Object-oriented refinement and proof using behaviour functions.

CLARK, Anthony <http://orcid.org/0000-0003-3167-0739>

Available from Sheffield Hallam University Research Archive (SHURA) at:
http://shura.shu.ac.uk/11931/

This document is the author deposited version. You are advised to consult the publisher's version if you wish to cite from it.

Published version


Copyright and re-use policy

See http://shura.shu.ac.uk/information.html
Object-Oriented Refinement and Proof using Behaviour Functions

Tony Clark
Department of Computing, University of Bradford
West Yorkshire, BD7 1DP, UK
a.n.clark@scm.brad.ac.uk

Abstract

This paper proposes a new calculus for expressing the behaviour of object-oriented systems. The semantics of the calculus is given in terms of operators from computational category theory. The calculus aims to span the gulf between abstract specification and concrete implementation of object-oriented systems using mathematically verifiable properties and transformations. The calculus is compositional and can be used to express the behaviour of partial system views. The calculus is used to specify, analyse and refine a simple case study.

1 Introduction

In [Gog75], [Ehr91] and [Gog90] Goguen et al. propose an abstract model of object systems based on standard constructions in Category Theory. They show how to use the constructions to build systems but do not propose a calculus for expressing and reasoning about them. In [Cla99a], [Cla99b] and [Cla99c] a calculus is proposed for expressing object systems based on Goguen’s work. The calculus was shown to support incremental system development based on features of Computational Category Theory [Ryd88]. The calculus does not have a formal semantics and therefore its link to the abstract object model is weak.

This paper develops a formal semantics for the $\lambda_o$-calculus. The semantics encodes the required categorical constructions as builtin operators and then uses them to express a number of features of object-oriented systems development including: (under-)specification; refinement; encapsulation; invariant properties.

This work contributes to the area of object-oriented systems development by providing a rigorous framework within which aspects of development can be defined and explored. In particular the $\lambda_o$-calculus aims to span the gulf between abstract specification and concrete implementation using mathematically verifiable refinement transformations. The $\lambda_o$-calculus can express partial views of a system and is therefore suitable as the basis for a semantics of UML [UML98] and as such can be seen as an extension of, or complimentary to, [Cla97] [Eva98] [Eva99] and [Lan98].

The builtin operators of the $\lambda_o$-calculus arise from Computational Category Theory. The reader is directed to [Bar90], [Ryd88] and [Gog89] for definitions of the appropriate constructs and to [Cla99a] for a discussion of how these constructs are used in the development of object-oriented systems.

This work differs from other approaches with similar aims. The Object Calculus [Bic97] uses similar categorical constructs but uses a logic rather than a $\lambda$-calculus to express models. Following Goguen, we propose that the behaviour of a system is a limit on a diagram of behaviours; diagrams are also used in [Ken99] [Ken97] where the aim is to express logical properties of data. Other calculi have been proposed as the basis for object-oriented systems, notably those defined in [Abu98]. The $\lambda_o$-calculus differs from other calculi in that it can express partial views of a system, incorporates non-determinism, solve constraints via equalizers and has a builtin notion of refinement via refinement morphisms.

The paper is structured as follows: section 2 gives an overview of the semantic model used for object-oriented systems; section 3 defines the $\lambda_o$-calculus used to express the model; section 4 defines a simple system requirements
that is used to demonstrate features of object-oriented system development using the $\lambda_o$-calculus in the rest of the paper; section 5 shows how a system invariant is verified; section 6 shows how mutual constraints can be achieved by composing sub-systems; section 7 shows how object-oriented encapsulation can be achieved using the refinement; finally sections 8 and 9 show how refinement achieves concrete data representation and an implementation in Java.

2 Behavioural Object-Oriented Model

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

The computation performed when a message is handled by an object depends on the object’s current state and causes the object to change state and produce output messages. If we observed an object over a period of time we would see a sequence of messages and state changes: $\ldots \sigma_1 \xrightarrow{(I_1,O_1)} \sigma_2 \xrightarrow{(I_2,O_2)} \sigma_3 \ldots$ where each $\sigma_j$ 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 $\sigma_j$ receiving messages $I_j$ causing a state change to $\sigma_{j+1}$ 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 $(t_1, t_2, v)$ where $t_1$ identifies the source object, $t_2$ identifies the target object and $v$ is the message data. Where any of the message components may be inferred from context they are elided.

Object systems are constructed from multiple objects interacting by passing messages. The state of an object system is a set of object states $S$. Computation in an object system occurs when the messages in set $I$ are sent to the objects in $S$ producing a new set of object states $S'$ and a collection of output messages $O$: $\ldots \xrightarrow{\{} I \xrightarrow{(O)} S \xrightarrow{\{} S' \xrightarrow{\{} \ldots$ Object-oriented designs represent non-deterministic computational systems. Object calculations are represented as a calculation graph where the nodes of the graph are labelled with sets of states and the edges are labelled with pairs of input and output message sets.

Object system calculations can be transformed by graph homomorphisms. Such transformations can be used as the basis of system composition operations based on graph products and coproducts. Equalizers can be used to constructively find equivalence proofs expressed in terms of graph homomorphisms. The behaviour of a system is expressed as a limit on a diagram consisting of calculation graphs and graph homomorphisms. System properties can be expressed by adding the required behaviour to the diagram and then showing that the limit is preserved.

The rest of this paper uses these features as the semantic basis of a calculus for expressing, verifying and transforming object-oriented system designs.

3 The $\lambda_o$-Calculus

The $\lambda_o$-calculus is a notation for expressing object-oriented system designs. It is a standard normal order $\lambda$-calculus [Han94] [Plo75] extended with built-in operators [Lan64] for constructing behaviour functions in terms of behaviour products, coproducts, equalizers and morphisms.

The syntax of the $\lambda_o$-calculus is given in figure 1. The semantics of the basic calculus is given as a convertibility relation between terms in appendix A. All $\lambda_o$-terms have a type given by the type theory defined in appendix A. The following sugar $e_1$ where $e_1 = e_2$ is translated as $(\lambda v.e_1)(\nu v.e_2)$. The following sugar case $e_1$ of $\ldots$ else $e_2$ end is translated as case $e_1$ of $\ldots$ $v$ -> $e_2$ end.

3.1 Object Calculations and Morphisms

An object calculation is a sequence of object state transitions caused as a result of a collection of objects receiving messages, changing state and sending messages. Given an object with identity $t$, state $v$ and behaviour $e$, if the object receives messages $I$, changes state and behaviour to $v'$ and $e'$, and produces output messages $O$ then $e \{(t,v)\} I = (e' \{(t,v')\}, O)$. 

Rigorous Object-Oriented Methods 2000
Object-Oriented Refinement and Proof using Behaviour Functions

\[
\begin{align*}
  e ::=& \text{expressions} \\
  v ::=& \text{variable} \\
  \lambda v.e ::=& \text{function} \\
  ee ::=& \text{application} \\
  \mu w.e ::=& \text{recursive definition} \\
  (e, \ldots, e) ::=& \text{tuple} \\
  ce. \ldots e ::=& \text{structure} \\
  \textbf{case} e \textbf{ of} \ p \to e; \ldots; p \to e \textbf{ end} ::=& \text{selection} \\
  (e,e)e ::=& \text{object morphism}
\end{align*}
\]

\[
\begin{align*}
  p ::=& \text{patterns} \\
  v ::=& \text{variable} \\
  cp \ldots p ::=& \text{structure} \\
  (p, \ldots p) ::=& \text{tuple}
\end{align*}
\]

\[
\begin{align*}
  v ::=& \text{values} \\
  \lambda w.e ::=& \text{function} \\
  ce. \ldots e ::=& \text{structure} \\
  (e, \ldots, e) ::=& \text{tuple} \\
  n ::=& \text{integer} \\
  t ::=& \text{tag}
\end{align*}
\]

\[
\begin{align*}
  \tau ::=& \text{types} \\
  B ::=& \text{basic type} \\
  T ::=& \text{tag type} \\
  V ::=& \text{all values} \\
  \{ \tau \} ::=& \text{set of } \tau \\
  (\tau, \ldots, \tau) ::=& \text{tuple type} \\
  [\tau] ::=& \text{sequence of } \tau \\
  \tau \to \tau ::=& \text{function type} \\
  \tau \odot \tau ::=& \text{disjoint union}
\end{align*}
\]

Figure 1: Syntax of $\lambda_a$

\[
\begin{align*}
  e \{(t, v)\} I = (e', S, O) \\
  e \{(t, \text{env } v)\} \overset{[I(t,v)]}{\longrightarrow} e' S
\end{align*}
\]

\[
\begin{align*}
  e S \overset{1}{\longrightarrow} e S \\
  e_1 S_1 \overset{m_1}{\longrightarrow} e_2 S_2 \\
  e_2 S_2 \overset{m_2}{\longrightarrow} e_3 S_3 \\
  e_1 S_1 \overset{m_1 + m_2}{\longrightarrow} e_3 S_3 \\
  e_1 S_1 \overset{m}{\longrightarrow} e_2 S_2 \\
  e_3(\phi_1(S_2)) \overset{\phi_2(m)}{\longrightarrow} e_4(\phi_1(S_2)) \forall S_1, m \\
  (\phi_1, \phi_2)e_1 = e_3
\end{align*}
\]

Figure 2: Object Calculations
A pre-system behaviour, of type $P$ is a function that expects to be supplied with a set of states. The result is a system behaviour of type $O$ that expects a set of input messages and produces a replacement system behaviour and a set of output messages. A system state of type $\Sigma$ is either a single object state or a pair of system states. A system message of type $M$ is either a set of object messages or a pair of system messages. System behaviour types are defined below:

\[
P = \{\Sigma\} \rightarrow (O, M)
\]

\[
O = M \rightarrow (O, M)
\]

\[
\Sigma = (T, V) \oplus (\Sigma, \Sigma)
\]

\[
M = \{(T, T, V)\} \oplus (M, M)
\]

Object calculations are represented by the transition relation $\vdash$ which is defined in figure 2. Each transition is labelled with sequences of trees of input output messages. The operator `env` associates all atomic state values with a name relative to a given behaviour function, the result is a partial function from names to values. An object calculation $e : S \xmapsto{m} e'$ is well formed when all output messages produced by each transition are input messages in the next transition.

Object calculation morphisms are pairs of functions $(\phi_1, \phi_2)$ such that $\phi_1$ is a mapping between object states and $\phi_2$ is a mapping between sequences of input output messages. Such a morphism can be applied to a behaviour function $e$ in $\lambda_n$ to produce a new function $(\phi_1, \phi_2)e$ whose behaviour is given in terms of a mapping on $e$-calculations as shown in figure 2. Composition of object calculation morphisms is defined component-wise as follows: $(\phi_1, \phi_2) \circ (\phi_3, \phi_4) = (\phi_1 \circ \phi_3, \phi_2 \circ \phi_4)$. The type of a calculation morphism is $\Phi = (\{\Sigma\} \rightarrow \{\Sigma\}, [M] \rightarrow [M])$.

### 3.2 Constructing Systems

The state of a system of objects is a set of binary trees. The leaves of each tree are labelled with object states. Views of the same object may occur at different leaves in the tree providing that they are consistent. A system state $S$ is consistent $\vdash S$ when it is a set of possible states for the same object $t: \vdash \bigcup_{i=1,n}\{(t, \rho_i)\}$, when it is the composition of two different object states: $\vdash \bigcup_{i=1,n}\{(t_1, \rho_i)\} \times \bigcup_{j=1,m}\{(t_2, \rho_j)\}$ such that $t_1 \neq t_2$, when it is the composition of two views of the same object such that attribute names occurring in both have the same values: $\vdash \bigcup_{i=1,n}\{(t_1, \rho_i)\} \times \bigcup_{j=1,m}\{(t_2, \rho_j)\}$ when $\rho_i(n) = \rho_j(n)$ for all $i, j, n \in \text{dom}(\rho_i) \cap \text{dom}(\rho_j)$, and finally when pair-wise decomposition of the state is well formed: $\vdash S_1 \times S_2 \times S_3$ when $\vdash S_1 \times S_2$, $\vdash S_1 \times S_3$ and $\vdash S_2 \times S_3$.

Systems are constructed from objects using the operators $\times : O \rightarrow O \rightarrow (O, \Phi, \Phi), \ast : O \rightarrow O \rightarrow (O, \Phi, \Phi)$ and $\text{eq} : O \rightarrow \Phi \rightarrow \Phi \rightarrow (O, \Phi)$. The semantics of these operators is given in terms of system calculations. Operator $\times$ is used to construct a system from its components, operator $\ast$ is used to construct alternative possible behaviours and eq...
is used to express system constraints. System construction is defined using the operators in figure 3 where ⊕ is disjoint set union and zip merges pairs of sequences to produce sequences of pairs. The operators prod : O → O → O and coprod : O → O → O are used to construct products and coproducts consisting of behaviour functions and associated behaviour morphisms. They are defined by extending the λ₀-convertibility relation as follows:

\[ \pi_1(\text{prod } e_1 e_2) = e_1 \]
\[ \pi_2(\text{prod } e_1 e_2) = e_2 \]
\[ e_1 \times e_2 = (\text{prod } e_1 e_2, \pi_1, \pi_2) \]
\[ t_1 e_1 = \text{coprod } e_1 e_2 \]
\[ t_2 e_2 = \text{coprod } e_1 e_2 \]
\[ e_1 + e_2 = (\text{coprod } e_1 e_2, t_1, t_2) \]

The system construction operator \( \text{prod} \) is used to express system constraints. System construction is defined using the operators in figure 3 where \( \oplus \) is disjoint set union and zip merges pairs of sequences to produce sequences of pairs. The operators \( \text{prod} : O \rightarrow O \rightarrow O \) and \( \text{coprod} : O \rightarrow O \rightarrow O \) are used to construct products and coproducts consisting of behaviour functions and associated behaviour morphisms. They are defined by extending the \( \lambda_0 \)-convertibility relation as follows:

The following proof uses the transition semantics of both sides of the equation to show that they are equivalent. From

\[ \pi_1(\text{prod } e_1 e_2) = e_1 \]
\[ \pi_2(\text{prod } e_1 e_2) = e_2 \]
\[ e_1 \times e_2 = (\text{prod } e_1 e_2, \pi_1, \pi_2) \]
\[ t_1 e_1 = \text{coprod } e_1 e_2 \]
\[ t_2 e_2 = \text{coprod } e_1 e_2 \]
\[ e_1 + e_2 = (\text{coprod } e_1 e_2, t_1, t_2) \]

The theory \( \lambda_0 \) is extended with equivalences for the underlying operators \( \text{prod} \), \( \text{coprod} \) and \( \text{eq} \). In each case a one step transition defines term equivalence, for example:

\[ (\text{prod } e_1 e_2) S \xrightarrow{\{I_{O}\}} e_3 S' \]
\[ \text{prod } e_1 e_2 S = (e_3 S', O) \]

Products and coproducts must observe some simple algebraic properties given in the following theorems.

**Theorem 1** Let \( e_1(t) \) be a terminal object behaviour defined: \( e_1(t)(*) = (e_1(t)(*), \emptyset) \). Then, \( e \times e_1(t) = e = e_1(t) \times e \).

The following proof shows that there exists an isomorphism between \( e \) and \( e \times e_1(t) \) (and equivalently \( e \) and \( e_1(t) \times e \)). Define an object morphism \( (\phi_1, \phi_2) \) as:

\[ \phi_1(e \times e_1(t), e) \]
\[ \phi_2(e) \]
\[ \phi_2(m) = \begin{cases} [I, O] & \text{when } m = [I, O] \land (e \times e_1(t), e) = \emptyset \land m = [I, O]. \\
\phi_2(m_1) \lor \phi_2(m_2) & \text{when } m = m_1 \lor m_2 \land m = [I, O]. \\
\phi_2(m_1) \land \phi_2(m_2) & \text{when } m = m_1 \land m_2 \land m = [I, O]. \end{cases} \]

The inverse \( (\phi_1^{-1}, \phi_2^{-1}) : e \rightarrow e \times e_1(t) \) is well defined and therefore: \( (\phi_1^{-1}, \phi_2^{-1}) \circ (\phi_1, \phi_2) = \text{Id}_e \) and \( (\phi_1, \phi_2) \circ (\phi_1^{-1}, \phi_2^{-1}) = \text{Id}_{e \times e_1(t)} \). A similar argument is used to show that \( e \times e_1(t) \) is isomorphic to \( e_1(t) \times e \). QED.

**Theorem 2** System composition is associative, i.e. \( (e_1 \times e_2) \times e_3 = e_1 \times (e_2 \times e_3) \)

The following proof sketch shows that the required isomorphism exists. Firstly define an object morphism \( (\phi_1, \phi_2) : (e_1 \times e_2) \\ e_3 \rightarrow e_1 \times (e_2 \times e_3) :\)

\[ \phi_1(\{(v_1, v_2), v_3\}) = \{(v_1, (v_2, v_3)\} \]
\[ \phi_2(m) = \begin{cases} [I] & \text{when } m = [I] \land \phi_1(\{(v_1, v_2), v_3\}) = \emptyset \land m = [I]. \\
\phi_2(m_1) \lor \phi_2(m_2) & \text{when } m = m_1 \lor m_2 \land m = [I]. \\
\phi_2(m_1) \land \phi_2(m_2) & \text{when } m = m_1 \land m_2 \land m = [I]. \end{cases} \]

It is straightforward to define \( (\phi_1^{-1}, \phi_2^{-1}) \). QED.

**Theorem 3** The system construction operator \( \times \) distributes over +, i.e. \( e_1 \times (e_2 + e_3) = (e_1 \times e_2) + (e_1 \times e_3) \). The following proof uses the transition semantics of both sides of the equation to show that they are equivalent. From

\[ e_1 S_1 \rightarrow e_1' S_1' \]
\[ e_2 S_2 \rightarrow e_2' S_2' \]
\[ e_3 S_3 \rightarrow e_3' S_3' \]

we get the following:

\[ (e_1 \times (e_2 + e_3))(S_1 \times (S_2 \oplus S_3)) \rightarrow (e_1' \times (e_2' + e_3'))(S_1' \times (S_2' \oplus S_3')) \]
\[ (e_1 \times e_2 + (e_1 \times e_3))(S_1 \times (S_2 \oplus S_3)) \rightarrow (e_1' \times e_2' + (e_1' \times e_3'))((S_1') \times (S_2' \oplus S_3')) \]

Since \( P \times (Q \oplus R) = ((P \times Q) \oplus (P \times R)) \) for any sets \( P, Q, R \) then we conclude that the two calculations given above are equivalent. QED.
3.3 System Refinement

System development through step-wise refinement is attractive since it allows abstract models to be developed early in the life-cycle and then refined to concrete implementations through a series of verified transformations. Consider two behaviour functions \( e_1 \) and \( e_2 \) such that \( e_2 \) is a more concrete version of \( e_1 \). Typically, the states of \( e_2 \) will be related to those of \( e_1 \) but will involve more components and inter-relationships. For example, object-oriented design promotes the use of encapsulation whereby structured data is implemented as a collection of objects whose detail is hidden behind method interfaces. The calculations of \( e_1 \) will be more abstract than those of \( e_2 \); \( e_1 \) may perform complex tasks in a single computation step whereas \( e_2 \) must observe implementation constraints imposed by the target system.

If \( e_1 \) is an abstract version of the required system behaviour and \( e_2 \) is a (relatively) concrete version then \( e_2 \) must do everything that \( e_1 \) can do subject to an appropriate transformation on states and calculations. Furthermore, if \( e_1 \) is complete then \( e_2 \) must not introduce any behaviour that is inconsistent with that defined by \( e_1 \).

A behaviour refinement from \( e_1 \) to \( e_2 \) is a pair of mappings \( (\gamma_1, \gamma_2) : e_1 \rightarrow e_2 \) such that for every abstract calculation: \( e_1, S_1 \xrightarrow{m} e_1' S_1' \) there exists a concrete calculation: \( e_2, S_2 \xrightarrow{\gamma_2(m)} e_2' S_2' \) such that \( \gamma_1(S_2) = S_1 \) and \( \gamma_1(S_2') = S_1' \).

**Theorem 4** If \( (\gamma_1, \gamma_2) : e_1 \rightarrow e_2 \) and \( (\gamma'_1, \gamma'_2) : e_2 \rightarrow e_3 \) are two refinements, then \( (\gamma_1 \circ \gamma'_1, \gamma_2 \circ \gamma'_2) : e_1 \rightarrow e_3 \) is also a refinement.

**Theorem 5** If \( (\gamma_1, \gamma_2) : e_1 \rightarrow e_2 \) is a refinement then \( (\gamma_1, \gamma_2) \times \text{Id}_e : e_1 \times e \rightarrow e_2 \times e \) is also a refinement.

3.4 Message Passing

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

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

Basing the semantic model on asynchronous message passing does not preclude synchronous message passing since an asynchronous model which incorporates replacement behaviours can implement synchronous messages [Agh86] [Agh91]. A message \( m \in O \) is sent synchronously when \( e \) is a behaviour that waits for an incoming message \( m' \) such that \( m' \) is the response to \( m \). When \( m' \) is received the behaviour reverts to its original functionality.

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

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

\[
agent\{(t, v)\} \{m\} =
\begin{align*}
\text{case } m \text{ of} \\
\ldots
\text{p}_1 \rightarrow \\
\text{let } p_2 \leftarrow e_1 \\
\text{in } e_2 \\
\ldots
\end{align*}
\]

A \texttt{let} expression occurs in the context of a behaviour, represented here by the function \texttt{agent}. The expression \texttt{e1} is a set of messages or a single message which is to be sent in response to receiving a message matching \texttt{p1}. The
behaviour \( agent \) may carry on handling messages\(^1\). Any incoming message matching \( p_2 \) is a response to the messages \( e_1 \); the response of \( agent \) is defined by \( e_2 \).

The semantics of \( \text{let} \) is defined by a syntax translation to the basic design notation:

\[
\begin{align*}
\text{agent} \{ (t, v) \} \{ m \} &= \\
\text{case } m \text{ of} \\
\quad \ldots \\
\quad p_1 \to (\text{agent} \{ (t, v) \} + \text{wait}, e_1) \\
\quad \text{where rec \ wait} \{ m \} = \\
\quad \text{case } m \text{ of} \\
\quad \quad p_2 \to e_2 \\
\quad \quad \text{else } (\text{wait}, \emptyset) \\
\quad \text{end} \\
\quad \ldots \\
\text{end}
\end{align*}
\]

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

4 Requirements and Initial Specification

Software to control a simple machine (see figure 4) for dispensing widgets is required. The machine consists of a store of widgets, 4 buttons, an output tray and a two-tone beeper. The buttons are labelled 0 – 3. In order to dispense a widget the operator must press the buttons 1, 2 and 3 in order. At any time the operator may cancel the operation by pressing 0. Widgets are removed from the store and delivered to the output tray when they are dispensed. If the operation succeeds the beeper makes a high beep otherwise the beeper makes a low beep. Each widget has a unique identity.

An initial attempt at the required behaviour is shown in figure 5. The behaviour function \( M \) has two state components \( s \) and \( o \) that are sets of widgets representing the store and output respectively. Input messages 0 – 2 cause no state change and no output messages. Input message 3 from source object \( t' \) causes a widget to be dispensed and added to the output tray \( o \) if available in the store \( s \). A boolean reply is sent to the source of the message causing a high beep \( (\text{true}) \) if successful and a low beep \( (\text{false}) \) if the operation failed. The initial behaviour is under specified since it includes the required behaviour, but also permits illegal sequences of buttons.

\(^1\)In fact we only require \( agent \) to handle messages which it sends to itself. The extra machinery for this feature is straightforward but would clutter the example so we omit it.

Rigorous Object-Oriented Methods 2000
\[ M \{\{(t, (s, o))\}\} = \]
\[
\text{case } m \text{ of }
\begin{align*}
0 & \rightarrow (M \{\{(t, (s, o))\}\}, 0) \\
1 & \rightarrow (M \{(t, (s, o))\}, 0) \\
2 & \rightarrow (M \{(t, (s, o))\}, 0) \\
(t', 3) & \rightarrow \\
\text{case } s \text{ of }
\begin{align*}
0 & \rightarrow (M \{(t, (s, o))\}, \{(t, t', \text{false})\}) \\
\{w\} \cup s' & \rightarrow (M \{(t, (s', o \cup \{w\})\}), \{(t, t', \text{true})\}) \\
\end{align*}
\]

end

end

5 A Simple System Invariant

A simple system property is that the number of widgets available in both the store and the output tray is an invariant, \textit{i.e.} pushing buttons cannot cause widgets to be introduced or lost. This can be expressed as a behaviour:

\[ I \{\{(t, w)\}\} = (I \{\{(t, w)\}\}, 0) \]

together with a behaviour morphism from \( M \) to \( I \) that translates an \( M \) state \( (s, o) \) to an \( I \) state \#s + \#o and is identity everywhere else. In order to show that \( I \) is an invariant we show that the limit on the diagram \( M \) is the same as the limit on the diagram \( M \rightarrow I \).

\textbf{Theorem 6} \( I \) is an invariant of \( M \).

The proof shows that there exists a total behaviour morphism \( \phi : M \rightarrow I \) such that a limit on the diagram containing \( M \) is unchanged (isomorphic to) a limit on the diagram containing \( \phi : M \rightarrow I \). The mapping is defined as follows:

\[
\phi_1 \{(t, (s, o))\} = \{(t, \#s + \#o)\} \\
\phi_2 (m) = \begin{cases} 
[i, 0] & \text{when } m = [] \\
\langle i, 0 \rangle & \text{when } m = [i, o] \\
\phi_2 (m_1) + \phi_2 (m_2) & \text{when } m = m_1 + m_2 
\end{cases}
\]

\textbf{Proposition 1} The behaviour morphism \( \phi \) defines a total graph homomorphism.

\begin{align*}
\phi_2 \{(t, (s, o))\} &= \{(t, \#(\{w\} \cup s') + \#o - \{w\})\} \\
&= \{(t, 1 + \#s' + \#o - 1\} \\
&= \phi_2 \{(t, (s', o'))\} + 1 - 1 \\
&= \phi_2 \{(t, (s', o'))\}
\end{align*}

When \( m = m_1 + m_2 \) then assume by induction that the invariant is true for both \( M \{\{(t, (s, o))\} \overset{m_1} \rightarrow M \{\{(t, (s', o'))\} \)

\text{and } M \{\{(t, (s, o))\} \overset{m_2} \rightarrow M \{\{(t, (s', o'))\} \text{ and is therefore true by definition for } M \{\{(t, (s, o))\} \overset{m_1 + m_2} \rightarrow M \{\{(t, (s', o'))\} \}

It remains to show that \( M \) is a limit on diagrams containing \( M \) and \( \gamma : M \rightarrow I \) respectively.

\textbf{Proposition 2} \( M \) is a limit on the diagram containing \( M \) and \( \text{Id}_M : M \rightarrow M \).
\[ P\{(t, \sigma)\}\{m\} = \]
\[
\text{case } m \text{ of} \\
0 \rightarrow (P\{(t, A)\}, 0) \\
1 \rightarrow \text{case } \sigma \text{ of } A \rightarrow (P\{(t, B)\}, 0) \text{ end} \\
2 \rightarrow \text{case } \sigma \text{ of } B \rightarrow (P\{(t, C)\}, 0) \text{ end} \\
3 \rightarrow \text{case } \sigma \text{ of } C \rightarrow (P\{(t, A)\}, 0) \text{ end} \\
\text{end}
\]

Figure 6: Legal Message Sequences

The proof follows directly from the properties of the identity morphism. Now consider the second diagram. Firstly construct a product \( M \times I \) in which nodes are labelled with states from the free product \( \text{states}(M) \times \text{states}(I) \). Note that the product contains states that are legal \{\{(t_1, (s, o)), (t_2, #s + \#o)\}\} and those that are not.

Now construct an equalizer \( e : L \rightarrow M \times I \) such that \( \gamma \circ \pi_1 \circ e = \pi_2 \circ e \):

\[
\begin{align*}
\text{L} & \xleftarrow{} \text{M} \times \text{I} \\
& \quad \searrow \pi_2 \qquad \swarrow \pi_1 \\
\text{M} & \xrightarrow{\gamma} \text{I}
\end{align*}
\]

The behaviour \( L \) is a limit on the diagram and contains just those states that are legal. The limit \( L \) is not exactly the same as \( M \) but there exists an isomorphism between them. Therefore we conclude that \( \gamma : M \rightarrow I \) is a property of \( M \). QED

6 Removing Illegal Message Sequences

The behaviour \( M \) is under specified since it permits buttons on the machine to be pressed in illegal sequences. Object-oriented design notations such as UML restrict behaviours such as \( M \) using state transition models that impose orderings on sequences of permitted message calls. The machine consists of three states referenced as 1, 2 and 3.

The initial state for the machine is 1. Button 1 may only be pressed in state 1 causing a state change to 2. Button 2 may only be pressed in state 2 causing a state change to 3. Button 3 may only be pressed in state 3 causing a state change to 1; a widget is dispensed as a side effect. Button 0 may be pressed in any state causing a change to state 1.

The state transition machine is expressed as a behaviour function \( P \) in figure 6. The behaviour describes a single state component \( \sigma \) whose value is the machine state. The behaviour handles all machine messages making appropriate state changes but otherwise does nothing.

Behaviour \( M \) includes all correct behaviour but also includes incorrect behaviour. Applying the constraint \( P \) to \( M \) will produce just the required behaviour. The constraint is applied by combining \( M \) and \( P \) using the behaviour combination operator \( \times \). This produces a new behaviour \( M' = M \times P \) shown in figure 7. The behaviour \( M' \) has a state \( (s, o, \sigma) \) that is the combination of states from \( M \) and \( P \). Message dispatch has been combined so that the conditions from both \( M \) and \( P \) are taken into account. The multiple patterns \( (0, \sigma) \) for all states \( \sigma \) has been combined into a single pattern \( (0, \_0) \) since the same transition occurs in all cases. Notice that the machine simply ignores buttons that are pressed out of sequence and that if an operator gets into trouble they may always press 0 to reset the machine.

7 Object-Oriented Encapsulation

The behaviour \( M' \) does not observe the principle of encapsulation since the state \( (s, o) \) accesses state components of both \( s \) and \( o \). In order to be object-oriented, the behaviour \( M' \) must reference both \( s \) and \( o \) as objects. Accessing and
changing state in both \( s \) and \( o \) must occur via message passing.

Although both \( s \) and \( o \) are represented as sets of widgets, they are used in different ways. These differences may result in radically different implementation strategies and so we implement each as a separate behaviour in figure 8. A store behaviour \( S \) handles messages \( \text{empty} \) and \( \text{get} \). The former replies with \( \text{true} \) when the store is empty and the latter replies with an element of the store selected at random. An output tray behaviour \( O \) handles a single message \( \text{store} \) containing a widget \( x \). The widget is added to the output tray.

The behaviour function \( M' \) must be refined in order to use references to the store and output tray. The result is a new behaviour function \( M'' \) shown in figure 9. When \( M'' \) receives a message \( 3 \) from object \( t' \) in state \( 3 \) it sends a message \( \text{empty} \) to the store and waits for the reply. If the store is not empty then \( M'' \) sends it another message \( \text{get} \) and waits for a widget to be returned. On receiving the widget \( x \), \( M'' \) sends a \( \text{store}(x) \) message to the output tray. Finally, \( M'' \) replies to \( t' \) with \( \text{true} \) or \( \text{false} \).

**Theorem 7** \( S \times M' \times O \) is a refinement of \( M' \)
A proof of correctness for a refinement is established by defining a refinement homomorphism $\gamma_1, \gamma_2)$:

$$\gamma_1 \{ (t, (s, o, \sigma)) \} = \{ (t, (s, o, \sigma)) \}$$

$$\gamma_2(m) = \begin{cases} 
(i, \emptyset) & \text{when } m = [] \\
(i, [0]) & \text{when } m = [0] \\
3, \text{get, empty, true, false} & \text{when } m = [3, \{\text{false}\}] \\
3, \text{get, empty, false, get, w, true, store(w)} & \text{when } m = [3, \{\text{true}\}] \\
\gamma_2(m_1) + \gamma_2(m_2) & \text{when } m = m_1 + m_2
\end{cases}$$

\textbf{Theorem 8} $(\gamma_1, \gamma_2) : M' \rightarrow S \times M'' \times O$ is a refinement homomorphism.

The proof must establish that for all transitions $M'(\gamma_1(W)) \xrightarrow{\gamma_2(m)} M'(\gamma_1(W'))$ there exists a transition $(S \times M'' \times O)W \xrightarrow{\gamma_2(\emptyset)} (S \times M'' \times O)W'$. The refinement is \textit{complete} if it holds for all transitions performed by $M'$ and is \textit{sound} if it holds for all transitions performed by $S \times M'' \times O$. The proof is by induction on the structure of $m$; we proceed by case analysis of $m$.

When $m = []$, since $\gamma_2(m) = [], W = W'$ and $\gamma_1(W) = \gamma_1(W')$. When $m = [0]$, then $\gamma_2(m) = [0, 0]$ and for any $W = \{(t, (t, s, t, o, \sigma))\}$, $W' = \{(t, (t, s, t, o, \sigma))\}$, therefore $M'(\gamma_1(W)) \xrightarrow{m} M'(\gamma_1(W'))$.

When $m = [1]$, then $\gamma_2(m) = [1, 0]$ and for any $W = \{(t, (t, s, t, o, \sigma))\}$, $W' = \{(t, (t, s, t, o, \sigma))\}$, therefore $M'(\gamma_1(W)) \xrightarrow{m} M'(\gamma_1(W'))$.

When $m = [2, 0]$, then $\gamma_2(m) = [2, 0]$ and for any $W = \{(t, (t, s, t, o, \sigma))\}$, $W' = \{(t, (t, s, t, o, \sigma))\}$, therefore $M'(\gamma_1(W)) \xrightarrow{m} M'(\gamma_1(W'))$.

When $m = [3, \{\text{false}\}]$, then $\gamma_2(m) = [3, \{\text{false}\}]$ and for any $W = \{(t, (t, s, t, o, \sigma))\}$, $W' = \{(t, (t, s, t, o, \sigma))\}$, therefore $M'(\gamma_1(W)) \xrightarrow{m} M'(\gamma_1(W'))$.

When $m = [3, \{\text{true}\}]$, then $\gamma_2(m) = [3, \{\text{true}\}]$ and for any $W = \{(t, (t, s, t, o, \sigma))\}$, $W' = \{(t, (t, s, t, o, \sigma))\}$, therefore $M'(\gamma_1(W)) \xrightarrow{m} M'(\gamma_1(W'))$.

When $m = m_1 + m_2$, we assume by induction that the theorem holds for the sequences $m_1$ and $m_2$ and therefore it holds for $m$ by the transitivity of $\xrightarrow{\cdot}$, QED

8 Concrete Data Representation

The behaviours $S$ and $O$ do not uphold the principle of concrete data representation. Java does not provide a representation for data structures that is fully consistent with sets. One possible refinement for $S$ and $O$ is to use the standard behaviour for indexed sets that can be implemented directly in Java using either arrays or vectors. The store and output tray behaviours are refined as shown in figure 10.
\[
S' \{ (t, (Q, i)) \} \{ m \} = \\
\begin{cases}
\text{case } m \text{ of} \\
(t', \text{empty}) \to (S' \{ (t, (Q, i)) \}, \{ (t', i = -1) \}) \\
(t', \text{get}) \to (S' \{ (t, (Q, i - 1)) \}, \{ (t, t', Q[i]) \}) \\
\end{cases}
\]

\[
O' \{ (t, (Q, i)) \} \{ m \} = \\
\begin{cases}
\text{case } m \text{ of} \\
\text{store}(x) \to (O' \{ ((Q[i] := x), i + 1) \}), \emptyset \\
\end{cases}
\]

Figure 10: Refinement of Store and Output Tray Behaviours

The states of \( S' \) and \( O' \) consist of an indexed set \( Q \) and an index \( i \). The values in \( Q \) are indexed by integers from 0 upwards and \( i + 1 \) is the size of the indexed set. Given an indexed set \( Q \), the value associated with index \( k \) is \( Q[k] \). The indexed set is extended with a value \( x \) indexed by \( k \) to produce \( Q[k] := x \).

\textbf{Theorem 9} \( S' \times M'' \times O' \) is a legal refinement of \( S \times M'' \times O \)

A proof follows from theorem 5 and the existence of two refinements from \( S \) to \( S' \) and from \( O \) to \( O' \). The refinement \( (\gamma_1, \gamma_2) : S \to S' \) is defined as follows:

\[
\gamma_1 \{ (t, (Q, i)) \} = \{ (t, \{ Q[j] \mid j \in 0 \ldots i \}) \}
\]

\[
\gamma_2(m) = m
\]

Notice that \( (\gamma_1, \gamma_2) \) reduces the non-determinism occurring in \( S \) since the get method always returns widgets in a particular order for a given state.

\section{Implementation in Java}

Composition of refinement morphisms by theorem 4 shows the resulting system \( S' \times M'' \times O' \) to be a valid refinement of the initial specification \( M \). The system is implemented by identifying a one-to-one correspondence between the design components and programming language components that give rise to the same behaviour. Each behaviour function corresponds to a Java class. Each state component of a behaviour function corresponds to a private field. Each message handled by a class corresponds to a public method.

The implementation is given in figure 11. During implementation we may identify certain behaviours that are consistent with behaviours provided by existing Java classes. This occurs here with \( S' \) and \( O' \) both of which are consistent with the behaviour of \texttt{Vector}.

\section{References}


class M {
    private Store s;
    private OutputTray o;
    private int state = 1;
    public M(Store s, OutputTray o) { this.s = s; this.o = o; }
    public void zero() { state = 1; }
    public void one() { if(state == 1) state = 2; else error(); }
    public void two() { if(state == 2) state = 3; else error(); }
    public void three() {
        if(state != 3)
            error();
        else {
            if(!s.empty()) o.store(s.get());
            state = 1;
        }
    }
}

class Store {
    private Vector q;
    public Store(Vector q) { this.q = q; }
    public boolean empty() { return q.size() == 0; }
    public Widget get() { return (Widget)q.removeElement(); }
}

class OutputTray {
    private Vector q = new Vector();
    public store(Widget x) { q.addElement(x); }
}

Figure 11: Implementation of Machine


A The Theory $\lambda_o$

Figure 12 defines a semantics for the $\lambda_o$-calculus using a convertibility relation between terms. Figure 13 defines a type theory for $\lambda_o$-terms.

$$
(\lambda v.e)e_2 = e_1[v := e_2] \quad \mu v.e = e[v := \mu v.e]
$$

$$
e_1 = \nu
\theta(p) = \nu
\frac{\text{case } e_1 \text{ of } \ldots p \rightarrow e_2 \ldots \text{ end} = e_2}{\lambda v.e = \lambda v.e'}
$$

$$
e_1 = e_1'
\theta_1 = \theta_1
\frac{e_2 = e_2'}{e_1e_2 = e'_1e'_2}
\mu v.e = \mu v.e'
$$

$$
e_i = e_i'
\forall i = 1, \ldots, n
\frac{(e_1, \ldots, e_n) = (e_1', \ldots, e_n')}{\text{case } e \text{ of } p_1 \rightarrow e_1; \ldots; p_n \rightarrow e_n \text{ end} =
\text{case } e' \text{ of } p_1 \rightarrow e_1'; \ldots; p_n \rightarrow e_n' \text{ end}}
$$

Figure 12: The Theory $\lambda_o$
\[
\begin{align*}
A \vdash v : A(v) & \quad A, v : \alpha \vdash e : \beta \\
A \vdash e_1 : \alpha \to \beta & \quad A \vdash \lambda v. e : \alpha \to \beta \\
A \vdash e_2 : \alpha & \quad A, v : \tau \vdash e : \tau \\
A \vdash e_1 e_2 : \beta & \quad A \vdash \mu v. e : \tau \\
A \vdash e_i : \tau_i \forall i = 1, \ldots, n & \quad A \vdash \tau \vdash (e_1, \ldots, e_n) : (\tau_1, \ldots, \tau_n) \\
A \vdash (e_1, \ldots, e_n) : (\tau_1, \ldots, \tau_n) & \quad A \vdash \sigma e_1 \ldots e_n : \tau \\
A \vdash e : \tau_1 & \quad A, v_i : \tau_i \vdash p_i : \tau \forall i = 1, \ldots, n \\
A, v_i : \tau_i \vdash e_i : \tau_2 \forall i = 1, \ldots, n & \quad A \vdash \text{case } e \text{ of } p_1 \to e_1 ; \ldots ; p_n \to e_n \text{ end : } \tau_2
\end{align*}
\]

Figure 13: Type Theory for $\lambda_o$