.
- [9] Joseph Goguen. Mathematical representation of hierarchically organised systems. In E. O. Attinger, editor, *Global Systems Dynamics*, pages 111–129. S. Karger, 1970.
- [10] Joseph Goguen. Minimal realization of machines in closed categories. *Bulletin of the American Mathematical Society*, 78(5):777–783, 1972.
- [11] Joseph Goguen. Systems and minimal realization. In *Proceedings, 1971 IEEE Conf. on Decision and Control*, pages 42–46, 1972.
- [12] Joseph Goguen. Discrete-time machines in closed monoidal categories. *Journal of Computer and System Sciences*, 10:1–43, 1975.
- [13] Joseph Goguen. Objects. *International Journal of General Systems*, 1:237–243, 1975.
- [14] Joseph Goguen. A categorial manifesto. *Mathematical Structures in Computer Science*, 1(1):49–67, 1991.
- [15] Joseph Goguen. Types as theories. In George Michael Reed, Andrew William Roscoe, and Ralph F. Wachter, editors, *Topology and Category Theory in Computer Science*, pages 357–390. Oxford University Press, 1991.

- [16] Joseph Goguen. Sheaf semantics for concurrent interacting objects. *Mathematical Structures in Computer Science*, 11:159–191, 1992.
- [17] Joseph Goguen and Rod Burstall. Institutions: Abstract model theory for specification and programming. *Journal of the Association for Computing Machinery*, 39(1):95–146, 1992.
- [18] Joseph Goguen and Răzvan Diaconescu. Towards an algebraic semantics for the object paradigm. In Hartmut Ehrig and Fernando Orejas, editors, *Recent Trends in Data Type Specification*. Springer-Verlag Lecture Notes in Computer Science 785, 1994.
- [19] Joseph Goguen and Grant Malcolm. Proof of correctness of object representations. In A. W. Roscoe, editor, *A Classical Mind: essays dedicated to C.A.R. Hoare*, chapter 8, pages 119–142. Prentice-Hall International, 1994.
- [20] John Gray. Fragments of the history of sheaf theory. In M.P. Fourman, C.J. Mulvey, and D.S. Scott, editors, *Applications of Sheaves*. Springer-Verlag Lecture Notes in Mathematics 753, 1980.
- [21] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
- [22] Joachim Lambek and Philip J. Scott. *Introduction to Higher Order Categorical Logic*. Cambridge University Press, 1986. Cambridge Studies in Advanced Mathematics, Volume 7.
- [23] Saunders Mac Lane. *Categories for the Working Mathematician*, volume 5 of *Graduate Texts in Mathematics*. Springer Verlag, 1971.
- [24] Saunders Mac Lane and Ieke Moerdijk. *Sheaves in Geometry and Logic*. Springer-Verlag, 1992.
- [25] Johan Lilius. A sheaf semantics for Petri nets. Technical Report A23, Dept. of Computer Science, Helsinki University of Technology, 1993.
- [26] Grant Malcolm and Joseph Goguen. Proving correctness of refinement and implementation. Technical Monograph PRG-114, Programming Research Group, Oxford University, 1994.
- [27] Antoni Mazurkiewicz. Traces, histories, graphs: instances of a process monoid. In M.P. Chytil and V. Koubek, editors, *Mathematical Foundations of Computer Science*. Springer-Verlag Lecture Notes in Computer Science 176, 1984.
- [28] Luis Monteiro and Fernando Pereira. A sheaf-theoretic model of concurrency. In *Proc. Logic in Computer Science (LICS '86)*. IEEE Press, 1986.
- [29] Benjamin C. Pierce. *Basic Category Theory for Computer Science*. MIT Press, 1991.
- [30] B.R. Tennison. *Sheaf Theory*, volume 20 of *London Mathematical Society Lecture Notes*. Cambridge University Press, 1975.

- [31] David A. Wolfram and Joseph A. Goguen. A sheaf semantics for FOOPS expressions (extended abstract). In M. Tokoro, O. Nierstrasz, P. Wegner, and A. Yonezawa, editors, *Proceedings of the ECOOP'91 Workshop on Object-Based Concurrent Computing*, pages 81–98. Springer-Verlag Lecture Notes in Computer Science 612, 1992.