A Semantic Framework for Object-Oriented Development

Tony Clark (a.n.clark@comp.brad.ac.uk)

April 14, 1999

Abstract

A categorical model of object-oriented systems is proposed and denoted using a λ-calculus. The model is used to provide a definition of design refinement. An example system is rigorously developed in Java from an initial user requirements by refining an initial design.

Contents

1 Introduction

Current object-oriented design notations such as OMT [Run91], Booch [Boo94] and UML [UML98] are syntax-bound and semantic-free in the sense that they typically employ a large and rigorously enforceable collection of construction rules, but rarely provide a model to explain what is being constructed. Whilst this omission clearly does not prevent such notations being used effectively in the development of object-oriented software systems, it must raise questions regarding the long-term viability of notations which are not adequately anchored in a semantic theory.

The aims of this work are to provide a semantic framework suitable for such notations and which can form a basis for rigorous object-oriented development. Our approach is to take as a starting point the computational behaviour of objects and to provide a semantic model of incremental system development.

A system is defined as the solution to a set of simultaneous equations which specify its computational behaviour and structure. We use category theory [Bar90] [Ryd88] [Gog89] as a tool to express the equations since this theory provides standard constructions and results which conveniently express semantics without getting unnecessarily entangled in issues of syntax.

An object-oriented design can be characterised with respect to the structure of system states and a given set of atomic computation steps. An initial system design tends to abstract away from structural and computational detail. One of the aims of object-oriented system development is to *refine* an initial design into an object-oriented design. This paper proposes a definition of object-oriented refinement which supports verifiable step-wise system development.

This paper is structured as follows. Section 2 identifies a number of key features which are common to all object-oriented design notations. These features are the motivation for the design of a behavioural object model described in section 3. The model expresses behaviour as objects in a category and refinement as a relationship between objects in a category. A notation based on the λ-calculus is proposed for denoting behaviours in section 4. Systems are built from collections of behaviour descriptions expressed as standard categorical constructions defined in section 5. A system is a collection of constraints on possible object behaviours. The overall system behaviour must satisfy all of the constraints. A standard result from category theory is described in section 6 which provides an algorithm for finding a system behaviour. Section 7 shows an example object-oriented design using all the design features defined in the paper. Finally, section 8 analyses the work, discusses related work and outlines future research.

The paper aims to be self contained with respect to the necessary category theory. Readers are directed to [Pri97] for an overview of graphical object-oriented design notations and to [Fie88] for an overview of λ-notation and functional programming.

2 Object-Oriented Design Features

Object-oriented designs, as expressed using a typical design notation such as UML, consist of a number of different models. Each model is used to express a different feature of the required system. Although the design notations differ syntactically, we propose that there are a number of characterising features common to all object-oriented systems. This section lists these features which are then formalised in the rest of the paper. A more detailed analysis of object-oriented features can be found in [Weg87], [Mey88], [Cla96], [Cla94]. The features listed below are discussed at length in [Cla99b].

Design Feature 1 *Objects have state consisting of names and values.*

Design Feature 2 *Messages are passed via named associations.*

Design Feature 3 *Object state transitions involve input and output messages.*

Design Feature 4 *A class represents a collection of objects with the same behaviour.*

Design Feature 5 *Two instances of the same class differ with respect to their identity.*

Design Feature 6 *Object-Oriented designs are (possibly) non-deterministic.*

Design Feature 7 *Object-Oriented designs support inheritance.*

Design Feature 8 *Object designs are compositional.*

Design Feature 9 *Object designs may be partial.*

Design Feature 10 *Communication may be synchronous or asynchronous.*

Design Feature 11 *Object designs can be inter-dependent.*

Design Feature 12 *Global behaviour satisfies all local behavioural constraints.*

Design Feature 13 *A design refinement involves soundness and completeness checks.*

3 Object Behaviour

Object-oriented designs denote the structure and behaviour of systems. Objects calculate by performing state transitions and communicate by passing messages. System behaviour is represented as a graph labelled with states and messages. Object-oriented behaviours conform to a particular class of states and transitions. Refinement transforms behaviours so that they are more object-oriented whilst preserving their meaning.

3.1 Object States

At any given moment in time, an object exists in a particular state. The state of an object provides a complete description of its type, its identity and its attributes.

All objects have a class (feature 4) which defines the behaviour of the object. It is possible to distinguish between instances of two different classes which both define the same attributes. Class identity is represented as a type tag α where each class is allocated a different tag.

All objects have an identity (feature 5). An object's identity distinguishes it from all other instances of the same class in the same state. Object identity is represented as an object tag τ where each object has a different tag.

All objects have attributes which are named values (feature 1). The attributes of an object determine the behaviour of the object when it receives a message. The attributes of an object may change when a message is processed. The attributes of an object are represented as a partial function ρ from attribute names to values. An attribute j is renamed i in ρ to produce a new attribute function $\rho[i/j]$ defined as follows:

$$\rho[i/j](k) = \begin{cases} \rho(i) & \text{when } k = j \\ \rho(k) & \text{otherwise} \end{cases}$$

A *state* is $<\alpha, \tau, \rho>$ and represents a partial view of an object. Two different views of the same object must have the same type and identity but may differ with respect to the attributes providing that the attribute views are consistent.

Consistency is defined as follows. Let $\leq$ (is more defined than) be a partial order on attribute descriptions such that $\rho_1 \leq \rho_2$ when $\rho_1(a) = \rho_2(a)$ for all attributes $a \in dom(\rho_2)$. Any two attribute descriptions are *consistent* when there is a greatest lower bound attribute description $\rho_1 \sqcap \rho_2$.

The partial order on attribute descriptions can be extended to states by requiring that $<\alpha_1, \tau_1, \rho_1> \leq <\alpha_2, \tau_2, \rho_2>$ holds if and only if $\alpha_1 = \alpha_2$, $\tau_1 = \tau_2$ and $\rho_1 \leq \rho_2$.

Object-oriented systems consist of sets of object states. A set is *well formed* when it contains only one state for a given object identity. We can define a partial order on well formed sets of object states as follows. Let Σ_1 and Σ_2 be two well formed sets of object states. The relation $\Sigma_1 \leq \Sigma_2$ holds when for each object state in Σ_2 there is an object state in Σ_1 which is consistent. This can be stated formally as:

$$\forall <\alpha_2, \tau_2, \rho_2> \in \Sigma_2 \bullet \exists <\alpha_1, \tau_1, \rho_1> \in \Sigma_1 \bullet \alpha_1 = \alpha_2 \wedge \tau_1 = \tau_2 \wedge \rho_1 \leq \rho_2$$

Given two well formed sets Σ_1 and Σ_2 of object states the greatest lower bound $\Sigma_1 \sqcap \Sigma_2$ is the smallest set containing all of the object states from both Σ_1 and Σ_2 where different views of the same object have been merged consistently.

3.2 Object Calculation Graphs

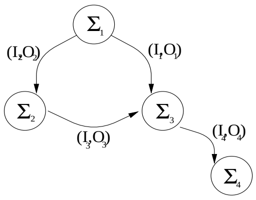
Systems are constructed as a collection of objects. Each object is a separate computational system with its own state (feature 1) modified in response to handling messages (feature 2). A message is a package of information sent from one object to another.

The computation performed when a message is handled by an object depends on the object's current state and causes the object to change state and produce output messages (feature 3). If we observed an object over a period of time we would see a sequence of messages and state changes: $\ldots \sigma_1 \overset{(I_1,O_1)}{\longmapsto} \sigma_2 \overset{(I_2,O_2)}{\longmapsto} \sigma_3 \ldots$ where each σ_j is an object state, I_j are input messages, and O_j are output messages. Such a sequence is an *object calculation* and describes a single object in state σ_j receiving messages I_j causing a state change to σ_{j+1} and producing output messages O_j.

A message consists of a source object, a target object and some message data. The source and target objects are identified by their object identity tags. For a given object system, the data items which can be passed as messages will be defined for each type of target object. A message, whether input or output, is represented as $<\tau_s, \tau_t, \nu>$ where τ_s identifies the source object, τ_t identifies the target object and ν is the message data.

Object systems are constructed from multiple objects interacting by passing messages. The state of an object system is a well formed set of object states Σ. Computation in an object system occurs when the messages in set I are sent to the objects in Σ producing a new set of object states Σ' and a collection of output messages O: $\ldots \longmapsto \Sigma \overset{(I,O)}{\longmapsto} \Sigma' \longmapsto \ldots$ Object-oriented designs represent non-deterministic computational systems (feature 6). We can therefore define all the possible object calculations performed by an object system O in response to handling sequences of input messages of length n.

Object calculations are represented as a *calculation graph* $O(n)$ where the nodes of the graph are labelled with well formed sets of states and the edges are labelled with pairs of input and output message sets. An example graph G_x is:



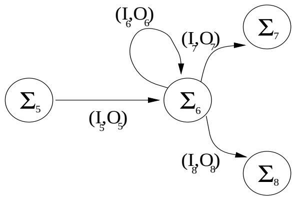
Starting in state Σ_1, the graph G_x can produce the following possible object calculations of length 2:

$$\Sigma_1 \overset{(I_1,O_1)}{\longmapsto} \Sigma_3 \overset{(I_4,O_4)}{\longmapsto} \Sigma_4$$
$$\Sigma_1 \overset{(I_2,O_2)}{\longmapsto} \Sigma_2 \overset{(I_3,O_3)}{\longmapsto} \Sigma_3$$

A graph $G = (N, E, s : E \to N, t : E \to N)$ is a set of nodes N, a set of edges E and a pair of mappings s which maps an edge to its source node, and t which maps an edge to its target node. A graph homomorphism $(\phi_n, \phi_e) : G_1 \to G_2$ is a mapping from graph G_1 to graph G_2 consisting of a pair of mappings $\phi_n : N_1 \to N_2$ and $\phi_e : E_1 \to E_2$ such that the following diagrams commute:

$$\begin{tikzcd} E_1 \arrow[r, "\phi_e"] \arrow[d, "t_2"'] & E_2 \arrow[d, "t_1"] \\ N_1 \arrow[r, "\phi_n"'] & N_2 \end{tikzcd} \qquad \begin{tikzcd} E_1 \arrow[r, "\phi_e"] \arrow[d, "s_2"'] & E_2 \arrow[d, "s_1"] \\ N_1 \arrow[r, "\phi_n"'] & N_2 \end{tikzcd}$$

Consider the following graph G_y:



We can define a graph homomorphism $\phi : G_x \to G_y$ such that $\phi_n = \{\Sigma_1 \mapsto \Sigma_5, \Sigma_2 \mapsto \Sigma_6, \Sigma_3 \mapsto \Sigma_6, \Sigma_4 \mapsto \Sigma_7\}$ and $\phi_e = \{(I_1, O_1) \mapsto (I_5, O_5), (I_2, O_2) \mapsto (I_5, O_5), (I_3, O_3) \mapsto (I_6, O_6), (I_4, O_4) \mapsto (I_7, O_7)\}$ so that $\phi(G_x)$ is included in G_y.

3.3 Object-Oriented Designs

Object-oriented designs must conform to certain structural and behavioural principles. The structure of all objects in the design should uphold the *principle of encapsulation* which requires that the format of data within each object is hidden and accessible only through its message interface. This implies that the value of each object attribute must be atomic and that objects refer to each other via their object identifiers.

The behaviour of objects in an object-oriented design should uphold the *principle of atomic communication* whereby a single system state transition, occurring in response to a message m, may only affect the state of the target of m. This implies that a design in which a single message affects objects other then the target must be refined in order to decompose the single transition into a sequence of transitions involving multiple messages.

3.4 Object Semantics

The meaning of an object is defined to be the calculation graph describing all of its possible behaviours. We will give a precise object semantics using simple constructions from category theory. In order to be self contained we include definitions of category, terminal object, functor, and natural transformation.

A *category* consists of a collection of objects[1], a collection of arrows and an infix binary associative operator $\circ$. Upper case letters A, B, ... are used to range over objects. Lower case letters f, g, ..., are used to range over arrows. Each arrow has a domain object A and a range object B and is written $f : A \to B$. The operator $\circ$ which maps a pair of arrows $f : A \to B$ and $g : B \to C$ to an arrow $g \circ f : A \to C$. Every object A in a category has an identity arrow $id_A : A \to A$ which is the left and right identity of $\circ$. An example category is **Int** whose objects are integers. There is an arrow $f : n \to m$ in **Int** for every pair of integers n and m such that $n \leq m$. Another example category is **Calc** whose objects are calculation graphs and whose arrows are graph homomorphisms.

A *terminal object* in a category is an object A such that for all objects B in the category there is an arrow $f : B \to A$. The terminal object in **Calc** is a graph with a single node labelled $\emptyset$ and a single edge (from $\emptyset$ to $\emptyset$) labelled with $(\emptyset, \emptyset)$. For each object in **Calc** there is exactly one arrow which maps all nodes to $\emptyset$ and all edges to $(\emptyset, \emptyset)$.

A *functor* consists of a source category $\mathbf{C}$ and a target category $\mathbf{D}$, and a function F_1 which maps objects of $\mathbf{C}$ to objects of $\mathbf{D}$, and function F_2 which maps arrows of $\mathbf{C}$ to arrows of $\mathbf{D}$. For every $\mathbf{C}$ arrow $f : A \to B$, $F_2(f) : F_1(A) \to F_1(B)$ in $\mathbf{D}$. For every $\mathbf{C}$ object A, $F_2(id_A) = id_{F_1(A)}$. For every pair of composable $\mathbf{C}$ arrows $g \circ f$, $F_2(g \circ f) = F_2(g) \circ F_2(h)$ in $\mathbf{D}$.

Given two functors $F : \mathbf{C} \to \mathbf{D}$ and $G : \mathbf{C} \to \mathbf{D}$ a *natural transformation* $\gamma : F \to G$ is defined as a family of arrows γ_A indexed by objects A of $\mathbf{C}$ such that $\gamma_A : C(A) \to D(A)$ for every object A of $\mathbf{C}$ and the following diagram

[1] The term *object* is used in the mathematical rather than the software sense.

commutes for all **C** arrows $f : A \to B$:

$$\begin{tikzcd} C(A) \arrow[r, "\gamma_A"] \arrow[d, "C(F)"'] & D(A) \arrow[d, "D(f)"] \\ C(B) \arrow[r, "\gamma_B"'] & D(B) \end{tikzcd}$$

An object O is described in terms of its calculations. A collection of graphs $O(0), O(1), O(2), \ldots, O(n)$ describe calculations arising out of sequences of messages of length $0, 1, 2, \ldots, n$. Consider two integers n and m such that $n \leq m$. Both integers produce calculation graphs $O(n)$ and $O(m)$. If the object O is well-behaved then there must be a graph homomorphism $\phi : O(n) \to O(m)$.

This leads us to define objects as functors from the category **Int**, whose objects are integers and morphisms $f : n \to m$ hold when $n \leq m$, to the category **Calc**, whose objects are object calculation graphs and morphisms are graph homomorphisms.

Let **Obj** be a category whose objects are functors from **Int** to **Calc** and whose arrows are natural transformations between functors. An object in **Obj** will be referred to as a *behaviour* and an arrow as a *behaviour morphism*.

Object-oriented design notations provide models which express objects in terms of states, associations and messages. The semantics of these models is provided by objects in **Obj** defined using standard categorical constructions.

3.5 Behaviour Refinement

Not all system designs are object-oriented. It is often convenient in the early stages of development to ignore the principles of encapsulation and atomic communication. By ignoring them it is possible to abstract away from computational details.

During system development it is necessary to modify a design in order that it conforms to the principles underlying object-orientation. Assuming that the initial design is correct, any modifications will change the structure and computation of the system but must preserve the overall behaviour.

This section defines *behavioural equivalence* in terms of behaviour isomorphisms. An equivalence relation on behaviours is then restricted to produce a directed refinement relation defined as adjoint functors between behaviours viewed as categories. A refinement relation can be used in system development in order to transform an initial design into an object-oriented design.

Consider two behaviours O_1 and O_2 such that $O_1 \equiv O_2$. Every computation performed by O_1 can be performed by O_2 and vice versa. This is defined by requiring equivalent behaviours to be related by a behaviour isomorphism: there are two behaviour morphisms $\gamma_1 : O_1 \to O_2$ and $\gamma_2 : O_2 \to O_1$ such that $\gamma_2 \circ \gamma_1 = O_{1id}$ and $\gamma_1 \circ \gamma_2 = O_{2id}$.

Software development is directed; we start with an initial design and wish to terminate with an object-oriented design. Verification of the process must ensure that all the initial behaviour is preserved (*completeness*) and that no junk behaviour is introduced (*soundness*). Equivalence can be relaxed slightly to reflect development as follows.

Any behaviour O can be viewed as a category in which the objects are behaviour states and arrows are sequences of message pairs. Category-hood follows from: every object Σ has an identity arrow $[]$; and for every pair of arrows $f : \Sigma_1 \to \Sigma_2$ and $g : \Sigma_2 \to \Sigma_3$ there is an arrow $g \circ f : \Sigma_1 \to \Sigma_3$ which is constructed as $f +\!\!+ g$; and the associativity of $\circ$ follows from the associativity of $+\!\!+$.

A refinement R is a functor $R : O_1 \to O_2$ between behaviour categories; O_2 is a more computational version if O_1. In order to check the refinement there must be an anti-refinement (or *coarsening*) $U : O_2 \to O_1$. U forgets the extra computational structure in O_2.

Let Σ_1 be a state of behaviour O_1 and Σ_2 be a state of behaviour O_2. Now consider two computations $f : \Sigma_1 \to U(\Sigma_2)$ and $g : R(\Sigma_1) \to \Sigma_2$ which are performed by behaviour O_1 and O_2 respectively. The refinement can be expressed as a diagram:

$$\begin{tikzcd} \Sigma_1 \arrow[r, "f"] \arrow[d] & U(\Sigma_2) \\ R(\Sigma_1) \arrow[r, "g"'] & \Sigma_2 \arrow[u] \end{tikzcd} \tag{1}$$

The diagram states that performing a computation in the source object is the same as translating the source state, performing the computation in the target object and then translating the target state. This issues are similar to those arising in compiler correctness [Sab97]. Given any Σ_1 the refinement is *sound* if for every f there exists a g and is *complete* if for every g there is an f.

4 Object-Oriented Design Notation

Rather than use a graphical design notation such as UML to denote behaviours, a textual design language is defined whose semantics is given by constructions in **Obj**. In principle, there are many different possible choices of language to denote constructions in **Obj**. One possibility is to use a form of modal logic where statements in the logic express properties about *multiple worlds*. A world can correspond to a set of object states and relationships between worlds correspond to transitions arising due to messages.

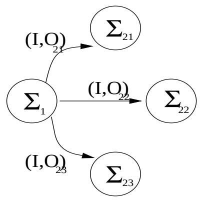
A problem with this approach is that formal logic tends to have a flat structure and does not lend itself to modular system construction. In addition, one of the strengths of formal logic, namely its ability to describe systems by abstracting away from computational detail, can be a weakness when we know the computational model which must be used.

Following Landin [Lan64] we take a different approach which is to use an extension of λ-notation as our design language. Since λ-calculi are the canonical programming languages this allows us to express the computational features of the design whilst the extensions abstract away from unnecessary computational design choices.

4.1 Behaviour Functions

A family of behaviours can be represented as a function with the following form: $M = \lambda i_1.\lambda i_2.\lambda i_3.\lambda i_4.N$ where $i_1 - i_4$ are parameters and N is the body. The first parameter is supplied with a type tag α. The result is a function which describes the behaviour of a class of objects. The second parameter is supplied with an object tag τ. The result is a function which describes the behaviour of a single object in all possible states. The third parameter is supplied with a value v which is the state of an object. The result is a function which describes the behaviour of a single object starting with a particular initial state v. The fourth parameter is supplied with a set of messages I. The result is a set of pairs: $M(\alpha)(\tau)(v)(I) = \bigcup_{i=1,n}\{(P_i, O_i)\}$ where P_i are *replacement behaviours* and O_i are corresponding output messages. The replacement behaviours are functions which determine the response to subsequent messages. Actor theory [Agh91] [Agh86] uses the same approach to functionally model concurrent systems.

Suppose that Σ_1 is a set of states described by P, Σ_{2i} is the set of states described by P_i for $i = 1, 3$ then the calculation graph which is described by the application of P is:



Subsequent components of the graph are constructed using the appropriate replacement object P_i. The body N of behaviour function handles messages by performing case analysis on the value of m. The body is of the form:

$$
\begin{array}{l}
\textbf{case } m \textbf{ of} \\
\quad p_1 \rightarrow e_1 \\
\quad p_2 \rightarrow e_2 \\
\quad \ldots \\
\quad p_n \rightarrow e_n \\
\quad \textbf{else } e \\
\textbf{end}
\end{array}
$$

where m is an expression whose value is a set of input messages[2], p_i are patterns which match input messages, e_i and e are *transition expressions* whose values are pairs of replacement object behaviour functions and sets of output messages.

[2] To avoid notational clutter singleton sets are identified with their element and messages whose source tag is unused are associated with their data compoment.

The operational semantics of **case** is as follows: m is evaluated and matched against all of the patterns p_i. Pattern matching produces a collection of variable bindings whose scope is the corresponding transition expression e_i. For each pattern which matches, the transition expression is evaluated to produce a collection of pairs (P, O). If no pattern matches then the optional default transition expression is evaluated. The result of evaluating **case** is the set of pairs resulting from evaluating transition expressions whose patterns match the input messages. A case expressions of the form **case** m **of else** e **end** is equivalent to just the expression e.

A transition expression denotes a collection of pairs of the form (P, O). In principle a transition expression can be of arbitrary complexity, however the following forms are frequently used:

e **when** c	where e is a transition expression and c is a boolean expression. The boolean expression acts as a guard on the transition.
e **whererec** b	where e is a transition expression and b is a collection of mutually recursive bindings whose scope is e.
(P, O)	where P is an expression denoting a replacement behaviour function and O is an expression denoting a set of output messages.

4.2 Example Behaviour Functions

A terminal object in **Obj** has no state and can respond only to an empty set of messages. It is defined below and is unique up to isomorphism since we do not specify a type or object tag:

$$\textbf{letrec } \mathit{empty}(\emptyset) = \{(\mathit{empty}, \emptyset)\}$$

Consider describing the behaviour of a single cell which stores a value. A cell object can be sent a message *set* which changes the value of its value and a message *get* which retrieves its value. The behaviour is as follows (note that we omit the source and target tags from message patterns when they are not used):

$$\begin{array}{l}
\textbf{letrec } \mathit{cell}(\alpha)(\tau)(v)(m) = \\
\quad \textbf{case } m \textbf{ of} \\
\quad\quad \mathit{set}(v') \to (\mathit{cell}(\alpha)(\tau)(v'), \emptyset) \\
\quad\quad <\tau', \tau, \mathit{get}> \to (\mathit{cell}(\alpha)(\tau)(v), \{<\tau, \tau', v>\}) \\
\quad\quad \textbf{else } (\mathit{cell}(\alpha)(\tau)(v), \emptyset) \\
\quad \textbf{end}
\end{array}$$

A class is created by sealing the type tags which occur in the object calculations. Suppose that α_1 is a type tag for a class of cell objects:

$$\textbf{let } \mathit{cellClass} = \mathit{cell}(\alpha_1)$$

A cell object is created by supplying the values of the attributes and the object tag:

$$\textbf{let } c_1 = \mathit{cellClass}(\tau_1)(0)$$

Suppose that we wish to set the contents of cell c_1 to 1 and then to retrieve its value. This is achieved by sending it messages from a hypothetical source object τ_0:

$$c_1(<\tau_0, \tau_1, \mathit{set}(1)>) = \{(c_2, \emptyset)\}$$

$$c_2(<\tau_0, \tau_1, \mathit{get}>) = \{(c_2, \{<\tau_1, \tau_0, 2>\})\}$$

4.3 Message Handling

Computation occurs in an object-oriented system in terms of message passing. A behaviour is expressed in the design notation as a function which maps incoming messages to a pair (P, O) where P is a replacement behaviour and O is a set of outgoing messages. Once the messages O have been produced, the behaviour is immediately ready to handle new incoming messages as specified by P.

The basic model of message handling is therefore *asynchronous*. This decision arises because object-oriented design notations can express both synchronous and asynchronous message passing. Typically there are different notations to express *send message and wait for reply* and *send message without waiting for reply*.

Basing the semantic model on asynchronous message passing does not preclude synchronous message passing since an asynchronous model which incorporates replacement behaviours can implement synchronous messages. A message $m \in O$ is sent synchronously when P is a behaviour that waits for an incoming message m' such that m' is the response to m. When m' is received the behaviour reverts to its original functionality.

Variations on the synchronous model described above are possible. For example, the waiting behaviour may permit a sub-set of the functionality, or may implement a priority based interrupt mechanism, or may allow the behaviour to send messages to itself.

The example program development described in this paper uses a form of synchronous message passing. It is convenient to add syntactic sugar to the design notation capturing this form of message passing. The sugar is a form of **let** expression occurring in the context of a behaviour function as follows:

$$
\begin{array}{l}
\textbf{letrec } \mathit{agent}(\alpha)(\tau)(\sigma)(m) = \\
\quad \textbf{case } m \textbf{ of} \\
\qquad \ldots \\
\qquad p_1 \rightarrow \\
\qquad\quad \textbf{let } p_2 \leftarrow e_1 \\
\qquad\quad \textbf{in } e_2 \\
\qquad \ldots \\
\quad \textbf{end}
\end{array}
$$

A **let** expression occurs in the context of a behaviour, represented here by the function *agent*. The expression e_1 is a set of messages or a single message which is to be sent in response to receiving a message matching p_1. The behaviour *agent* may carry on handling messages[3]. Any incoming message matching p_2 is a reponse to the messages e_1; the response of *agent* is defined by e_2.

The semantics of **let** is defined by a syntax translation to the basic design notation:

$$
\begin{array}{l}
\textbf{letrec}\ \mathit{agent}(\alpha)(\tau)(\sigma)(m) = \\
\quad \textbf{case}\ m\ \textbf{of} \\
\qquad \ldots \\
\qquad p_1 \rightarrow (\mathit{agent}(\alpha)(\tau)(\sigma) + \mathit{wait}, e_1) \\
\qquad\quad \textbf{whererec}\ \mathit{wait}(m) = \\
\qquad\qquad \textbf{case}\ m\ \textbf{of} \\
\qquad\qquad\quad p_2 \rightarrow e_2 \\
\qquad\qquad\quad \textbf{else}\ (\mathit{wait}, \emptyset) \\
\qquad\qquad \textbf{end} \\
\qquad \ldots \\
\quad \textbf{end}
\end{array}
$$

The locally created behaviour *wait* is used to extend *agent* with a handler for the response to messages e_1. Typically, when the response occurs, e_2 will revert back to the original behaviour *agent*.

4.4 System Execution

A variety of system execution mechanisms are possible using object designs in the format described above. Typically we wish to inject a single message into a system and then observe the messages which emerge. Suppose that, given a system of objects expressed as a behaviour function o and a set of messages m that: o/m is a set of messages produced by restricting m to those whose targets are in o; and, $o\backslash m$ is $m - (o/m)$, *i.e.* the messages in m whose targets are not in o. The function *exec* is supplied with a system of objects o and a set of initial messages m and produces a sequence of messages:

$$
\begin{array}{l}
\textbf{letrec}\ \mathit{exec}(o)(m) = \\
\quad \textbf{let}\ (o', m') = o(m) \\
\quad \textbf{in}\ (m' \backslash o') : (\mathit{exec}(o')(m'/o'))
\end{array}
$$

4.5 Semantics of Behaviour Functions

The meaning of a behaviour function is defined by a partial mapping from λ-terms to behaviours. Before defining the semantics we establish some terminology. A set of behaviours which differ only with respect to object tags is referred to as a *class behaviour*. A set of class behaviours which differ only with respect to the class tags is referred to as a *family of behaviours*.

[3] In fact we only require *agent* to handle messages which it sends to itself. The extra machinery for this feature is straightforward but would clutter the example so we omit it.

Suppose that M is a behaviour function $M = \lambda i_1.\lambda i_2.\lambda i_3.\lambda i_4.N$. Let $[\![M(\alpha)(\tau)(v)]\!]$ be the behaviour constructed by supplying all possible sequences of messages to object τ in state v. Let $[\![M(\alpha)(\tau)]\!]$ be the behaviour constructed by supplying the object τ with all possible message sequences in all possible states. Let $[\![M(\alpha)]\!]$ be the class behaviour formed by supplying all possible instances of α in all possible states with all possible message sequences. Finally, let $[\![M]\!]$ be the family of behaviours constructed by supplying M with all possible class tags, object tags, states and message sequences.

4.6 Behaviour Morphisms

Arrows in **Obj** are families of graph homomorphisms which must be well-behaved with respect to message sequences (see section 3.4). This is expressed by stating that behaviour morphisms are natural transformations. Let O_1 and O_2 be behaviours (*i.e.* functors from **Int** to **Calc**). A morphism $\gamma : O_1 \rightarrow O_2$ from O_1 to O_2 is a family of graph homomorphisms γ_n for each object in **Int** such that for any arrow $f : n \rightarrow m$ in **Int** the following diagram commutes:

$$\begin{tikzcd}
O_1(n) \arrow[r, "\gamma_n"] \arrow[d, "O_1(f)"'] & O_2(n) \arrow[d, "O_2(f)"] \\
O_1(m) \arrow[r, "\gamma_m"'] & O_2(m)
\end{tikzcd}$$

An arrow is defined in the design language as a homomorphism implemented as a pair of functions (f, g) where f maps sets of states and g maps pairs of sets of messages. Behaviour transformation is performed by applying an arrow to a behaviour to produce a new behaviour. The application of (f, g) to the behaviour function M_1 produces a behaviour function M_2 which makes the following diagram commute for any α, τ and v:

$$\begin{tikzcd}
M_1(\alpha)(\tau)(v) \arrow[r, "(f,g)"] \arrow[d, "{[\![\cdot]\!]}"'] & M_2(\alpha)(\tau)(v) \arrow[d, "{[\![\cdot]\!]}"] \\
G_1 \arrow[r, "(f,g)"'] & G_2
\end{tikzcd}$$

Application of arrows to produce class behaviours and families of behaviours follows from an extension of the above definition. Given a homomorphism between graphs we can uniquely extend this to a homomorphism between sets of graphs (class behaviours) and then sets of sets of graphs (families of behaviours).

For each behaviour function there is exactly one possible morphism to the terminal behaviour function *empty*: *term* $= (\mathbf{K}(\emptyset), \mathbf{K}(\emptyset, \emptyset))$

5 Systems

Object-oriented systems are compositional (feature 8). Composition can occur in order to extend the possible system behaviour and also occur in order to restrict possible system behaviour. Extension can occur when new methods or attributes are added to a class. Extension also occurs when partial behaviours are combined to produce a "larger" behaviour (feature 9). Restriction occurs when behaviours are composed and required to behave consistently (feature 11).

This section shows how systems are constructed from sub-systems. The system building operations are defined in terms of standard constructs from category theory. The design language is extended with system building operators.

5.1 Products

Given objects A and B, a product is an object $A \times B$ together with two arrows $\pi_1 : A \times B \to A$ and $\pi_2 : A \times B \to B$ such that for any object C with morphisms $f : C \to A$ and $g : C \to B$ there is a unique arrow $u : C \to A \times B$ such that the following diagram commutes:

$$\begin{tikzcd}
& C \arrow[ddl, "f"'] \arrow[d, "u"] \arrow[ddr, "g"] & \\
& A \times B \arrow[dl, "\pi_1"] \arrow[dr, "\pi_2"'] & \\
A & & B
\end{tikzcd}$$

Following the standard product construction for two graphs (see [Bar90]), products in **Calc** are constructed as follows. Given two calculation graphs G_1 and G_2, a product $G_1 \times G_2$ is a calculation graph whose nodes and edges are labelled with pairs of labels from G_1 and G_2 respectively. For every node $n \in G_1$ and node $n_2 \in G_2$ there is a node $n \in G_1 \times G_2$ such that $\mathit{label}(n) = (\mathit{label}(n_1), \mathit{label}(n_2))$. For every edge $e_1 \in G_1$ and edge $e_2 \in G_2$ there is an edge $e \in G_1 \times G_2$ such that $\mathit{label}(e) = (\mathit{label}(e_1), \mathit{label}(e_2))$. The source and target nodes of e correspond to the pairing of the corresponding source and target nodes of e_1 and e_2. The projection arrows are graph homomorphisms which project onto the first and second co-ordinates of the labels respectively.

This leads us to define product of two behaviours O_1 and O_2 for all n as follows:

$$(O_1 \times O_2)(n) = O_1(n) \times O_2(n)$$

Unfortunately, this can produce inconsistent system states. A product state could be formed by composing two views of the same object:

$$(\{<\alpha, \tau, \{x \mapsto 1\}>\}, \{<\alpha, \tau, \{x \mapsto 2\}>\})$$

in which the object τ associates the attribute x simultaneously with the values 1 and 2. Furthermore, a tree structure is imposed on system states which is undesirable since system composition becomes non-associative: $(O_1 \times O_2) \times O_3 \not\equiv O_1 \times (O_2 \times O_3)$.

We propose a structure for system composition which ensures consistent states. When two systems are composed, the resulting behaviour has the largest consistent state. Let *merge* be the calculation graph homomorphism $(\mathit{merge}_1, \mathit{merge}_2)$ which is defined as follows:

$$\mathit{merge}_1(\Sigma_1, \Sigma_2) = \Sigma_1 \sqcap \Sigma_2$$

$$\mathit{merge}_2((I_1, O_1), (I_2, O_1)) = (I_1 \cup I_2, O_1 \cup O_2)$$

The composition of behaviour functions is defined by a design language operator $\times$ as follows:

$$[\![M_1 \times M_2]\!] = \mathit{merge}([\![M_1]\!] \times [\![M_2]\!])$$

The combination of behaviour composition and consistency allows a system development technique to be compositional. For example we may define two views of the same object. Each view is specific with respect to a different aspect of the object's behaviour, otherwise each view leaves the object free to perform any behaviour. The composition of the two views, as defined by a product, allows each view to constrain the free behaviour of the other.

Object-oriented design notations define partial object behaviours (feature 9). A partial behaviour describes the state changes and messages which occur when a message is sent to an object. The description is *partial* because it does not apply to all possible system states. A total description is constructed by composing a sufficiently complete collection of partial descriptions.

The product operator $\times$ combines partial behaviours using the morphism *merge*. For example, a two-dimensional point object can be defined as the composition of independent behaviours describing the x-view and y-view respectively.

5.2 Coproducts

Given two objects A and B, a coproduct is an object $A + B$ together with two object morphisms $\iota_1 : A \to A + B$ and $\iota_2 : A \to A + B$ such that for any object C and arrows $f : A \to C$ and $g : B \to C$ there is a unique arrow $u : A + B \to C$ such that the following diagram commutes:

$$\begin{tikzcd}
A \arrow[dr, "\iota_1"] \arrow[ddr, "f"'] & & B \arrow[dl, "\iota_2"'] \arrow[ddl, "g"] \\
& A + B \arrow[d, "u"] & \\
& C &
\end{tikzcd}$$

A coproduct of two calculation graphs G_1 and G_2 in **Calc** is a calculation graph $G_1 + G_2$ which contains all of the nodes and edges of G_1 and G_2. The nodes and edges are labelled in order to record whether they originated from G_1 or G_2. This allows the arrow u to test the origin of a node or edge in order to apply the appropriate arrow f or g. Behaviour coproducts are defined for all n as follows:

$$(O_1 + O_2)(n) = O_1(n) + O_2(n)$$

Coproducts ensure that the calculations performed by the component objects are represented separately by labelling the states and edges. In an object design we will often not be interested in the origin of a transition and can therefore lose the labels which encode this information. The result is a restriction on the coproduct which produces an object for which we cannot necessarily guarantee a unique arrow u.

The implication of a non-unique arrow u is that the design is non-deterministic which is a required feature (feature 6) of object-oriented design systems. This leads to an operator $+$ which is used to merge two objects to produce a (possibly) non-deterministic composite object.

Suppose that $G_1 + G_2$ is a coproduct of G_1 and G_2 such that nodes and edges from G_1 and G_2 are labelled *pink* and *blue* respectively. A de-labelling operation can be expressed as a morphism *delabel* defined as a forgetful graph homomorphism $(\mathit{strip}, \mathit{strip})$ which is defined as $\mathit{strip}(\mathit{pink}(v)) = v$ and $\mathit{strip}(\mathit{blue}(v)) = v$. Given object calculation functions M_1 and M_2 then we define the operator $+$ as follows:

$$[\![M_1 + M_2]\!] = \mathit{delabel}([\![M_1]\!] + [\![M_2]\!])$$

5.3 Equalizers

Object-oriented design notations often allow the engineer to produce different views of the same component using different modelling notations. The behaviour of the resulting system is constructed as the largest set of behaviours consistent with all possible views.

Consistency between views is achieved using *equalizers*. Let A and B be two objects and let $f : A \to B$ and $g : A \to B$ be two object morphisms. An equalizer of the arrows is an object E together with a morphism $e : E \to A$ such that $f \circ e = g \circ e$ and for any object E' and arrow $h : E' \to A$ such that $f \circ h = g \circ h$ there is a unique arrow u such that the following diagram commutes:

$$\begin{tikzcd}
E \arrow[r, "e"] & A \arrow[r, "f", "g"'] & B \\
E' \arrow[u, "u"] \arrow[ur, "h"'] & &
\end{tikzcd}$$

In **Calc** an equalizer between two graph homomorphisms $f : G_1 \to G_2$ and $g : G_1 \to G_2$ is a graph G and a homomorphism $e : G \to G_1$ such that e picks out the largest sub-graph of G_1 which produces the same image under f

and g. If the equalizer is defined in terms of the largest sub-graph then the homomorphism u is unique up to isomorphism.

The design language provides a builtin operator *eq* which constructs equalizers. If M_1 is a function implementing an object and f, g are pairs of functions from M_1 to another object M_2 then $(N, e) = eq(M_1)(f)(g)$ such that $[\![N]\!]$ and e is an equalizer for $[\![M_1]\!]$, $[\![M_2]\!]$, f and g.

Equalizers can be used to force consistency. System design often leaves the behaviour of a local component under-defined by permitting a variety of computations. When components are composed, the possible computations are reduced by requiring pair-wise consistency. The result is still under-defined, however certain illegal computations are ruled out.

An example of consistent behaviour is constructed using a categorical construction called a *pullback*. Consider three objects O, O_1 and O_2:

$$\begin{array}{ccc} O_1 & \xrightarrow{f} & O \\ & & \uparrow g \\ & & O_2 \end{array}$$

where the behaviour of O_1 and O_2 can be understood in terms of O. A pullback is an object O' with two arrows k and l such that the following diagram commutes:

$$\begin{array}{ccc} O_1 & \xrightarrow{f} & O \\ k \uparrow & & \uparrow g \\ O' & \xrightarrow[l]{} & O_2 \end{array}$$

and for any other object O'' with arrows $k' : O'' \to O_1$ and $l' : O'' \to O_2$ for which the diagram commutes there exists a unique arrow $u : O'' \to O'$.

The pullback object O' is the largest object behaviour which makes the arrows $f \circ k$ and $g \circ l$ equal. Therefore pullback objects can be used to construct the smallest composite behaviour from O_1 and O_2 which makes the behavioural views f and g equivalent. For example, this may arise when requiring that an argument value in two different method calls must be the same. This occurs in an object-oriented design language such as UML where a variable occurs on a design more than once.

6 System Behaviour

A system is composed of multiple objects. Each object is designed by providing multiple views of its behaviour. The different views of an object's behaviour are not independent. The dependencies are constraints which hold between the different views and which serve to rule out possible combined behaviours. The constraints can be viewed as simultaneous equations affecting the overall behaviour of the system. A solution to the equations is a system behaviour (feature 12). This section describes how constraints on system components are expressed as diagrams in **Obj**, defines system behaviour as a limit on a system diagram and describes an algorithm for constructing limits in the design language.

6.1 System Diagrams

The behaviour of a system is described by an object in **Obj**. Objects in **Obj** are non-deterministic partial views of object systems. Consistent with current object-oriented development processes, we claim that the meaning of an object-oriented design is the composition of the meanings of the constituent design models.

Systems consist of a collection of co-ordinated objects. An overall system transition will typically involve a collection of controlled individual object transitions. Such co-ordination implies a constraint on the free behaviour of the individual objects. The constraints can be expressed as behaviour morphisms which express how one behaviour can be interpreted in terms of another.

A single behaviour may be the source of multiple morphisms. This may be used to "glue" together individual behaviours and also to require them to interact. A single behaviour may be the target of multiple morphisms. This may be used to require individual morphisms to behave consistently under certain circumstances.

A system is expressed as a collection of **Obj** objects and morphisms between them. A system *diagram* is a graph whose nodes are labelled with behaviours and whose edges are labelled with behaviour morphisms.

6.2 Limits of Diagrams

In order to solve the equations we use the categorical construct of limits which has been proposed by Goguen in [Gog90] as a means for expressing the behaviour of a system. In general, a diagram Δ is a graph with a mapping from the nodes of Δ to objects and a mapping from the edges of Δ to morphisms between the source and target objects. Let Δ_n represent an object N in the diagram Δ and Δ_e represent a morphism. A *cone* on a diagram Δ is an object A (which is not necessarily in the diagram) together with, for each object Δ_n a morphism $\gamma_n : A \to \Delta_n$ such that for all edges $e : n \to m$ in Δ the following diagram commutes:

$$\begin{tikzcd} & A \arrow[dl, "\gamma_n"'] \arrow[dr, "\gamma_m"] & \\ \Delta_n \arrow[rr, "\Delta_e"'] & & \Delta_m \end{tikzcd}$$

A cone on a system expressed as a diagram is a composite behaviour which is consistent with the behaviours on the diagram. Notice however, the there is no condition on *which* particular behaviour to pick for the cone. There are many possible choices for cones including the behaviour with no states or transitions (this is an *initial object* which can always be mapped to another behaviour using the empty morphism).

A *cone arrow* on a diagram Δ from cone $\gamma_n : A \to \Delta_n$ on Δ to cone $\gamma'_n : A' \to \Delta_n$ is an object morphism $\gamma : A' \to A$ such that for all nodes n in Δ the triangle below commutes:

$$\begin{tikzcd} A \arrow[dr, "\gamma_n"'] & & A' \arrow[ll, "\gamma"'] \arrow[dl, "\gamma'_n"] \\ & \Delta_n & \end{tikzcd}$$

Cone arrows allow us to place restrictions on cones. For example if there is a cone arrow $\gamma : A' \to A$ we know that the behaviour described by A' is more restricted than that described by A.

A *limit* on a diagram Δ is a cone A, on Δ such that for any other cone A' on Δ there is a unique cone arrow $u : A' \to A$. The definition of a limit captures the behaviour of a system because it must contain all of the behaviours on the diagram and must obey all of the constraints on the diagram.

6.3 Constructing Limits

A standard result of category theory is that any category having a terminal object, binary products and equalizers of pairs of arrows has all finite limits [Ryd88]. This result allows the construction of a limit on a diagram providing that these simple properties hold in the appropriate category. An algorithm for constructing limits of object-oriented design diagrams is described in [Cla99b]. The algorithm constructs limits incrementally using a terminal object, products and equalizers.

7 An Example Design

This section shows how the design language can be used to give a meaning to a small object-oriented design expressed as a static class diagram and a collection of dynamic object interaction diagrams. We present a requirement for the example application, produce a collection of object-oriented models and then construct the corresponding behaviours.

7.1 Software Requirements

Software to control a library is required. The library has readers who may borrow copies of books. At any given time each reader has a number of books on loan. New readers may join the library at any time. The library has a number of copies of books. Each book has a unique title. A copy is either on the shelf in the library or is being borrowed by a reader. Libraries operate a shares readership policy whereby joining one library permits readers to borrow books at all participating libraries.

7.2 Initial Behaviour

An initial analysis of the requirements leads us to design a simple structure for a system. A library system consists of a single object with a state (R, B) consisting of readers R and books B. Each reader is a pair (n, C) where n is a name and C is a set of borrowed copies. Each book is a pair (n, i) where n is a name and i is the number of shelved copies.

In the initial system design we will treat R and B as lookup tables. Let T be a table implemented as a set of pairs containing keys and values. The keys of a table are produced by the operator *dom*. Table lookup $T \bullet k$ is defined and has the value v when there exists exactly one pair $(k, v) \in T$. Table extension is defined as: $T[k \mapsto v] \equiv T \cup \{(k, v)\}$. The following sugar for adding and removing elements in a table will be useful:

$$T[k \oplus v] \equiv \begin{cases} T[k \mapsto T \bullet k \cup \{v\}] & \text{when } \mathit{isSet}(T \bullet k) \\ T[k \mapsto T \bullet k + v] & \text{when } \mathit{isInt}(T \bullet k) \end{cases}$$

$$T[k \ominus v] \equiv \begin{cases} T[k \mapsto T \bullet k - \{v\}] & \text{when } \mathit{isSet}(T \bullet k) \\ T[k \mapsto T \bullet k - v] & \text{when } \mathit{isInt}(T \bullet k) \end{cases}$$

Initial system behaviour can be decomposed into the sucess and failure modes. The design operator $+$ allows us to define these modes separately and then combine them.

$$
\begin{array}{l}
\textbf{letrec } \mathit{libOk}(\alpha)(\tau)(R, B)(m) = \\
\quad \textbf{case } m \textbf{ of} \\
\quad\quad \mathit{addReader}(n) \rightarrow (\mathit{libOk}(\alpha)(\tau)(R[n \mapsto \emptyset], B), \emptyset) \textbf{ when } n \notin \mathit{dom}(R) \\
\quad\quad \mathit{addBook}(n) \rightarrow (\mathit{libOk}(\alpha)(\tau)(R, B[n \mapsto 0]), \emptyset) \textbf{ when } n \notin \mathit{dom}(B) \\
\quad\quad \mathit{addCopy}(n) \rightarrow (\mathit{libOk}(\alpha)(\tau)(R, B[n \oplus 1]), \emptyset) \textbf{ when } n \in \mathit{dom}(B) \\
\quad\quad \mathit{borrow}(n_1, n_2) \rightarrow \\
\quad\quad\quad (\mathit{libOk}(\alpha)(\tau)(R[n_1 \oplus n_2], B[n_2 \ominus 1]), \emptyset) \\
\quad\quad\quad\quad \textbf{when } n_1 \in \mathit{dom}(R) \,\&\, n_2 \in \mathit{dom}(B) \\
\quad\quad \mathit{return}(n_1, n_2) \rightarrow \\
\quad\quad\quad (\mathit{libOk}(\alpha)(\tau)(R[n_1 \ominus n_2], B[n_2 \oplus 1]), \emptyset) \\
\quad\quad\quad\quad \textbf{when } n_1 \in \mathit{dom}(R) \,\&\, n_2 \in \mathit{dom}(B) \\
\quad\quad \textbf{else } (\mathit{libOk}(\alpha)(\tau)(R, B), \emptyset) \\
\quad \textbf{end}
\end{array}
$$

Each of the state transitions in *libOk* is guarded by a condition. If any of these conditions do not hold then the library should not change state. In practice, such error transitions would send a message back to the source of the message.

$$
\begin{array}{l}
\textbf{letrec } \mathit{libFail}(\alpha)(\tau)(R,B)(m) = \\
\quad \textbf{case } m \textbf{ of} \\
\quad\quad \mathit{addReader}(n) \rightarrow (\mathit{libFail}(\alpha)(\tau)(R,B),\emptyset) \textbf{ when } n \in \mathit{dom}(R) \\
\quad\quad \mathit{addBook}(n) \rightarrow (\mathit{libFail}(\alpha)(\tau)(R,B),\emptyset) \textbf{ when } n \in \mathit{dom}(B) \\
\quad\quad \mathit{addCopy}(n) \rightarrow (\mathit{libFail}(\alpha)(\tau)(R,B),\emptyset) \textbf{ when } n \notin \mathit{dom}(B) \\
\quad\quad \mathit{borrow}(n_1,n_2) \rightarrow \\
\quad\quad\quad (\mathit{libFail}(\alpha)(\tau)(R,B),\emptyset) \\
\quad\quad\quad\quad \textbf{when } n_1 \notin \mathit{dom}(R) \mid n_2 \notin \mathit{dom}(B) \\
\quad\quad \mathit{return}(n_1,n_2) \rightarrow \\
\quad\quad\quad (\mathit{libFail}(\alpha)(\tau)(R,B),\emptyset) \\
\quad\quad\quad\quad \textbf{when } n_1 \notin \mathit{dom}(R) \mid n_2 \notin \mathit{dom}(B) \\
\quad\quad \textbf{else } (\mathit{libFail}(\alpha)(\tau)(R,B),\emptyset) \\
\quad \textbf{end}
\end{array}
$$

The initial design is then constructed by composing the two class alternative class behaviours: $\mathit{library} = \mathit{libOk} + \mathit{libFail}$

7.3 Refinement Of Initial Behaviour

Although the initial design specifies the correct system behaviour it fails to qualify as an object-oriented design. The reason for this is that the system state is not structured as a collection of communicating objects.

A library object contains a set of books, each book is a pair (n,i). Pairs are structured data which can only exist in an object-oriented program as an object. Furthermore, the initial design does not uphold the principle of encapsulation. The representation of books (as pairs) is known to the library object.

The initial design is refined to become more object-oriented. The state must be changed to implement all structured data components as objects. The behaviour must be changed correspondingly to uphold the principle of encapsulation.

Given a state (R,B) in the source behaviour, the refinement acts as identity on R and transforms $B = \{n_1 \mapsto i_1,\ldots,n_k \mapsto i_k\}$ into a set of object identifiers $\{\tau_1,\ldots,\tau_k\}$ and introduces new objects $\tau_1 \mapsto (n_1,i_1),\ldots,\tau_k \mapsto (n_k,i_k)$ to the system state. A book behaviour is as follows:

$$
\begin{array}{l}
\textbf{letrec } \mathit{book}(\alpha)(\tau)(n,i)(m) = \\
\quad \textbf{case } m \textbf{ of} \\
\quad\quad <\tau',\tau,\mathit{getName}> \rightarrow (\mathit{book}(\alpha)(\tau)(n,i),\{<\tau,\tau',n>\}) \\
\quad\quad \mathit{borrow} \rightarrow (\mathit{book}(\alpha)(\tau)(n,i-1),\emptyset) \textbf{ when } i > 0 \\
\quad\quad \mathit{addCopy} \rightarrow (\mathit{book}(\alpha)(\tau)(n,i+1),\emptyset) \\
\quad\quad \textbf{else } (\mathit{book}(\alpha)(\tau)(n,i),\emptyset) \\
\quad \textbf{end}
\end{array}
$$

The successful library behaviour is modified to take account of book objects. The transitions for *libOk* are redefined as follows. There is no change to the *addReader* transition. The initial design uses set membership to test for the existence of a book. This must now be implemented as a private method of the library:

$$
\begin{array}{l}
<\tau', \tau, \mathit{findBook}(\emptyset, n)> \rightarrow (\mathit{libOk}(\alpha)(\tau)(R, B), \{<\tau, \tau', \mathit{noBook}>\}) \\
<\tau', \tau, \mathit{findBook}(\{o\} \cup S, n_1)> \rightarrow \\
\quad \textbf{let } n_2 \leftarrow <\tau, o, \mathit{getName}> \\
\quad \textbf{in if } n_1 = n_2 \\
\quad\quad \textbf{then } (\mathit{libOk}(\alpha)(\tau)(R, B), \{<\tau, \tau', \mathit{book}(o)>\}) \\
\quad\quad \textbf{else } (\mathit{libOk}(\alpha)(\tau)(R, B), \{<\tau', \tau, \mathit{findBook}(S, n_1)>\})
\end{array}
$$

When a library recieves an *addBook* message with a name n which does not already exist then a new book object is created. We assume that τ'' is a new object identifier and that β is the type tag for books:

$$
\begin{array}{l}
\mathit{addBook}(n) \rightarrow \\
\quad \textbf{let } \mathit{noBook} \leftarrow <\tau, \tau, \mathit{findBook}(n)> \\
\quad \textbf{in } (\mathit{libOk}(\alpha)(\tau)(R, B \cup \{\tau''\}) \times \mathit{book}(\beta)(\tau'')(n, \emptyset), \emptyset)
\end{array}
$$

A copy of a book may be added when there is a book currently registered in the library. The book is found using *findBook* and then is sent a message *addCopy*:

$$
\begin{array}{l}
\mathit{addCopy}(n) \rightarrow \\
\quad \textbf{let } \mathit{book}(b) \leftarrow <\tau, \tau, \mathit{findBook}(n)> \\
\quad \textbf{in } (\mathit{libOk}(\alpha)(\tau)(R, B), \{<\tau, b, \mathit{addCopy}>\})
\end{array}
$$

A copy is borrowed by finding the book using *findBook* and then modifying both the reader and the book. The reader can be modified directly because the representation is known to the library object. The book is modified by sending it a message *borrow*.

$$
\begin{array}{l}
\mathit{borrow}(n_1, n_2) \rightarrow \\
\quad \textbf{let } \mathit{book}(b) \leftarrow <\tau, \tau, \mathit{findBook}(n_2)> \\
\quad \textbf{in } (\mathit{libOk}(\alpha)(\tau)(R[n_1 \oplus n_2], B), \{<\tau, b, \mathit{borrow}>\}) \\
\quad\quad \textbf{when } n_2 \in \mathit{dom}(R)
\end{array}
$$

Correspondingly, a copy is returned by directly removing the copy from the reader and sending a *return* message to the book.

$$
\begin{array}{l}
\mathit{return}(n_1, n_2) \rightarrow \\
\quad \textbf{let } \mathit{book}(b) \leftarrow <\tau, \tau, \mathit{findBook}(n_2)> \\
\quad \textbf{in } (\mathit{libOk}(\alpha)(\tau)(R[n_1 \ominus n_2], B), \{<\tau, b, \mathit{addCopy}>\}) \\
\quad\quad \textbf{when } n_2 \in \mathit{dom}(R)
\end{array}
$$

This completes the refinement of the behaviour *libOk*. The behaviour *libFail* is refined using the same principles using the outcomes from set membership of R and sending *findBook* messages.

7.4 Correctness Proof

In order to show that the refinement is correct we must show that it is sound and complete as described in section 3.5. In particular we must show that the diagram 1 holds.

The following source state is used: $\{\tau \mapsto (R, B)\}$ where R is a set of readers and B is the set $\{n_1 \mapsto i_1, \ldots, n_k \mapsto i_k\}$. The corresponding target state is $\{\tau \mapsto (R, T)\} \cup O$ where T is the set of object identifiers $\{\tau_1, \ldots, \tau_k\}$ and O is the state $\{\tau_1 \mapsto (n_1, i_1), \ldots, \tau_k \mapsto (n_k, i_k)\}$.

To prove soundness and completeness with respect to diagram 1 we must establish that given the source state, for every arrow g there is an arrow f which makes the diagram commute and vice versa. This is done for each system function independently. The format of each proof is similar; the rest of this section states the conditions for each function and proves the first two sound and complete.

In order that the refinement of *addReader* is sound and complete we must have the following:

$$
\begin{tikzcd}
\{\tau \mapsto (R, B)\} \arrow[r, "addReader(n)"] \arrow[d] & \{\tau \mapsto (R[n \mapsto \emptyset], B)\} \\
\{\tau \mapsto (R, T)\} \cup O \arrow[r, "addReader(n)"'] & \{\tau \mapsto (R[n \mapsto \emptyset], T)\} \cup O \arrow[u]
\end{tikzcd}
\tag{2}
$$

The proof of 2 follows from the definition of the source and target behaviours of the refinement. The refinement of *addBook* is sound and complete when:

$$
\begin{tikzcd}
\{\tau \mapsto (R, B)\} \arrow[r, "addBook(n)"] \arrow[d] & \{\tau \mapsto (R, B[n \mapsto 0])\} \\
\{\tau \mapsto (R, T)\} \cup O \arrow[r, "c \circ addBook(n)"'] & \begin{array}{c}\{\tau \mapsto (R, T \cup \tau'')\} \cup \\ O[\tau'' \mapsto (n, 0)]\end{array} \arrow[u]
\end{tikzcd}
\tag{3}
$$

The proof of 3 is by induction on the size of the set B and the length of the computation c. There are two cases to consider:

1. When $B = \emptyset$ the computation c is $[noBook]$ and therefore the proposition holds by definition.

2. When B is non-empty then we assume that $B = \{n' \mapsto i'\} \cup B'$ for some $n' \neq n$ and that the proposition holds for B, therefore there is some computation c' such that there is a computation $g' : c' \circ addBook(n)$ which is both sound and complete with respect to the state $\{\tau \mapsto (R, B')\}$. By the definition of both behaviours there is a computation:

$$c'' = [findBook(\{\tau'\}, n), getName, n', noBook]$$

such that $c'' \circ c' \circ \mathit{addBook}(n)$ exists.

Therefore we conclude that the refinement of *addBook* is sound and complete. The conditions for the soundness and completeness of refinements for system function *addCopy* is expressed as diagram 4.

$$
\begin{tikzcd}
\{\tau \mapsto (R, B[n_j \mapsto i_j])\} \arrow[r, "addCopy(n_j)"] \arrow[d] & \{\tau \mapsto (R, B[n_j \oplus 1])\} \\
\begin{array}{c}\{\tau \mapsto (R,T)\} \cup \\ O[\tau_j \mapsto (n_j, i_j)]\end{array} \arrow[r, "c \circ addCopy(n_j)"'] & \begin{array}{c}\{\tau \mapsto (R,T)\} \cup \\ O[\tau_j \mapsto (n_j, i_j + 1)]\end{array} \arrow[u]
\end{tikzcd}
\tag{4}
$$

The condition for *borrow* is expressed in diagram 5.

$$
\begin{tikzcd}
\{\tau \mapsto (R, B[n_2 \oplus i])\} \arrow[r, "borrow(n_1,n_2)"] \arrow[d] & \{\tau \mapsto (R[n_1 \oplus n_2, B[n_2 \ominus 1])\} \\
\begin{array}{c}\{\tau \mapsto (R,T)\} \cup \\ O[\tau_2 \mapsto (n_2, i)]\end{array} \arrow[r, "c \circ borrow(n_1,n_2)"'] & \begin{array}{c}\{\tau \mapsto (R[n_1 \oplus n_2, T)\} \cup \\ O[\tau_2 \mapsto (n_2, i - 1)]\end{array} \arrow[u]
\end{tikzcd}
\tag{5}
$$

The proof of 4 and 5 are by induction on the size of the set B. The condition and proof for *return* are similar to those for *borrow*.

7.5 Completing the Refinement

The behaviour defined in section 7.3 is a refinement of the initial behaviour, but is not an object-oriented behaviour. The reader component R of the system state must be refined to a set of objects. This section outlines the refinement step. A reader behaviour is defined as follows:

$$
\begin{array}{l}
\textbf{letrec } \mathit{reader}(\alpha)(\tau)(n, C)(m) = \\
\quad \textbf{case } m \textbf{ of} \\
\quad\quad <\tau', \tau, \mathit{getName}> \to (\mathit{reader}(\alpha)(\tau)(n, C), \{<\tau, \tau', n>\}) \\
\quad\quad \mathit{borrow}(c) \to (\mathit{reader}(\alpha)(\tau)(n, C \cup \{c\}), \emptyset) \\
\quad\quad \mathit{return}(c) \to (\mathit{reader}(\alpha)(\tau)(n, C - \{c\}), \emptyset) \\
\quad\quad \textbf{else } (\mathit{reader}(\alpha)(\tau)(n, C), \emptyset) \\
\quad \textbf{end}
\end{array}
$$

The *libOk* behaviour is refined as follows. The transitions for *findReader*, *addBook* and *addCopy* are unchanged. Since readers are represented as objects, a new message handler *findReader* is added which behaves like *findBook*. The message handler for *borrow* is refined to use both *findBook* and *findReader*:

$$\begin{array}{l} \mathit{borrow}(n_1, n_2) \rightarrow \\ \quad \textbf{let } \mathit{book}(b) \leftarrow <\tau, \tau, \mathit{findBook}(n_2)> \\ \quad \textbf{in let } \mathit{reader}(r) \leftarrow <\tau, \tau, \mathit{findReader}(n_1)> \\ \quad\quad \textbf{in } (\mathit{libOk}(\alpha)(\tau)(R, B), \{<\tau, b, \mathit{borrow}>, <\tau, r, \mathit{borrow}(n_2)>\}) \end{array}$$

The message handler for *borrow* is modified as follows:

$$\begin{array}{l} \mathit{return}(n_1, n_2) \rightarrow \\ \quad \textbf{let } \mathit{reader}(r) \leftarrow <\tau, \tau, \mathit{findReader}(n_1)> \\ \quad \textbf{in let } \mathit{book}(b) \leftarrow <\tau, \tau, \mathit{findBook}(n_2)> \\ \quad\quad \textbf{in } (\mathit{libOk}(\alpha)(\tau)(R, B), \{<\tau, b, \mathit{return}>, <\tau, r, \mathit{return}(n_2)>\}) \end{array}$$

7.6 Behavioural and Structural Analysis

The refinement described in sections 7.3 and 7.5 produce an object-oriented behaviour description. Such a behaviour is analysed for common features which can be factored out. Structural and behavioural properties factored in this way indicate that inheritance may be used when the behaviour is implemented using a concrete programming language.

Consider the behaviours *book* and *reader*. Both provide a state component n which is used to index into collections of behavioural instances using the message *getName*. This indicates that there is a common behaviour *named* and projection morphisms:

$$\begin{tikzcd} & \mathit{named} & \\ \mathit{book} \arrow[ur] & & \mathit{reader} \arrow[ul] \end{tikzcd}$$

In an implementation *named* will occur as a super-class of both *book* and *reader*. The definition of *named* is as follows:

$$\begin{array}{l} \textbf{letrec } \mathit{named}(\alpha)(\tau)(n)(m) = \\ \quad \textbf{case } m \textbf{ of} \\ \quad\quad <\tau', \tau, \mathit{getName}> \rightarrow (\mathit{named}(\alpha)(\tau)(n), \{<\tau, \tau', n>\}) \\ \quad\quad \textbf{else } (\mathit{named}(\alpha)(\tau)(n), \emptyset) \\ \quad \textbf{end} \end{array}$$

Analysis of *libOk* indicates that the message handlers for *findBook* and *findReader* are very similar. Subsequent definition of the common behaviour *named* suggests that both *findBook* and *findReader* can be replaced by a single behaviour defined entirely in terms of *named*.

Consider a behaviour functor F_1 which acts on system states by projecting all book objects to equivalent named objects by forgetting the copy count. F_1 acts as identity on all arrows except that $\mathit{findBook}(O, n)$ is replaced by $\mathit{find}(O, n)$, $\mathit{book}(b)$ is replaced by $\mathit{found}(b)$ and *noBook* is replaced by *notFound*.

In order for F_1 to be valid, it must be sound and complete with respect to indexing into collections of books. Therefore, for any system state Σ, the following diagram must commute:

$$\begin{tikzcd} \Sigma \arrow[r, "{[\mathit{findBook}(O,n)] \mathbin{+\!\!+} c}"] \arrow[d] & \Sigma \arrow[d] \\ F_1(\Sigma) \arrow[r, "{[\mathit{find}(O,n)] \mathbin{+\!\!+} F_1(c)}"'] & F_1(\Sigma) \end{tikzcd}$$

Similarly, a behaviour functor F_2 is defined to project states and calculations involving indexing readers. Both sets of indexing calculations are produced by the following behaviour:

$$\begin{array}{l} <\tau', \tau, \mathit{find}(\emptyset, n)> \to (\mathit{libOk}(\alpha)(\tau)(R,B), \{<\tau, \tau', \mathit{notFound}>\}) \\ <\tau', \tau, \mathit{find}(\{o\} \cup S, n_1)> \to \\ \quad \mathbf{let}\ n_2 \leftarrow <\tau, o, \mathit{getName}> \\ \quad \mathbf{in}\ \mathbf{if}\ n_1 = n_2 \\ \qquad \mathbf{then}\ (\mathit{libOk}(\alpha)(\tau)(R,B), \{<\tau, \tau', \mathit{found}(o)>\}) \\ \qquad \mathbf{else}\ (\mathit{libOk}(\alpha)(\tau)(R,B), \{<\tau', \tau, \mathit{find}(S, n_1)>\}) \end{array}$$

The shared readership policy is expressed as behaviour morphisms which require all libraries to share the same R component. Consider the behaviour *cell* which contains a single value and responds to messages which set and access the value. Two libraries can be projected onto cell:

$$\begin{tikzcd} \mathit{lib}_1 \arrow[dr, "\gamma_1"'] & & \mathit{lib}_2 \arrow[dl, "\gamma_2"] \\ & \mathit{cell} & \end{tikzcd}$$

The object morphisms γ_1 and γ_2 project both libraries onto a single cell whose object identity is the same in each case and whose value is the readers of both lib_1 and lib_2. Library messages which access and update readers are mapped by γ_1 and γ_2 to corresponding access and update messages from *cell*. All other library messages are ignored by *cell* since they do not change the value of the readers.

System behaviour is constructed by taking a limit on a diagram containing all of the component behaviours and constraints between them. The constraint expressed using γ_1 and γ_2 above requires that the readers in lib_1 and lib_2 are always the same. This is shown as S on the pullback diagram:

$$\begin{tikzcd} & S \arrow[dl] \arrow[dr] & \\ lib_1 \arrow[dr, "\gamma_1"'] & & lib_2 \arrow[dl, "\gamma_2"] \\ & cell & \end{tikzcd}$$

There are a number of implementation choices for the shared readership policy whose behaviour is defined by S. If the programming language supports shared data between class instances (such as `static` in Java) then the R component of a library class may be shared. Alternatively, a library may be able to access all other instances of the library class and therefore send appropriate messages when its readers are updated. Finally, a controller object may be used to contain all libraries and federate messages between them. The controller would be responsible for ensuring that all libraries have consistent readers.

7.7 Implementation

Once the library system has been specified, refined and analysed it is possible to implement the behaviour as an object-oriented program. Any object-oriented language is suitable. Each independent behaviour is defined as a class. The state components of the behaviour are defined as fields and the message handlers are defined as methods. Any common behaviour is defined using inheritance. The implementation of the library system is defined in appendix A. The main features of the implementation are:

- The class `Named` defines the common behaviour for readers and books.
- The attribute `readers` in `Library` is declared `static` so that libraries implement the shared readership policy.
- The class `Library` defines a method `find` which is used to index both readers and books.

8 Conclusion

8.1 Review

The aims of this work are to define a semantic framework which is suitable for rigorous object-oriented development. In order to do this, we have taken a behavioural view of object-oriented systems and defined object behaviour as a graph of states and transitions arising from message passing. Object behaviours have been defined using standard constructions in category theory. Systems behaviours arise from the solution to a collection of simultaneous equations which constrain a collection of freely defined object behaviours. We have shown that such constraints can be expressed as behaviour morphisms on a diagram whose solution is constructed as a limit. Behaviour equivalence and refinement has been defined using isomorphisms and adjoint functors respectively. Refinement is a relation between a source and target behaviours whereby both have the same functionality and the target is closer to an implementation in terms of object-oriented design principles.

A notation for expressing object designs has been proposed as a λ-notation extended with built-in operators for system construction. The semantics of the notation is given by constructions in category theory. The notation has been used to express a small but representative object-oriented design.

The design notation is highly expressive and facilitates a variety of approaches to system design. In particular it allows systems to be designed using modular units and then composed using the operators $\times$ and $+$. The system constraints are enforced using an operator *eq*.

Section 2 lists a collection of features which are essential to object-oriented design. Current graphical object-oriented design notations offer these features which are the motivation for the behavioural model of object systems defined in section 3. This leads to the claim that the semantics of current object-oriented design notations are represented by the model and therefore the design notation which is defined in this paper.

8.2 Analysis

The model which is proposed for object-oriented systems is universal and representative of other approaches to the semantic definition of dynamic systems. In particular, concurrent object systems are often expressed as labelled transition graphs. The use of such a semantic model to express the semantics of current graphical design notations is new and offers potentially fruitful feedback for the invention and modification of such notations.

There is always a tension between the simplicity of an informally defined notation (as represented by the class of graphical design notations) and the notational overload of a rigorously defined notation (as represented by the design notation used in this paper). The use of λ-notation can offer some help in this regard since it is higher-order (and can therefore encode very high-level control abstractions) and has a distinguished history of being sweetened through the use of *syntactic sugar*.

A question arises regarding the expressiveness of the proposed notation with respect to logic notations. Certainly with respect to complex control issues, λ-based approaches permit the construction of respectable control abstractions such as replacement behaviours, which are not readily available in standard logics. By making the λ-notation non-deterministic, either as part of the execution mechanism or by encoding it using sets, we claim that many of the useful properties relating to logic based abstraction from computational mechanisms are inherited by a notation such as that proposed in this paper.

It is envisaged that the semantic model and notation which is defined in this paper would be used in conjunction with the graphical design notations currently available. The benefits of graphical notations lie in their ease of assimilation, a property which arises directly from their approximate nature. A suitable development process may be to use graphical notation as a first attempt at designing a system and then to clarify the meaning of system components and system composition using the model and notation proposed here.

8.3 Related Work

The analysis and semantic foundations of object-oriented designs and development is currently an active research area [Rui95]. For example [Cit95] shows how message diagrams (equivalent to UML collaboration diagrams) can be given a semantics in terms of a partial order on events; [Bou95] shows how the specification language Larch can be used to give a formal semantics to static object diagrams; and, [Mor96b] [Mor96a] can be used to produce executable object-oriented designs.

The use of category theory to capture the essential characteristics of systems dates back to Goguen [Gog75] who updated the approach to address concurrent object-oriented systems in [Gog90]. Sheaf theory is a general mechanism for making global observations about locally defined phenomena. In addition to Goguen, sheaves are used in [Mal96] and [Ehr91]. Category theory is used to express static properties of object-oriented designs in [Pie96]. Kent [Ken99] [Ken97] uses a graphical notation called *constraint diagrams* to express system properties. The diagrams achieve a similar aim to equalisers as used in this paper.

The use of operators such as $\times$ and $+$ to construct system descriptions dates back to the specification language *clear* [Bur77b] which lead to the OBJ family of specification languages [Gog99] which differ from the approach taken in this paper by using a semantic framework based on rewriting terms in order sorted equational logic.

A related approach which addresses object-oriented system execution is the use of modal logics; examples are Object Calculus [Bic97], [Cla97] and [Lan98]. This approach differs from that taken here in that it uses a modal logic framework to express and analyse object execution. By abstracting away from notational issues we are able to select a notation (executable or otherwise) as appropriate.

A number of researchers such as [Dup97] have used first order logical notations for expressing the semantics of object-oriented design notations. Although this approach will capture the behaviour of abstract systems, these notations do not have an executable semantics and are weak at capturing temporal system properties.

A number of researchers such as [Par83] and [Luq93] advocate program transformation and step-wise program refinement in the system development process. Transformation of programs in a functional notation started with [Bur77a] and continued with [Bir87]. Refinement of UML object-oriented designs using Z and the Object-Calculus is described in [Eva98] and [Eva99]. Both of these approaches use logic as the design language this is contrasted with the more computational approach taken in this paper. A general approach to program refinement is described in [Mor90].

8.4 Future Plans

A next step in this work is to develop a proof theory for the design notation which can be used to establish system properties. The theory will be used as the basis of an interpreter for the language since design animation can be viewed as a restricted form of proof. Other types of property include: querying whether or not a particular message is ever generated, identifying the circumstances under which a system state arises, and establishing that the system is deterministic and therefore ready for translation to a concrete programming language. The work described in [Cla99a] discusses how properties of behaviour diagrams can be established.

A proof theory is also required in order to establish a rigorously defined development process. This could take the form of a refinement relation between system diagrams. One diagram can be viewed as a refinement of another when determinacy and execution detail is increased whilst remaining consistent with the original behaviour. The work described in [Cla99a] gives an example of how an object-oriented design expressed as a collection of λ-function behaviours can be refined.

References

[Agh86] Agha, G.: Actors: *A Model of Concurrent Computation in Distributed Systems.* MIT Press, 1986.

[Agh91] Agha, G.: The Structure and Semantics of Actor Languages. In proceedings of REX School/Workshop on Foundations of Object-Oriented Languages, Lecture Notes in Computer Science 489, Springer-Verlag, 1991.

[Bar90] Barr, M. & Wells, C.: *Category Theory for Computing Science.* Prentice Hall International Series in Computer Science, 1990.

[Bic97] Bicarregui, J., Lano, K. & Maibaum, T.: Towards a Compositional Interpretation of Object Diagrams. Technical Report, Department of Computing, Imperial College of Science, Technology and Medicine, 1997.

[Bir87] Bird, R., S.: *A Calculus of Functions for Program Derivation.* Oxford University Programming Research Group Monograph, 1987.

[Boo94] Booch, G.: *Object-Oriented Analysis and Design with Applications*, 2nd edition. Benjamin/Cummings Publishing Company Inc., 1994.

[Bou95] Bourdeau, R. & Cheng, B.: A Formal Semantics for Object Model Diagrams. IEEE Transactions on Software Engineering, 21(10), 1995.

[Bur77a] Burstall, R. M. & Darlington, J. A Transformation System for Developing Recursive Programs. Journal of ACM, 24(1), 1977.

[Bur77b] Burstall, R. M. & Goguen, J. A.: Putting Theories Together to Make Specifications. In the proceedings of IJCAI '77, 1977.

[Cit95] Citrin, W., Cockburn, A., von Kanel, J. & Hauser, R.: Formalized Temporal Message Flow Diagrams. Software Practice and Experience, 25(12), 1995.

[Cla94] Clark, A. N.: A Layered Object-Oriented Programming Language. GEC Journal of Research, 11(3), The General Electric Company p.l.c., pp 173 – 180, 1994.

[Cla96] Clark, A. N.: *Semantic Primitives for Object-Oriented Programming Languages.* PhD Thesis, Queen Mary and Westfield College, University of London, 1996.

[Cla97] Clark, A. N. & Evans, A. S.: Semantic Foundations of the Unified Modelling Language. In the proceedings of the First Workshop on Rigorous Object-Oriented Methods: ROOM 1, Imperial College of Science Technology and Medicine, London, June, 1997.

[Cla99a] Clark, A. N.: A Semantics for Object-Oriented Systems. Presented at the Third Northern Formal Methods Workshop. September 1998. To appear in BCS FACS Electronic Workshops in Computing, 1999.

[Cla99b] Clark, A. N.: A Semantics for Object-Oriented Design Notations. Technical report, 1999.

[Dup97] Dupuy, S.: Chabre-Peccoud, M. and Ledru, Y., Integrating OMT and Object-Z. In the proceedings of the First Workshop on Rigorous Object-Oriented Methods, Imperial College of Science Technology and Medicine, London, June 1997.

[Ehr91] Ehrich, H-D., Goguen, J. A. & Sernadas, A.: A Categorical Model of Objects as Observed Processes. In the proceedings of REX School/Workshop on Foundations of Object-Oriented Languages, Lecture Notes in Computer Science 489, Springer-Verlag, 1991.

[Eva98] Evans, A. S.: Reasoning with UML Class Diagrams. In WIFT '98, IEEE Press, 1998.

[Eva99] Evans, A. S. & Lano, K. C.: Rigorous Development in UML. To appear in the proceedings of the ETAPS '99, FASE Workshop, 1999.

[Fie88] Field, A. J. & Harrison P. G.: *Functional programming.* Addison-Wesley, International Computer Science Series, 1988.

[Gog75] Goguen, J.: Objects. International Journal of General Systems, 1(4):237–243, 1975.

[Gog89] Goguen, J.: A Categorical Manifesto. Technical Report PRG-72, Programming Research Group, Oxford University, March 1989.

[Gog90] Goguen, J. A.: Sheaf Semantics for Concurrent Interacting Objects. Mathematical Structures in Computer Science, 1990.

[Gog99] Goguen, J. A., Winkler, T., Meseguer, J., Futstsugi, K. & Jouannaud, J-P.: Introducing OBJ. Technical report at `http://www-cse.ucsd.edu/users/goguen/pubs/iobj.ps.gz`.

[Ken99] Kent, S. & Gil J.: Visualising Action Contracts in Object-Oriented Modelling. To appear in the IEE Software Journal, 1999.

[Ken97] Kent, S.: Constraint Diagrams: Visualising Invariants in Object-Oriented Models. In the proceedings of OOPSLA 97, ACM Press, 1997.

[Lan64] Landin P.: The Next 700 Programming Languages. Communication of the ACM, 9(3), 1966, pp 157 – 166.

[Lan98] Lano, K. & Bicarregui, J.: UML Refinement and Abstraction Transformations. In the proceedings of the Second Workshop on Rigorous Object-Oriented Methods: ROOM 2, Bradford, May, 1998.

[Luq93] Luqi, V. B. & Yehudai, A.: Using Transformations in Specification-Based Prototyping. IEEE Transactions on Software Engineering, 19(5), 1993.

[Mal96] Malcolm G.: Interconnections of Object Specifications in *Formal Methods and Object Technology*, (eds.) Goldsack, S. J. & Kent, S. J., Springer-Verlag, 1996, pp 205 – 226.

[Mey88] Meyer, B.: *Object-Oriented Software Construction*. Prentice Hall International Series in Computer Science, 1988.

[Mor96a] Moreira A. & Clark R. G.: LOTOS in the Object-Oriented Analysis Process. In *Formal Methods and Object Technology*, (eds.) Goldsack, S. J. & Kent, S. J., Springer-Verlag, 1996, pp 33 – 46.

[Mor96b] Moreira, A. & Clark, R.: Adding Rigour to Object-Oriented Analysis. Software Engineering Journal, September 1996.

[Mor90] Morgan, C.: *Programming from Specifications*. Prentice-Hall, 1990.

[Par83] Partsch, H. & Steinbruggen, R. Program Transformation Systems. ACM Computing Surveys, 15(3), 1983.

[Pie96] Piessens F. & Steegmans E.: Categorical Semantics for Object-Oriented Data Specifications. In *Formal Methods and Object Technology*, (eds.) Goldsack, S. J. & Kent, S. J., Springer-Verlag, 1996, pp 302 – 316.

[Pri97] Priestley, M.: *Practical Object-Oriented Design*. McGraw-Hill, 1997.

[Rui95] Ruiz-Delgado, A., Pitt, D. & Smythe, C.: A Review of Object-Oriented Approaches in Formal Specification. The Computer Journal, 38(10), 1995.

[Run91] Rumbaugh, J.: *Object-Oriented Modeling and Design*. Prentice Hall International, 1991.

[Ryd88] Rydeheard, D. E. & Burstall, R. M.: *Computational Category Theory*. Prentice Hall International Series in Computer Science, 1988.

[Sab97] Sabry, A. & Wadler, P.: A Reflection on Call-by-Value. ACM Transactions on Programming Languages and Systems, 19(5), pp 111 – 136,1997.

[UML98] The UML Notation version 1.1, UML resource center, http://www.rational.com.

[Weg87] Wegner, P.: Dimensions of Object Based Language Design. In Meyrowtiz N. (ed.), ACM Symposium on Object-Oriented Programming: Systems, Languages and Applications, ACM, 1987, pp 168 – 182.

A Library Implementation in Java

```
class Named {
  private String name;
  public Named(String name)
  { this.name = name; }
  public String getName()
  { return name; }
}

class Book extends Named {
  private int copies = 0;
  public Book(String name)
  { super(name); }
  public void borrow()
  {
    if(copies > 0)
      copies = copies - 1;
    else throw new Error("no copies left");
  }
  public void addCopy()
  { copies = copies + 1; }
}

class Reader extends Named {
  private Vector copies = new Vector();
  public Reader(String name,Vector copies)
  {
    super(name);
    this.copies = copies;
  }
  public void borrow(String name)
  { copies.addElement(name); }
  public void ret(String name)
  { copies.removeElement(name); }
}
```

```
class Library {
  private static Vector readers = new Vector();
  private Vector books = new Vector();
  public void addReader(String name)
  { readers.addElement(new Reader(name,new Vector())); }
  public void addBook(String name)
  { books.addElement(new Book(name)); }
  public void addCopy(String bookName)
  {
    Book book = (Book)find(bookName,books);
    if(book != null)
      book.addCopy();
    else throw new Error("cannot find book");
  }
  private Named find(String name,Vector table)
  {
    Named named = null;
    for(int i = 0; (named == null) && (i < table.size()); i++) {
      Named n = (Named)table.elementAt(i);
      if(n.getName().equals(name))
        named = n;
    }
    return named;
  }
  public void borrow(String readerName,String bookName)
  {
    Reader reader = (Reader)find(readerName,readers);
    Book book = (Book)find(bookName,books);
    if((reader != null) & (book != null)) {
      reader.borrow(bookName);
      book.borrow();
    }
    else throw new Error("illegal name in borrow");
  }
  public void ret(String readerName,String bookName)
  {
    Reader reader = (Reader)find(readerName,readers);
    Book book = (Book)find(bookName,books);
    if((reader != null) & (book != null)) {
      reader.ret(bookName);
      book.addCopy();
    }
    else throw new Error("illegal name in ret");
  }
}
```