Sheaf Semantics for Concurrent Interacting Objects

Joseph A. Goguen*
Dept. of Computer Science & Engineering
University of California at San Diego

Abstract: This paper uses concepts from sheaf theory to explicate phenomena in concurrent systems, including object, inheritance, deadlock, and non-interference, as used in computer security. The approach is very general, and applies not only to concurrent object oriented systems, but also to systems of differential equations, electrical circuits, hardware description languges, and much more. Time can be discrete or continuous, linear or branching, and distribution is allowed over space as well as time. Concepts from category theory help to achieve this generality: objects are modeled by sheaves; inheritance by sheaf morphisms; systems by diagrams; and interconnections by diagrams of diagrams. In addition, behaviour is given by limit, and the result of interconnection by colimit. The approach is illustrated with many examples, including a semantics for a simple concurrent object-based programming language.

1 Introduction

Many popular formalisms for concurrent systems are *syntactic* (or "formal") in the sense that they represent systems by expressions, and then reason about systems by manipulating the corresponding expressions. For example, Milner's CCS [36], Hoare's CSP [30] and Bergstra's ACP [5] provide *process algebras*, which represent systems by expressions in which the primitives for process combination are implicitly defined by sets of equations; a quite different formal approach to concurrency is Girard's linear logic [12].

What we call *semantic*, or *model theoretic*, approaches, provide complete sets of possible behaviours for systems. Such approaches have received less attention than syntactic approaches, but are important as standards against which to test the soundness and completeness of syntactic approaches, and also for defining basic general concepts, such as deadlock and information flow. Moreover, they are closer to our physical intuition, can often describe examples in simple and natural ways, and integrate easily with such additional considerations as data structure, objects and constraints. Trace models, as used in CSP [30] and other process algebras, are a prototypical example. From this point of view, Petri nets [40], (labelled) transition systems [45], and synchronisation trees [36] can also be seen as syntactic.

Actually, things are not quite so simple, because the approaches that we have lumped together as "syntactic" really have varying degrees of semantics. For example, transition systems and synchronisation trees have been used as semantics for CCS and CSP; also, CSP has a "preferred" model, based on failures and refusals [30]. Petri nets have been used as models for linear logic (e.g., [35]), and set theoretic models have been given for Hewitt's actor approach [1]. Moreover, CCS expressions have been used as models for temporal logic. One person's syntax is another person's semantics.

*Thanks also to the Programming Research Group, Oxford University. The research reported in this paper has been supported in part by grants from the Science and Engineering Research Council, and the Fujitsu Corporation.

This paper proposes a new model theoretic approach to concurrency based on sheaves. Sheaf theory developed in mathematics for studying relationships between local and global phenomena, and has been applied in algebraic geometry, differential geometry, analysis, and even logic. It has also been given an abstract form using category theory [29, 28], which among other things provides some general results about limits that are used in this paper. From the point of view of concurrency theory, it seems suggestive to think of sheaves as a generalisation of trace models. Sheaves handle real time systems, and variation over space as well as over time, either discrete or continuous, in fact over any topological space, in a very natural way. The definition of sheaf involves a generalisation of the prefix closure condition for traces, and introduces an important new idea, called the *sheaf condition*.

Our main motivation for using sheaf theory in Computing Science is the desire to give semantics for "new generation" systems, such as object-based concurrent information systems, programming languages, and operating systems. This approach allows integrating concurrency with objects, data abstraction, information hiding, etc., and also helps to illuminate the notion of inheritance. Having such a semantic approach can be a significant help when designing a new language; for example, a sheaf theoretic approach helped with the design of FUNNEL, a hardware description language [42].

The approach is *declarative* and *constraint based*, and does not require distinguishing inputs and outputs. This makes it easy to treat applications such as constraint logic programming and electrical circuits that are essentially *relational*, in the sense that they involve finding solutions for a number of simultaneous relations (which are not just functions); in particular, there may be more than one solution, or no solution. It seems possible that functional input/output thinking has held back progress in this area, by making it more difficult to treat systems that involve the interdependent origination of their observed behaviour. Another goal has been to obtain a general model theoretic framework, within which general concepts such as deadlock and non-interference can be defined independent of formalism, and within which a wide variety of approaches can be compared. It may be worth emphasising that true concurrency can be modeled, and that we are not at all dependent upon interleaving. In particular, Lilius [34] has shown how to model Petri nets in our sheaf theoretic framework.

This paper builds on a much earlier paper [15] which used sheaf theory as part of a research programme on "Categorical General Systems Theory" [13, 14, 18]. My interest in this area was revived by the desire to give a semantics for FOOPS (a Functional Object Oriented Programming System) [22, 25] and for the Rewrite Rule Machine, a multi-grain hierarchical massively parallel graph rewriting machine (see [19] and [17]).

The main points made in this paper about the relationship between sheaves and objects are the following:

- objects give rise to sheaves;
- inheritance relations correspond to sheaf morphisms;
- systems are diagrams of sheaves; and
- the behaviour of a system is given by the limit of its diagram.

Here "diagram" and "limit" are to be understood in the sense of category theory. We also treat the interconnection of systems using colimits. Although this paper tries to give some philosophical motivation for these points, it does not attempt a serious philosophical treatment.

This paper has more definitions and examples than results. But the variety of examples may be surprising, and some of them illustrate a simple new approach to an important application area, the semantics of distributed concurrent (possibly object oriented) systems. Some of the definitions may also be surprising for their generality, including a notion of security that generalises the Goguen-Meseguer non-interference approach [20, 21]; indeed, this seems to be the most general such definition in the literature, and it applies, for example, to the security of real time distributed concurrent (possibly object oriented) databases. A definition of deadlock is also given, which again seems more general than anything in the literature.

Category theory has by now been used in many studies of concurrency; for example, see [45, 35, 11]. But as far as I know, only Monteiro and Pereira [37] have previously studied concurrency using sheaves; however, their approach does not seem to be closely related to the present paper.

Prerequisites and Notation

Basic category theory and some intuition for concurrency are needed to read this paper. The former can be found many places, including [32], which requires some mathematical sophistication, and [26], which may be especially recommended because it discusses sheaves, though in a different formulation from ours, and because it begins rather gently. An introduction to category theory for computing scientists is [3]. Some underlying intuitions for basic categorical concepts are given in [16], and an overview of the Computing Science categorical literature is given in [39]. A basic introduction to concurrency using CSP is [30].

We will use semicolon (";") to denote composition of functions, so that $(f;g)(x) = g(f(x))$; more generally, we let semicolon denote composition in any category, and we let 1_A denote the identity morphism at the object A. In set theory, we let ω denote the set of natural numbers, $\{0, 1, 2, ...\}$, and we let $\#S$ denote the cardinality of a set S.

Acknowledgements

Special thanks to Rod Burstall for "active listening" at the Dôme Café in Oxford while I tried to explain these ideas in preparation for a lecture on this subject at the UK-Japan Workshop on Concurrency, in September 1989. Thanks also to Hans-Dieter Ehrich, Amilcar Sernadas, José Fiadeiro, Felix Costa, Tom Maibaum and others in the ESPRIT sponsored ISCORE project (e.g., see [8, 9, 10]) for reawakening my interest in this area, and for their encouraging comments. Thanks to Susanna Schwab (neé Ginali) for a number of very useful suggestions, thanks to Jeremy Jacob and David Wolfram for noticing a number of bugs and infelicities in an earlier draft, and thanks to David Benson for showing me drafts of a paper in preparation, and the MSc thesis of Rakesh Dubey, both of which influenced the final draft of this paper. Thanks to Răzvan Diaconescu for several valuable suggestions, especially the proof of Theorem 29. Thanks also to Frances Page and Joan Arnold for help with preparation of the manuscript, and especially with the diagrams.

This paper is dedicated with affection to Professor Erwin Engeler, and was given as a lecture on the occasion of his sixtieth birthday, in March 1990.

2 Sheaves and Objects

Let us begin by asking how it is that we come to know about ordinary everyday objects. Consider a comfortable leather armchair C. Perhaps you see it briefly through a doorway, with only its left profile visible. Perhaps you later see it from the front. Maybe you eventually sit in it and notice its leathery smell, but perhaps you never see its back or bottom, and most likely you have no idea how it is stuffed. In fact, what you have is a collection of observations of certain attributes over certain regions of space-time; you never "have" the object as a whole. We can formalise such observations as functions $f\colon U \to A$ from some domain (of space-time) to some set A of attributes. If attributes $A_1, ..., A_n$ are observed, then we have $A = A_1 \times ... \times A_n$. Note that this is a *semantic* approach, based on direct observation of behaviour, rather than a *syntactic* approach, based on some sort of description or abstraction of behaviour.

The possible domains U are partially ordered by inclusion, and typically are closed under finite intersections and arbitrary unions, i.e., they form what is called a topological space. But for most of this paper, the following assumption about domains for observation is sufficient:

Definition 1: A **base** for observation is a family of sets partially ordered by inclusion. □

The following examples describe some typical bases for observations that are of interest in Computing Science.

Example 2: For discrete time systems, the base consisting of intervals of natural numbers starting from an initial time 0 is often appropriate. Intuitively, 0 might represent the time when the system was created, or first observed, and the various intervals starting from 0 might represent periods of continuous observation of the system. Thus,

$$\mathcal{I}_0(\omega) = \{\emptyset, \{0\}, \{0,1\}, \{0,1,2\}, ...\} \cup \{\omega\},$$

where the set ω of all natural numbers is the domain for observations over a complete (infinite) life cycle. The notation $[n] = \{0, ..., n-1\}$ for $n > 0$, and $[0] = \emptyset$, is often convenient when using this base. □

Example 3: Let $\mathcal{I}_0^f(\omega) = \mathcal{I}_0(\omega) - \{\omega\}$m i.e., the *finite* intervals starting from 0. □

Example 4: The base consisting of semi-open intervals of real numbers starting at time 0 is often appropriate for real time systems,

$$\mathcal{I}_0(\mathcal{R}^+) = \{[0,r) \mid r \geq 0\} \cup \{\mathcal{R}^+\},$$

where $\mathcal{R}^+$ denotes the non-negative real numbers. □

Example 5: A base consisting of certain unions of rectangles may be useful for describing the behaviour of certain systems distributed over one dimension in space and one dimension in time. In this example, $\mathcal{I}$ is the set of all subsets $U \subseteq \omega \times \omega$ that satisfy the following two conditions:

1. For $t \in \omega$, let $h(U,t) = \#\{h \mid (t,h) \in U\}$; then for each $t \in \omega$, we require that $h(U,t)$ is finite, and that $\{h \mid (t,h) \in U\} = [h(U,t)]$.

2. Given $t, t' \in \omega$ such that $|t - t'| = 1$, then $|h(U,t) - h(U,t')| \leq 1$.

These conditions say that each set in the base is a union of 1-by-n rectangles (for $n \geq 0$), such that the heights of two adjacent rectangles always differs by at most one; see Figure 1. We leave it to the reader to check that this base actually is a topological space. □

The bases of Examples 2 and 4 are topological spaces, while the base of Example 3 is not, because it is not closed under arbitrary unions.

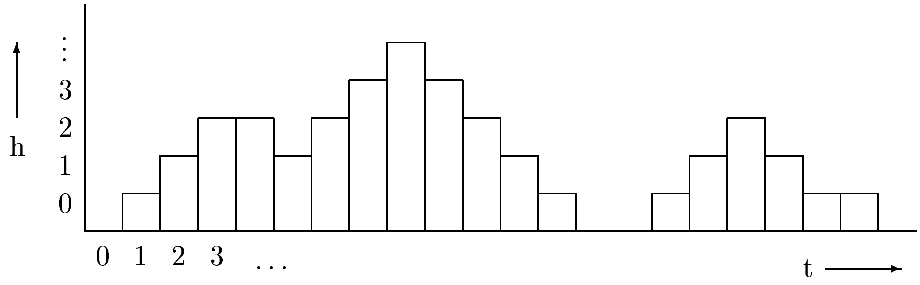


Figure 1: An Open Set

Example 6: If P is a preordered set (i.e., a set with a given relation $\leq$ that is reflexive and transitive), then the downclosed sets of P form a topological space, where a set $D \subseteq P$ is **downclosed** iff $d \in D$ and $d' \leq d$ imply $d' \in D$. This generalises Example 2 above. Let us denote this topology by $\mathcal{T}(P)$. An intuition that often works for this topology is that $d' \leq d$ means that "if you are now at d then you must have been at d'"; for example, this works when D is time. □

For example, this construction can be used to define bases that reflect branching (non-linear) time, as used in temporal logic.

Any base can be embedded in a unique least topological space, by closing under finite intersections and arbitrary unions. Hence, little is lost by assuming that the base is a topological space. Also, it will be important for future discussions to note that any base can be regarded as a subcategory of the category $\mathcal{S}et$ of sets, with the domains as its objects, and their inclusions as its morphisms.

Given any inclusion $U \subseteq V$ and any observation $f\colon V \to A$, we can form the *restriction* of f to U, denoted $f\lceil U$; this is a function $U \to A$ which has the same values as f, but is only defined on the points in U.

Given an object (in the intuitive sense) $\mathcal{O}$, we let $\mathcal{O}(U)$ denote its set of observations over U, and given an inclusion $i\colon U \hookrightarrow V$, we let $\mathcal{O}(i)\colon \mathcal{O}(V) \to \mathcal{O}(U)$ denote the function that maps each $f\colon V \to A$ to its restriction $f\lceil U$. Then $\mathcal{O}$ satisfies the following two equations:

$$\mathcal{O}(i;j) = \mathcal{O}(j);\mathcal{O}(i)$$

where $j\colon V \hookrightarrow W$ is another inclusion and where ";" indicates the composition of functions; and

$$\mathcal{O}(1_U) = 1_{\mathcal{O}(U)}$$

where 1_U denotes the identity function on U. These two equations say that $\mathcal{O}$ is a *contravariant functor* on the base $\mathcal{T}$ viewed as a category with inclusions as its morphisms. Since each $\mathcal{O}(U)$ is a set, we can view $\mathcal{O}$ as a set-valued functor $\mathcal{O}\colon \mathcal{T}^{op} \to \mathcal{S}et$, where $\mathcal{T}^{op}$ denotes the opposite category to $\mathcal{T}$. Although set-valued functors are often especially convenient (e.g., because one can apply the Yoneda lemma), the above formulation also suggests a natural generalisation, in which attributes might have other structure, such as that of a vector space, topological space, or Banach space. This is described in the following:

Definition 7: A **presheaf** is a functor $\mathcal{O}\colon \mathcal{T}^{op} \to \mathcal{C}$, where $\mathcal{T}$ is called its **base category** and $\mathcal{C}$ its **structure category**. If $i\colon U \hookrightarrow V$ is in $\mathcal{T}$, then $\mathcal{O}(i)\colon \mathcal{O}(V) \to \mathcal{O}(U)$ is called the

restriction morphism induced by i. Given a preobject $\mathcal{O}$, the notation $f\lceil U$ is usually clearer than $\mathcal{O}(i)(f)$, where $f \in \mathcal{O}(V)$ and $i\colon U \hookrightarrow V$, and this notation is much used in the following. □

This definition encapsulates the insight that observations are closed under restriction; this is a generalisation of the *prefix closure* condition of trace models. The fact that this condition can be formulated as functoriality allows the natural application of a number of basic results in category theory.

In all of our examples, the elements of each $\mathcal{O}(U)$ are functions, and in most of our examples, the structure category $\mathcal{C}$ is the category $\mathcal{S}et$ of sets. Section 2.2 below describes some modest (but somewhat more sophisticated) assumptions that allow us to handle other structures.

This approach can handle distribution over space, which is useful for studying multi-processor computer systems, and it can also take account of continuity, linearity, or other special structure that observations may have, by appropriate choice of $\mathcal{T}$ and $\mathcal{C}$. In fact, all of the presheaves in the examples of this paper arise in the following way:

Definition 8: A **preobject** $\mathcal{O}$ over a base $\mathcal{T}$ with **attribute object** $A = A_1 \times ... \times A_n$ is a presheaf of the form

$$\mathcal{O}(U) = \{h\colon U \to A_1 \times ... \times A_n \mid K(h)\},$$

where the morphisms $\mathcal{O}(i)$ are restriction maps, and the relation K expresses some property of functions, embodying the "laws" that $\mathcal{O}$ satisfies; the elements of A may be called **states, attributes** or **events**, depending on the context. □

For example, K might arise from the laws defining a logic gate, an electronic device, or the distribution of heat on a sphere.

When $\mathcal{T} = \mathcal{I}_0(\omega)$, the elements of $\mathcal{O}(\{0, ..., i\})$ can be thought of as *traces* of the behaviour of $\mathcal{O}$ over U. (The states in these traces may be sets of more elementary states, or may have some other complex structure.) Intuitively, if $U \subseteq V$ and $h \in \mathcal{O}(U)$ and $h' \in \mathcal{O}(V)$, then $h = h'\lceil U$ means that the states in h' can evolve from those in h.

Sheaf theory suggests that the following additional condition may be quite fundamental for some applications:

Definition 9: An **object** is a preobject $\mathcal{O}$ such that its base $\mathcal{T}$ is a topological space, and such that given $U_i \in \mathcal{T}$ and $f_i \in \mathcal{O}(U_i)$ for all $i \in I$ such that $U = \bigcup_{i \in I} U_i$ and such that $f_i\lceil(U_i \cap U_j) = f_j\lceil(U_i \cap U_j)$ for all $i, j \in I$, then there is a unique $f \in \mathcal{O}(U)$ such that $f\lceil U_i = f_i$ for all $i \in I$. This condition is called the **sheaf condition**. If the index set I is restricted to be finite, then the corresponding condition is called the **finite sheaf condition**. □

The sheaf condition says that any set of pairwise consistent local observations can be "fused" together into a unique observation over the union of their domains. A deterministic input/output system has a subset of attributes (its inputs and/or states) such that knowing their values at some point in time uniquely determines the values of the other attributes (the outputs). However, the sheaf condition does not exclude non-determinism, in either the weak sense that there is more than one $f' \in \mathcal{O}(V)$ such that $f'\lceil U = f$ for some given $f \in \mathcal{O}(U)$ with $U \subseteq V$, or in the stronger sense that there is no subset of the attributes which is sufficient to determine all of each $f\colon U \to A$ in $\mathcal{O}(U)$. Note that non-determinism is *not* being modeled by sets of values in this discussion, although that could certainly be done for some applications if desired. Also, notice that if $\mathcal{T}$ is a compact topological space, then the finite and infinite sheaf conditions are equivalent; however, compactness does not seem to be very common in the applications.

The sheaf condition appears to be satisfied by the behaviours of all naturally arising systems from Computing Science. This "Sheaf Hypothesis" is similar to the so-called Church (or Church-Turing) thesis, that all intuitively computable functions are computable in the precise sense of (say) Turing machines. Indeed, the claim of Scott and others that computable functions are continuous can be seen as a special case of our claim (see Section 3.3 for some related discussion). However, the sheaf condition is *not* satisfied by certain *properties* of systems, when they are expressed as presheaves. For example, we will see in Example 16 that the property of *fairness* gives a presheaf that is not a sheaf. However, the finite sheaf condition *is* satisfied by such examples, and seems to be satisfied by all naturally arising *properties* of systems in Computing Science. Such property presheaves also satisfy the so-called *separation condition*, which is the sheaf condition with "there exists a unique" replaced by just "there exists a".

Summing up the above discussion, our first main principle, which has much support from mathematical experience with many different kinds of geometrical object, as well as strong intuitions from concurrency theory in Computing Science, is that

$$\boxed{\text{OBJECTS GIVE RISE TO SHEAVES.}}$$

Sheaves can be thought of as a kind of "phase space" for objects. This sense of "object" might better be called an "object template" or "ideal object," because it describes all possible behaviours. When an object is actually part of a system, it may not exhibit all of its potential behaviours, because some of these may be inconsistent with constraints imposed by other objects. Also, objects in the present sense do not have unique identifiers; we will see later on that when part of a system, an object acquires a unique identifier (it will be the node that the object labels in the diagram of the system).

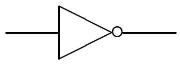
2.1 Some Examples

This subsection gives a number of examples illustrating the above definitions; some come from Computing Science, while others come from Electrical Engineering. Some more sophisticated examples are given in Section 4.

Example 10: (Invertor): For the discrete time case, $\mathcal{T} = \mathcal{I}_0(\omega)$, and when there is no delay, the behaviour of an invertor is described by the following sheaf,

$$\mathcal{O}(I) = \{f\colon I \to 2 \times 2 \mid i \in I \text{ and } f(i) = \langle t, t' \rangle \text{ imply } t' = \neg t\},$$

where $I \in \mathcal{T}$, $2 = \{0, 1\}$, and $\neg$ denotes negation (i.e., $\neg t = 1 - t$). The traditional picture is



If we impose a unit delay, we get instead the following:

$$\mathcal{O}(I) = \{f\colon I \to 2 \times 2 \mid i, i+1 \in I \text{ and } f(i) = \langle t, t' \rangle \text{ imply } f(i+1) = \langle t'', \neg t \rangle\}.$$

□

Example 11: (Gates): More generally, if $L(f_1, ..., f_n)$ is any Boolean-valued function of Boolean variables $f_1, \ldots, f_n$ then we can form a discrete time "L-gate" as follows, again for $I \in \mathcal{T} = \mathcal{I}_0(\omega)$:

$$\mathcal{O}(I) = \{f\colon I \to 2^{n+1} \mid i \in I \text{ implies } f_{n+1}(i) = L(f_1(i), ..., f_n(i))\},$$

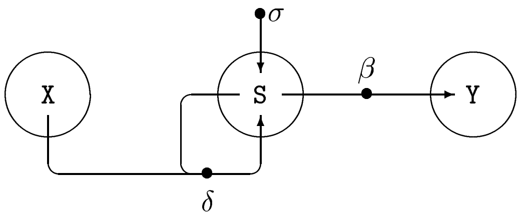
where $f(i) = \langle f_1(i), ..., f_{n+1}(i)\rangle$. For example, an invertor arises by taking $L(t) = \neg t$, an AND-gate arises from $L(t, t') = t \wedge t'$, and an OR-gate arises from $L(t, t') = t \vee t'$.

For the unit delay case, we instead define, for $I \in \mathcal{I}_0(\omega)$,

$$\mathcal{O}(I) = \{f\colon I \to 2^{n+1} \mid i, i+1 \in I \text{ implies } f_{n+1}(i+1) = L(f_1(i), ..., f_n(i))\}.$$

For example, a unit delayor arises by taking $L(t) = t$. □

Example 12: (Automata): More generally still, we can use automata to model dynamic digital circuits whose behaviour depends on state as well as on input. The "ADJ diagram[1]" (i.e., algebraic signature[2]) given below describes deterministic (not necessarily finite state) automata with initial state σ, transition function δ, and output function β. Such an automaton A consists of three sets, A_X, A_S, and A_Y whose elements are called *inputs*, *states* and *outputs* respectively, plus an element $A_\sigma \in A_S$ and functions $A_\beta\colon A_S \to A_Y$ and $A_\delta\colon A_X \times A_S \to A_S$, called the *output* and *state transition functions*, respectively. When just one automaton is under consideration and there is no danger of confusion, we will write X, S, Y for A_X, A_S, A_Y, and σ, β, δ for $A_\sigma, A_\beta, A_\delta$ respectively.



Here again time is discrete, so we let $\mathcal{T} = \mathcal{I}_0(\omega)$. The sheaf for this system is given, for $I \in \mathcal{T}$, by

$$\mathcal{O}(I) = \left\{ f\colon I \to X \times S \times Y \;\middle|\; \begin{array}{l} f_2(0) = \sigma \\ f_3(i) = \beta(f_2(i)) \text{ for all } i \in I \\ f_2(i+1) = \delta(f_1(i+1), f_2(i)) \text{ for all } i, i+1 \in I \end{array} \right\}$$

where f_1, f_2, f_3 respectively denote the X, S, Y components of f. Automata can be used, for example, to define digital devices that have internal states, such as flip-flops. □

Example 13: (Non-deterministic Automata): The only difference between deterministic and non-deterministic automata is that the transition function has the form $\delta\colon X \times S \to 2^S$ instead of $\delta\colon X \times S \to S$; that is, it returns a *set* of states rather than a single state. To get the sheaf for such a system, we replace the last line of the definition of $\mathcal{O}(I)$ in Example 12 above by

$$f_2(i+1) \in \delta(f_1(i+1), f_2(i)) \quad \text{for all} \quad i, i+1 \in I$$

Because $\delta(f_1(i+1), f_2(i))$ can be empty, it may happen that there is no $h' \in \mathcal{O}([i+1])$ such that $h'\lceil[i] = h$ for some given $h \in \mathcal{O}([i])$; even for deterministic automata, it can happen that there is more than one such h', but for non-deterministic automata, there can be multiple h' having the same input components. □

[1]This name was suggested by Cliff Jones for a kind of diagram introduced by Goguen, Thatcher, and Wagner [24] in their study of abstract data types. (The reason for the name "ADJ" is that the set {Goguen, Thatcher, Wagner, Wright} called itself ADJ at that time.)

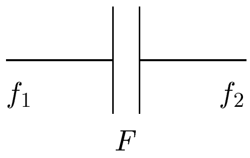
[2]A **many sorted signature** Σ consists of a set S of sorts and a family of sets $\Sigma_{w,s}$ of operation symbols, one for each $w \in S^*$ and $s \in S$; we say that $\sigma \in \Sigma_{w,s}$ has **arity** w and **value sort** s. An operation symbol $\sigma \in \Sigma_{w,s}$ is interpreted as an operation $A_\sigma\colon A_w \to A_s$, where $A_w = A_{s_1} \times ... \times A_{s_n}$ when $w = s_1...s_n$. See [24] for more details.

We can also model electrical components, using the real time base $\mathcal{I}(\mathcal{R}^+)$. For example:

Example 14: (Capacitor): Here, for $I \in \mathcal{I}(\mathcal{R}^+)$,

$$\mathcal{O}(I) = \left\{ f : I \to \mathcal{R}^3 \;\middle|\; \begin{array}{l} f \text{ is } C^\infty \text{ on } I, \text{ and} \\ f_3 = F \bullet \frac{d}{dt}(f_1 - f_2) \end{array} \right\},$$

where F is the capacitance, f_3 is the current, f_1, f_2 are voltages, and where a C^∞ function has continuous derivatives of every order on its domain. The following is the traditional picture:



Classical electrical engineering, which encompasses only linear devices, such as capacitors, inductors, and resistors, leads to systems of first order linear differential equations, of the general form

$$a_1 \frac{dx_1}{dt} + a_2 \frac{dx_2}{dt} + \cdots + a_n \frac{dx_n}{dt} + b_1 x_1 + b_2 x_2 + \cdots + b_n x_n = c\ .$$

In this setting, we could choose to exploit a more sophisticated structure category: because the differential equations involved are linear, the spaces of solutions are vector spaces; and because they also have a topological structure, we could let $\mathcal{C}$ be the category of topological vector spaces; an even better choice would be the category of Banach spaces (See [46] for information on functional analysis, including Banach spaces, and Section 2.2 below for an approach to handling such additional structure.) □

In much the same way, we could look at an object of solutions to some partial differential equation on a smooth manifold, for example, describing the flow of heat on the surface of a sphere. This would involve a 3-dimensional space[3], say embedded in 4-dimensional Euclidean space. See [15] for further examples along similar lines. We now return to discrete time.

Example 15: (Networks): Networks are usually built from sites and links[4]. Clearly the sites can be regarded as objects, and for many purposes it is also convenient to regard the links as objects. For example, it is not possible to send arbitrary signals over a real link, because it has a certain bandwidth, certain conventions about how signals are represented, certain physical properties, etc. Thus, we represent a network by a graph whose nodes include both the sites and the links of the network, and whose edges represent a connection from a node to a link. For example, consider a network with four sites, labelled $P1, P2, P3, P4$, and four links, labelled $F1, F2, F3, F4$, as in the figure of Example 40. We can regard such a graph as defining a *preorder*, with $n > n'$ iff there is a link from n to n'. For example, $P1 > F1$. The downclosed sets then give a topology, in which an open neighborhood N is a collection of sites and links such that if a site is in N, then all the links connected to it are also in N. The topologies of various SIMD and MIMD multi-processor architectures can be handled in this way.

[3]The third dimension is for time.

[4]The words "node" and "edge" would perhaps be clearer, but would conflict with the use of these words in connection with graphs later in this discussion.

Real networks are *dynamic* in the sense that some sites and/or links may sometimes be "down" and other times be "up." Some networks "dynamically reconfigure" themselves. It might seem that such networks cannot be modeled with the fixed topologies discussed above. However, they can be, by fixing in advance all potential communication links, and including "up/down" as part of the state of each node. For example, if the potential sites are $\{S_i \mid i \in \omega\}$, and if there is a potential link $L_{i,j}$ between each pair (i, j) of sites, then we let

$$N = \{S_i \mid i \in \omega\} \cup \{L_{i,j} \mid i, j \in \omega\}$$

where $L_{i,j} < S_i, S_j$ for all $i, j \in \omega$. Similarly, for a "star" topology, we would let

$$N = \{S_i \mid i \in \omega\} \cup \{\star\}$$

where $\star$ denotes the central node, with $\star < S_i$ for each $i \in \omega$. $\Box$

If P is a totally ordered set (i.e., for all $d, d' \in P$ either $d \leq d'$ or $d' \leq d$ (or both)), then the finite sheaf condition is automatically satisfied in the downclosed topology $\mathcal{T}(P)$. It follows from this that any presheaf over the base $\mathcal{I}_0^f(\omega)$ is a sheaf, because any non-trivial infinite union would have union ω, which has been excluded. However, not all presheaves over $\mathcal{I}_0(\omega)$ and $\mathcal{I}_0(R^+)$ are sheaves, because interesting phenomena can appear "at infinity" that do not appear in the finite approximations, as shown in the following:

Example 16: (Fair Scheduler): We use the base $\mathcal{I}_0(\omega)$, and consider a "fair scheduler" $\mathcal{F}$ for two events a, b. What we mean by fairness here is that if an a occurs, then eventually a b must occur, and if a b occurs, then eventually an a must occur. This means that

$$\mathcal{F}(\omega) = (a^+b^+ + b^+a^+)^\omega,$$

which is a concatenation of strings, the first consisting of some a's followed by some b's or else some b's followed by some a's, and so on forever. It is interesting now to notice that for each $n \in \omega$,

$$\mathcal{F}([n]) = \{a, b\}^n,$$

i.e., *any* behaviour is possible on each finite interval $\{0, ..., n-1\}$, including all a's, and all b's.
If we now suppose that $\mathcal{F}$ is a sheaf, then the sheaf condition implies that

$$\mathcal{F}(\omega) = \{a, b\}^\omega,$$

i.e., that *any* behaviour is also possible in the limit. But this contradicts the definition of $\mathcal{F}$; therefore $\mathcal{F}$ is *not* a sheaf. However, it does satisfies the finite sheaf and gluing conditions. $\Box$

Example 17: (Stacks): We can give an implementation for a stack that is distributed in space, as well as in time, using the base of Example 5, by defining

$$\mathcal{O}(U) = \{f : U \to \omega \mid h \in h(U, t) \cap h(U, t+1) \Rightarrow f(t, h) = f(t+1, h)\}.$$

(It might not appear so at first, but this example does satisfy the infinite sheaf condition.) $\Box$

2.2 Structure Categories

This subsection presents some assumptions on the structure category $\mathcal{C}$ that will make everything work just as well as in the set case. (This material can be skipped on a first reading, as it requires somewhat more sophisticated category theory.)

Because Definition 8 involves an attribute object $A = A_1 \times ... \times A_n$, we must assume that structure categories have finite products. In fact, we will see later that we need limits over all small (or at least finite) diagrams, i.e., we need *(finite) completeness*. All of the proposed structure categories mentioned so far in fact satisfy these assumptions, so they are not very restrictive in practice.

Definition 18: A **structure category** is a (finitely) complete category $\mathcal{C}$ together with a functor $\mathcal{U}\colon \mathcal{C} \to \mathcal{S}et$, called its **forgetful functor**, that has a left adjoint $F\colon \mathcal{S}et \to \mathcal{C}$, called its **free functor**. $\Box$

For example, if $\mathcal{C}$ is the category of vector spaces, then $\mathcal{U}$ gives the underlying set of a vector space, and F gives the free vector space using a given set as basis. In this connection, Definition 9 should be made more precise, by replacing "$u\colon U \to A_1 \times ... \times A_n$" by "$u\colon U \to \mathcal{U}(A_1 \times ... \times A_n)$".

Proposition 19: If $\mathcal{C}$ is a structure category, then its forgetful functor $\mathcal{U}$ preserves products, and more generally, all limits.

Proof: It is well known that right adjoints preserve limits; e.g., see [32] or [26]. $\Box$

This means that the underlying set of a product object in $\mathcal{C}$ can be taken to be the product of the underlying sets of the component objects; for example, we can get a product of two vector spaces by giving a vector space structure to the product of their underlying sets. In the following section, we will see that limits give the behaviour of systems; hence they are quite basic to our approach.

Here is how the definition of object looks in the present general setting:

Definition 20: A preobject $\mathcal{O}\colon \mathcal{T}^{op} \to \mathcal{C}$ with structure category $\mathcal{C}$ and with base $\mathcal{T}$ a topological space satisfies the **finite sheaf condition** iff the following is an equalizer diagram in $\mathcal{C}$,

$$\mathcal{O}(U \cup V) \longrightarrow \mathcal{O}(U) \times \mathcal{O}(V) \rightrightarrows \mathcal{O}(U \cap V)$$

for all $U, V \in \mathcal{T}$, where the first arrow is the tupling of the restrictions to U and V, and the next two are $\pi_1; \upharpoonright(U \cap V)$ and $\pi_2; \upharpoonright(U \cap V)$. $\mathcal{O}$ is a **sheaf** iff the following is an equalizer diagram in $\mathcal{C}$,

$$\mathcal{O}(U) \longrightarrow \Pi_j\ \mathcal{O}(U_j) \rightrightarrows \Pi_{j,j'}\ \mathcal{O}(U_j \cap U_{j'}).$$

whenever $U = \bigcup_j U_j$, where the first arrow is the tupling of the restrictions, and the next two are given as compositions of projections with restrictions. This is called the **sheaf condition**. $\Box$

It seems worth noting at this point that every presheaf is contained in a least sheaf, called its *enveloping sheaf*; more precisely, there is a left adjoint to the inclusion functor of sheaves into presheaves, called the *sheafification* functor; see [44] for some further details.

3 System, Behaviour and Interconnection

The previous section presented objects as coherent collections of possible observations. This section considers systems of such objects, and in particular, it considers how their joint behaviour is determined by the behaviours of their components. It also considers relationships of inheritance between objects, and how to interconnect systems. A very general compositionality result is proved for these notions. This approach seems especially convenient for systems that are highly concurrent, such as electrical circuits (whose component objects are capacitors, resistors, inductors, etc.) and digital circuits (whose components are flip flops, invertors, AND gates, etc.).

3.1 Systems

This subsection argues that systems can be modeled by diagrams of sheaves. In order to consider systems as diagrams, we need an appropriate notion of morphism between objects; these must be able to express the kinds of relationships between objects that arise when they are connected together to form systems; we argue that such relationships can be regarded as instances of *inheritance*. We use preobject morphisms to express relationships between component objects in a system, defined as follows:

Definition 21: Given preobjects $\mathcal{O}$ and $\mathcal{O}'$ over the same base $\mathcal{T}$, a **morphism** $\varphi\colon \mathcal{O} \to \mathcal{O}'$ is a family $\varphi_U\colon \mathcal{O}(U) \to \mathcal{O}'(U)$ of maps, one for each $U \in \mathcal{T}$, such that for each $i\colon U \to V$ in $\mathcal{T}$, the diagram

$$\begin{tikzcd}
\mathcal{O}(V) \arrow[r, "\varphi_V"] \arrow[d, "\mathcal{O}(i)"'] & \mathcal{O}'(V) \arrow[d, "\mathcal{O}'(i)"] \\
\mathcal{O}(U) \arrow[r, "\varphi_U"] & \mathcal{O}'(U)
\end{tikzcd}$$

commutes in $\mathcal{C}$. When $\mathcal{O}$ and $\mathcal{O}'$ are objects, we may also call φ an **object morphism** or a **sheaf morphism**. This gives rise to categories $\mathcal{P}re\mathcal{O}bj(\mathcal{T},\mathcal{C})$ and $\mathcal{O}bj(\mathcal{T},\mathcal{C})$ of preobjects and objects, respectively, over a given base $\mathcal{T}$ and structure category $\mathcal{C}$. $\square$

Actually, such a φ is just a natural transformation from the functor $\mathcal{O}$ to the functor $\mathcal{O}'$.

Lemma 22: Given preobjects $\mathcal{O}$ and $\mathcal{O}'$ with attribute objects A and A' respectively, and given $a\colon A \to A'$, then defining $a_U\colon [U \to A] \longrightarrow [U \to A']$ by $f \mapsto f;a$, i.e., by $a_U(f) = f;a$, gives a morphism iff $U \in \mathcal{T}$ and $f \in \mathcal{O}(U)$ imply $f;a \in \mathcal{O}'(U)$. $\square$

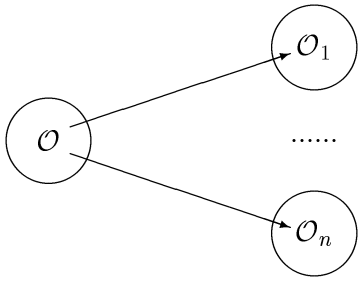
Definition 23: Given preobjects $\mathcal{O}$ and $\mathcal{O}'$ with attribute objects $A = \prod_{j\in J} A_j$ and $A' = \prod_{j\in J'} A_j$, respectively, with $J' \subseteq J$, if $a\colon A \to A'$ sending $\langle a_j \mid j \in J\rangle$ to $\langle a_j \mid j \in J'\rangle$ induces a morphism by the method of Lemma 22, then it is called a **projection morphism**. $\square$

For example, if $J = \{1, ..., n\}$ and $J' = \{1, ..., m\}$ with $m < n$, then we may think of the projection map $\mathcal{O} \to \mathcal{O}'$ as "forgetting" the attributes $A_{m+1}, ..., A_n$ of $\mathcal{O}$, or in a dual but perhaps more suggestive language, as expressing the *inheritance* by $\mathcal{O}$ of the attributes $A_1, ..., A_m$ from $\mathcal{O}'$, with $A_{m+1}, ..., A_n$ added as its own local attributes. All of the morphisms that occur in the examples of this paper are projections. Non-projection morphisms describe a kind of generalised inheritance that may involve combining attributes and changing representation. These considerations motivate our second main principle, that

$\boxed{\text{MORPHISMS REPRESENT INHERITANCE.}}$

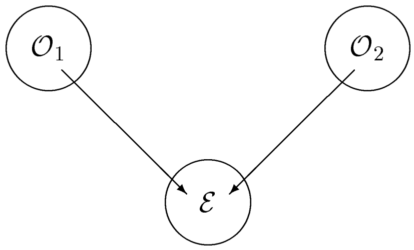
In particular, $\mathcal{O}$ inherits from $\mathcal{O}'$ iff $\varphi\colon\ \mathcal{O} \to \mathcal{O}'$ (the apparent reversal of direction here arises from the duality between "forgetting" and inheritance mentioned above).

In much the same way, *multiple inheritance* arises from multiple morphisms, as illustrated by the following diagram,



which we describe by saying that $\mathcal{O}$ inherits from $\mathcal{O}_1, ..., \mathcal{O}_n$.

Now let us return to the question of how objects form systems: in order for two objects to interact, they must each inherit something (some attributes and behaviour) from a third object; i.e., they must share a "common language" before they can speak to each other. This may be pictured as follows:

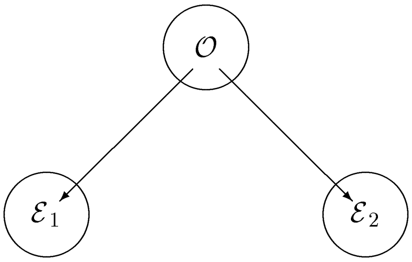


Let us call this a "valley" or "V" diagram. In many cases, the shared "language" $\mathcal{E}$ consists of *all possible* behaviours having states in some attribute object, that is, it has the form

$$\mathcal{E}(I) = \{f\colon\ I \to A \mid M(f)\}$$

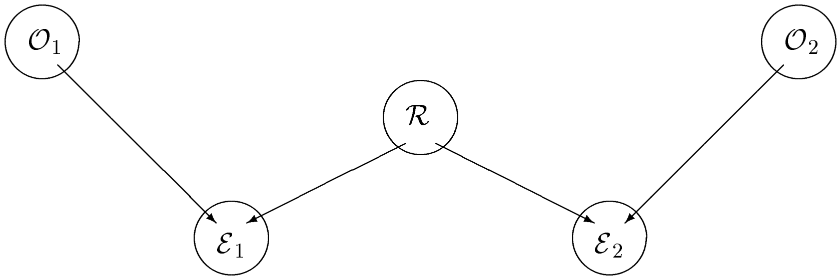
for each $I \in \mathcal{T}$, where the relation M may, for some application areas, express some minimal conditions (such as being linear, bounded, or continuously differentiable) and where A contains the states of some communication medium, such as the real numbers or an appropriate alphabet of events. It is common that $M(f)$ holds for all f, and in general we expect M to be such that $J \subseteq I$ and $M(f)$ imply $M(f{\restriction}J)$; the latter condition is sufficient for $\mathcal{E}$ to be a functor. It may help to think of $\mathcal{E}$ as the general object of all possible event streams or traces.

Another common case is that some relationship holds between two languages. This may be pictured as follows:



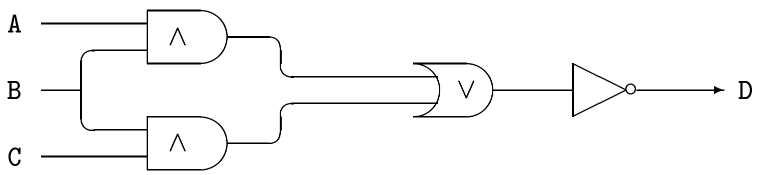
Let us call this a "peak" diagram. For example, an invertor is a relation between two objects of discrete time Boolean valued event streams. Similarly, AND an OR gates are relations between three such objects.

A more general way for two objects to communicate is for there to be a "boundary object" that can translate between their languages, or at least pass along some information, based on the existence of a relationship between the two languages. The system diagram for this is a "double valley" or "W" diagram, as follows:

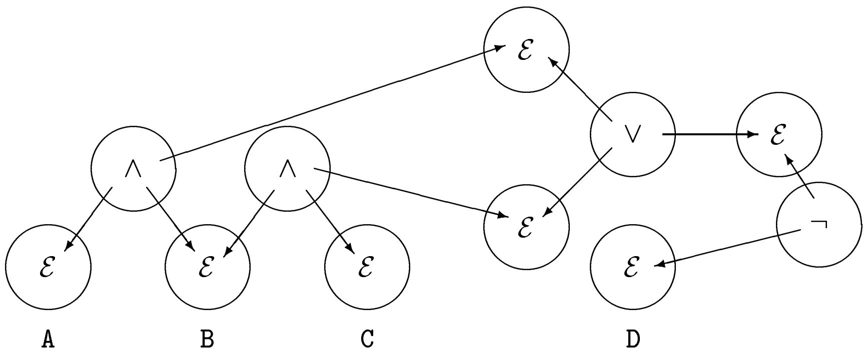


in which $\mathcal{O}_i$ has language $\mathcal{E}_i$ (for $i = 1,2$) and $\mathcal{R}$ is a relation between these languages. For example, $\mathcal{E}_1$ might be all streams over discrete time and $\mathcal{E}_2$ all streams over continuous time, with $\mathcal{R} \to \mathcal{E}_1$ and $\mathcal{R} \to \mathcal{E}_2$ giving the translation of a discrete time interval $\{0, 1, ..., n\}$ to a continuous time interval $[0, n+1)$. Similarly, $\mathcal{R}$ might translate between two different clocks, thus allowing for asynchronous communication. For another example, $\mathcal{R}$ might represent the summarisation of complex data in $\mathcal{O}_1$ to simple statistics in $\mathcal{O}_2$. $\mathcal{O}_1$ and $\mathcal{O}_2$ can be thought of as two different perspectives on a more complex object $\mathcal{R}$ or as providing a form of non-strict inheritance. What is called "overriding" in object oriented programming can probably be handled in this way. (These ideas were inspired by work of Leigh Star [41] on "boundary objects" in the sociology of science.)

Example 24: Now let us consider the system that in traditional notation would be described by the following diagram,



where $\wedge, \vee$ label AND and OR gates, respectively, corresponds to the following diagram in the category of objects over $\mathcal{I}_0(\omega)$:



□

These considerations motivate our third main principle, that

$$\boxed{\text{SYSTEMS ARE DIAGRAMS.}}$$

In line with our attempt to keep the category theoretic prerequisites of this paper to a minimum, we may explain the above with the following:

Definition 25: A **system** $\mathcal{S}$ consists of a graph with nodes $n \in N$ labelled by (pre)objects $\mathcal{S}_n$ and with edges $e\colon n \to n'$ labelled by morphisms $\varphi_e\colon \mathcal{S}_n \to \mathcal{S}_{n'}$. □

All of the (pre)objects and morphisms in $\mathcal{S}$ are assumed to have the same base and the same structure category. (We will later give a more sophisticated definition of system.)

3.2 Behaviour

This subsection argues that the behaviour of a system is given by the limit of its diagram. Let us begin by constructing an object that describes all the possible behaviours of a system. So let us assume a system $\mathcal{S}$ with objects $\mathcal{S}_n$ for $n \in N$, and let us choose some fixed domain $I \in \mathcal{T}$. Then a possible behaviour of the system over I is a choice of one behaviour for each object, say $f_n\colon I \to A_n$ in $\mathcal{S}_n(I)$, such that this family f_n is mutually consistent, in the sense that for each morphism $\varphi_e\colon \mathcal{S}_n \to \mathcal{S}_{n'}$ we have $\varphi_e(f_n) = f_{n'}$. Let us call such a family $\{f_n \mid n \in N\}$ a **consistent net of points** in $\mathcal{S}$.

Thus, the object of behaviours of the system has, for each $I \in \mathcal{T}$,

$$\mathcal{L}(I) = \left\{ \ \{f_n \mid n \in N\} \ \middle| \ \begin{array}{l} f_n \in \mathcal{S}_n(I) \text{ and } \varphi_e\colon \mathcal{S}_n \to \mathcal{S}_{n'} \\ \text{imply } \varphi_e(f_n) = f_{n'} \end{array} \right\},$$

that is, it contains all of the consistent nets of points over I.

When the structure category $\mathcal{C}$ is $\mathcal{S}et$, it is well known (e.g., [32, 26]) that $\mathcal{L}(I)$ is (a construction for) the *limit* of the diagram $\mathcal{S}(I)$ in which each node n is labelled by $\mathcal{S}_n(I)$ and each edge $e\colon n \to n'$ is labelled by $\varphi_e\colon \mathcal{S}_n(I) \to \mathcal{S}_{n'}(I)$. (One might say that the limit is "trying to make the diagram commute".)

We now draw upon another general result from category theory, showing that limits of diagrams of preobjects are computed "pointwise," i.e., we have the following, in which $\lim_n \mathcal{S}_n$ denotes the limit of a system $\mathcal{S}$ of objects $\mathcal{S}_n$, including their morphisms:

Proposition 26: Any diagram $\mathcal{S}$ of presheaves with values in a structure category $\mathcal{C}$ has a limit, and in fact, for each $U \in \mathcal{T}$,

$$\left(\lim_n \mathcal{S}_n\right)(U) = \lim_n(\mathcal{S}_n(U)) .$$

Proof: This follows directly from a well known result about limits in functor categories that is proved, for example, in [32] and [26], using the assumption that the structure category has limits. □

This motivates our fourth main principle, that

$$\boxed{\text{BEHAVIOUR IS LIMIT.}}$$

Proposition 26 and our assumption that $\mathcal{C}$ has limits imply that every system has a behaviour object $\mathcal{L}$. But this does not mean that every system actually exhibits behaviour over every $I \in \mathcal{T}$; for example (assuming $\mathcal{C} = \mathcal{S}et$), it is possible that $\mathcal{L}(I) = \emptyset$ for some, or even for all, $I \in \mathcal{T}$.

To illustrate this principle, the **parallel composition** of two objects, $\mathcal{O}_1 \parallel \mathcal{O}_2$, is the sheaf given by their product, $\mathcal{O}_1 \times \mathcal{O}_2$, and their **synchronised parallel connection** is given by the limit of a valley diagram, where synchronising events occur in the bottom object.

The relationship between global and local behaviour that arises between a system and its component objects should be distinguished from the global/local relationship stated in the sheaf condition of Definition 9. The first concerns the behaviour of multi-object systems, through diagrams and their limits, while the second concerns "glueing together" behaviour over domains.

3.3 Interconnection

The principles that objects are sheaves, systems are diagrams, and behaviour is limit are all taken from some earlier work in categorical General System Theory [13, 14, 18]. Another principle from this work is that interconnecting systems corresponds to taking colimits in the category of systems, where sharing is indicated by inclusion maps from shared parts into the systems that share them. The papers [13, 14, 18] develop some very general results in this setting, including the so-called Interconnection and Behaviour Theorems, which are given below[5]. We apply this material to show that the behaviour of a sheaf at a limit point is the limit of its behaviours at approximating points. (This is more technical than most of the rest of the paper, and some readers may wish to skip it on a first reading.)

In order to have a category of systems, we first need to define morphisms of systems. It is convenient to do this in the general setting of diagrams over an arbitrary category[6] $\mathcal{S}$, as follows:

Definition 27: A **diagram** in a category $\mathcal{S}$ is a functor $D\colon B \to \mathcal{S}$ from some **base** category B. Given diagrams $D_0\colon B_0 \to \mathcal{S}$ and $D_1\colon B_1 \to \mathcal{S}$, a **morphism** from D_0 to D_1 consists of a functor $F\colon B_0 \to B_1$ and a natural transformation $\eta\colon F;D_1 \Rightarrow D_0$.

Given three diagrams, $D_i\colon B_i \to \mathcal{S}$ for $i = 0,1,2$, the **composition** of the morphisms $\langle F_1, \eta_1\rangle\colon D_0 \to D_1$ and $\langle F_2, \eta_2\rangle\colon D_1 \to D_2$ is a morphism $\langle F_1;F_2, (F_1 \circ \eta_2);\eta_1\rangle\colon D_0 \to D_2$. □

[5]Actually, the results given here are somewhat more general than those in [13, 14, 18], because the restriction to so-called interconnection morphisms has been removed.

[6]In particular, the objects of $\mathcal{S}$ need not be sheaves or presheaves.

Here the operation ";" on natural transformations is their vertical composition, whereas "$\circ$" is their horizontal composition; e.g., see [32]. Although Definition 25 defined a system to be a kind of labelled graph, it is not difficult to see that any such labelled graph extends uniquely to a functor with source the category of paths in the given graph; and conversely, any functor can be considered a labelling of the underlying graph of its source category. So Definition 27 is consistent with Definition 25.

The direction of the arrows $\eta_n\colon D_1(F(n)) \to D_0(n)$ may seem counter-intuitive at first, but, for example, in the common case of an inclusion from a one node diagram into a system, it corresponds to need for translating the language of an object in the system into that of the object in the one node diagram.

It is not difficult to check the following:

Fact 28: Composition of diagram morphisms is associative and has identities, so that we have a **category of diagrams in** $\mathcal{S}$, denoted $Dgm(\mathcal{S})$. $\square$

The following basic result guarantees that in the setting of this paper, where the category $\mathcal{S}$ of systems in complete, we can always interconnect systems. (The elegant proof is due to Răzvan Diaconescu.)

Theorem 29: (Interconnection Theorem) If $\mathcal{S}$ is (finitely) complete, then $Dgm(\mathcal{S})$ is also (finitely) cocomplete, i.e., has all (finite) colimits.

Proof: First, we define a functor $P\colon Cat^{op} \to Cat$ by

$$P(B) = [B \to \mathcal{S}]^{op} \text{ for any category } B, \text{ and}$$
$$P(F) = [F \to \mathcal{S}]^{op}\colon [B_1 \to \mathcal{S}]^{op} \to [B_0 \to \mathcal{S}]^{op} \text{ for any functor } F\colon B_0 \to B_1.$$

Noting that P is a (strict) indexed category (in the sense of [43]), and that $Dgm(\mathcal{S}) = \mathrm{Flat}(P)$ (again, see [43] for this notation), we can use Theorem 2 of [43] to show the (finite) cocompleteness of $\mathrm{Flat}(P)$ by checking the hypotheses of that theorem:

1. Cat is cocomplete.
2. $[B \to \mathcal{S}]^{op}$ is (finitely) cocomplete for any category B because $\mathcal{S}$ is assumed (finitely) complete.
3. P is locally reversible because $(\mathcal{S}^F)^{op}\colon [B_1 \to \mathcal{S}]^{op} \to [B_0 \to \mathcal{S}]^{op}$ has a left adjoint, because $\mathcal{S}^F\colon [B_0 \to \mathcal{S}] \to [B_1 \to \mathcal{S}]$ has a right adjoint, because any functor $B_0 \to \mathcal{S}$ has a right Kan extension along any functor $R\colon B_0 \to B_1$ by the Kan Extension Theorem (see Theorem 1 on page 233 of [32]) because $\mathcal{S}$ is assumed complete.

$\square$

Taking colimits in the category $Dgm(\mathcal{S})$ corresponds to interconnecting systems. For example, interconnecting two relations (represented by "peak" diagrams) over a common object gives a "double peak" diagram, whose behaviour (i.e., limit) gives the relation which is the composition of the two given relations, as illustrated in Figure 2.

We may summarise the above discussion in the following principle, from [13, 14, 18] (see also [16]):

$$\boxed{\text{INTERCONNECTION IS COLIMIT.}}$$

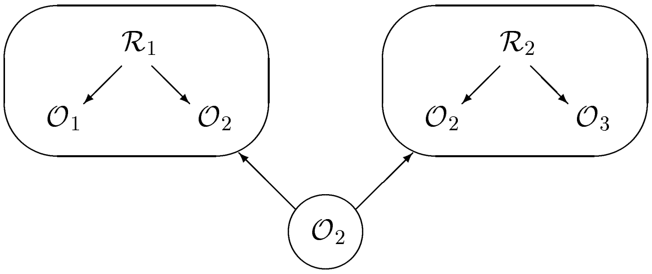


Figure 2: The Composition of Two Relations

We will soon need the following result about limits:

Proposition 30: Given a complete category $\mathcal{S}$, then "limit of" is a functor

$$\mathrm{Lim}\colon Dgm(\mathcal{S})^{op} \longrightarrow \mathcal{S}$$

which is right adjoint to the functor

$$\lceil \_ \rceil : \mathcal{S} \longrightarrow Dgm(\mathcal{S})^{op}$$

which sends each object C to the diagram consisting of just one node labelled C.

Proof: This is just a reformulation of the universal property of limits. In particular, note that the functor category $Dgm(\mathcal{S})[D, \lceil C \rceil]$ is the category of all cones $\eta\colon \lceil C \rceil \Rightarrow D$ over D. □

Intuition about systems suggests that one can calculate the behaviour of a system from the behaviours of its components. This intuition is actually a precise theorem in the present formal setting, stated as follows:

Theorem 31: (Behaviour Theorem) Let $D\colon I \to Dgm(\mathcal{S})^{op}$ be a (small) diagram of diagrams (i.e., systems) over a complete category $\mathcal{S}$. Then (in a hopefully suggestive notation)

$$\mathrm{Lim}(\mathrm{colim}\ D) = \lim(D; \mathrm{Lim}).$$

Proof: As noted in the proof of Proposition 19, right adjoints preserve limits; also, limits in $Dgm(\mathcal{S})^{op}$ are colimits in $Dgm(\mathcal{S})$. □

This result is a very general "*Compositionality Theorem,*" in the sense of giving very general conditions under which behaviours of parts can be composed to give the behaviour of the whole.

Let us consider a special case, where I is the discrete base with just two nodes, say 0,1. Then a functor $D\colon I \to Dgm(\mathcal{S})^{op}$ consists of just two diagrams, D_0 and D_1. If we denote their colimit in $Dgm(\mathcal{S})$ by $D_0 + D_1$, then we have the following formula:

$$\lim(D_0 + D_1) = \lim(D_0) \times \lim(D_1).$$

It may be amusing, and perhaps surprising, that Theorem 31 can be used to prove that the behaviour of a sheaf at a "limit point" is the limit of the behaviours leading up to it. Given a presheaf $\mathcal{O}\colon \mathcal{T}^{op} \to \mathcal{C}$ with $\mathcal{C}$ complete, let $U = \bigcup_{j\in J} U_j$ in $\mathcal{T}$, for some index set J. Let I have the shape

$$\{0\} \leftleftarrows \{1\}$$

We will now define a diagram D in $Dgm(\mathcal{O})$ with base I. Let D_0 be the diagram with base the set J and with $j \in J$ labelled by $\mathcal{O}(U_j)$; let D_1 be the diagram with base $J \times J$ and with $\langle j, j'\rangle$ labelled by $\mathcal{O}(U_j \cap U_{j'})$. Next, define $d\colon D_1 \to D_0$ as follows: on the bases, d_F is the projection $\langle j, j'\rangle \mapsto j$; and on the objects, $(d_\eta)_{\langle j,j'\rangle}$ is the restriction morphism $\mathcal{O}(U_j) \to \mathcal{O}(U_j \cap U_{j'})$. Define $d'\colon D_1 \to D_0$ similarly: on the bases, d'_F is the projection $\langle j, j'\rangle \mapsto j'$; and on the objects, $(d'_\eta)_{\langle j,j'\rangle}$ is the restriction morphism $\mathcal{O}(U_{j'}) \to \mathcal{O}(U_j \cap U_{j'})$.

If we apply the limit functor Lim to this diagram D (of diagrams) with base I, and then take its limit, we get the following diagram in $\mathcal{C}$,

$$L \longrightarrow \Pi_j\ \mathcal{O}(U_j) \rightrightarrows \Pi_{j,j'}\ \mathcal{O}(U_j \cap U_{j'}).$$

The equaliser formulation of the sheaf condition (Definition 20) says that $\mathcal{O}$ is a sheaf iff $L \cong \mathcal{O}(U)$ whenever $U = \bigcup_j U_j$, and the Behaviour Theorem says that L is the limit of the following diagram

$$\begin{tikzcd}[column sep=small]
\mathcal{O}(U_1) \arrow[dr] \arrow[ddrr] & & \mathcal{O}(U_2) \arrow[dl] \arrow[dr] & & \mathcal{O}(U_3) \arrow[dl] \arrow[ddll] & & \ldots \\
& \mathcal{O}(U_1 \cap U_2) & & \mathcal{O}(U_2 \cap U_3) & & & \ldots \\
& & \mathcal{O}(U_1 \cap U_3) & & & & \ldots
\end{tikzcd}$$

where all arrows are restriction morphisms. From this, we conclude that $\mathcal{O}$ is a sheaf iff its behaviour $\mathcal{O}(U)$ at U is the limit of its behaviours $\mathcal{O}(U_j)$ at U_j, whenever $U = \bigcup_j U_j$. For example, if $\mathcal{T} = \mathcal{I}_0(\omega)$ and $U_j = [n]$, then we have $\omega = \bigcup_n [n]$, so that $\mathcal{O}(\omega)$ is the limit of the $\mathcal{O}([n])$.

3.4 Discussion

We have considered systems at three different levels: (1) *objects*, as collections of possible observations; (2) *systems*, as collections of interacting objects; and (3) *interconnections*, as systems of interacting systems. A wide variety of systems can be treated in this way, including digital hardware, electrical circuits, and (as shown in the next section) concurrent programming languages. Concepts from category theory have helped achieve this generality: we model objects as sheaves; systems as diagrams; and interconnections as diagrams of diagrams. In addition, behaviour is given by limit, and the result of interconnection by colimit. Although we have not done so here, it is possible to iterate these constructions to obtain hierarchical systems of arbitrary depth; see [13, 14, 18].

Our approach to systems is *declarative* or *constraint based* in the sense that behaviour arises through "mutual effects" or "interdependent origination" rather than through the propagation of causes and effects; in particular, we do not assume that all devices have inputs and outputs, and hence we are not limited to simple functional devices[7]. In this setting, what it means for a system to *satisfy* a specification (i.e., a presheaf representing some property, such as fairness) is that if we interconnect that property with the system, then the resulting behaviour is the same as (i.e., is isomorphic to) the behaviour of the original system.

[7] Of course, ours is not the only approach with this property. For example, Gordon [27] has used higher order relations to study digital circuits. But the point does seem worth a bit of emphasis.

It is interesting to look at so-called "internal choice" and "external choice" in the context of limits of diagram of sheaves. We have already noted that non-determinism simply corresponds to there being more than one function to choose from in some set $\mathcal{O}_n(U)$. When $\mathcal{O}_n$ participates in some larger system, some of these elements may no longer be consistent with overall system behaviour – i.e., they may not be in the n component of any consistent net of points for the system. More interestingly, the n component may be completely determined by the behaviour of the rest of the system. In this case, we may say that "external choice" is being exerised. But if several values for the nth component remain, then we may say that $\mathcal{O}_n$ has "internal choice". Under a constraint oriented view of systems, the distinction between internal and external choice appears somewhat artificial, and may depend on the point of view taken. (Formulating the above discussion for an arbitrary structure category requires some additional concepts that are discussed in Section 4.2.)

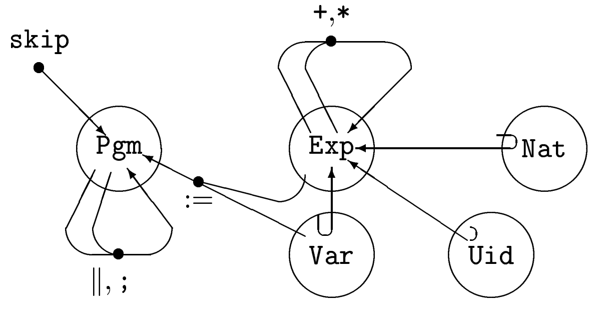
It is worth noting that the sheaves that arise from some particular model of concurrency, such as some kind of transition system, typically form a complete subcategory of the category of all sheaves over the appropriate base; in general, there will be some sheaves that do not correspond to any system of the given kind.

4 Semantics and Properties of Systems

Now that we have stated, motivated and illustrated all four of the main principles of this paper, let us apply them to some examples of more immediate interest to Computing Science. This section shows how to give semantics for a simple concurrent language executing in a distributed environment.

The objects considered in the previous sections were *closed*, in the sense that their attributes can be divided into inputs and outputs such that the values on the inputs uniquely determine those on the outputs. But modern programming requires *open systems*, the objects of which only constrain what happens under certain conditions, and leave the rest unconstrained. (Of course, non-determinism, where outputs are only partially constrained, is also possible.) Openness in this sense is similar to what happens when a number of objects are connected to an Ethernet, and each responds only to those messages addressed to it.

The syntax of our simple concurrent programming language is given by the (order sorted) signature[8] Σ of the following ADJ diagram:



Here `Uid` is the sort for object names, `Var` for variables, `Nat` for natural numbers, `Exp` for expressions, and `Pgm` for programs. Also, `+` and `*` are operations on expressions, while `;` and $\|$

[8] An order sorted signature has, in addition to the data of a many sorted signature, a partial ordering on the set S of sorts, and some assumptions about the consistency of overloaded operation symbols. This ADJ diagram is therefore augmented to indicate subsort relationships with "hooked" arrows. (See [23] for details of order sorted algebra.)

are operations on programs, $:=$ is assignment, `skip` is the instruction that does nothing, and $\hookrightarrow$ indicates a *subsort* relation (based on order-sorted algebra [23]). We also assume that the sorts `Uid`, `Var` and `Nat` are populated by a countably infinite number of mutually disjoint constants. Typical elements of sort `Uid` are `Tom`, `Dick` and `Harry`, and of `Var` are X, Y, Z, while the elements of sort `Nat` are of course 0,1,2,... . We let $|\Sigma|$ denote the set of all symbols in Σ; hence $|\Sigma|^*$ denotes the set of all finite strings of elements from Σ. Unless otherwise indicated, variables w, w', w_1, etc. range over $|\Sigma|^*$.

Our language is the algebra T_Σ of all Σ-terms, constructed just like the order sorted term algebra in [23], except that in order to simplify the transition rules to be given below, we will use reverse Polish (i.e., Polish postfix, or Łukasiewicz) notation for the terms of sort `Pgm`, which will represent programs. For example, the program

$$X := Y + 2 \parallel Y := X + 2$$

appears in $T_{\Sigma,\mathtt{Pgm}}$ as the string

$$\parallel := X \mathbin{+} Y\ 2 := Y \mathbin{+} X\ 2$$

To simplify the notation, we will sometimes write $u \in \mathtt{Uid}$ instead of $u \in T_{\Sigma,\mathtt{Uid}}$ and similarly for $X \in \mathtt{Var}$ and the other sorts. So $n \in \mathtt{Nat}$ iff $n \in \omega$. Also, we abbreviate $T_{\Sigma,\mathtt{Pgm}}$ by P_Σ.

For simplicity, we will use discrete time in this example, so that $\mathcal{T} = \mathcal{I}_0(\omega)$. Then the appropriate event stream object $\mathcal{E}$ is given, for $I \in \mathcal{T}$, by

$$\mathcal{E}(I) = \{f \mid f \colon I \to P_\Sigma\}.$$

Next, we give the objects that define the semantics of the various features of the language. For this purpose, we will use the following "transition notation" in defining objects $\mathcal{O}$ over the base $\mathcal{I}_0(\omega)$: if A is the attribute object of $\mathcal{O}$, then $a \mapsto a'$ (for $a, a' \in A$) means that if $f(i) = a$ for some $f \in \mathcal{O}(I)$, then there are some $I' \supseteq I$ and $f' \in \mathcal{O}(I')$ such that $i + 1 \in I'$ and $f' \restriction I = f$ and $f'(i+1) = a'$. (This implies that each basic operation takes unit time; but it is easy to redefine $\mapsto$ so that different operations take different times, or so that operations may take as long as they like.) In general, a and a' in the notation $a \mapsto a'$ are not elements of A, but rather may be *patterns* that define a set of transitions. For example,

$$w{+}nmw' \mapsto wkw' \quad \textbf{if} \quad n, m \in \omega \quad \textbf{and} \quad k = n + m$$

means that if there exist strings $w, w' \in |\Sigma|^*$ and some numbers n, m such that $f(i) = w{+}nmw'$, then there is some f' such that $f'(i) = w{+}nmw'$ and $f'(i+1) = wkw'$ where $k = n + m$. Also, we will let

$$\mapsto \ a$$

indicate that the initial state is a, i.e., that if $f \in \mathcal{O}(I)$ is defined at 0, then $f(0)$ matches the pattern a; usually a is just a single attribute in this notation.

The object $\mathcal{O}$ defined by a set of transition relations is then the least family[9] $\mathcal{O}(U) \subseteq \mathcal{E}(U)$ of sets of term-valued functions, for each $U \in \mathcal{T}$, satisfying the transition relations and closed under restriction, i.e., such that if $U \subseteq V$ and $f \in \mathcal{O}(V)$ then $f \restriction U \in \mathcal{O}(U)$.

We now define a series of objects, one for each feature of the language, which when put together give a system whose semantics is that of the language.

[9] This exists because the conditions defining it are all positive, i.e., they are Horn clauses.

Example 32: (Parallel Composition) This feature is defined by an object denoted $\mathcal{P}$ having attribute object P_Σ and satisfying

$$\begin{array}{lcl} w_1 \parallel \texttt{skip}\ w\ w_2 & \mapsto & w_1'\ w\ w_2' \\ w_1 \parallel w\ \texttt{skip}\ w_2 & \mapsto & w_1'\ w\ w_2' \end{array}$$

for $w_1, w_1', w, w', w_2, w_2' \in P_\Sigma$. This just says that when either of the two parallel programs is completed, the parallel constructor can be eliminated; computation by the other program can proceed in parallel with the elimination. Note that this object is a subobject of $\mathcal{E}$; we will use the resulting inclusion morphism to relate it to other objects. □

Example 33: (Sequential Composition) This object is denoted $\mathcal{Q}$, has attribute object P_Σ again, and is defined by

$$\begin{array}{lcl} w_1\ ;w_2\ w_3\ w_4 & \mapsto & w_1'\ ;w_2'\ w_3\ w_4' \ \ \textbf{if}\ \ w_1 \neq \texttt{skip} \\ w_1\ ;\texttt{skip}\ w_3\ w_4 & \mapsto & w_1'\ w_3\ w_4' \end{array}$$

This says that the first program must be completed before the execution of second can begin. It does not constrain what can be done by the first program. Again, its projection is the inclusion. □

Example 34: (Assign) Assignment is defined by a family of objects $\mathcal{G}^X$, one for each $X \in \texttt{Var}$. $\mathcal{G}^X$ has attribute object $\omega \times P_\Sigma$ and is defined by the following:

$$\begin{array}{lcl} & \mapsto & \langle 0, w\rangle \\ \langle m, w := Xnw'\rangle & \mapsto & \langle n, w''\texttt{skip}\,w'''\rangle \\ \langle n, wpXw'\rangle & \mapsto & \langle n, w''pnw'''\rangle \ \ \textbf{if}\ \ p \neq := \end{array}$$

where $n \in \omega$ and $p \in \Sigma$. This says the each variable initially has value 0, that an assignment of n to X causes the object $\mathcal{G}^X$ to remember n, and that the variable X can be replaced by its value, unless it occurs just after :=. The projection is from the second component of the pair in the state. □

Example 35: (Adder) Let $\omega_\perp = \omega \cup \{\perp\}$ and let $\omega_\perp \times P_\Sigma$ be the attribute object. Then a typical adder object, $\mathcal{A}^{\texttt{Fred}}$ for $\texttt{Fred} \in \texttt{Uid}$, may be defined as follows:

$$\begin{array}{lcl} \langle s, w_1\texttt{Fred}\,w_2\rangle & \mapsto & \langle \perp, w_1' s w_2'\rangle \ \ \textbf{if}\ \ s \neq \perp \\ \langle \perp, w_1\texttt{+}nmw_2\rangle & \mapsto & \langle s, w_1'\texttt{Fred}\,w_2'\rangle \ \ \textbf{where}\ \ s \text{ is the number } n{+}m \\ \langle s, w\rangle & \mapsto & \langle s, w'\rangle \ \ \textbf{if}\ \ w, w' \in P_{\Sigma - \texttt{Fred}} \end{array}$$

where $n, m \in \omega$. This says that Fred performs just one addition at a time, and allows other operations to occur concurrently. If additions are ready to be performed, then he must perform one; he replaces the chosen expression +nm by his name and memorises the sum. When he sees his name, he replaces it by the number he has memorised, forgets that number, and becomes ready to do another sum. Notice that Fred also prohibits anyone else (to whom he communicates) from using his name. Again, projection is from the second component of the pair in the state. □

Example 36: (Multiplier) A typical multiplier, say $\mathcal{M}^{\texttt{Tom}}$ for $\texttt{Tom} \in \texttt{Uid}$, is defined just like the adder `Fred`, except that `+` is replaced by `*` and `Fred` is replaced by `Tom`. □

We have now defined all the features of the language. Unfortunately, what we have still allows unprogrammed transitions to occur. However, we can introduce a "frame object" to keep things the same unless they are deliberately changed:

Example 37: (Frame) This object is denoted $\mathcal{F}$, has attribute object P_Σ, and is defined by

$$\begin{aligned} & w_1 ... w_n \mapsto w'_1 ... w'_n \;\; \textbf{where} \\ & \quad w_i = w'_i \text{ for } i = 1, ..., n \;\; \textbf{unless} \\ & \quad\quad w_i \in \texttt{Uid} \;\; \textbf{or} \;\; w'_i \in \texttt{Uid} \;\; \textbf{or} \;\; w_i \in \{:=, \|, \texttt{;}, \texttt{skip}\}, \end{aligned}$$

where w_i, w'_i are here in Σ^* rather than in P_Σ. □

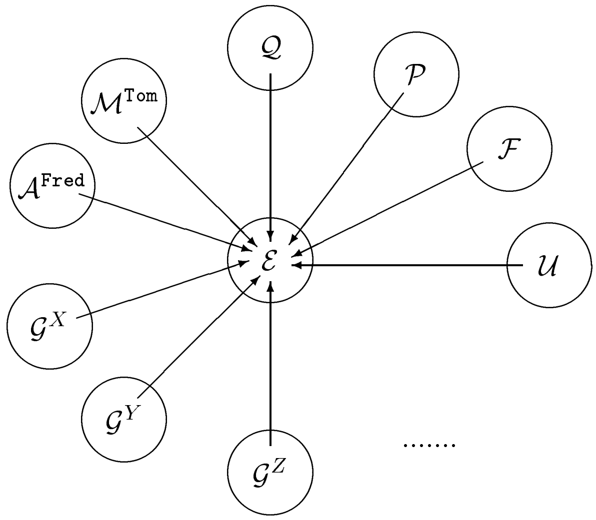
Finally, in order to "close" the system, we can add a "shop steward" who disallows unregistered workers:

Example 38: (Union) This object denoted $\mathcal{U}$ is defined, for $I \in \mathcal{T}$, by

$$\mathcal{U}(I) = \{f \mid f\colon I \to P_{\Sigma-(\texttt{Uid}-W)}\},$$

where W is the set of union members, which might for example be $\{\texttt{Fred}, \;\texttt{Tom}\}$. □

So our system looks as follows,



and the limit of this diagram evaluates programs in our little language. In the limit object, at each instant of time, each object has a copy of the same program, and possibly some internal state, such as the value of a variable. As time progresses, the program is simplified as worker objects process its parts. This processing is concurrent and distributed. Moreover, the objects for variables are true objects in the sense of object oriented programming, although very simple ones. The consistent nets of points in the limit object can be seen as the *run time states* of the computation. If we add more workers, then programs can be executed more quickly. Note that these objects can be seen as (infinite state) automata.

Of course, this is a simple example. But seems clear that the same techniques will extend to much more complex languages. For example, it is easy to add more language constructs, such as loops. Another interesting feature to add would be `abort`. Also, it seems that the semantics of the functional and object oriented language FOOPS [22, 25] and of the Rewrite Rule Machine (see [19] and [17]) can be developed in a similar way, and I hope these will be discussed in future papers. See [7] for some other applications of sheaf theory to concurrent systems.

4.1 Deadlock

It is important that many real systems should be *deadlock free*, in the sense that they do not get into dead states. Definitions of deadlock in the literature tend to be syntactic, and as far as I am aware, there is no definition that is sufficiently general to encompass all of the kinds of system considered in this paper. The following proposes one:

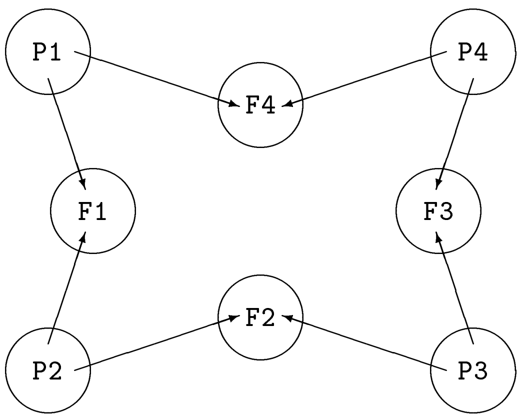
Definition 39: If $\mathcal{L}$ is the limit object of a system, then the system has **deadlock** at $h \in \mathcal{L}(U)$ iff for all $U' \supseteq U$ and all $h' \in \mathcal{L}(U')$ such that $h' \lceil U = h$, we have $h'(i) = h'(i')$ for all $i, i' \in U' - U$. Furthermore, the system **terminates** at $h \in \mathcal{L}(U)$ iff $(\forall U' \supseteq U)(\forall h' \in \mathcal{L}(U'))\ h' \lceil U \neq h$. $\Box$

Deadlock at h says that if the system has evolved through the events in h, then evolution into a different state is impossible. Terminating at h is a more drastic form of deadlock in which there is no possible future behaviour of the system that extends h. For example, consider a short circuited 6 volt battery: it cannot have both a 6 and a 0 volt potential difference between its terminals. Thus, a discrete time system in a state where in the next instant a switch will be thrown that produces a short circuit is terminal at that state. (Of course, a more realistic model would show a large current flow followed by the decay of either the battery and/or the wiring, but in the simple model chosen for this example, a short circuit is an inconsistency which precludes any future behaviour.)

The following is a common example in the Computing Science literature on concurrency:

Example 40: (Dining Philosophers): This somewhat fanciful situation involves four philosophers supported in a research institute with a circular table, the center of which always contains a plate of food. This food must be eaten seated at the table with one fork in each hand. The table has four forks, one between each two adjacent chairs. Philosophers are asynchronous processes that think and eat.

Let us now construct a formal model of this situation: Let $\mathcal{P} = \{P1, P2, P3, P4\}$ be the names of the four philosophers, and let $\mathcal{F} = \{F1, F2, F3, F4\}$ be names for the four forks. Then the following is a diagram for this situation as a system:



The fork object between philosophers P and P' has states $s \in \{P, P', \perp\}$ with the following transitions,

$$\begin{array}{lcl} s & \mapsto & s \\ \perp & \mapsto & s \\ s & \mapsto & \perp \end{array}$$

That is, a fork can remain as it is, can be picked up, and can be put down.

Each philosopher P has states $s = \langle l, n, r \rangle$ for $l, r \in \mathcal{P}_\perp = \mathcal{P} \cup \{\perp\}$ and $n \in \{t, e\}$ (t and e stand for "thinking" and "eating"[10] respectively), and has the following transitions:

$$\begin{array}{lcll}
 & \mapsto & \langle \perp, t, \perp \rangle & \\
\langle l, n, r \rangle & \mapsto & \langle l', n, r' \rangle & \textbf{if } l, l', r, r' \neq P \\
\langle l, t, r \rangle & \mapsto & \langle l', e, r' \rangle & \textbf{if } l, l', r, r' \neq P \\
\langle l, e, \perp \rangle & \mapsto & \langle l', e, P \rangle & \textbf{if } l, l' \neq P \\
\langle l, e, P \rangle & \mapsto & \langle l', e, P \rangle & \textbf{if } l, l' \neq P \\
\langle \perp, e, P \rangle & \mapsto & \langle l, e, P \rangle & \\
\langle P, e, P \rangle & \mapsto & \langle \perp, t, \perp \rangle &
\end{array}$$

The first rule says that philosophers are born thinking, without forks. The second rule says that if a philosopher has no forks, then he can continue in his present state, without constraining the actions of his two neighboring philosophers. The third rule says if he has no forks and is thinking, then he may become hungry, again without constraining his neighbors. The fourth rule says that if he has no forks, then he can pick up his right fork if it is available, without constraining his left neighbor (note that, by the second rule, he need not do so). The fifth rule says that he can remain in the state of having just a right fork, without constraining his left neighbor (but he does not need to so remain, because l can be any fork state). The sixth rule says that if he has a right fork, then he can pick up his left fork, if it is available. The last rule says that if he has both forks and is eating, then he can put them both down and think. (A slightly more accurate model might add a rule saying that if he has both forks he can continue to hold them and eat; under the above rules, he has only one unit of time in which to eat.)

The object for each philosopher has two projections, from its first and third components, as shown in the diagram above.

Let $\mathcal{L}$ denote the limit of this system, and let $I = \{0, ..., i\} \in \mathcal{I}_0(\omega)$. Then officially, the elements of $\mathcal{L}(I)$ are 8-tuples of functions on I, four of them 3-tuples for the philosophers, and four of them elements of $\mathcal{P}_\perp$, for the forks. However, it is equivalent to consider 16-tuple valued functions on I, by flattening the 3-tuples. In fact, these 16-tuples have a lot of redundancy, due to the constraints imposed by their representing consistent nets of points, and it suffices to consider 8-tuples of the form

$$s = \langle f_1, f_2, f_3, f_4, p_1, p_2, p_3, p_4 \rangle$$

with $f_i \in \mathcal{P}_\perp$ and $p_i \in \{t, e\}$, where f_i is the state of the i^{th} fork and p_i is the middle component of the state of the i^{th} philosopher. Hereafter, we will feel free to call p_i "the state of P_i" and to call such 8-tuples "states of the system". Notice that when the system is in state s, then each philosopher P_i (for $i = 1, 2, 3, 4$) is in the state $\langle f_{i-1}, p_i, f_{i+1} \rangle$, where the subscripts are understood modulo 4.

There is a unique state of the system in which each philosopher is in his initial state, namely

$$\langle \perp, \perp, \perp, \perp, t, t, t, t \rangle$$

and we will call this the "initial state of the system".

It is now easy to see that the state in which each philosopher holds his right fork is a reachable deadlock state. To see that it is reachable, it suffices to give a sequence transitions from the initial state to it. In fact, two parallel transitions are enough: in the first, each philosopher becomes hungry, and in the second, each philosopher picks up his right fork:

$$\langle \perp, \perp, \perp, \perp, t, t, t, t \rangle \mapsto \langle \perp, \perp, \perp, \perp, e, e, e, e \rangle \mapsto \langle P_1, P_2, P_3, P_4, e, e, e, e \rangle .$$

[10] A more accurate description would be "hungry and in a process that may lead to eating."

Let us denote this last state by x. To see that it is a deadlock state, we note that only the second rule applies to x. This is checked by seeing which lefthand sides of rules match $\langle P_{i-1}, e, P_{i+1}\rangle$, where the subscripts are again understood modulo 4. Therefore, the only possible transitions from x return the system to state x, because the corresponding righthand side of the second rule is again $\langle P_{i-1}, e, P_{i+1}\rangle$. This analysis also shows that the system is not terminal.

It is possible to modify this example to use real time, by giving a slightly different interpretation for the transition notation $\mapsto$; however, we omit this. □

Deadlock can also arise when testing to see if some system satisfies some property by interconnecting the system with the property and taking the limit: if the property is *inconsistent* with the system, in the sense that the system has no states with the given property, then the interconnected system exhibits deadlock.

4.2 Information Flow and Security

Another important property of real systems is *security*: we may want to be sure that access to certain confidential information is protected, and that certain unauthorised actions are prevented. For example, we don't want an electronic bank robber to be able to discover which are the largest accounts, and then withdraw funds from them. It is now rather well recognised that "non-interference" assertions can be used to express security properties of (monolithic) sequential systems [20, 21]. An active research topic is the extension of such assertions to more general classes of system. Definition 43 below extends non-interference to a much wider class of system, such as distributed object oriented databases, and appears to be more general than anything else in the literature. (But see [31, 38] for two other very general approaches.) First, we need some auxiliary material:

Definition 41: Let $\varphi\colon \mathcal{O} \to \mathcal{O}'$ be a morphism of set valued presheaves. Then the **image presheaf** of φ, denoted $\varphi(\mathcal{O}) \subseteq \mathcal{O}'$, is defined by $\varphi(\mathcal{O})(U) = \varphi_U(\mathcal{O}(U))$; note that $\varphi(\mathcal{O})(U) \subseteq \mathcal{O}'(U)$ for each $U \in \mathcal{T}$. □

This definition can be generalised to a suitable structure category $\mathcal{C}$, and is used in the last equation of Definition 43 below. But first note the following:

Fact 42: The image of a morphism of sheaves is a sheaf. □

Now the main concept:

Definition 43: Given a system $\mathcal{S}$ with objects $\mathcal{S}_n$ for $n \in N$, with morphisms $\varphi_e\colon \mathcal{S}_n \to \mathcal{S}_{n'}$ for $e\colon n \to n'$, and with behaviour (i.e., limit) $\mathcal{L}$ having projections $\pi_n\colon \mathcal{L} \to \mathcal{S}_n$, then $\mathcal{S}_m$ is **non-interfering with** $\mathcal{S}_k$, written $\mathcal{S}_m \not\rightsquigarrow \mathcal{S}_k$, iff the following holds:

> let $\mathcal{L}'$ be the limit of the subsystem of $\mathcal{S}$ from which $\mathcal{S}_m$ and all morphisms to and from $\mathcal{S}_m$ have been omitted; let $\pi'_n\colon \mathcal{L}' \to \mathcal{S}_n$ be its projections, for $n \in N - \{m\}$; then $\pi_k(\mathcal{L}) = \pi'_k(\mathcal{L}')$.

□

What this says[11] is that the behaviour of the system looks the same from the object $\mathcal{S}_k$ with the object $\mathcal{S}_m$ omitted as it does with the object $\mathcal{S}_m$ present; i.e., there is no flow of information

[11] It may help to think of "k" as a "crook" and "m" as a "market analyst" whose information k seeks to steal.

from $\mathcal{S}_m$ to $\mathcal{S}_k$. This definition is general enough to apply to data dependency analysis for compilers of concurrent languages onto distributed systems, and to the flow of information in natural language conversation, along the lines suggested by situation semantics [4] and the work of Dretske [6]. Indeed, it seems possible that sheaf theory could help in providing a natural semantics for situation theory.

5 Conclusions

Our sheaf approach provides a semantic, i.e., model theoretic, foundation for concurrent distributed computing by (possibly active) objects, without commitment to any particular notation or conceptualisation for concurrency. In general, such an approach should be closer to our physical intuition, and can provide standards against which to measure the soundness and completeness of syntactic systems.

Sheaf theory has been used in mathematics to study relationships between local and global phenomena, for example, in algebraic geometry, differential geometry, and even logic; the subject has also been developed in an abstract form using category theory. The theory of topoi, originally developed by Lawvere and Tierney (see [33], [26], [2]) is perhaps the most exciting development in this respect. An interesting topic for future research is to see what the theory of topoi can tell us about concurrency. For example, one should be able to reason about a system using the internal intuitionistic logic of the corresponding topos of sheaves.

Concepts from category theory have helped us achieve generality: objects have been modeled by sheaves; inheritance by sheaf morphisms; systems by diagrams; and interconnections by diagrams of diagrams. In addition, behaviour is given by limit, and the result of interconnection by colimit. The approach is illustrated with many examples, including a semantics for a simple concurrent object-based programming language.

The definitions, examples and results in this paper are just a beginning. Yet the variety of examples may be surprising. Moreover, one example illustrates an important class of applications, namely the semantics of concurrent, distributed object oriented systems. Some of the definitions may also be surprising for their generality, including a notion of security that generalises the Goguen-Meseguer non-interference approach [20, 21] from sequential systems to (for example) real time distributed concurrent object oriented databases. A very general definition of deadlock is also given. It is interesting that these concepts are so easily stated in a purely semantic form.

References

[1] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT, 1986.

[2] Michael Barr and Charles Wells. *Toposes, Triples and Theories.* Springer, 1984. Grundlehren der mathematischen Wissenschafter, Volume 278.

[3] Michael Barr and Charles Wells. *Category Theory for Computing Science.* Prentice-Hall, 1990.

[4] Jon Barwise and John Perry. *Situations and Attitudes.* MIT, 1983. Bradford Books.

[5] Jan Bergstra and Jan Willem Klop. Process algebra for synchronous communication. *Information and Control*, 60:190–137, 1984.

[6] Fred Dretske. *Knowledge and the Flow of Information.* MIT, 1981. Bradford Books.

[7] Hans-Dieter Ehrich, Joseph Goguen, and Amilcar Sernadas. A categorial theory of objects as observed processes. In J.W. de Bakker, Willem P. de Roever, and Gregorz Rozenberg, editors, *Foundations of Object Oriented Languages*, pages 203–228. Springer, 1991. Lecture Notes in Computer Science, Volune 489; Proceedings, REX/FOOL Workshop, Noordwijkerhout, the Netherlands, May/June 1990.

[8] Hans-Dieter Ehrich and Amilcar Sernadas. Algebraic implementation of objects over objects. In J.W. de Bakker, Jan Willem deRoever, and Gregorz Rozenberg, editors, *Proceedings, REX Workshop on Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*, pages 239–266. Springer, 1990. Lecture Notes in Computer Science, Volume 430.

[9] Hans-Dieter Ehrich, Amilcar Sernadas, and Christina Sernadas. Objects, object types, and object identification. In Hartmut Ehrig et al., editors, *Categorical Methods in Computer Science with Aspects from Topology*, pages 142–156. Springer, 1989. Lecture Notes in Computer Science, Volume 393.

[10] Hans-Dieter Ehrich, Amilcar Sernadas, and Christina Sernadas. From data types to object types. *Journal of Information Processing and Cyberntics*, 26(1/2):33–48, 1990.

[11] Gian Luigi Ferrari. *Unifying Models of Concurrency*. PhD thesis, University of Pisa, 1990.

[12] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 46:1–102, 1986.

[13] Joseph Goguen. Mathematical representation of hierarchically organized systems. In E. Attinger, editor, *Global Systems Dynamics*, pages 112–128. S. Karger, 1971.

[14] Joseph Goguen. Categorical foundations for general systems theory. In F. Pichler and R. Trappl, editors, *Advances in Cybernetics and Systems Research*, pages 121–130. Transcripta Books, 1973.

[15] Joseph Goguen. Objects. *International Journal of General Systems*, 1(4):237–243, 1975.

[16] Joseph Goguen. A categorical manifesto. *Mathematical Structures in Computer Science*, 1(1):49–67, March 1991. Also, Programming Research Group Technical Monograph PRG-72, University of Oxford, March 1989.

[17] Joseph Goguen. Semantic specifications for the rewrite rule machine. In Aki Yonezawa and Takayasu Ito, editors, *Concurrency: Theory, Language and Architecture*, pages 216–234, 1991. Proceedings of a U.K.–Japan Workshop; Springer, Lecture Notes in Computer Science, Volume 491.

[18] Joseph Goguen and Susanna Ginali. A categorical approach to general systems theory. In George Klir, editor, *Applied General Systems Research*, pages 257–270. Plenum, 1978.

[19] Joseph Goguen, Sany Leinwand, José Meseguer, and Timothy Winkler. The Rewrite Rule Machine, 1988. Technical Report Technical Monograph PRG-76, Programming Research Group, Oxford University, 1989.

[20] Joseph Goguen and José Meseguer. Security policies and security models. In Marvin Schafer and Dorothy D. Denning, editors, *Proceedings, 1982 Symposium on Security and Privacy*, pages 11–22. IEEE Computer Society, 1982.

[21] Joseph Goguen and José Meseguer. Unwinding and inference control. In Dorothy D. Denning and Jonathan K. Millen, editors, *Proceedings, 1984 Symposium on Security and Privacy*, pages 75–86. IEEE Computer Society, 1984.

[22] Joseph Goguen and José Meseguer. Unifying functional, object-oriented and relational programming, with logical semantics. In Bruce Shriver and Peter Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 417–477. MIT, 1987. Preliminary version in *SIGPLAN Notices*, Volume 21, Number 10, pages 153-162, October 1986.

[23] Joseph Goguen and José Meseguer. Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. Technical Report SRI-CSL-89-10, SRI International, Computer Science Lab, July 1989. Originally given as lecture at *Seminar on Types*, Carnegie-Mellon University, June 1983; many draft versions exist.

[24] Joseph Goguen, James Thatcher, and Eric Wagner. An initial algebra approach to the specification, correctness and implementation of abstract data types. Technical Report RC 6487, IBM T.J. Watson Research Center, October 1976. In *Current Trends in Programming Methodology, IV*, Raymond Yeh, editor, Prentice-Hall, 1978, pages 80-149.

[25] Joseph Goguen and David Wolfram. On types and FOOPS. In William Kent Robert Meersman and Samit Khosla, editors, *Object Oriented Databases: Analysis, Design and Construction*, pages 1–22. North Holland, 1991. Proceedings, IFIP TC2 Conference, Windermere, UK, 2-6 July 1990.

[26] Robert Goldblatt. *Topoi, the Categorial Analysis of Logic*. North-Holland, 1979.

[27] Michael J.C. Gordon. Why higher-order logic is a good formalism for specifying and verifying hardware. In George Milne and P.A. Subrahmanyam, editors, *Formal Aspects of VLSI Design*. North-Holland, 1986.

[28] John Gray. Sheaves with values in a category. *Topology*, 3(1):1–18, 1965.

[29] Alexandre Grothendieck. Catégories fibrées et descente. In *Revêtements étales et groupe fondamental, Séminaire de Géométrie Algébraique du Bois-Marie 1960/61, Exposé VI.* Institut des Hautes Études Scientifiques, 1963. Reprinted in Lecture Notes in Mathematics, Volume 224, Springer, 1971, pages 145–94.

[30] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[31] Jeremy Jacob. A security framework. In *Proceedings, 1989 Computer Security Foundations Workshop*, pages 98–111. MITRE, 1988. Franconia, New Hampshire.

[32] Saunders Mac Lane. *Categories for the Working Mathematician*. Springer, 1971.

[33] F. William Lawvere. Introduction. In F. William Lawvere, C. Maurer, and Gavin Wraith, editors, *Model Theory and Topoi*, pages 3–14. Springer, 1975. Lecture Notes in Mathematics, Volume 445.

[34] Johan Lilius. Sheaf semantics for Petri nets. Technical report, Helsinki University of Technology, 1991.

[35] José Meseguer and Ugo Montanari. Petri nets are monoids: A new algebraic foundation for net theory. In *Proceedings, Symposium on Logic in Computer Science*. IEEE Computer Society, 1988. Full version in Report SRI-CSL-88-3, Computer Science Laboratory, SRI International, January 1988; submitted to *Information and Computation*.

[36] Robin Milner. *A Calculus of Communicating Systems*. Springer, 1980. Lecture Notes in Computer Science, Volume 92.

[37] Luis Monteiro and Fernando Pereira. A sheaf-theoretic model of concurrency. Technical Report CSLI-86-62, Center for the Study of Language and Information, Stanford University, October 1986.

[38] Colin O'Halloran. A calculus of information flow. Technical report, Royal Signals and Radar Establishment, Malvern, 1990.

[39] Benjamin C. Pierce. A taste of category theory for computer scientists. Technical Report CMU-CS-90-113, Carnegie-Mellon University, 1990.

[40] W. Reisig. *Petri Nets: An Introduction*. Springer, 1986. EATCS Monographs on Theoretical Computer Science.

[41] Susan Leigh Star. The structure of ill-structured solutions: heterogeneous problem-solving, boundary objects and distributed artificial intelligence. In Michael Huhns and Les Gasser, editors, *Distributed Artificial Intelligence*, volume 3, pages 37–54. Morgan Kauffmann, 1988.

[42] Victoria Stavridou, Joseph Goguen, Steven Eker, and Serge Aloneftis. FUNNEL: A CHDL with formal semantics. In *Proceedings, Advanced Research Workshop on Correct Hardware Design Methodologies*, pages 117–144. IEEE, 1991. Turin.

[43] Andrzej Tarlecki, Rod Burstall, and Joseph Goguen. Some fundamental algebraic tools for the semantics of computation, part 3: Indexed categories. *Theoretical Computer Science*, 91:239–264, 1991. Also, Monograph PRG-77, August 1989, Programming Research Group, Oxford University.

[44] B.R. Tennison. *Sheaf Theory*. Cambridge, 1975. London Mathematical Society Lecture Notes Series, 20.

[45] Glynn Winskel. A compositional proof system on a category of labelled transition systems. *Information and Computation*, 87:2–57, 1990.

[46] Kosaku Yosida. *Functional Analysis*. Springer, 1968. Second Edition.

Contents