A Semantics for Object-Oriented Systems

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

1 Introduction

Current object-oriented design notations such as OMT [14], Booch [3] and UML [16] 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 basis which is 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 [1] [15] [9] as a tool to express the equations since this theory provides standard constructions and results which conveniently express semantics without getting unnecessarily entangled in syntactical issues.

Furthermore, the aspects of category theory which we use have a constructive flavour which allows us to program up the results and use them to express systems as non-deterministic specifications. A functional language is used to express the specifications since it supports refinement transformations which are used to produce a deterministic implementation from a specification.

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.

Our 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.

Early results show that our approach is constructive in the sense that we can produce program elements which correspond closely to semantic components. This leads us to believe that we can produce a system development technique which involves mapping elements of design notations to executable components whose semantics are expressed using categorical concepts and whose proof theory and transformation system is expressed in terms of a clear semantic model.

This paper reports early stages of this research and defines a semantic model, outlines how the model can be used as a semantic basis for current object-oriented design notations and gives a simple example of how the model facilitates program development. The paper is structured as follows:

- Sections 2 – 6 define the basic semantic model which we use to express the behaviour of object systems.
- Section 7 outlines how we express properties of systems, how we might establish the properties and how we can view system development as a series of semantic preserving transformations.
- Section 8 outlines how we can generalise the semantic model to include dynamic system behaviour in terms of class instantiation.
- Section 9 describes how we can use the model to interpret the standard features which are employed by typical object-oriented design notations.
- Section 10 gives a constructive interpretation to the categorical tools employed in defining the semantic model and uses them to specify and subsequently implement a very simple object-oriented program.
- Section 11 concludes by describing further work which is planned and compares our approach with related research.

2 Objects

Systems are constructed as a collection of objects. Each object is a separate computational system with its own state which is modified in response to handling messages. A snapshot of an object is (τ, ms, σ) where τ is an object identifier, ms is a sequence of messages, referred to as a *message history* and σ is the state of the object. All objects have a unique object identifier. There is no restriction on the contents of an object state; we may think of it as a tuple of simple data items.

Objects are computational systems which perform calculations in response to receiving messages. A message is a package of information which is sent from one object to another. When the message is handled by the receiver, the result is a change in state (and possibly new objects and further messages). If we observed a single object, it would consume messages over a period of time and possibly change state each time a message is consumed. This gives rise to *object calculations* which are sequences of object states of the form:

$$\ldots \longmapsto (\tau, m : ms, \sigma) \longmapsto (\tau, ms, \sigma') \longmapsto \ldots$$

in which the object with identifier τ consumes the next message m causing its state to change from σ to σ'. All object calculations have a starting state, but may not have an end state if the object continues to exist indefinitely.

Objects have a number of messages which they recognise and can handle. In an object-oriented program these correspond to all the possible legal method calls for an object. Correspondingly, an object can be associated with a set of the possible message histories which it can receive during its life-time. Suppose that we represent such a set of message histories as M, and the set of possible object histories for a deterministic object O as $O(M)$.

A set of message histories M has the following structure: suppose that we select a history $ms \in M$ and then generate all of the history prefixes $prefixes(ms)$; in order for M to be well formed, all of the prefixes $ms' \in prefixes(ms)$ must be also in M, $ms' \in M$. Now, consider two sets of message histories for the same object M_1 and M_2 such that $M_1 \subseteq M_2$, *i.e.* there is an inclusion function $f : M_1 \hookrightarrow M_2$. Given two sets of object calculations $O(M_1)$ and $O(M_2)$, by definition all the calculations in $O(M_1)$ are contained in $O(M_2)$ so $O(f) : O(M_1) \hookrightarrow O(M_2)$.

Now consider non-deterministic objects, or *object specifications*. Given a message history $m : ms$ and an object state $(\tau, m : ms, \sigma)$ there may be many possible transitions:

$$\begin{array}{l} (\tau, m : ms, \sigma) \longmapsto (\tau, ms, \sigma_1) \\ (\tau, m : ms, \sigma) \longmapsto (\tau, ms, \sigma_2) \\ (\tau, m : ms, \sigma) \longmapsto (\tau, ms, \sigma_3) \end{array}$$

The meaning of an object specification $O(M)$ is sets of object calculation sets. Each calculation set $C \in O(M)$ is a complete deterministic description of one possible behaviour of the object O with respect to the message histories M. For any pair of message history sets: $f : M_1 \hookrightarrow M_2$, $O(f) : O(M_1) \hookrightarrow O(M_2)$ where f maps sets of calculations in $O(M_1)$ to sets of calculations in $O(M_2)$.

In conclusion, objects and object specifications are functors from the category of sets of message histories with inclusions as morphisms to the categories of object calculations and object specification calculations. Since the former appears to be a special case of the latter, we will use object specifications from now on.

3 Object Morphisms

Consider two objects O_1 and O_2. Let M_1 and M_2 be two sets of message histories such that $f : M_1 \hookrightarrow M_2$ and therefore $O_1(f) : O_1(M_1) \hookrightarrow O_1(M_2)$ and $O_2(f) : O_2(M_1) \hookrightarrow O_2(M_2)$. An *object morphism* $\gamma : O_1 \rightarrow O_2$ is a family of mappings between object calculations, γ_M, indexed by sets of message histories M. The morphisms are defined so that the following diagram commutes:

$$
\begin{tikzcd}
O_1(M_1) \arrow[r, "\gamma_{M_1}"] \arrow[d, "O_1(f)"'] & O_2(M_1) \arrow[d, "O_2(f)"] \\
O_1(M_2) \arrow[r, "\gamma_{M_2}"'] & O_2(M_2))
\end{tikzcd}
$$

A morphism γ_M will associate each of the calculation sets in $O_1(M)$ with a suitable calculation set in $O_2(M)$. We can think of this as consistently relabelling the object tags, consistently modifying the message history (within the same history class) and consistently modifying the object states. Notice that a mapping must associate *all* calculation sets from $O_1(M)$ with *some* calculation set in $O_2(M)$ and can thus be used as a constraint, $\gamma_M : O_1(M) \rightarrow O_2(M)$ picks out the possible calculation sets from $O_2(M)$ which are allowed by $O_1(M)$.

A sufficient condition for γ to exist between two objects O_1 and O_2 occurs when O_1 represents an extension of the same object. The extension is expressed in terms of the object state by adding extra components, for example when a two-dimensional point is extended with a z co-ordinate to become a three-dimensional point. In such cases, the morphism γ_M is constructed from a projection on the state which simply drops the extra state components, *i.e.* the mapping $(x, y, z) \mapsto (x, y)$.

Consider the case where $M = M_1 = M_2$ but O_2 is only defined with respect to some sub-set of M. This corresponds to the case where O_1 is an extension of O_2 and has added some new methods. In this case the morphism $\gamma_M : O_1(M) \rightarrow O_2(M)$ must (uniquely) associate calculations involving the new messages in $O_1(M)$ with error calculations in $O_2(M)$.

4 Systems

A system of objects is constructed by composing individual objects such that the original objects can be recovered. Consider a pair of completely independent objects O_1 and O_2 which represent sets of possible calculation sets. An *object system* which is made up of these two objects will have calculations which are made up of all pairs of calculations such that the original calculations can be recovered simply by projecting the pairs onto their components:

$$\begin{cases} (\tau_1, m : ms, \sigma) \longmapsto (\tau_1, ms, \sigma) \longmapsto \ldots \\ (\tau_2, m : ms, \sigma) \longmapsto (\tau_2, ms, \sigma) \longmapsto \ldots \end{cases}$$

Alternatively, two objects may be inter-related. In order for this to be the case all calculations for the objects will have the same object identifier. The system will consist of single calculations which are produced by merging corresponding calculations from both objects. For example, if the first object contains a calculation with an object state (τ, ms_1, σ_1), and the second object contains a calculation with an object state (τ, ms_2, σ_2) then the calculations can be merged to produce an object state $(\tau, ms, \sigma_1 \times \sigma_2)$ when $ms = ms_1 = ms_2$ and $\sigma_1 \times \sigma_2$ is the smallest object state containing the information from both σ_1 and σ_2. The original calculations can be recovered by dropping the extra state components in each case. Note that not all the calculations from both "views" of the same object may be in the product since they have to agree on message histories.

Systems of objects can be expressed in terms of object *products*. Given independent objects O_1 and O_2, the product is an object $O_1 \times O_2$ together with two object morphisms $\pi_1 : O_1 \times O_2 \to O_1$ and $\pi_2 : O_1 \times O_2 \to O_2$ such that for any object O with morphisms $f : O \to O_1$ and $g : O \to O_2$ there is a unique arrow $u : O \to O_1 \times O_2$ such that the following diagram commutes:

$$\begin{tikzcd}
& O \arrow[d, "u"] \arrow[ddl, "f"'] \arrow[ddr, "g"] & \\
& O_1 \times O_2 \arrow[dl, "\pi_1"] \arrow[dr, "\pi_2"'] & \\
O_1 & & O_2
\end{tikzcd}$$

We can think of the product as containing calculations whose behaviour is a pairing of component object calculations. Given a collection of independent objects, a product can be constructed pair-wise, forming a single system object which contains the behaviour of its components.

5 System Behaviour

In general, a system is not composed from a collection of completely independent objects. Objects can be used to place restrictions on the behaviour of other objects and to extend the state or message histories of other objects. These interactions are expressed on an object diagram which is a graph whose nodes are labelled with objects and whose edges are labelled with object morphisms.

Suppose that we have an object O which represents sets of object calculation sets. We wish to restrict the collection of O calculations. The restriction can be expressed as an object R_1 with a morphism $\gamma_1 : R_1 \to O$ which picks out just those calculations from O which are allowed by R_1 (possibly after applying some mapping). Given any number of these restrictions, say 7, we can produce a diagram:

$$\begin{tikzcd}
 & R_1 \arrow[d, "\gamma_1"] & R_2 \arrow[dl, "\gamma_2"] \\
R_7 \arrow[r, "\gamma_7"] & O & R_3 \arrow[l, "\gamma_3"] \\
R_6 \arrow[ur, "\gamma_6"] & R_5 \arrow[u, "\gamma_5"] & R_4 \arrow[ul, "\gamma_4"]
\end{tikzcd}$$

The behaviour of the complete system is the object which makes all of the constraints hold. We can think of the restrictions γ_i as simultaneous equations which apply to the object O and the system as a set of calculation sets which is a solution to the equations. In order to solve the equations we use the categorical construct of limits which has been proposed by Goguen in [8] 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 O (which is not necessarily in the diagram) together with, or each object Δ_n a morphism $\gamma_n : O \to \Delta_n$ such that for all edges $e : n \to m$ in Δ the following diagram commutes:

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

A cone on a system expressed as a diagram of objects is a collection of object calculations which contains some of the behaviour of all the objects on the diagram and which observes all of the restrictions on the diagram. Notice however, the there is no condition on *which* particular collection of calculations to pick for the cone. There are many possible choices for cones including an object which contains all possible calculation sets which are consistent with the diagram and also the object which contains no calculations at all (this is an *initial object* which can always be mapped to another object using the empty mapping).

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

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

Cone arrows allow us to place restrictions on cones. For example if there is a cone arrow $\gamma : O' \to O$ we know that the calculations performed by O' are more restricted than those performed by O.

A *limit* on a diagram Δ is a cone O, on Δ such that for any other cone O' on Δ there is a unique cone arrow $u : O' \to O$. The definition of a limit captures the behaviour of a system because it must contain all of the objects on the diagram and must obey all of the constraints on the diagram. Furthermore, a limit contains as many possible calculations without adding any extra state components to objects or changing the messages which objects can handle.

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 [15]. This theorem allows us to construct the limit on a diagram providing that certain simple properties hold.

A *terminal object* in the category of objects is an object with a single possible behaviour for any message history. We could represent this as:

$$(\tau, m : ms, ()) \longmapsto (\tau, ms, ()) \longmapsto \ldots$$

where m and ms are any message and message history respectively. A terminal object represents the degenerate case where there are no objects on a diagram; the resulting behaviour is unobservable in the sense that all messages are ignored.

A product can always be found for two objects since the product either contains pairs of calculations or merges corresponding calculations which arise from observing the same object.

An *equalizer* for two object morphisms is defined as follows. Let $f : O_1 \to O_2$ and $g : O_1 \to O_2$ be two arrows which map calculations performed by O_1 to calculations performed by O_2. An equalizer of f and g is an object O together with an arrow $j : O \to O_1$ such that for any arrow $h : O' \to O_1$ there is a unique arrow $k : O' \to O$ such that the following diagram commutes:

$$\begin{tikzcd}
O \arrow[dr, "j"] & & \\
& O_1 \arrow[r, "{f,g}"] & O_2 \\
O' \arrow[uu, "k"] \arrow[ur, "h"'] & &
\end{tikzcd}$$

An equalizer for object morphisms $f, g : O_1 \to O_2$ is an object O and an object morphism $j : O \to O_1$. The object O can be viewed as a sub-set of O_1 calculations and j is an inclusion such that f and g agree on the sub-set O. The extra component of the definition O' and k force O to be the *largest* sub-set which satisfies the constraint.

In conclusion, we have established that the behaviour of a system of objects is expressed as a limit on a diagram containing the component objects and relationships between them. The limit can be constructed using the categorical constructs of terminal object, product and equializer.

6 Communication

Objects communicate by sending messages. A source object sends a message to a target object via a communication channel. We may think of objects as having two message streams: an input stream (message history) and an output stream. Since elements of the output stream are produced as a result of handing an input message, we may think of the output stream as being part of an object's state; however, we will extend the model of object calculations to isolate the output stream:

$$(\tau, m : ms, \sigma, M) \longmapsto (\tau, ms, \sigma', M')$$

where M and M' are sets of output messages. The messages M' are produced as a result of handling the input message m.

A wide variety of design choices arise when modelling object communication. In particular, we must distinguish between concurrent systems and sequential systems. In a concurrent system, multiple messages may be received by a single object and the order of system execution may be non-deterministic since objects can perform computations at the same time. We will limit the discussion to the description of object systems which will eventually be realised using sequential implementation languages. As a result, we can simplify the definition of output messages, because an object may produce at most a single object:

$$(\tau, m : ms, \sigma, o) \longmapsto (\tau, ms, \sigma', o)$$

where o and o' are single output messages or some unique value denoting no output message (which will be omitted from a state description).

Communication between two objects occurs when the output message produced by the first object is received by the second. The following pair of calculation schemas show how this works:

$$\begin{cases} (\tau_1, m : ms_1, \sigma_1) \longmapsto (\tau_1, ms_1, \sigma_1', o) \longmapsto (\tau_1, ms_1, \sigma_1') \\ (\tau_2, o : ms_2, \sigma_2) \longmapsto (\tau_2, o : ms_2, \sigma_2) \longmapsto (\tau_2, ms_2, \sigma_2') \end{cases}$$

Initially, the object τ_1 is currently active and consumes the next input message m which produces an output message o which we assume has τ_2 as its target. Therefore, the next input message waiting to be consumed by the object τ_2 is o. Note that we are analysing sequential systems so only one object may be active at any given time. Furthermore, we restrict our systems even further by requiring that any output message must be handled by the next system transition. We do not require that message passing is synchronous since this can easily be modelled by sending a return message including result data.

Objects are defined to produce output messages when certain conditions arise in their input. However, the objects do not necessarily know the target of the output message. We require a mechanism which allows us to plug together two objects using a communication medium. This is easily achieved using a system object which controls the flow of messages between objects. Consider the case where we have two objects O_1 and O_2. Both objects produce messages on their output and we wish to connect the output of one object to the input of the other. This is achieved by defining a system object O and two object morphisms f and g:

$$\begin{tikzcd} & O \arrow[dl, "f"'] \arrow[dr, "g"] & \\ O_1 & & O_2 \end{tikzcd}$$

The system object O is a restriction on the freely generated system $O_1 \times O_2$ which requires that whenever an output message is produced by one of the objects it is the next input message to be handled by the other object. We can apply several separate restrictions using this technique and the overall behaviour of the system will require that all the restrictions hold.

7 System Properties

During system development, the software engineer must establish properties and perform system transformations. Current object-oriented design notations provide little support for these activities due to their weak semantics. A category theoretic framework for expressing the behaviour of object-oriented systems facilitates the definition of system properties and transformations. This section gives a flavour of the structures which can be defined in order to support system development.

The behaviour of an object-oriented system is expressed as a limit O on the diagram Δ containing all system objects and morphisms between them. A system property can be expressed as a limit P on a second diagram Δ' which describes just the behaviour which is required by the property. If the system exhibits all of the required behaviour then all of the behaviour expressed by P must be contained in O. Over the message histories M it is possible to associate every calculation in $O(M)$ to some calculation in $P(M)$. This is expressed as an object morphism $\gamma : O \rightarrow P$. If the morphism is *onto*, then the entire property P is true for object O:

$$\begin{tikzcd} O \arrow[r, "\gamma"] \arrow[d, "\eta_n"'] & P \arrow[d, "\eta'_n"] \\ \Delta_n & \Delta'_n \end{tikzcd}$$

Given two properties $\gamma_1 : O \rightarrow P_1$ and $\gamma_2 : O \rightarrow P_2$ then a property which describes those behaviours from P_1 and P_2 which hold in O at the same time is a *pushout* P with object morphisms $\eta_1 : P_1 \rightarrow P$ and $\eta_2 : P_2 \rightarrow P$ which makes the following diagram commute:

$$\begin{tikzcd} O \arrow[r, "\gamma_1"] \arrow[d, "\gamma_2"'] & P_1 \arrow[d, "\eta_1"] \\ P_2 \arrow[r, "\eta_2"'] & P \end{tikzcd}$$

During development, a system description is refined by applying modifications. Assuming that the process does not go wrong and therefore no backtracking is required, there are two major types of refinement modification which can be applied to a system O_1 to produce O_2. Firstly, extra properties may be added to a system for example by extending the state components in objects. In this case, each of the calculations which are contained in O_2 can be transformed into a calculation in O_1 by forgetting the extra state, therefore there is an object morphism $\gamma : O_2 \rightarrow O_1$. Secondly, a system may be made more deterministic by removing unwanted calculations in which case all of the calculations which are present in O_2 will also be present in O_1 and therefore there exists an object morphism $\gamma : O_2 \rightarrow O_1$.

8 Classes

A *class* is a collection of objects each of which has the same behaviour but a different identity. An object is brought into existence by instantiating a class; class instantiation can occur at any time during the life-time of a system and is performed by an object making a request for a new object of a certain type. Classes are added to the system model described so far by a simple generalisation as follows.

Rather than build systems out of individual objects we can generalise our model and construct systems out of classes. Just as an object denotes all of the possible calculations which it can perform, a class denotes all of the possible objects which it can create. We assume that all possible objects exist, but that an object can only start to compute after another object has caused it to come into existence using a message *new*. A system always starts with a single active object. All other objects must be created as a result of messages initiated by the root object.

The following three objects calculations show how class instantiation is modelled. Initially, object τ_1 is active; it creates a new object τ_2 and sends it a message m_1 then becomes dormant. Object τ_2 becomes active and handles the message m_1 when it is sent and then instantiates the object τ_2 before sending it a message m_2 and becoming dormant. Finally, object τ_3 handles message m_2.

$$(\tau_1, new(\tau_2) : m_1 : ms_1, \sigma_1) \longmapsto (\tau_1, m_1 : ms_1, \sigma_1) \longmapsto (\tau_1, ms_1, \sigma_1) \longmapsto \ldots$$

$$\begin{array}{l}
(\tau_2, m_1 : new(\tau_3) : m_2 : ms_2, \sigma_2) \longmapsto \\
\quad (\tau_2, m_1 : new(\tau_3) : m_2 : ms_2, \sigma_2) \longmapsto \\
\quad\quad (\tau_2, new(\tau_3) : m_2 : ms_2, \sigma_2') \longmapsto \\
\quad\quad\quad (\tau_2, m_2 : ms_2, \sigma_2') \longmapsto \\
\quad\quad\quad\quad (\tau_2, ms_2, \sigma_2') \longmapsto \ldots
\end{array}$$

$$\begin{array}{l}
(\tau_3, m_2 : ms_3, \sigma_3) \longmapsto \\
\quad (\tau_3, m_2 : ms_3, \sigma_3) \longmapsto \\
\quad\quad (\tau_3, m_2 : ms_3, \sigma_3) \longmapsto \\
\quad\quad\quad (\tau_3, m_2 : ms_3, \sigma_3) \longmapsto \\
\quad\quad\quad\quad (\tau_3, m_2 : ms_3, \sigma_3) \longmapsto \\
\quad\quad\quad\quad\quad (\tau_3, ms_3, \sigma_3') \longmapsto \ldots
\end{array}$$

9 Design Notations

The model for object-oriented systems described in this paper can be used to give a meaning to current object-oriented design notations. Such notations often provide a collection of models which are used to express information about both the static and dynamic behaviour of systems. However, the notations are often defined without a clear indication of the underlying object-oriented model and so it is difficult to see how they can be used with any rigour. The rest of this section gives an outline of how the semantics described in this paper can used as the basis for providing a meaning for these notations and thereby provide a framework for the analysis of object-oriented designs.

- Object-oriented notations often provide a static modelling language which expresses the structure of classes in terms of their interfaces. The semantics of such *static models* is given as freely generated classes (see §8) where messages can make arbitrary modifications to object state as described in §2.

- Classes in static models are often organised into generalisation hierarchies where a sub-class is linked to its super-class. These links correspond to morphisms (see §3) between classes such that the calculations performed by sub-class instances are equivalent to the calculations performed by super-class instances after a projection.

- Classes are linked using *associations* which indicate that instances of the classes may communicate during system execution. Such associations correspond to freely constructed systems of objects which communicate as described in §6.

- The dynamic behaviour of object-oriented systems are often expressed using models which express the traces of messages between objects. These are restrictions on the calculations which may be performed by objects in a system as described in §5. Unlike current design notations, the semantic model proposed in this paper explains how individual dynamic descriptions can be composed to produce an overall system behaviour.

- The dynamic behaviour of a class of objects is often expressed using state transition models. These models are used to restrict the sequence of messages which may legally be handled by an object and to express the state changes which occur as a result of handling a message in a particular state. Our model has parameterised object behaviours with respect to message histories and therefore provided a concrete framework within which message restrictions can be expressed and studied.

10 Program Development

This section shows how the proposed model of object-oriented system specification and development can be used to produce a program using step-wise refinement. The approach validates the use of the categorical machinery by showing how a system can be constructed from reusable definitions for categorical product and equalizer.

The example system defines a simple class for two-dimensional points. We start by defining the behaviour of the system which corresponds to just the interface of $Point$, *i.e.* all instances of $Point$ arbitrarily change state in response to receiving any message.

Next, we define two independent constraints on the behaviour. Firstly, R_1 defines that when a $Point$ object receives a message $setx$ it must appropriately set the value of the x co-ordinate. Secondly, R_2 places a similar constraint on receipt of a $sety$ message.

We require that the behaviour of $Point$ objects in the complete system satisfy both constraints at the same time. This is achieved by constructing a limit on the diagram containing the definitions of $Point$, R_1 and R_2. We will show that we can define procedures for constructing products and equalizers and then use these to construct a limit. The situation is shown in the following diagram where X is a product of R_1 and R_2 and L is constructed by finding an equalizer for arrows $f \circ \pi_1$ and $g \circ \pi_2$:

$$
\begin{tikzcd}
& Point & \\
R_1 \arrow[ur, "f"] & & R_2 \arrow[ul, "g"'] \\
& X \arrow[ul, "\pi_1"] \arrow[ur, "\pi_2"'] & \\
& L \arrow[u, "i"'] &
\end{tikzcd}
$$

We start the program development with underspecified objects. This is achieved using a simple functional programming language. The language is standard except for the following features:

- Underspecification is achieved using the non-deterministic operator *sel* which arbitrarily selects an element from a set. The operator *fail* is used to induce failure.
- Objects are tagged and untagged with unique labels using the operators *seal* and *unseal*.

- Objects are represented as tagged records where $\oplus$ is used to concatenate records. Name lookup in records shadows on the right and the concatenation operator can be undone.

The class *Point* has a state x and y and a message interface containing *setx* and *sety*. In the following specification, the tag τ is supplied each time the class is instantiated. Each instance of *Point* has a different tag. Note that *Point* is underspecified since the methods use *sel* to select arbitrary integer co-ordinates.

$$\begin{aligned} \mathit{Point}(\tau)(n_1, n_2) = \mathit{seal}(\tau)\{ & x = n_1; \\ & y = n_2; \\ & \mathit{setx}(x) = \mathit{Point}(\tau)(\mathit{sel}(N), \mathit{sel}(N)); \\ & \mathit{sety}(y) = \mathit{Point}(\tau)(\mathit{sel}(N), \mathit{sel}(N))\} \end{aligned} \tag{1}$$

Each restriction is defined as a separate class which extends the definition of *Point* in order to restrict the appropriate method:

$$R_1(\tau)(n_1, n_2) = (\mathit{Point}(\tau)(n_1, n_2)) \oplus (\mathit{seal}(\tau)\{\mathit{setx}(x) = R_1(\tau)(n_1 + x, n_2)\}) \tag{2}$$

$$R_2(\tau)(n_1, n_2) = (\mathit{Point}(\tau)(n_1, n_2)) \oplus (\mathit{seal}(\tau)\{\mathit{sety}(y) = R_2(\tau)(n_1, y + n_2)\}) \tag{3}$$

We define morphisms f and g from the extensions to the class *Point*. Each morphism recovers the original definitions of *setx* and *sety* by peeling back the extra method layer:

$$f(\tau)(o) = \mathbf{let}\ r \oplus \{\mathit{setx} = \_\} = \mathit{unseal}(\tau)(o)\ \mathbf{in}\ \mathit{seal}(\tau)(r) \tag{4}$$

$$g(\tau)(o) = \mathbf{let}\ r \oplus \{\mathit{sety} = \_\} = \mathit{unseal}(\tau)(o)\ \mathbf{in}\ \mathit{seal}(\tau)(r) \tag{5}$$

The system is now complete and is described by the diagram containing *Point*, R_1, R_2 and the arrows f and g. In order to complete the system we must find a limit on this diagram. In order to do this we can construct a product $X = R_1 \times R_2$ and then find an equalizer of the arrows $f \circ \pi_1$ and $g \circ \pi_2$. Consider constructing a product of two class definitions:

$$\begin{aligned} & \mathit{product}(c_1, c_2) = (c, \pi_1, \pi_2) \\ & \quad \mathbf{where} \\ & \qquad c(\tau_1, \tau_2)(v_1, v_2) = \mathbf{let}\ \begin{aligned} o_1 &= c_1(\tau_1)(v_1) \\ o_2 &= c_2(\tau_2)(v_2) \end{aligned} \\ & \qquad\qquad\qquad\qquad\quad\ \mathbf{in}\ (o_1, o_2) \\ & \qquad \pi_1(\tau)(o_1, o_2) = o_1 \\ & \qquad \pi_2(\tau)(o_1, o_2) = o_2 \end{aligned} \tag{6}$$

The product of two classes produces a triple containing a class and two projection arrows. Note that if the product class "fuses" together two parts of the same larger class then we can simplify the product by merging the tags τ_1 and τ_2 and instances o_1 and o_2.

An equalizer of two arrows is constructed using the following program:

$$
\begin{array}{l}
\mathit{equalize}(c, a_1, a_2) = (c', a) \\
\quad \textbf{where} \\
\quad\quad c'(\tau)(v) = \textbf{let } o = c(\tau)(v) \\
\quad\quad\quad\quad\quad\quad \textbf{in if } a_1(\tau)(o) = a_2(\tau)(o) \\
\quad\quad\quad\quad\quad\quad\quad \textbf{then } o \\
\quad\quad\quad\quad\quad\quad\quad \textbf{else } \mathit{fail} \\
\quad\quad a(\tau)(o) = o
\end{array}
\tag{7}
$$

The definition of *equalize* constructs a new class c' whose instances are just those instances of c which produce the same object under morphisms a_1 and a_2. We can now construct the limit on the diagram using the following construction:

$$
\begin{array}{l}
(L, i) = \mathit{equalize}(X, f \circ \pi_1, g \circ \pi_2) \\
\quad \textbf{where} \\
\quad\quad (X, \pi_1, \pi_2) = \mathit{product}(R_1, R_2)
\end{array}
\tag{8}
$$

The limit is a non-deterministic program which can be thought of as performing generate-and-test when used to construct instances. Although the limit has an executable semantics, it is an *infeasible program* due to the amount of backtracking necessary. We shall now show how program transformation can be used to produce a feasible deterministic implementation of the limit on the diagram.

From 8 we note that the arrow i is not required for the purposes of generating a class definition so we omit it. We unfold *product* and *equalize* using their definitions 6 and 7 to produce 9.

$$
\begin{array}{l}
L(\tau)(v) = \textbf{let } o = X(\tau)(v) \\
\quad\quad\quad\quad \textbf{in if } (f \circ \pi_1)(\tau)(o) = (g \circ \pi_2)(\tau)(o) \\
\quad\quad\quad\quad\quad \textbf{then } o \\
\quad\quad\quad\quad\quad \textbf{else } \mathit{fail}
\end{array}
\tag{9}
$$

From 9 we unfold X using its definition from *product* to produce 10.

$$
\begin{array}{l}
L(\tau_1, \tau_2)(v_1, v_2) = \textbf{if } f(\tau_1)(R_1(\tau_2)(v_1)) = g(\tau_2)(R_2(\tau_2)(v_2)) \\
\quad\quad\quad\quad\quad\quad\quad\quad \textbf{then } (R_1(\tau_1)(v_1), R_2(\tau_2)(v_2)) \\
\quad\quad\quad\quad\quad\quad\quad\quad \textbf{else } \mathit{fail}
\end{array}
\tag{10}
$$

From 10 we unfold f and g using their definitions 4 and 5. In order to do this we introduce two new local variables o_3 and o_4 in 11.

$$
\begin{array}{l}
L(\tau_1, \tau_2)(v_1, v_2) = \\
\quad \textbf{let } o_1 = R_1(\tau_1)(v_1) \\
\quad \phantom{\textbf{let }} o_2 = R_2(\tau_2)(v_2) \\
\quad \textbf{in let } o_3 = \textbf{let } r \oplus \{setx = \_\} = unseal(\tau_1)(o_1) \textbf{ in } seal(\tau_1)(r) \\
\quad \phantom{\textbf{in let }} o_4 = \textbf{let } r \oplus \{sety = \_\} = unseal(\tau_2)(o_2) \textbf{ in } seal(\tau_2)(r) \\
\quad\quad \textbf{in if } o_3 = o_4 \\
\quad\quad\quad \textbf{then } (o_1, o_2) \\
\quad\quad\quad \textbf{else } fail
\end{array}
\tag{11}
$$

From 11 we note that $o_3 = o_4$ only under the circumstances that $\tau_1 = \tau_2$ so we can replace the tags with a single tag τ. We unfold R_1 and R_2 from their definitions 2 and 3 to produce 12.

$$
\begin{array}{l}
L(\tau)((n_1, n_2)(m_1, m_2)) = \\
\quad \textbf{let } o_1 = (Point(\tau)(n_1, n_2)) \oplus (seal(\tau)\{setx(x) = R_1(\tau)(n_1 + x, n_2)\}) \\
\quad \phantom{\textbf{let }} o_2 = (Point(\tau)(m_1, m_2)) \oplus (seal(\tau)\{sety(y) = R_2(\tau)(m_1, m_2 + y)\}) \\
\quad \textbf{in let } o_3 = \textbf{let } r \oplus \{setx = \_\} = unseal(\tau_1)(o_1) \textbf{ in } seal(\tau_1)(r) \\
\quad \phantom{\textbf{in let }} o_4 = \textbf{let } r \oplus \{sety = \_\} = unseal(\tau_2)(o_2) \textbf{ in } seal(\tau_2)(r) \\
\quad\quad \textbf{in if } o_3 = o_4 \\
\quad\quad\quad \textbf{then } (o_1, o_2) \\
\quad\quad\quad \textbf{else } fail
\end{array}
\tag{12}
$$

From 12 we can see the definition of $Point$ requires that $o_3 = o_4$ only if $(n_1, n_2) = (m_1, m_2)$ so we can replace these pairs with a single pair. Furthermore, the non-deterministic method definitions $setx$ and $sety$ from $Point$ are shadowed by deterministic definitions from R_1 and R_2 respectively. Therefore, we may drop these components from $Point$ in the definition of L given in 13.

$$
\begin{array}{l}
L(\tau)(n_1, n_2) = \\
\quad \{x = n_1; \\
\quad\; y = n_2; \\
\quad\; setx(x) = L(\tau)(n_1 + x, n_2); \\
\quad\; sety(y) = L(\tau)(n_1, y + n_2)\}
\end{array}
\tag{13}
$$

11 Conclusion

The aims of this research are to identify a semantic model which is suitable for object-oriented software development and particularly suitable for analysing object-oriented designs.

We have laid the foundations of achieving this aim by describing a model of object-oriented systems based on objects as computational entities. We have shown that this model is compositional and how it can be used to establish system properties and to define system refinement. We have outlined how the model can be used to provide a semantic framework for the major features which are found in current object-oriented design notations.

We have chosen to use category theory to express the semantic model because this allows us to focus on semantic issues whilst leaving open issues relating to the notation used to express the model. Such a notation may be logical such as a modal logic or first order such as VDM and Z. Alternatively, a suitable notation may be a calculus based on λ-notation.

We have provided a very simple example program refinement which shows how the model may be put into practice. The example shows the utility of using standard results from category theory since we have been able to "code up" the categorical constructs and use them to produce a program.

The next stage of this research is to investigate how the techniques scale-up by using them to develop some small-scale object-oriented applications. Also, we would like to investigate a close coupling between object-oriented design notations such as UML and the proposed semantic model since this will produce a mechanism for analysing the UML designs.

The analysis and semantic foundations of object-oriented designs and development is currently an active research area [13]. For example [5] shows how message diagrams (equivalent to UML collaboration diagrams) can be given a semantics in terms of a partial order on events; [4] shows how the specification language Larch can be used to give a formal semantics to static object diagrams; and, [12] can be used to produce executable object-oriented designs.

Our approach contributes to this area by identifying a potentially fruitful collaboration of ideas. The use of category theory to capture the essential characteristics of systems dates back to Goguen [10] who updated the approach to address object-oriented systems in [8]. We take a slightly less abstract view in that we wish to focus on the development of sequential object-oriented programs rather than capturing general object-oriented behaviour.

A related approach which addresses object-oriented system execution is the use of modal logics; examples are Object Calculus [2], [6] and [11]. This approach differs from that which is taken in this work 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 [7] 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.

References

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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