A Semantics for Object-Oriented Design Notations

Tony Clark (a.n.clark@comp.brad.ac.uk)
Formal Methods Group
Department of Computing
University of Bradford
West Yorkshire BD7 1DP

March 4, 1999

Abstract

Current graphical object-oriented design notations are syntax-bound and semantic-free since they tend to focus on design representation rather than on the meaning of the design. This paper proposes a meaning for object-oriented designs in terms of object behaviours represented as constructions in category theory. A new design language is proposed, based on λ-notation, whose semantics is given by object behaviours. An example application is constructed as both a graphical design and using the design language.

Submitted to BCS FACS Journal January 1999.

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.

This approach has the benefit of focussing on the semantics of object-oriented systems, unlike other approaches which propose particular languages, for example Z or modal logic, as their starting point. We claim that this leads to a fundamental model of object-oriented systems behaviour which can be denoted using a variety of languages, including Z, modal logics and concrete programming languages, which are chosen to suit the development method or application.

The approach is highly compositional which allows the semantics of system components to correspond very closely to the design elements which are used to denote them. This is in contrast to other approaches, for example those based on first order logic, in which the distinction between system components is blurred.

Although the proposed semantic model can be denoted by any suitable language, this paper proposes a λ-notation for object designs. The semantics of the notation is given in terms of a category of object behaviours.

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. 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 which uses all the design features which have been 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 which are common to all object-oriented systems. This section discusses 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].

Most object-oriented design notations provide models for expressing *static* and *dynamic* properties of the required system. The static properties of an object include a description of its *state space*. At any given time an object is in exactly one state consisting of a collection of named values:

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

Objects perform computations in response to messages. A message is exchanged between a sender and a receiver which are both objects. Often the design notation will require that the exchange occurs via a named associations:

Design Feature 2 *Computation occurs through messages passed via named associations.*

On receiving a message, an object performs a computation. The actions depend on the current state of the object when the message is received. On completion, the object is left in a new state and sends a collection of output messages:

Design Feature 3 *Computation at an object is described by state transitions involving input messages and output messages.*

Objects, even those with the same state and behaviour, can differ in two respects. Firstly, objects with the same behaviour are grouped into classes; they are referred to as *instances* of the class. Secondly, two instances of the same class always differ with respect to their *identity*:

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.*

Typically, a design is underspecified in the sense that it may express more than one sequence of events for every message. Indeed, this feature provides a characteristic difference between a *design* and a *program*. UML provides the *Object Constraint Language* (OCL) which is based on first order logic; the OCL can be used to define computations in terms of conditions on the pre- and post-states of objects, thereby leaving the *how* of computation to a later stage of development:

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

One of the key features of the object-oriented paradigm is *inheritance* which is a relationship between classes. Inheritance occurs in designs and supports *reuse* and *polymorphism*:

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

Most object-oriented design notations recognise that compositionality is a key feature in developing large systems. Designs are typically composed of a collection of sub-systems. Many notations allow different *views* of the same system to be expressed as different models, for example computation in terms of message sequences between objects or in terms of state transitions at a single object:

Design Feature 8 *Object designs are compositional.*

Designs often provide a view of system execution in terms of the response of a single object to a given message. UML provides *object interaction diagrams* which express sequences of messages which occur between a collection of objects. Such views of computation are often dependent on the current state of computation, several views may be used to describe the messages arising from the same initial message in different system states. Each view represents a partial definition of the receiver's response to the initial message; a complete definition is formed through composition:

Design Feature 9 *Object designs may be partial.*

Static object models express object states and associations. The associations provide a communication medium through which object interactions take place. Communication occurs through message passing which may be synchronous or asynchronous:

Design Feature 10 *Execution requires message communication which may be synchronous or asynchronous.*

Object designs place restrictions on the behaviour of single objects. Object behaviour is further restricted by design composition, especially when different views of the same computation are merged. One view may leave an object feature underspecified whilst the other makes it deterministic:

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

The behaviour of an object-oriented system is described by a collection of models. Each model represents a collection of constraints on the behaviour of the system at various points during computation. The overall system behaviour is the result of finding a solution to all of the constraints:

Design Feature 12 *The global behaviour described by an object-oriented design satisfies all of the locally specified behavioural constraints.*

The rest of this paper provides a semantic model based on the features which have been identified above.

3 Object Behaviour

Object-Oriented designs denote the structure and behaviour of systems. This section defines a semantic model of object structure and behaviour. Object behaviour is a graph which is labelled with object states and messages.

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 which contains 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) which is modified in response to handling messages (feature 2). A message is a package of information sent from one object to another.

The computation which is 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.

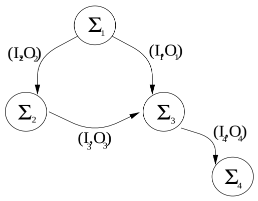
A message consists of a source object, a target object and some message data. The source and target objects are identified by their object identity tags. For a given object system, the data items which can be passed as messages will be defined for each type of target object. A message, whether input 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 which interact 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 which are 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:



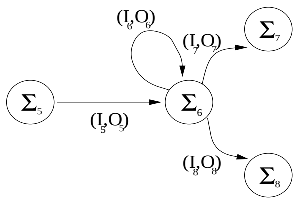
Starting in state Σ_1, the graph G_x can produce the following possible object calculations of length 2:

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

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 Semantics

The meaning of an object is defined to be the calculation graph which describes 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.

3.3.1 Categories

A *category* consists of:

- A collection of objects[1]. Upper case letters A, B, ... are used to range over objects; and
- A collection of arrows. 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 $f : A \to B$; and
- A binary associative 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.

3.3.2 Terminal Objects

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 $(\emptyset, \emptyset)$.

3.3.3 Functors

A *functor* consists of:

- A source category **C** and a target category **D**; and
- A function F_1 which maps objects of **C** to objects of **D**; and
- A function F_2 which maps arrows of **C** to arrows of **D** such that the following conditions hold:
  - For every **C** arrow $f : A \to B$, $F_2(f) : F_1(A) \to F_1(B)$ in **D**; and
  - For every **C** object A, $F_2(id_A) = id_{F_1(A)}$; and
  - For every pair of composable **C** arrows $g \circ f$, $F_2(g \circ f) = F_2(g) \circ F_2(h)$ in **D**.

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

3.3.4 Natural Transformations

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 commutes for all $\mathbf{C}$ arrows $f : A \to B$:

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

3.3.5 Objects as Functors

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** which are defined using standard categorical constructions.

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**. Although the language is more expressive than current object-oriented design notations, section 7 shows a correspondence with graphical notations.

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 parameters are supplied with values as follows:

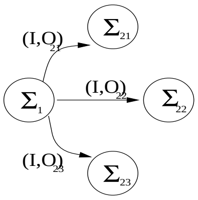
1. The first parameter is supplied with a type tag α. The result is a function which describes the behaviour of a class of objects.

2. 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.

3. 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.

4. 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] uses exactly 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, p_i are patterns which match sets of input messages, e_i and e are *transition expressions* whose values are pairs of replacement object behaviour functions and sets of output messages.

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 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.

A case expressions of the form **case** m **of else** e **end** is equivalent to just the expression e.

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:

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

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 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 from the system. Suppose that, given a system of objects 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 which emerge:

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

4.4 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 the following 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*.

Suppose that M is a behaviour function:

$$M = \lambda i_1.\lambda i_2.\lambda i_3.\lambda i_4.N$$

then 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.

4.5 Behaviour Morphisms

Arrows in **Obj** are families of graph homomorphisms which must be well-behaved with respect to message sequences (see section 3.3.5). 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 $\gamma : O_1 \to O_2$ from O_1 to O_2 is a family of graph homomorphisms γ_n for each object in **Int** such that for any arrow $f : n \to m$ 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 example, suppose that we wish to produce a new behaviour *cellx* which stores a value called x, and responds to messages *setx* and *getx*. This is achieved as follows, where $\mathit{pair}(p, q)(v_1, v_2) = (p(v_1), q(v_2))$:

$$\begin{array}{l}
\mathbf{let}\ \mathit{ctox} = (\mathit{map}(f), \mathit{pair}(\mathit{map}(g_1), \mathit{map}(g_2))) \\
\quad \mathbf{where} \\
\qquad f(<\tau_1, \tau_2, \rho>) = <\tau_1, \tau_2, \rho[x/v]> \\
\qquad g_1(<\tau_1, \tau_2, \mathit{set}(v)>) = <\tau_1, \tau_2, \mathit{setx}(v)> \\
\qquad g_1(<\tau_1, \tau_2, \mathit{get}>) = <\tau_1, \tau_2, \mathit{getx}> \\
\qquad g_2(<\tau_1, \tau_2, \nu>) = <\tau_1, \tau_2, \nu>
\end{array}$$

$$\mathbf{let}\ \mathit{cellx} = \mathit{ctox}(\mathit{cell})$$

A behaviour *celly* can be produced by applying morphism *ctoy* to *cell*. The morphism *ctoy* is defined by renaming v to y, *set* to *sety* and *get* to *gety*.

For each behaviour function there is exactly one possible morphism to the terminal behaviour function *empty*:

$$\mathbf{let}\ \mathit{term} = (\mathbf{K}(\emptyset), \mathbf{K}(\emptyset, \emptyset))$$

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 and we give examples of their use.

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:

$$
\begin{tikzcd}
 & C \arrow[dl, bend right, "f"'] \arrow[d, "u"] \arrow[dr, bend left, "g"] & \\
A & A \times B \arrow[l, "\pi_1"] \arrow[r, "\pi_2"'] & B
\end{tikzcd}
$$

5.1.1 Behaviour Products

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 leads to 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.

The following shows how the product operator is used to construct a two dimensional point behaviour from components *cellx* and *celly*:

$$\textbf{let } (\mathit{point}, \pi_x, \pi_y) = \mathit{cellx} \times \mathit{celly}$$

The resulting function *point* describes a family of behaviours. All object states have attributes x and y. The messages which can be sent to the objects are: *setx*; *sety*; *getx*; and *gety*. Notice that the product operator $\times$ forces the x attribute to be left unchanged when a *sety* message is received because the function *cellx* does not recognise the message. Similarly, the y attribute is unaffected by the *setx* message.

5.1.2 Combining Partial Behaviours

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.

For example, suppose that we want to describe a cell which can respond to two messages: *set* and *get*. The *get* message retrieves the current value in the cell. The *set* message supplies a value v and causes the cell to change to $v/2$ if the value is even and v if the value is odd. This can be expressed as follows:

$$
\begin{array}{l}
\textbf{let } \mathit{cellEven}(\alpha)(\tau)(v)(m) = \\
\quad \textbf{case } m \textbf{ of} \\
\quad\quad \{<\tau', \tau, \mathit{get}>\} \cup m' \rightarrow \\
\quad\quad\quad (\mathit{cellEven}(\alpha)(\tau)(v), \{<\tau, \tau', v>\}) \\
\quad\quad \{<\tau', \tau, \mathit{set}(w)>\} \cup m' \rightarrow \\
\quad\quad\quad (\mathit{cellEven}(\alpha)(\tau)(w/2), \emptyset) \textbf{ when } \mathit{even}(w) \\
\quad\quad \{<\tau', \tau, \mathit{set}(w)>\} \cup m' \rightarrow \\
\quad\quad\quad (\mathit{cellEven}(\alpha)(\tau)(\top), \emptyset) \textbf{ when } \mathit{odd}(w) \\
\quad\quad \textbf{else } (\mathit{cellEven}(\alpha)(\tau)(v), \emptyset) \\
\quad \textbf{end}
\end{array}
$$

The behaviour *cellEven* handles the message $\mathit{set}(w)$ in two mutually exclusive ways. Firstly, if the value w is even the current value of the state variable v is updated to $w/2$. Secondly, if the value of w is odd then the state variable v becomes $\top$. The value $\top$ is special in the design notation and represents the non-deterministic selection of a value from all possible values. In effect, we do not know what *cellEven* will do when it receives an odd value and therefore we allow any possible state change.

A behaviour *cellOdd* defines the behaviour of a cell when it is supplied with an odd value:

$$
\begin{array}{l}
\textbf{let } \mathit{cellOdd}(\alpha)(\tau)(v)(m) = \\
\quad \textbf{case } m \textbf{ of} \\
\quad\quad \{<\tau', \tau, \mathit{get}>\} \cup m' \rightarrow \\
\quad\quad\quad (\mathit{cellEven}(\alpha)(\tau)(v), \{<\tau, \tau', v>\}) \\
\quad\quad \{<\tau', \tau, \mathit{set}(w)>\} \cup m' \rightarrow \\
\quad\quad\quad (\mathit{cellOdd}(\alpha)(\tau)(\top), \emptyset) \textbf{ when } \mathit{even}(w) \\
\quad\quad \{<\tau', \tau, \mathit{set}(w)>\} \cup m' \rightarrow \\
\quad\quad\quad (\mathit{cellOdd}(\alpha)(\tau)(w), \emptyset) \textbf{ when } \mathit{odd}(w) \\
\quad\quad \textbf{else } (\mathit{cellOdd}(\alpha)(\tau)(v), \emptyset) \\
\quad \textbf{end}
\end{array}
$$

Finally, a deterministic behaviour is constructed by forming a product of *cellEven* and *cellOdd*:

$$\textbf{let } (\mathit{cellEvenOdd}, \pi_e, \pi_o) = \mathit{cellEven} \times \mathit{cellOdd}$$

5.1.3 Object Communication

Object-oriented execution occurs by means of message passing. Design notations express the potential for message passing between objects using associations. Associations are modelled in **Obj** as a named attribute whose value contains the object identity of the associated objects.

Message passing occurs when a behaviour produces an output message whose target is given by the value of an association. Messages may be synchronous or asynchronous (feature 10). An asynchronous message is sent and the source object does not wait for a reply. In the case of synchronous messages, an object must arrange to wait for a reply before continuing to process any incoming messages.

For example, a cashpoint machine is associated to a number of accounts. The object identity tags of the accounts are maintained in the machine database. A customer can make an enquiry regarding the balance of their account by supplying their pin number:

$$
\begin{array}{l}
\textbf{letrec } \mathit{cashpoint}(\alpha)(\tau)(\mathit{db})(m) = \\
\quad \textbf{case } m \textbf{ of} \\
\quad\quad \{<\tau', \tau, \mathit{enquire}(\mathit{pin})>\} \cup m' \rightarrow \\
\quad\quad\quad (\mathit{wait}, \{<\tau, \mathit{lookup}(\mathit{pin})(\mathit{db}), \mathit{balance}>\}) \\
\quad\quad\quad\quad \textbf{whererec } \mathit{wait}(m) = \\
\quad\quad\quad\quad\quad \textbf{case } m \textbf{ of} \\
\quad\quad\quad\quad\quad\quad \{<\tau'', \tau, \mathit{balance}(n)>\} \cup m' \rightarrow \\
\quad\quad\quad\quad\quad\quad\quad (\mathit{cashpoint}(\alpha)(\tau)(\mathit{db}), \{<\tau, \tau', n>\} \cup m') \\
\quad\quad\quad\quad\quad\quad \textbf{else } (\mathit{wait}, m) \\
\quad\quad\quad\quad\quad \textbf{end} \\
\quad\quad \textbf{else } (\mathit{cashpoint}(\alpha)(\tau)(\mathit{db}), \emptyset) \\
\quad \textbf{end}
\end{array}
$$

The cashpoint behaviour shows how replacement behaviours are used to handle synchronous messages. On receiving an *enquire* message, the replacement *wait* is used to store up messages until a balance is returned by the appropriate account.

5.2 Coproducts

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

such that the following digram commutes:

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

5.2.1 Behaviour Coproducts

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:

$$\mathit{strip}(\mathit{pink}(v)) = v$$

$$\mathit{strip}(\mathit{blue}(v)) = v$$

Given object calculation functions M_1 and M_2 then we define the operator $+$ as follows:

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

5.2.2 Non-Deterministic Behaviours

Suppose a cell is to be designed which can be sent a message *mod* causing it to modify its value. Modification can be defined to add 1 to the value:

$$\begin{array}{l}
\textbf{letrec } \mathit{cellInc}(\alpha)(\tau)(v)(m) = \\
\quad \textbf{case } m \textbf{ of} \\
\quad\quad \{<\tau', \tau, \mathit{mod}>\} \cup m' \rightarrow \\
\quad\quad\quad (\mathit{cellInc}(\alpha)(\tau)(v+1), \emptyset) \\
\quad\quad \textbf{else } (\mathit{cellInc}(\alpha)(\tau)(v), \emptyset) \\
\quad \textbf{end}
\end{array}$$

Alternatively, modification can be defined to subtract 1 from the value:

$$\begin{array}{l}
\textbf{letrec } \mathit{cellDec}(\alpha)(\tau)(v)(m) = \\
\quad \textbf{case } m \textbf{ of} \\
\quad\quad \{<\tau', \tau, \mathit{mod}>\} \cup m' \rightarrow \\
\quad\quad\quad (\mathit{cellDec}(\alpha)(\tau)(v-1), \emptyset) \\
\quad\quad \textbf{else } (\mathit{cellDec}(\alpha)(\tau)(v), \emptyset) \\
\quad \textbf{end}
\end{array}$$

If we wish to leave the choice between increment and decrement open as a design choice then the cell can be defined as non-deterministic using the + operator:

$$\textbf{let } \mathit{cellNonDet} = \mathit{cellInc} + \mathit{cellDec}$$

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 $f : A \rightarrow B$ and $g : A \rightarrow B$ be two object morphisms. An equalizer of the arrows is an object E together with a morphism $e : E \rightarrow A$ such that $f \circ e = g \circ e$ and for any object E' and arrow $h : E' \rightarrow A$ such that $f \circ h = g \circ h$ there is a unique arrow u such that the following diagram commutes:

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

5.3.1 Behaviour Equalizers

In **Calc** an equalizer between two graph homomorphisms $f : G_1 \rightarrow G_2$ and $g : G_1 \rightarrow G_2$ is a graph G and a homomorphism $e : G \rightarrow G_1$ 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.

5.3.2 Consistent Behaviours

Equalizers can be used to make two ways of viewing the same behaviour consistent. For example, a two-dimensional point behaviour can be mapped onto a single cell so that the x or the y co-ordinate becomes the value of the cell. Suppose that we define *point* as follows:

$$
\begin{array}{l}
\textbf{let } point(\alpha)(\tau)(x,y)(m) = \\
\quad \textbf{case } m \textbf{ of} \\
\quad\quad \emptyset \rightarrow (point(\alpha)(\tau)(x,y), \emptyset) \\
\quad\quad \{<\tau', \tau, getx>\} \cup m' \rightarrow point(\alpha)(\tau)(x,y)(m') \otimes \{<\tau, \tau', x>\} \\
\quad\quad \{<\tau', \tau, gety>\} \cup m' \rightarrow point(\alpha)(\tau)(x,y)(m') \otimes \{<\tau, \tau', y>\} \\
\quad\quad \{<\tau', \tau, setx(v)>\} \cup m' \rightarrow point(\alpha)(\tau)(v,y)(m') \\
\quad\quad \{<\tau', \tau, sety(v)>\} \cup m' \rightarrow point(\alpha)(\tau)(x,v)(m') \\
\quad\quad \textbf{else } (point(\alpha)(\tau)(x,v), \emptyset) \\
\quad \textbf{end}
\end{array}
$$

where $(P, O_1) \otimes O_2 = (P, O_1 \cup O_2)$. The following pair of arrows define homomorphisms which map *point* onto *cell*. Arrow f maps the x co-ordinate onto the value of the cell, *getx* to *get*, *setx* to *set* and all other messages to *unknown*. Arrow g maps the y co-ordinate onto the value of the cell and, *gety* to *get*, *sety* to *set* and all other messages to *unknown*. Arrow f is defined below:

$$
\begin{array}{l}
\textbf{let } f = (map(f_1), pair(map(f_2), map(id))) \\
\quad \textbf{where} \\
\quad\quad f_1(<\alpha, \tau, \rho>) = <\alpha, \tau, \rho/\{x\}> \\
\quad\quad f_2(<\tau_1, \tau_2, getx>) = <\tau_1, \tau_2, get> \\
\quad\quad f_2(<\tau_1, \tau_2, setx(v)>) = <\tau_1, \tau_2, set(v)> \\
\quad\quad f_2(<\tau_1, \tau_2, gety>) = <\tau_1, \tau_2, unknown> \\
\quad\quad f_2(<\tau_1, \tau_2, sety(v)>) = <\tau_1, \tau_2, unknown>
\end{array}
$$

An equalizer for f and g will require that the x and y co-ordinates are always the same value, *i.e.* any changes which are made to an object described by the equalizer must affect both the x and the y co-ordinate *in tandem*:

$$\textbf{let}\ (tandem, e) = eq(point)(f)(g)$$

The object *tandem* describes object calculations whose message input sets are elements of the following inductively defined set M: $\emptyset \in M$; $\{getx\} \in M$; $\{gety\} \in M$; $\{setx(v), sety(v)\} \in M$; if $m_1, m_2 \in M$ then $m_1 \cup m_2 \in M$.

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.

This section provides an algorithm for constructing limits on a diagram and gives an example. The algorithm is derived from a proof in [Ryd88] which shows how a colimit can be constructed on a diagram.

Let Δ be a diagram on a category with a terminal object, all finite products and finite equalizers. If the diagram has no edges (and consists only of nodes) then the limit is constructed by forming the product of all node labels on the diagram. If the diagram is empty then the limit is a terminal object of the category.

If the diagram contains edges then let $e : p \to q$ be an edge with an associated arrow $f : \Delta_p \to \Delta_q$ and let Δ' be Δ with e removed. Assume inductively that the diagram Δ' has a limit $\eta_n : A \to \Delta_n$. Consider a parallel pair of arrows: $\eta_q : A \to \Delta_q$ and $f \circ \eta_p : A \to \Delta_q$. Let the equalizer of these arrows be $h : B \to A$ which is shown on the following diagram:

$$\begin{tikzcd}
 & B \arrow[d, "h"] & \\
 & A \arrow[dl, "\eta_p"'] \arrow[dr, "\eta_q"] & \\
\Delta_p \arrow[rr, "f"'] & & \Delta_q
\end{tikzcd}$$

We wish to show that $\nu_n = \eta_n \circ h : B \to \Delta_n$ is a limit on the diagram Δ. We show that for any other cone there must be a unique arrow to B. Let $\nu'_n : B' \to \Delta_n$ be another code on Δ. By definition, since η_n is a limit there must be a unique arrow $\mu : B' \to A$ such that for all nodes $n \in \Delta$ $\eta_n \circ u = \nu'_n$. Also, by the definition of equalizers since $f \circ \eta_p \circ u = \eta_q \circ u$ then there must be a unique arrow $v : B' \to B$ which makes the following diagram commute:

$$\begin{tikzcd}
 & B \arrow[dd, "h"] & \\
 & & B' \arrow[ul, "v"'] \arrow[dl, "u"] \arrow[ddll, "\nu_p"] \arrow[dd, "\nu_q"] \\
 & A \arrow[dl, "\eta_p"'] \arrow[dr, "\eta_q"] & \\
\Delta_p \arrow[rr, "f"'] & & \Delta_q
\end{tikzcd}$$

Since v is unique and $\nu_n : B' \to \Delta_n$ is an arbitrary cone on the diagram we must conclude that $\nu_n : B \to \Delta_n$ is a limit on the diagram Δ.

Suppose that we are constructing a library system. The system consists of two behaviours. The *library* behaviour describes library objects in terms of book titles, borrowers, copies on shelves and records of books currently on loan. The *person* behaviour describes people who may or may not be members of the library. We wish to express the constraint that all borrowers are people but not all people are borrowers. We sketch the design and show how the limit is constructed.

The *library* and *person* behaviours are defined as functions:

$$\textbf{letrec } \mathit{library}(\tau)(\mathit{books}, \mathit{borrowers}, \mathit{copies}, \mathit{onloan})(m) = \ldots$$

$$\textbf{letrec } \mathit{person}(\tau)(\mathit{name}, \mathit{age}, \mathit{sex})(m) = \ldots$$

The constraint is expressed as a pair of mappings with a simple common target behaviour *unit* which stores a single value v and ignores all messages:

$$\textbf{letrec } \mathit{unit}(v)(m) = \{(\mathit{unit}(v), \emptyset)\}$$

The behaviour morphism $f : \mathit{library} \to \mathit{unit}$ projects multiple library objects to a single *unit* object whose attribute v is the union of all borrowers. The morphism $g : \mathit{person} \to \mathit{unit}$ maps all people to a single *unit* object whose attribute v a sub-set of the people object tags non-deterministically selected from all possible sub-sets. In both cases f and g map messages to $(\emptyset, \emptyset)$ since we are not interested in the dynamic behaviour of libraries and people. This is expressed on the following diagram:

$$\begin{tikzcd}
\mathit{library} \arrow[dr, "f"'] & & \mathit{person} \arrow[dl, "g"] \\
 & \mathit{unit} &
\end{tikzcd}$$

Figure 1 shows how a limit is constructed for this diagram as a process involving 8 steps. Nodes on the diagram are labelled with behaviours or X which denotes behaviour product. Edges on the diagram are labelled with behaviour morphisms.

The initial diagram (i) is empty and the limit is therefore a terminal object *empty*. When *library* is added in (ii) the limit is the object *library*. In (iii) the *unit* object is added and a limit is constructed as a product $\mathit{library} \times \mathit{unit}$. The arrow f is added in (iv) and an equalizer is used to construct a limit *obj* in (v). The object *person* is added and the limit in (vi) is constructed as a product $\mathit{obj} \times \mathit{person}$. The arrow g is added in (vii) and finally a limit of the original diagram is constructed as an equalizer *limit* in $(viii)$.

The limit L is constructed in the design language using the operators $\times$ and *eq* as follows:

$$\begin{array}{l}
\textbf{let } (o_1, \pi_1, \pi_2) = \mathit{library} \times \mathit{unit} \\
\textbf{let } (\mathit{obj}, e_1) = \mathit{eq}(o_1)(f \circ \pi_1)(\pi_2) \\
\textbf{let } (o_2, \pi_3, \pi_4) = \mathit{obj} \times \mathit{person} \\
\textbf{let } (\mathit{limit}, e_2) = \mathit{eq}(o_2)(\pi_2 \circ e_1 \pi_3)(g \circ \pi_4)
\end{array}$$

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 is required to control a chemical factory. Lorry-loads of chemicals x and y arrive at the factory and are stored in tanks. Each tank has a storage capacity which must not be exceeded. Single measures of x and y are transferred to a mixer tank where they are mixed to produce a single unit of chemical z. Chemical z is transferred from the mixer tank to a storage tank awaiting collection and distribution.

7.2 Static Model

Analysis of the requirements leads to the identification of two classes: *factory* and *tank*. Further analysis of *tank* identifies two sub-classes: *storage* and *mixer*. The class *factory* has the following methods:

- *addx* which adds chemical to a storage tank containing chemical x. This method returns *true* when the operation succeeds.
- *addy* which is the same as *addx* but uses a different storage tank.

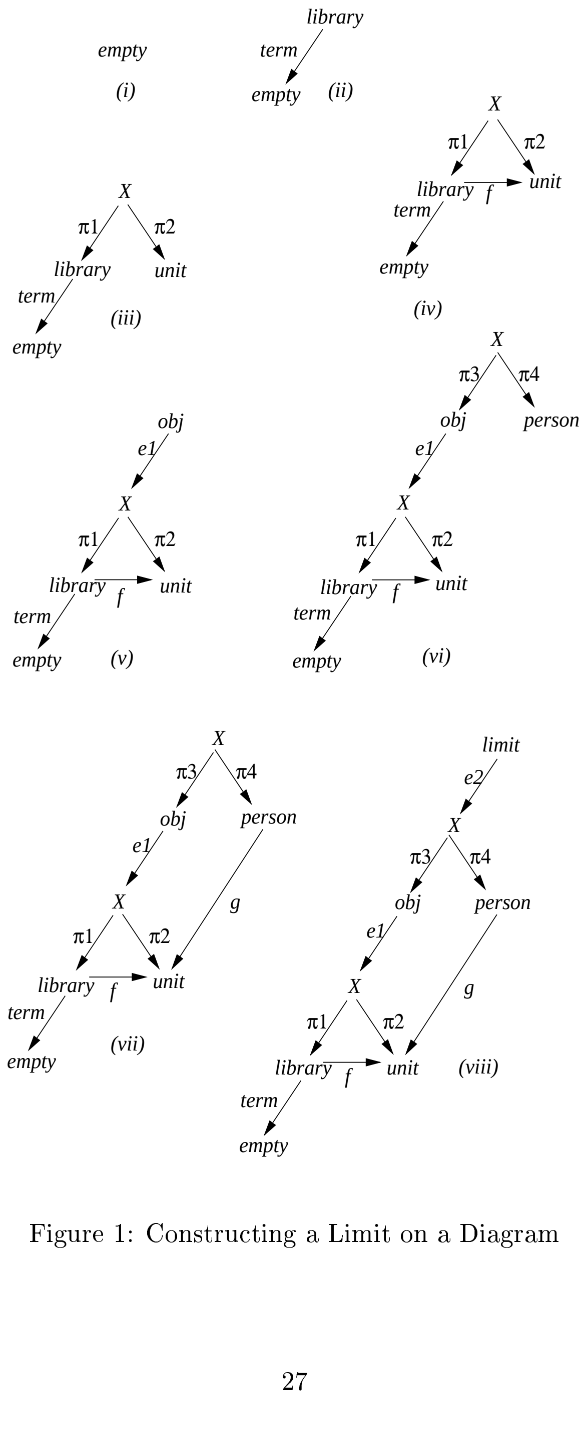


Figure 1: Constructing a Limit on a Diagram

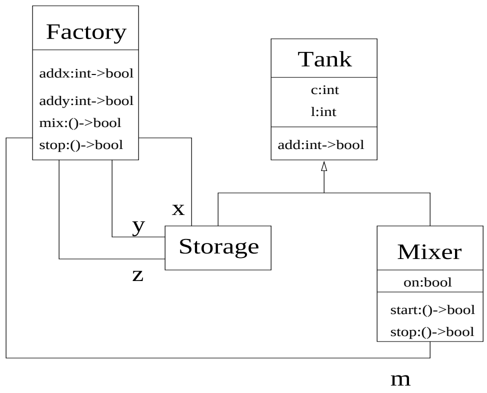


Figure 2: Static class diagram for the factory application

- *mix* which transfers 1 unit of x and 1 unit of y from the storage tanks to the mixer tank and starts the mixing process. The method returns *true* when the operation succeeds.

- *stop* which stops the mixing tank and transfers the contents to a storage tank for chemical z. This method returns *true* when the operation is successful.

To support these operations, *factory* has three associations named x, y and z with *storage* and a single association named m with *mixer*.

The class *tank* has a state consisting of c which is an integer representing the current tank contents, and l which is the capacity limit. The class *tank* has a single method called *add* which adds chemical to the tank. The method returns *true* when the operation is successful. Chemical is removed from a tank by supplying a negative value to *add*.

The class *mixer* has a boolean state variable *on* and two methods *start* and *stop* which both return *true* when they are successful.

Graphical object-oriented design notations such as UML express this information on a *static class diagram*. The static diagram for the factory application is shown in figure 2.

7.3 Dynamic Models

Each of the factory methods leads to a collection of object interaction diagrams. Each diagram expresses the sequence of messages which occur under a particular condition. A message is expressed as follows:

$$[b]n : e = m(e_1, \dots, e_n)$$

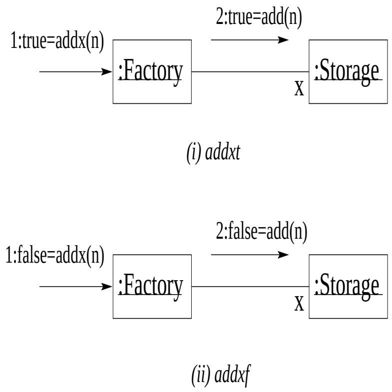


Figure 3: Object interaction diagrams for *addx*

where b is a conditional expression which acts as a guard on the message; n is an integer which determines the message ordering on the diagram; e is a value expression representing the return value of the message; m is the name of the message; and, e_i are the arguments of the message. A *ground* interaction is produced by consistent substitution for all variables on a diagram.

- The method *addx* can be described using two behaviours. The first is named *addxt* and describes the successful behaviour of *addx*. The second is named *addxf* and describes the unsuccessful behaviour. The object interaction diagrams are shown in figure 3.

- The method *addy* is described in the same way as *addx*. The result is two behaviours *addyt* and *addyf*.

- The method *mix* is described using a single interaction diagram shown in figure 4. Note that *mix* is only successful when the x and y tanks are not empty and the mixing tank is not full or currently mixing. Variables a and b are *true* when units of x and y are available. Variables c and d are *true* when the mixing tank can be loaded and then started.

- The method *stop* is described by a single interaction diagram shown in figure 5. The operation is successful when the mixing tank can be stopped, emptied and the unit of chemical z can be transferred to the z tank. Variables a and b are *true* when the mixing tank can be stopped and unloaded. Variable c is *true* when the chemical can be loaded into tank z.

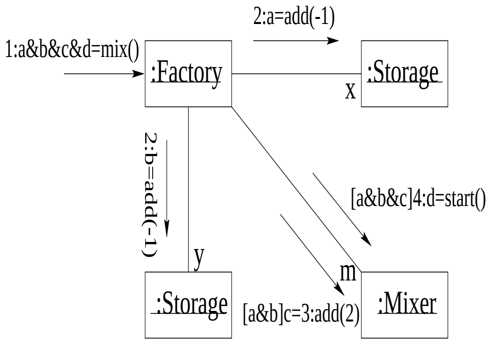


Figure 4: Object interaction diagram for *mix*

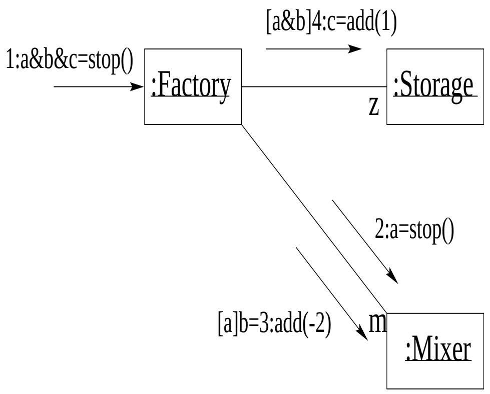


Figure 5: Object interaction diagram for *stop*

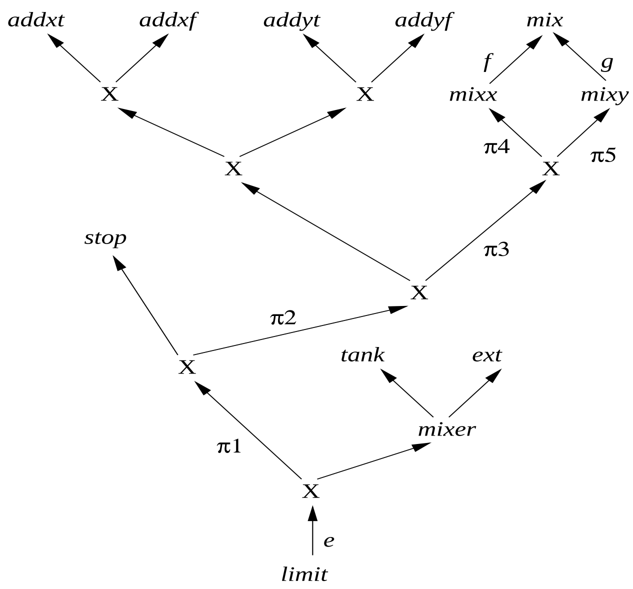


Figure 6: A factory design diagram

7.4 Factory Behaviours

Figure 6 shows the component behaviours of the factory design and how they are combined using products and equalizers to produce a single factory behaviour. A limit *limit* is constructed from the behaviours and products shown on the diagram and an equalizer constructed from the following two arrows: $f \circ \pi_4 \circ \pi_3 \circ \pi_2 \circ \pi_1$ and $g \circ \pi_5 \circ \pi_3 \circ \pi_2 \circ \pi_1$. This section describes how the behaviours are defined as functions in the design language.

7.4.1 Addx and addy

The behaviour for *addx* is decomposed into two sub-behaviours *addxt* and *addxf* which describe the success and failure of the operation respectively. The behaviour *addxt* is defined as follows:

$$
\begin{array}{l}
\textbf{let } \mathit{addxt}(\alpha)(\tau)(x)(m) = \\
\quad \textbf{case } m \textbf{ of} \\
\quad\quad \{<\tau', \tau, \mathit{addx}(n)>\} \cup m' \to (\mathit{wait}, \{<\tau, x, \mathit{add}(n)>\}) \\
\quad\quad\quad \textbf{whererec } \mathit{wait}(m) = \\
\quad\quad\quad\quad \{<x, \tau, \mathit{true}>\} \cup m' \to (\mathit{addxt}(\alpha)(\tau)(x), \{<\tau, \tau', \mathit{true}>\} \cup m') \\
\quad\quad\quad\quad \{<x, \tau, \mathit{false}>\} \cup m' \to (\mathit{addxt}(\alpha)(\tau)(\top), m') \\
\quad\quad\quad\quad \textbf{else } (\mathit{wait}, m) \\
\quad\quad \textbf{else } (\mathit{addxt}(\alpha)(\tau)(x), \emptyset) \\
\quad \textbf{end}
\end{array}
$$

The behaviour *addxt* receives a message *addx(n)* from object τ' and uses a replacement behaviour called *wait* to implement synchronous method call and return. The behaviour *wait* processes messages until it receives *true* or *false* from the x tank. If it receives *true* then the behaviour reverts to *addxt* and returns a message to τ'. Otherwise, if the return from x is *false*, the behaviour of *wait* is undefined. If *addxt* receives any message other than *addx* the result is undefined.

The behaviour *addxf* is defined in the same way as *addxt* above except that the definition of *wait* for *addxf* is defined when x returns *false* and undefined when it returns *true*.

Together, *addxt* and *addxf* define all possible behaviours for which occur when a factory receives *addx*. The behaviour *addxt* forces the correct behaviour when the x tank reports success and permits any behaviour when the x tank reports failure. The behaviour *addxf* forces the correct behaviour when the x tank reports failure and permits any behaviour when the x tank reports success.

The behaviour for *addy* is defined in exactly the same way as *addx* except that all occurrences of x are replaced with y. Note that this could be achieved using a renaming behaviour morphism.

7.4.2 Mix

The behaviour of *mix* is decomposed into the following sub-behaviours: *mix*, *mixx* and *mixy*. From the design in figure 4 we know that *mix* succeeds when tanks x and y are not empty and tank m is not full or currently mixing. One way of expressing this on a behaviour diagram is to define a single behaviour *mix* for mixing from a single arbitrary tank. Then two behaviours *mixx* and *mixy* specialise *mix* for the x and y tanks respectively. Behaviour morphisms translate *mixx* and *mixy* onto *mix*. A limit on the diagram will force mixing from x and y to succeed simultaneously in order for the *mix* operation to succeed. The behaviour *mix* is shown in figure 7. The value of t_1 is a storage tank object tag and the value of t_2 is a mixer tank object tag.

$$
\begin{array}{l}
\mathbf{letrec}\ \mathit{mix}(\alpha)(\tau)(t_1, t_2)(m) = \\
\quad \mathbf{case}\ m\ \mathbf{of} \\
\quad\quad \{<\tau', \tau, \mathit{mix}>\} \cup m' \rightarrow (\mathit{wait}, \{<\tau, t_1, \mathit{add}(-1)>\}) \\
\quad\quad \mathbf{whererec}\ \mathit{wait}(m) = \\
\quad\quad\quad \mathbf{case}\ m\ \mathbf{of} \\
\quad\quad\quad\quad \{<t_1, \tau, \mathit{true}>\} \cup m' \rightarrow (\mathit{wait}, \{<\tau, t_2, \mathit{add}(1)>\}) \\
\quad\quad\quad\quad\quad \mathbf{whererec}\ \mathit{wait}(m) = \\
\quad\quad\quad\quad\quad\quad \mathbf{case}\ m\ \mathbf{of} \\
\quad\quad\quad\quad\quad\quad\quad \{<t_2, \tau, \mathit{true}>\} \cup m' \rightarrow (\mathit{wait}, \{<\tau, t_2, \mathit{start}>\}) \\
\quad\quad\quad\quad\quad\quad\quad\quad \mathbf{whererec}\ \mathit{wait}(m) = \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad \mathbf{case}\ m\ \mathbf{of} \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \{<t_2, \tau, \mathit{true}>\} \cup m' \rightarrow \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad (\mathit{mix}(\alpha)(\tau)(t_1, t_2), \{<\tau, \tau', \mathit{true}>\} \cup m') \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \{<t_2, \tau, \mathit{false}>\} \cup m' \rightarrow \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad (\mathit{mix}(\alpha)(\tau)(t_1, t_2), \{<\tau, \tau', \mathit{false}>\} \cup m') \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \mathbf{else}\ (\mathit{wait}, m) \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad \mathbf{end} \\
\quad\quad\quad\quad\quad\quad\quad \{<t_2, \tau, \mathit{false}>\} \cup m' \rightarrow \\
\quad\quad\quad\quad\quad\quad\quad\quad (\mathit{mix}(\alpha)(\tau)(t_1, t_2), \{<\tau, \tau', \mathit{false}>\} \cup m') \\
\quad\quad\quad\quad\quad\quad\quad \mathbf{else}\ (\mathit{wait}, m) \\
\quad\quad\quad\quad\quad\quad \mathbf{end} \\
\quad\quad\quad\quad \{<t_1, \tau, \mathit{false}>\} \cup m' \rightarrow (\mathit{mix}(\alpha)(\tau)(t_1, t_2), \{<\tau, \tau', \mathit{false}>\} \cup m') \\
\quad\quad\quad\quad \mathbf{else}\ (\mathit{wait}, m) \\
\quad\quad\quad \mathbf{end} \\
\quad\quad \mathbf{else}\ (\mathit{mix}(\alpha)(\tau)(t_1, t_2), \emptyset) \\
\quad \mathbf{end}
\end{array}
$$

Figure 7: The behaviour function *mix*

$$
\begin{array}{l}
\mathbf{letrec}\ \mathit{stop} = \mathit{stopt} + \mathit{stopf} \\
\quad \mathbf{whererec} \\
\quad\quad \mathit{stopt}(\alpha)(\tau)(m,z)(\{<\tau',\tau,\mathit{stop}>\} \cup n') = (\mathit{wait}, \{<\tau,m,\mathit{stop}>\}) \\
\quad\quad\quad \mathbf{whererec} \\
\quad\quad\quad\quad \mathit{wait}(\{<m,\tau,\mathit{true}>\} \cup n') = (\mathit{wait}, \{<\tau,m,\mathit{add}(-2)>\}) \\
\quad\quad\quad\quad\quad \mathbf{whererec} \\
\quad\quad\quad\quad\quad\quad \mathit{wait}(\{<m,\tau,\mathit{true}>\} \cup n') = (\mathit{wait}, \{<\tau,z,\mathit{add}(1)>\}) \\
\quad\quad\quad\quad\quad\quad\quad \mathbf{whererec} \\
\quad\quad\quad\quad\quad\quad\quad\quad \mathit{wait}(\{<z,\tau,\mathit{true}>\} \cup n') = \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad (\mathit{stopt}(\alpha)(\tau)(m,z), \{<\tau,\tau',\mathit{true}>\}) \\
\quad\quad \mathit{stopf}(\alpha)(\tau)(m,z)(\{<\tau,\tau,\mathit{stop}>\} \cup n') = (\mathit{wait}, \{<\tau,m,\mathit{stop}>\}) \\
\quad\quad\quad \mathbf{whererec}\ \mathit{wait}(n) = \\
\quad\quad\quad\quad \mathbf{case}\ n\ \mathbf{of} \\
\quad\quad\quad\quad\quad \{<m,\tau,\mathit{false}>\} \cup n' \to (\mathit{stopf}, \{<\tau,\tau',\mathit{false}>\}) \\
\quad\quad\quad\quad\quad \{<z,\tau,\mathit{false}>\} \cup n' \to (\mathit{stopf}, \{<\tau,\tau',\mathit{false}>\}) \\
\quad\quad\quad\quad\quad \{<z,\tau,\mathit{true}>\} \cup n' \to (\mathit{stopf}, \{<\tau,\tau',\mathit{true}>\}) \\
\quad\quad\quad\quad\quad \mathbf{else}\ (\mathit{wait}, \emptyset) \\
\quad\quad\quad\quad \mathbf{end}
\end{array}
$$

Figure 8: The behaviour *stop*

7.4.3 Stop

The behaviour of *stop* succeeds when the mixing tank can be stopped, two units of chemical can be removed from the mixing tank and a single unit of chemical can be added to the z tank. If any of these operations fail then the *stop* operation fails. The behaviour function for *stop* provides an example of using $+$ and is shown in figure 8.

7.4.4 Tanks

The behaviour of *tank* and *storage* are defined as a single function which is supplied with different type tags for the two classes. The function is as follows:

$$
\begin{array}{l}
\mathbf{letrec}\ \mathit{tank}(\alpha)(\tau)(c,l)(m) = \\
\quad \mathbf{case}\ m\ \mathbf{of} \\
\quad\quad \{<\tau',\tau,\mathit{add}(n)>\} \cup m' \to \\
\quad\quad\quad (\mathit{tank}(\alpha)(\tau)(c,l), \{<\tau,\tau',\mathit{false}>\}) \\
\quad\quad\quad\quad \mathbf{when}\ (n+c<0)\ \mathbf{or}\ (n+c>l) \\
\quad\quad\quad (\mathit{tank}(\alpha)(\tau)(n+c,l), \{<\tau,\tau',\mathit{true}>\}) \\
\quad\quad\quad\quad \mathbf{when}\ (n+c \geq 0)\ \mathbf{and}\ (n+c \leq l) \\
\quad\quad \mathbf{else}\ (\mathit{tank}(\alpha)(\tau)(\top), \emptyset) \\
\quad \mathbf{end}
\end{array}
$$

Finally, the behaviour *mixer* is a sub-class of *tank* (feature 7). Sub-classes are defined using behaviours by extending the inherited behaviour. The extension is achieved as a product of *tank* and a new behaviour *ext*. The behaviour *ext* is an extension which defines just the behaviour for the methods *start* and *stop*:

$$
\begin{array}{l}
\textbf{letrec } \mathit{ext}(\alpha)(\tau)(\mathit{on})(m) = \\
\quad \textbf{case } m \textbf{ of} \\
\quad\quad \{<\tau', \tau, \mathit{start}>\} \cup m' \rightarrow \\
\quad\quad\quad (\mathit{ext}(\alpha)(\tau)(\mathit{on}), \{<\tau, \tau', \mathit{false}>\}) \textbf{ when } \mathit{on} \\
\quad\quad\quad (\mathit{ext}(\alpha)(\tau)(\mathit{true}), \{<\tau, \tau', \mathit{true}>\}) \textbf{ when } \mathit{not}(\mathit{on}) \\
\quad\quad \{<\tau', \tau, \mathit{stop}>\} \cup m' \rightarrow \\
\quad\quad\quad (\mathit{ext}(\alpha)(\tau)(\mathit{false}), \{<\tau, \tau', \mathit{true}>\}) \textbf{ when } \mathit{on} \\
\quad\quad\quad (\mathit{ext}(\alpha)(\tau)(\mathit{on}), \{<\tau, \tau', \mathit{false}>\}) \textbf{ when } \mathit{not}(\mathit{on}) \\
\quad\quad \textbf{else } (\mathit{ext}(\alpha)(\tau)(\mathit{on}), \emptyset) \\
\quad \textbf{end}
\end{array}
$$

8 Conclusion

8.1 Review

The aims of this work are to define a semantic framework which is suitable for current object-oriented design notations. 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 allowing the focus of attention to be placed on semantic rather than syntactic issues. Systems are defined to be behaviours arising 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.

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].

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.

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 [Cla99] 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 [Cla99] gives an example of how an object-oriented design expressed as a collection of λ-function behaviours can be refined.

References

[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.

[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.

[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.

[Cla99] 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.

[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.

[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.

[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.

[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.

[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.

[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.