SOFTWARE VERIFICATION RESEARCH CENTRE

DEPARTMENT OF COMPUTER SCIENCE

THE UNIVERSITY OF QUEENSLAND

Queensland 4072
Australia

TECHNICAL REPORT

No. 95-39

An Extended Semantic Foundation For Object-Z

Alena Griffiths

October, 1995

Phone: +61 7 365 1003
Fax: +61 7 365 1533

An Extended Semantic Foundation For Object-Z

Alena Griffiths

Software Verification Research Centre
Dept. of Computer Science
University of Queensland, 4072, Australia.

Abstract

The industrial potential of object-oriented formal methods is enormous. The potential of these methods has largely gone unrealised however, due to weaknesses in the underlying semantic foundations. This paper addresses that problem. We identify the "hard" problems in the area as: (i) the need to model object identity; (ii) the relationship between structure and behaviour; and, (iii) the (sometimes conflicting) notions of modularity and compositionality. The semantic foundations of Object-Z are then extended to provide a framework in which these ideas are accommodated.

1 Introduction

Object-oriented formal specification has been touted [9, 4, 7, 8] as a means by which formal methods might be applied on an industrial scale. The principal arguments advanced in favour of this claim are that:

1. the main barrier to the industrial scale use of formal methods is complexity (i.e. the effort required increases rapidly with the size of the system being developed), and the modularity afforded by object orientation can help to ameliorate this problem;
2. object orientation facilitates reuse, so there is potential for the reuse of formally verified software components.

Whilst these arguments are compelling, they both depend upon the existence of semantic foundations in which compositionality is well understood. That is, the ability to decompose systems into modular objects is of little use without a means of composing them to retrieve the original system. Likewise, the ability to reuse verified software components depends on an ability to give an interpretation to component level properties at the system level.

Published works on the semantics of object-oriented formal specification languages (of which the author is aware) suffer from one or both of the following limitations.

- Many [3, 2, 11] do not support object identity. Without support for this fundamental concept, the extent to which the languages defined by such semantics are object-oriented is questionable.

- Some provide a compositional semantics for structure [10, 2], but fail to do so for behaviour. That is, whilst the structure of a system is described in terms of the structure of its individual objects, the behaviour of the system tends to be described as a relation on the state of the system as a whole, rather than as a composition of the behaviour of individual objects.

The contribution of this paper is a framework in which the tri-partite nature of objects (entities which have state, behaviour and unique identity) is reconciled with the need for compositionality. This contribution follows on from earlier work on the semantics of Object-Z [5, 6], but the ideas should not be seen as language specific. The similarities between the "look-and-feel" of specifications developed using the major object-oriented formal specification languages has been noted[8].

Section 2 presents the architecture of Object-Z semantics. We introduce the idea of a distinction between the denotation (or model) of a syntactic component, and the meaning of that component. The latter is mathematically equivalent to the former, but has the advantage that it presents concepts like structure and behaviour in an intuitive way. We also present an outline of the stages involved in assigning a mathematical meaning to an Object-Z specification. This section provides a context in which the remainder of the paper may be interpreted.

Section 3 explores the relationship between modularity and compositionality, and argues for a dual notion of object operations. That is, we show that by adopting a model where operations consist of: (i) an internal state transition; and, (ii) an interaction with the environment; we are then able to give a meaning to classes which is independent of the remainder of the specification. Moreover, we can define a composition function which composes the meaning of the specification from the (modular) meaning of the constituent classes.

Section 4 reviews and extends the semantic universe of Object-Z, as presented in [6]. Based on the semantic universe for Z [1, 13], the Object-Z semantic universe extends the set *Type* with an extra partition for object identities, defines a new domain called *Message*, and extends the *carrier* function accordingly. The denotations for an operation, a class, and a specification are defined.

Section 5 presents a formalisation of the meaning of a class. The meaning is independent of the environment in which the class is used, and so is truly modular. The meaning of a class defines both the structure and behaviour of objects of that class, and is shown to be directly obtainable from the denotation of the class.

Section 6 presents a construction for the meaning of a system, given a system object and the meaning of the classes used in the specification of the system. As with classes, an account is given of both system structure and system behaviour, relating the two notions. The composition of system behaviour from constituent class behaviour is presented in outline. Full details are deferred to Section 7.

Section 8 concludes the paper, and suggests applications of the ideas and avenues for further research.

2 The Object-Z Semantic Framework

In this section we outline the semantic architecture of Object-Z. Our purpose is to clarify the process by which meaning is assigned to Object-Z specifications. A knowledge of the general principles underlying object-oriented formal specification is assumed — the reader is referred to [9]. Later sections in this paper will indicate which part of that process they address. The semantic framework is illustrated in Figure 1.

Figure 1: The Object-Z Semantic Framework

As indicated in Figure 1, an analysis of an Object-Z specification begins with a consideration of the concrete syntax. By ignoring representation details, and focusing on essential information, the concrete syntactic representation of a specification may be reduced to an abstract syntactic one. For example, the concrete syntax of an Object-Z declaration is given by the production:

```
Declaration → Word ':'  Term
```

where `Word` is a decorated variable. The corresponding abstract syntactic construct is:

Declaration
$word : Word$
$term : Term$

The abstract syntax tree that results from this procedure forms the starting point for semantic analysis, and so is inside what we call the semantic universe.

In Object-Z (as in Z), it is valid to declare a variable as belonging to any set which may be described by a well-formed term. For example, we could declare $x? : \{1, 2, 3\}$. The type of the word $x?$ must then be inferred from the type of the term $\{1, 2, 3\}$. The process of type inference is well-defined in Z, and because of the way in which the type structure of Object-Z has been defined, can be extended to Object-Z. Via this process of type inference, a typed syntax tree may be derived from the abstract syntax tree. The mapping is not total, as some abstract syntax trees may not satisfy the additional well-formedness rules associated with this phase and will be discarded.

Associated with each element of typed syntax is a denotation or model. The model is restricted in accordance with predicates in the state and operation schemata. The structure of some of these models is defined in Subsection 4.5. That subsection also justifies the applicability of Z semantics to the task of model restriction.

The last step in the process of assigning meaning to Object-Z specifications involves an interpretation of the denotations. That is, the denotations provide all the information we need. The process of model interpretation simply involves the "mathematical massaging" necessary to transform that information so that properties like structure, behaviour, identity and compositionality are readily apparent. In a sense, the model interpretation justifies the choice of denotation, because it demonstrates that the important properties have indeed been captured within the denotation. The rules for model interpretation are given in Sections 5 and 6.

A final comment should be made about the semantic universe. We view denotational semantics as a process which takes an abstract syntax tree, and assigns a mathematical meaning to it. Because this process involves input, output, transformations and intermediate stages, Figure 1 includes all these things within the region entitled "SEMANTIC UNIVERSE". The basic elements of this semantic universe are presented in Section 4.

3 Modularity and Compositionality

In this section we consider the notions of modularity and compositionality. We explain how a reference semantics for object identity facilitates these ideas. Then, in order to accommodate these notions within our semantics, we propose a dual view of operations, and draw a distinction between objects as independent entities and objects playing the role of "system object".

3.1 Concept Definitions

The term "modularity" embodies the notion that a system may be understood by understanding its separate parts or modules. Modularity is achieved by designing modules so they are maximally cohesive (i.e. all parts of the module contribute to the expression of a single concept), and minimally coupled (i.e. the extent to which modules depend on one another is minimised). In object-oriented specification, classes are the natural basis of modularity.

Related to the notion of modularity is the notion of compositionality. This concept says that the meaning of a system is a function of the meaning of its components. In object-oriented systems the natural syntactic components are classes. The meaning of a class is a homogeneous set of objects which each exhibit a certain structure and behaviour. The meaning of an object-oriented system is a heterogenous set of interacting objects which can be collectively considered to exhibit structure and behaviour. In order to claim the title “compositional”, the semantics must express this system meaning as a function of the meaning of each class.

3.2 A Reference Semantics for Object Identity

Underlying the semantics of Object-Z is the idea that if a variable a is declared to be of type class A, then the value of a is an object identity [5, 6]. This in turn requires the existence of an implicit object map, which is a function mapping identities to the underlying values (i.e. the attributes and operations) of objects they identify. The values may change over time, but the identities persist.

Apart from intuitively capturing notions like object persistence, and facilitating the modelling of object sharing and object self-reference, a reference semantics also enables one to take a more modular view. That is, a class A may define objects whose attributes include references to other objects. Changes in the state of those other objects will not cause a change in the state of objects of class A, since the references remain constant. This means that the possible evolutions of an object's state are exhaustively defined by the operations in the class to which the object belongs. This property is called encapsulation. In short, it means that one needn't look further than the class to discover how an object's state will be affected.

3.3 Operations Have Two Parts

The previous subsection showed that encapsulation allows one to confine one's attention to a class, for the purpose of giving a meaning to the evolution of an object's state. However, encapsulation is not sufficient to render a system fully modular. That is, the operations defined in a class may prescribe the invocation of an operation on one of the referenced objects. As such, the meaning of such operations is not confined to the referencing object — it is not inherently modular.

What we require is a denotation of an operation that is in some sense independent of the surrounding environment, even though when we later compose objects, we will not want to forget the interactive effect of those operations.

In order to achieve this goal, we propose a dual notion of operations. Operations are considered to consist of two parts. The first part is internal, and describes a relation on the state of the attributes. The second part is external, and describes an interaction with the environment. Either part may be null, so that every operation can be considered uniformly.

The description of the external part of an operation, in the meaning of a class, is largely syntactic. That is, at the class level we record the identity of the object with which the interaction occurs, and the name of the operation invoked. The description recorded is sufficiently detailed so that when objects are composed to yield a system, the contributions of these interactions to system behaviour are then interpreted.

It is these ideas which the formal structures presented in remaining sections describe.

3.4 Objects: Independent Entities or Systems?

One of the effects of adopting the dual notion of operations proposed in the previous section is that an object may be considered in two ways.

Firstly, we have the concept of an object as an independent entity or process. It will interact with other objects in the system, but can be assigned a meaning independent of those other objects. Its meaning (in this limited sense) is fully determined by its class.

Secondly, we have the idea that an object might in fact be the system object. In this case that object provides a "window" onto the total system. Its visible operations represent the interface of the system with its (unspecified) environment, and the messages it can send to other objects will determine which of their possible behaviours are utilised in this system. Moreover, the attributes the system object has will determine which other objects will be included in the system (which may in turn require the inclusion of other objects, etc.), and so the system object also determines system structure.

When an object is designated the system object, it still has meaning as an independent entity. But one must also view it as a starting point for system analysis, that is, as a starting point for the process whereby system meaning is composed from individual object meaning. In Section 6 we will resolve this problem by defining a composition function which uses the designated system object to determine the generation of structure and behaviour.

4 The Object-Z Semantic Universe

The previous sections presented the main idea on which this paper rests: that a formal treatment of modularity and compositionality is essential; and that the way to achieve this in the presence of a reference model for object identity is to adopt a dual notion of operations, and to use the "system object" as the starting point for system composition.

In this section we introduce some machinery necessary to achieve the formal treatment desired. That is, we present the Object-Z semantic universe. This universe was described in [6], and is predicated upon aspects of Z semantics [1, 13]. The semantic universe contains the basic elements from which the semantics is constructed. The fundamental elements of this universe — the names, types, and values — were described in [6]. We review aspects of that work which are crucial to this paper, and present some modifications and extensions to facilitate the structures presented in Sections 5 and 6.

4.1 Names

The names in an Object-Z specification were defined in [6]. For this paper we simplify the treatment, so that the basic entity is a word. We have:

$$Word == Name \times Role$$

where,

$$[Name]$$
$$Role ::= pre \mid post \mid in \mid out \mid none$$

Role captures the idea that a variable may denote the value of the state before or after operation invocation, or may be an input or output variable.

4.2 Types

To accommodate objects, the set *Type*, as defined in [1], is extended in [6] to provide partitions for the type of object identities (*O_type*), and the types of the underlying object values (*Cl_type*).

In this paper we adopt a more conservative view of type. As: (i) there is no syntactic component whose value is an underlying object value; (ii) all "object" variables are object identities; and, (iii) all other variables are of Z types; the types of all expressions in an Object-Z specification can be described without the need for the *Cl_type* partition.

Of course, the value of the underlying objects must still be represented in the semantic universe — this is done in Subsection 4.5. But, we choose not to capture this notion via the mechanism of types and associated values.

Therefore we have:

$$\begin{aligned} Type ::= {} & givenT\langle\langle Name\rangle\rangle \\ & \mid powerT\langle\langle Type\rangle\rangle \\ & \mid cproductT\langle\langle \mathrm{seq}\ Type\rangle\rangle \\ & \mid schemaT\langle\langle Signature\rangle\rangle \\ & \mid objectT\langle\langle Name\rangle\rangle \end{aligned}$$

where $Signature == Word \mathrel{\ffun} Type$. For later convenience we define $O\_type == \mathrm{ran}\ objectT$.

4.3 Values

With each type we associate a set of values, called the "carrier" of the type. In [6] a function *carrier* (based on the Z function of the same name) was defined inductively to support this association.

Object types are treated similarly to given types. That is, we associate a set of object identities with each element of *O_type*. The values associated with the complex types are then generated recursively. The details of the carrier function are not repeated here — the reader is referred to [6].

In Object-Z, the set of carrier sets is called $\mathcal{W}$, and the distributed union of $\mathcal{W}$ yields all values of interest, a set we call $\mathcal{V}$. For later convenience, $\mathbb{O}$ is defined as $\bigcup \mathrm{ran}(O\_type \lhd carrier)$.

An extension which provides a nil reference for each class (i.e. an identity which has no underlying object associated with it) is required. For this purpose we define:

$$\begin{array}{|l} nil : Name \rightarrow \mathbb{O} \\ \hline \forall\, n : Name \bullet nil(n) \in (objectT \mathbin{;} carrier)(n) \end{array}$$

The existence of a function *nil* is guaranteed by the axiom of choice.

4.4 Messages

A question left unanswered in [6], which we answer now, is the issue of how to represent external operation calls. Consider

B

$$a_1, a_2 : A$$
$$\ldots$$

$$Op1 \mathrel{\widehat=} a_1.Opa$$
$$Op2 \mathrel{\widehat=} a_1.Opa \parallel a_2.Opa$$
$$Op3 \mathrel{\widehat=} a_1.Opa \mathbin{[\!]} a_2.Opa$$
$$Op4 \mathrel{\widehat=} a_1.Opa \mathbin{⨟} a_2.Opa$$
$$\ldots$$

A

$$state : \ldots$$

Opa

$$\Delta(state)$$

In this specification $Op1$, when considered inside class B, says little more than that a message is sent to a_1 telling it to invoke operation Opa. One must go to class A to learn that Opa denotes a transition on the state of a_1. Moreover, $Op1$ has no effect on the evolution of the state of an object of class B, since it does not change its internal state. On the other hand, $Op1$ contains vital information about the way a set of objects of classes A and B will interact, and so is important when we compose objects to yield systems. In order to achieve the twin aims of modularity and compositionality, we propose to include within the denotation of class B a record of external operation calls. These external operation calls are referred to as messages.

The abstract syntax of a message is:

$$
\begin{aligned}
Message ::=\ & nil \\
& \mid simple\langle\!\langle Word \times Name\rangle\!\rangle \\
& \mid parallel\langle\!\langle Message \times Message\rangle\!\rangle \\
& \mid choice\langle\!\langle Message \times Message\rangle\!\rangle \\
& \mid sequence\langle\!\langle Message \times Message\rangle\!\rangle \\
& \mid hiding\langle\!\langle Message \times \mathbb{F}\ Word\rangle\!\rangle \\
& \mid rename\langle\!\langle Message \times (Word \mathrel{\rightarrowtail\!\!\!\!\twoheadrightarrow} Word)\rangle\!\rangle
\end{aligned}
$$

A "nil" message is included so that *every* operation in a class can be considered to consist of an internal transition and an external message. Examples of simple, parallel, choice and sequential messages are contained above in $Op1, Op2, Op3$ and $Op4$ respectively. The $Word$ and $Name$ in the domain of $simple$ refer to the variable name and operation name respectively. Note that the choice operator is non-deterministic and angelic, and the parallel operator permits communication between complementary variables in different arguments (i.e. if the first argument to the parallel operator contained a variable $x?$ and the second contained $x!$, these values would be equated). The hiding operator restricts the interface of an operation, and renaming is used to tailor the interface of an operation. We do not include an explicit construct for operation conjunction, which is commonly used in Object-Z specifications. This is because the effect of conjunction may be simulated using the parallel operator and hiding.

Wherever a message occurs, it will refer to a set of variables whose values will be object identities. For example, in the operation expression $a.Opa$, the set of object variables referred to is $\{a\}$. In

$a.Opa \parallel b.Opb$ the object variables are $\{a, b\}$. We define a function $vars$ to extract the set of affected object variables from a $Message$.

$$
\begin{array}{|l}
vars : Message \rightarrow \mathbb{P}\ Word \\
\hline
\forall\, m : Message \bullet \\
\quad m = nil \Rightarrow vars(m) = \varnothing \\
\quad m \in \operatorname{ran} simple \Rightarrow vars(m) = \{\pi_1(simple^{\sim}(m))\} \\
\quad m \in \operatorname{ran} parallel \Rightarrow vars(m) = vars(\pi_1(parallel^{\sim}(m))) \cup vars(\pi_2(parallel^{\sim}(m))) \\
\quad m \in \operatorname{ran} sequence \Rightarrow vars(m) = vars(\pi_1(sequence^{\sim}(m))) \cup vars(\pi_2(sequence^{\sim}(m))) \\
\quad m \in \operatorname{ran} hiding \Rightarrow vars(m) = vars(\pi_1(hiding^{\sim}(m))) \\
\quad m \in \operatorname{ran} rename \Rightarrow vars(m) = vars(\pi_1(rename^{\sim}(m)))
\end{array}
$$

The typed syntax of an external operation call, which is deduced during the type inference phase, associates a type with each of the object variables in the message.

$$
\begin{array}{|l}
\hline
\textit{Message\_expr} \\
msg : Message \\
var\_types : Signature \\
\hline
vars(msg) = \operatorname{dom} var\_types \\
\operatorname{ran} var\_types \subseteq O\_type \\
\hline
\end{array}
$$

4.5 Denotations

We define a variable valuation (often referred to as a "binding") to be an assignment of values to a set of words. That is,

$$Valuation == Word \mathrel{+\!\!\!\!\twoheadrightarrow} \mathcal{V}$$

In Object-Z, the model (or denotation) of a schema is an element of $Schema\_den$. It is the set of valuations consistent with the type (i.e. the signature) and predicates of the schema.

$$Schema\_den == \mathbb{P}\ Valuation$$

In Z, the process for obtaining the set of valuations associated with a schema is well defined. Because of the treatment of object identity in Object-Z, this process can be readily extended to Object-Z.

The previous subsection gave the typed syntax of an external operation call. This was the set $Message\_expr$. The associated value of an element of $Message\_expr$ (a syntactic entity) is an element of $Message\_val$ (a semantic entity). $Message\_val$ consists of the message itself, together with an assignment of values to the object variables affected by the message. In short, a message value gives the identity of each object affected, together with the structure of the message that affects the object. We have:

$$
\begin{array}{|l}
\hline
\textit{Message\_val} \\
msg : Message \\
var\_vals : Valuation \\
\hline
vars(msg) = \operatorname{dom} var\_vals \\
\hline
\end{array}
$$

Associated with each element of $Message\_expr$ is a denotation. This denotation consists of a subset of $Message\_val$. We have:

$$Message\_den == \mathbb{P}\, Message\_val$$

The association between message expressions and denotations is given by the following function:

$$
\begin{array}{|l}
message\_den : Message\_expr \rightarrow Message\_den \\
\hline
\forall\, me : Message\_expr;\ mv : Message\_val \bullet \\
\quad mv \in message\_den(me) \Leftrightarrow \\
\quad\quad me.msg = mv.msg \\
\quad\quad \mathrm{dom}\, mv.var\_vals = \mathrm{dom}\, me.var\_types \\
\quad\quad mv.var\_vals \subseteq me.var\_types \mathbin{;} carrier \mathbin{;} \in^{\sim}
\end{array}
$$

The final two lines of the above definition ensure that the function var_vals prescribes values for the words in the domain of the var_type function, and that the values prescribed are consistent with their type.

In line with the desire to treat operations uniformly, and to be seen as consisting of two parts, we have:

$$
\begin{array}{|l}
\hline
\multicolumn{1}{l}{Op\_denotation} \\
internal : Schema\_den \\
external : Message\_den \\
\hline
\end{array}
$$

The denotation of a class contains valuations for the state schema, a subset of which satisfy the initial predicate, and a set of operations.

$$
\begin{array}{|l}
\hline
\multicolumn{1}{l}{Class\_denotation} \\
states : Schema\_den \\
init\_states : Schema\_den \\
ops : Name \twoheadrightarrow\!\!\!\!\!\!\!\;\;| \;\; Op\_denotation \\
\hline
init\_states \subseteq states \\
\#\, \mathrm{dom}\, (\!|\, states \,|\!) = 1 \\
\hline
\end{array}
$$

The second predicate in the above schema insists that each valuation in $states$ is a valuation for the same set of attributes.

The denotation of a specification identifies the system object, gives the class dictionary, and records global definitions.

$$
\begin{array}{|l}
\hline
\multicolumn{1}{l}{Specification\_denotation} \\
sys\_ob : \mathbb{O} \\
cdict : Name \twoheadrightarrow\!\!\!\!\!\!\!\;\;| \;\; Class\_denotation \\
env : Word \twoheadrightarrow\!\!\!\!\!\!\!\;\;| \;\; \mathcal{V} \\
\hline
\end{array}
$$

5 The Meaning of a Class

The previous section saw the definition of the basic elements from which the semantics of Object-Z is constructed. Denotations for each of the syntactic entities that make up a specification were described, and similarities with Z were relied upon to show that the denotation for each part of an Object-Z specification could be deduced. Referring to Figure 1, we have thus far described the semantic framework down to the level of "Denotation". This section extends this the semantics for classes. That is, we provide the model interpretation functions necessary to deduce the meaning of a class from its denotation.

5.1 Interpreting Operations

The denotation of a class contains a list of available operations and their denotations. In this subsection we wish to interpret the operation denotations. That is, we wish to present an alternative but equivalent mathematical entity — a "meaning" — which captures the operation concept more intuitively than the denotation does.

We require the meaning of an operation to be given for some particular value of the object's state. The reason for this is that we later want to be able to describe the value of an object in terms of the value of its state, together with a description of enabled operations.

Therefore, we define an enabled operation value as follows:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{En\_op\_val}} \\
\hline
\textit{pre\_val} : \textit{Valuation} \\
\textit{internal} : \textit{Valuation} \leftrightarrow (\textit{Valuation} \times \textit{Valuation}) \\
\textit{external} : \textit{Message\_den} \\
\textit{inputs} : \mathbb{P}\,\textit{Valuation} \\
\textit{post\_vals} : \mathbb{P}\,\textit{Valuation} \\
\textit{outputs} : \mathbb{P}\,\textit{Valuation} \\
\hline
\operatorname{dom} \textit{internal} = \textit{inputs} \\
\operatorname{dom} \operatorname{ran} \textit{internal} = \textit{post\_vals} \\
\operatorname{ran} \operatorname{ran} \textit{internal} = \textit{outputs} \\
\hline
\end{array}
$$

The *pre_val* is the value of the state before the operation occurs. Possible inputs, outputs, and resultant values of state variables are given by *inputs*, *outputs* and *post_vals* respectively. The nature of the transition is described by *internal*, a fact captured by the predicates.

So an operation denotation can be interpreted as defining a set of elements of *En_op_val* — one for each value of the state in which the operation is enabled. Before this interpretation can be formally defined, we require an auxiliary function which restricts the bindings in a set to bindings for just the variables playing a specified role. The function is:

$$
\begin{array}{|l}
sub\_val : Role \rightarrow (\mathbb{P}\, Valuation \pfun \mathbb{P}\, Valuation) \\
\hline
\forall\, r : Role \bullet \\
\quad \mathrm{dom}\, sub\_val(r) = \{pv : \mathbb{P}\, Valuation \mid \#\, \mathrm{dom} (\!|\ pv\ |\!) = 1\} \\
\quad \forall\, pv : \mathrm{dom}\, sub\_val(r) \bullet \\
\qquad sub\_val(r)(pv) = \{v : Valuation \mid \exists\, v_0 : pv \bullet \\
\qquad\qquad\qquad\qquad\qquad\qquad (\mathrm{dom}\, v_0) \rhd \{r\} = \mathrm{dom}\, v \\
\qquad\qquad\qquad\qquad\qquad\qquad v \subseteq v_0\}
\end{array}
$$

The process of operation interpretation may now be defined as follows:

$$
\begin{array}{|l}
op\_interpret : Op\_denotation \rightarrow \mathbb{P}\, En\_op\_val \\
\hline
\forall\, od : Op\_denotation;\ ev : En\_op\_val \bullet \\
\quad ev \in op\_interpret(od) \Leftrightarrow \\
\qquad ev.pre\_val \in sub\_val(pre)(od.internal) \\
\qquad ev.internal = \{v\colon od.internal \mid sub\_val(pre)(\{v\}) = ev.pre\_val \bullet \\
\qquad\qquad (sub\_val(in)(\{v\}), (sub\_val(post)(\{v\}), sub\_val(out)(\{v\})))\} \\
\qquad ev.message \subseteq \{m\colon od.external \mid \exists\, v : od.internal \bullet \\
\qquad\qquad sub\_val(pre)(\{v\}) = ev.pre\_val \\
\qquad\qquad m.var\_vals \subseteq v\}
\end{array}
$$

The final condition in the above definition relates to messages and ensures that each of the bindings for variables in the messages of an *En_op_val* are consistent with at least one of the bindings for the variables in the internal transition of that same *En_op_val*.

5.2 Class Values

From the denotation of a class, we can use the operation interpretation defined in the previous subsection to derive what we refer to as a class value. This structure is not yet worthy of the title "meaning", but is a necessary intermediate stage. We have:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\ Class\_val\ }} \\
states : Schema\_den \\
init\_states : Schema\_den \\
ops : Name \ffun (\mathbb{P}\, En\_op\_val) \\
\hline
init\_states \subseteq states \\
\#\, \mathrm{dom} (\!|\ states\ |\!) = 1 \\
\hline
\end{array}
$$

Note that for each operation defined in a class, there will be many enabled operation values, one for each possible value of the object's state in which the operation is enabled.

The mapping from a *Class_denotation* to a *Class_val* involves an application of the operation interpretation, and is defined as follows:

$$
\begin{array}{|l}
alter\_op\_form : Class\_denotation \rightarrow Class\_val \\
\hline
alter\_op\_form = \lambda\, cd : Class\_denotation \bullet \\
\quad (\mu\, Class\_val \mid states = cd.states \\
\qquad\qquad init\_states = cd.init\_states \\
\qquad\qquad ops = cd.ops \fcmp op\_interpret)
\end{array}
$$

5.3 Object Values

An object value gives the underlying value for an object at some point in its evolution. This comprises the value of the object's state, together with a list of enabled operations.

$$
\begin{array}{|l}
\textit{Object\_val} \\
\hline
\textit{state} : \textit{Valuation} \\
\textit{ops} : \textit{Name} \mathrel{+\!\!\!\!\twoheadrightarrow} \textit{En\_op\_val} \\
\hline
\forall\, \textit{ev} : \operatorname{ran} \textit{ops} \bullet \textit{ev}.\textit{pre\_val} = \textit{state}
\end{array}
$$

Note that according to the above definition there may be operations defined in a class which do not appear in an object value associated with the class. This will occur whenever *state* does not satisfy the precondition[1] of the operation.

The relationship between object values and class values is as follows:

$$
\begin{array}{|l}
\_\,\textit{is\_ob\_of\_class}\,\_ : \textit{Object\_val} \leftrightarrow \textit{Class\_val} \\
\hline
\forall (\textit{ov}, \textit{cv}) : \textit{Object\_val} \times \textit{Class\_val} \bullet \\
\quad \textit{ov}\ \textit{is\_ob\_of\_class}\ \textit{cv} \Leftrightarrow \\
\quad\quad \textit{ov}.\textit{state} \in \textit{cv}.\textit{states} \\
\quad\quad \forall (n, o) : \textit{ov}.\textit{ops} \bullet \\
\quad\quad\quad o \in \textit{cv}.\textit{ops}(n) \\
\quad\quad \forall\, n : \operatorname{dom} \textit{cv}.\textit{ops} \bullet \\
\quad\quad\quad ((\exists\, \textit{ev} : \textit{cv}.\textit{ops}(n) \bullet \textit{ev}.\textit{pre\_val} = \textit{ov}.\textit{state}) \Rightarrow \\
\quad\quad\quad\quad n \in \operatorname{dom} \textit{ov}.\textit{ops})
\end{array}
$$

That an object is in one of the permissible initial states is captured by the following relationship:

$$
\begin{array}{|l}
\_\,\textit{is\_initial\_ob\_of\_class}\,\_ : \textit{Object\_val} \leftrightarrow \textit{Class\_val} \\
\hline
\forall (\textit{ov}, \textit{cv}) : \textit{Object\_val} \times \textit{Class\_val} \bullet \\
\quad \textit{ov}\ \textit{is\_initial\_ob\_of\_class}\ \textit{cv} \Leftrightarrow \\
\quad\quad \textit{ov}\ \textit{is\_ob\_of\_class}\ \textit{cv} \\
\quad\quad \textit{ov}.\textit{state} \in \textit{cv}.\textit{init\_states}
\end{array}
$$

So, a class value defines a set of object values, which we refer to as the objects of the class.

$$
\begin{array}{|l}
\textit{objects} : \textit{Class\_val} \rightarrow \mathbb{P}\, \textit{Object\_val} \\
\hline
\textit{objects} = \lambda\, \textit{cv} : \textit{Class\_val} \bullet \\
\quad \{\textit{ov} : \textit{Object\_val} \mid \textit{ov}\ \textit{is\_ob\_of\_class}\ \textit{cv}\}
\end{array}
$$

The initial objects of the class are distinguished as follows:

$$
\begin{array}{|l}
\textit{initial\_objects} : \textit{Class\_val} \rightarrow \mathbb{P}\, \textit{Object\_val} \\
\hline
\textit{initial\_objects} = \lambda\, \textit{cv} : \textit{Class\_val} \bullet \\
\quad \{\textit{od} : \textit{Object\_val} \mid \textit{ov}\ \textit{is\_initial\_ob\_of\_class}\ \textit{cv}\}
\end{array}
$$

[1] We use the term "precondition" in a special way because in Object-Z, preconditions are in fact enabling conditions.

5.4 Object Behaviour

A labelled transition is a triple (o_1, l, o_2) which represents the evolution of an object from object value o_1 to object value o_2 via the transition labelled l. The label is a 4-tuple consisting of an input vector, an operation name, a message value and an output vector. These concepts may be formalised thus:

$$Transition == Object\_val \times Label \times Object\_val$$
$$Label == Valuation \times Name \times Message\_val \times Valuation$$

A class value defines a set of labelled transitions, and so defines object behaviour. This relationship may be formalised:

$$\begin{array}{|l}
LT : Class_val \rightarrow \mathbb{P}\ Transition \\
\hline
\forall\, cv : Class_val;\ o_1, o_2 : Object_val;\ in, out : Valuation;\ n : Name;\ m : Message_val \bullet \\
\quad (o_1, (in, n, m, out), o_2) \in LT(cv) \Leftrightarrow \\
\qquad \{o_1, o_2\} \subseteq objects(cv) \\
\qquad (o_2.state, out) \in o_1.ops(n).internal \,(\!|\, \{in\} \,|\!) \\
\qquad m \in o_1.ops(n).external \\
\qquad m.var_vals \subseteq \bigcup\{o_1.pre_val, in, out, o_2.pre_val\}
\end{array}$$

In the above definition the predicate on the third line insists that transitions are only defined between objects of the same class. The fourth line says that the state of o_2 and the outputs out results from the application of one of the operations in o_1, given the input vector in. The final two predicates relate to the message: the first ensures that the message is one of the possible messages emitted when o_1 performs operation n; and the second ensures that the value of object variables in the message is consistent with the other variable valuations.

5.5 Class Meaning

We now have the necessary tools to define the meaning of a class. A class value defines a set of possible object values, a distinguished subset of which comprise the objects satisfying the initialisation predicate, and a set of transitions. In short, a class describes structure and behaviour.

$$\begin{array}{|l}
meaning : Class_val \rightarrow (\mathbb{P}\ Object_val \times \mathbb{P}\ Object_val \times \mathbb{P}\ Transition) \\
\hline
meaning = \lambda\, cv : Class_val \bullet \\
\quad (objects(cv), initial_objects(cv), LT(cv))
\end{array}$$

It was stated at the beginning of this section that the goal was to interpret the meaning of a class, given its denotation. We call this process "class interpretation", and define it as follows:

$$\begin{array}{|l}
class_interpret : Class_denotation \rightarrow (\mathbb{P}\ Object_val \times \mathbb{P}\ Object_val \times \mathbb{P}\ Transition) \\
\hline
class_interpret = alter_op_form \,\S\, meaning
\end{array}$$

6 The Meaning of a System

The previous section defined a means of interpreting class denotations in order to deduce the meaning of a class. This section outlines a similar procedure for the specification as a whole. The procedure relies on the constructs presented in Section 5. That is, instead of interpreting a specification denotation "from scratch", we use the meaning of the classes (i.e. the object values and associated transitions) that comprise the specification, and compose that (i.e. the individual objects and transitions) to yield the meaning of a specification.

6.1 Specification Values

At any point in its evolution, the value of a system described by an Object-Z specification will consist of two parts: an object map describing the identity and value of all objects in the system; and a distinguished system object.

$$
\begin{array}{l}
\textit{Spec\_val} \\
\hline
\textit{sys\_ob} : \mathbb{O} \\
\textit{object\_map} : \mathbb{O} \pfun \textit{Object\_val} \\
\hline
\textit{sys\_ob} \in \operatorname{dom} \textit{object\_map}
\end{array}
$$

For a given specification, we want to be able to say precisely which elements of *Spec_val* are allowable values for the system described by that specification. Some auxiliary machinery is required for this task.

Firstly, given a set of object values, we want to be able to scan the *state* variables of all those object values and extract all object identities. This set of object attributes, together with the system object, represents the set of identities of all objects in a system. The function is defined as follows:

$$
\begin{array}{l}
\textit{object\_attributes} : \mathbb{P}\, \textit{Object\_val} \rightarrow \mathbb{P}\, \mathbb{O} \\
\hline
\textit{object\_attributes} = \lambda\, \textit{ovs} : \mathbb{P}\, \textit{Object\_val} \bullet \\
\quad \textbf{let}\ \textit{value\_set} == \bigcup \{ \textit{ov} : \textit{ovs} \bullet \bigcup \operatorname{ran} \textit{ov.state} \} \bullet \\
\qquad \textit{ob\_id\_vals}(\textit{value\_set})
\end{array}
$$

In the above definition, the function *ob_id_vals* sorts through a set of possibly complex values and returns all object identities. The definition of the function is deferred to an appendix.

Secondly, we require a function which takes an object identity and returns the name of the class to which the referred object belongs.

$$
\begin{array}{l}
\textit{name\_of\_class} : \mathbb{O} \rightarrow \textit{Name} \\
\hline
\forall\, o : \mathbb{O};\ n : \textit{Name} \bullet \\
\quad (o, n) \in \textit{name\_of\_class} \Leftrightarrow o \in \textit{objectT} \mathbin{;} \textit{carrier}(n)
\end{array}
$$

Having defined the necessary machinery, we can now say that an element $sv \in \textit{Spec\_val}$ is a valid value for a specification *spec* if $sv \in \textit{spec\_vals}(\textit{spec})$, where *spec_vals* is defined as follows:

$$
\begin{array}{|l}
spec\_vals : Specification\_denotation \rightarrow \mathbb{P}\, Spec\_val \\
\hline
\forall\, sd : Specification\_denotation;\ sv : Spec\_val \bullet \\
\quad sv \in spec\_vals(sd) \Leftrightarrow \\
\quad\quad sd.sys\_ob = sv.sys\_ob \\
\quad\quad \forall\, oid : object\_attributes(\mathrm{ran}\, sv.object\_map) \bullet \\
\quad\quad\quad (oid \in \mathrm{dom}\, sv.object\_map \lor \exists\, c : \mathrm{dom}\, sd.cdict \bullet oid = nil(c)) \\
\quad\quad \forall\, oid : \mathrm{dom}\, sv.object\_map \bullet \\
\quad\quad\quad (sv.object\_map(oid)\ is\_ob\_of\_class \\
\quad\quad\quad\quad\quad alter\_op\_form(sd.cdict(name\_of\_class(oid)))) \\
\quad\quad\quad (oid = sv.sys\_ob \lor \\
\quad\quad\quad\quad \exists\, oid_1 : object\_attributes(\mathrm{ran}\, sv.object\_map) \bullet oid = oid_1)
\end{array}
$$

The above function associates a set of system values with a specification denotation. As such, it gives the possible structures of the system. These structures depend upon the system object. That is, all object maps must contain the system object, and then only those objects whose presence is directly or indirectly "required" by the system object.

The first of the universally quantified predicates insists that all references contained in object values appearing in the object map are valid. That is, the references must be to objects whose identities appear in the domain of the object map, or the references must be nil.

The second universal quantification captures the reverse idea: that all object references in the domain are "needed"; and also ensures that object identities refer to object values of the correct class. This latter condition depends crucially upon the relation *is_ob_of_class* which was defined in the previous section, and was used to describe the set of object values that a class defines.

As with classes, we distinguish the set of *Spec_val*'s which represents the possible initial configurations of the system.

$$
\begin{array}{|l}
init\_spec\_vals : Specification\_denotation \rightarrow \mathbb{P}\, Spec\_val \\
\hline
\forall\, sd : Specification\_denotation;\ sv : Spec\_val \bullet \\
\quad sv \in init\_spec\_vals(sd) \Leftrightarrow \\
\quad\quad sv \in spec\_vals(sd) \\
\quad\quad \forall\, oid : \mathrm{dom}\, sv.object\_map \bullet \\
\quad\quad\quad sv.object\_map(oid)\ is\_initial\_ob\_of\_class \\
\quad\quad\quad\quad\quad alter\_op\_form(sd.cdict(name\_of\_class(oid)))
\end{array}
$$

6.2 System Behaviour

To define system behaviour we introduce the notion of a specification transition. A specification transition is a triple (sv_1, l, sv_2) which represents the evolution of a system from the value sv_1 to the value sv_2 via the transition labelled l. Note that messages are not included in specification labels. Formally,

$$
\begin{aligned}
&Spec\_trans == Spec\_val \times Spec\_label \times Spec\_val \\
&Spec\_label == Valuation \times Name \times Valuation
\end{aligned}
$$

We want to define the set of transitions that a specification defines. This definition is presented at two levels of abstraction. At the top level, to decide that (sv_1, l, sv_2) is an allowable transition for a system, we need to ensure that sv_1 and sv_2 both represent possible values for the system at some point in its evolution. Thus:

$$
\begin{array}{|l}
ST : Specification\_denotation \rightarrow \mathbb{P}\, Spec\_trans \\
\hline
\forall\, sd : Specification\_denotation;\ sv_1, sv_2 : Spec\_val;\ inputs, outputs : Valuation;\ n : Name \bullet \\
\quad (sv_1, (inputs, n, outputs), sv_2) \in ST(sd) \Leftrightarrow \\
\qquad \{sv_1, sv_2\} \subseteq spec\_vals(sd) \\
\qquad sv_2\ is\_an\_effect\_of(sd)\,(sv_1, (inputs, n, outputs))
\end{array}
$$

The second conjunct in the above definition uses the function *is_an_effect_of*, which represents the second layer of abstraction. We first present an outline of the formal definition of this function, and then discuss its meaning.

$$
\begin{array}{|l}
is\_an\_effect\_of : Specification\_denotation \rightarrow (Spec\_val \leftrightarrow (Spec\_val \times Label)) \\
\hline
\forall\, sd : Specification\_denotation \bullet \\
\quad \forall\, sv_1, sv_2 : Spec\_val;\ inputs, outputs : Valuation;\ n : Name \bullet \\
\qquad sv_2\ is\_an\_effect\_of\ (sv_1, (inputs, n, outputs)) \Leftrightarrow \\
\qquad\quad \exists\, in, out : Valuation;\ m : Message\_val \bullet \\
\qquad\qquad in \subseteq inputs \\
\qquad\qquad out \subseteq outputs \\
\qquad\qquad (sv_1.sys\_ob, (in, n, m, out), sv_2.sys\_ob) \in \\
\qquad\qquad\quad LT(alter\_op\_form(sd.cdict(name\_of\_class(sv_1.sys\_ob)))) \\
\qquad\qquad \mathbf{let}\ rem\_om_1 == \{sv_1.syb\_ob\} \lhd sv_1.object\_map; \\
\qquad\qquad\quad rem\_om_2 == \{sv_2.syb\_ob\} \lhd sv_2.object\_map \bullet \\
\qquad\qquad\quad m.msg = nil \Rightarrow (rem\_om_1, rem\_om_2) \in nil\_msg \\
\qquad\qquad\quad m.msg \in \operatorname{ran} simple \Rightarrow \\
\qquad\qquad\qquad (rem\_om_1, rem\_om_2, m, inputs, outputs) \in simple\_msg(sd) \\
\qquad\qquad\quad m.msg \in \operatorname{ran} parallel \Rightarrow \\
\qquad\qquad\qquad (rem\_om_1, rem\_om_2, m, inputs, outputs) \in parallel\_msg(sd) \\
\qquad\qquad\quad m.msg \in \operatorname{ran} choice \Rightarrow \\
\qquad\qquad\qquad (rem\_om_1, rem\_om_2, m, inputs, outputs) \in choice\_msg(sd) \\
\qquad\qquad\quad m.msg \in \operatorname{ran} sequence \Rightarrow \\
\qquad\qquad\qquad (rem\_om_1, rem\_om_2, m, inputs, outputs) \in sequence\_msg(sd) \\
\qquad\qquad\quad m.msg \in \operatorname{ran} hiding \Rightarrow \\
\qquad\qquad\qquad (rem\_om_1, rem\_om_2, m, inputs, outputs) \in hiding\_msg(sd) \\
\qquad\qquad\quad m.msg \in \operatorname{ran} rename \Rightarrow \\
\qquad\qquad\qquad (rem\_om_1, rem\_om_2, m, inputs, outputs) \in rename\_msg(sd)
\end{array}
$$

This function formalises the intuition that the behaviour of a system is a composition of (i) the individual behaviours of constituent objects; and (ii) the interactions between those objects. The function takes a specification denotation and produces a relation between system structures (i.e. between elements of *Spec_val*). We say sv_2 is an effect of (sv_1, l) if when we apply the operation described in the label l to sv_1, sv_2 can result. Essentially, this is a relation between object maps with distinguished system objects.

The system objects of related system structures must be related by the meaning of the class to which they belong. That is, the underlying values of the system objects will occur in a transition in the set defined by LT. There will be a message m associated with this transition.

Removing the system object from the object maps in sv_1 and sv_2, one must then ensure that the resulting mini-object-maps (rem_om_1 and rem_om_2) are also in a similar relation. This time however, the label l will be partially determined by the message m. The message (in the simple case) will identify the object which is to play the role of system object in the new mini-systems. The purpose of the auxiliary functions *nil_msg*, *simple_msg*, *parallel_msg*, etc. is to define the extraction of the label from the message, and insist that the new mini-systems recursively satisfy the *is_an_effect_of* relation. The details of these auxiliary functions have been deferred to the next section.

6.3 System Meaning

Using the framework outlined in the previous subsections, we are now in a position to formally define the meaning of a system.

A system describes a set of specification values which represent system structure. A subset of these values is distinguished because they satisfy the initialisation predicates. Finally, a set of transitions capturing system behaviour is described. We have:

$$system\_meaning : Specification\_denotation \rightarrow (\mathbb{P}\, Spec\_val \times \mathbb{P}\, Spec\_val \times \mathbb{P}\, Spec\_trans)$$

$$\begin{array}{l} system\_meaning = \lambda\, sd : Specification\_denotation \bullet \\ \quad (spec\_vals(sd), init\_spec\_vals(sd), ST(sd)) \end{array}$$

7 The Semantics Of Object-Z Messages

This section contains details of the auxiliary functions used in the the function *is_an_effect_of* which is defined in Subsection 6.2.

7.1 Auxiliary Definitions

To render the presentation more concise, we introduce a few definitions.

$$\begin{array}{l} Map == \mathbb{O} \nrightarrow Object\_val \\ Msg\_argument == Map \times Map \times Message\_val \times Valuation \times Valuation \end{array}$$

A *Msg_argument* includes two object maps (representing the value of the system before and after the message is executed), a message, and valuations for input and output vectors.

Some message protocols require that complementary variables in the input and output bindings take equal values. Formally,

$$\_\, compatible\_with \,\_ : Valuation \leftrightarrow Valuation$$

$$\begin{array}{l} \mathrm{dom}\, compatible\_with = \{v : Valuation \mid \forall (n, r) : \mathrm{dom}\, v \bullet r = in\} \\ \mathrm{ran}\, compatible\_with = \{v : Valuation \mid \forall (n, r) : \mathrm{dom}\, v \bullet r = out\} \\ \forall\, v_1 : \mathrm{dom}\, compatible\_with;\ v_2 : \mathrm{ran}\, compatible\_with \bullet \\ \quad v_1\ compatible\_with\ v_2 \Leftrightarrow \\ \quad\quad \forall (n_1, r_1) : \mathrm{dom}\, v_1;\ (n_2, r_2) : \mathrm{dom}\, v_2 \bullet \\ \quad\quad\quad n_1 = n_2 \Rightarrow v_1(n_1, r_1) = v_2(n_2, r_2) \end{array}$$

7.2 The Nil Message

The message *nil* has no effect on the object map. This may appear straightforward, but it is important to note that it semantically enforces the notion that the value of an object will not change unless an operation is expressly invoked upon that object. We have:

$$
\begin{array}{|l}
nil\_msg : Map \leftrightarrow Map \\
\hline
\forall\, om_1, om_2 : Map \bullet \\
\qquad (om_1, om_2) \in nil\_msg \Leftrightarrow om_2 \oplus om_1 = om_2
\end{array}
$$

This definition insists that objects whose identities appear in both object maps have equivalent values. The reason the object maps are not simply equated is that identities may have been passed in or out of the system in the internal state transition associated with the sending of this *nil* message. This would require an expansion or contraction of the object map, but must not change the value of objects that persist. That the expansion or contraction happens properly is ensured by the requirements in the definition of the function *spec_vals*. Any arguments passed to this relation for checking will have satisfied *spec_vals* — this is ensured in the function ST.

7.3 The Simple Message

A simple message identifies an object and invokes an operation upon it. The function *simple_msg* defines a relation between object maps which captures the semantics of the operation call. The key idea is that the object identified by the message can be considered a system object, and so we can use the object maps to construct two "smaller" elements of *Spec_val* which must in turn satisfy the *is_an_effect_of* relation. We have:

$$
\begin{array}{|l}
simple\_msg : Specification\_denotation \rightarrow \mathbb{P}(Msg\_argument) \\
\hline
\forall\, sd : Specification\_denotation \bullet \\
\qquad \forall\, rem\_om_1, rem\_om_2 : Map;\ m : Message\_val;\ in, out : Valuation \bullet \\
\qquad\qquad (rem\_om_1, rem\_om_2, m, in, out) \in simple\_msg(sd) \Leftrightarrow \\
\qquad\qquad\qquad \mathbf{let}\ new\_sys\_ob == m.var\_vals(\pi_1(simple^\sim(m.msg))); \\
\qquad\qquad\qquad\quad new\_op\_name == \pi_2(simple^\sim(m.msg)) \bullet \\
\qquad\qquad\qquad\quad \mathbf{let}\ new\_sv_1 == (\mu\, Spec\_val \mid sys\_ob = new\_sys\_ob \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad object\_map = rem\_om_1) \\
\qquad\qquad\qquad\qquad new\_sv_2 == (\mu\, Spec\_val \mid sys\_ob = new\_sys\_ob \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad object\_map = rem\_om_2) \\
\qquad\qquad\qquad\qquad new\_label == (in, new\_op\_name, out) \bullet \\
\qquad\qquad\qquad\qquad\qquad\qquad new\_sv_2\ is\_an\_effect\_of\ (new\_sv_1, new\_label)
\end{array}
$$

7.4 Complex Messages

We use the term complex messages to describe messages generated recursively from other messages. In describing the relation on object maps which these messages generate, we need to (recursively) depend on the relations generated by each type of individual message. The purpose of the following function is to decide which relation to satisfy, based on the type of message input. We have:

$$
\begin{array}{l}
complex\_msg : Specification\_denotation \rightarrow \mathbb{P}(Msg\_argument) \\
\hline
\forall sd : Specification\_denotation \bullet \\
\quad \forall rem\_om_1, rem\_om_2 : Map;\ m : Message\_val;\ in, out : Valuation \bullet \\
\quad\quad (rem\_om_1, rem\_om_2, m, in, out) \in complex\_msg(sd) \Leftrightarrow \\
\quad\quad\quad m.msg = nil \Rightarrow (rem\_om_1, rem\_om_2) \in nil\_msg \\
\quad\quad\quad m.msg \in \mathrm{ran}\ simple \Rightarrow \\
\quad\quad\quad\quad (rem\_om_1, rem\_om_2, m, in, out) \in simple\_msg(sd) \\
\quad\quad\quad m.msg \in \mathrm{ran}\ parallel \Rightarrow \\
\quad\quad\quad\quad (rem\_om_1, rem\_om_2, m, in, out) \in parallel\_msg(sd) \\
\quad\quad\quad m.msg \in \mathrm{ran}\ choice \Rightarrow \\
\quad\quad\quad\quad (rem\_om_1, rem\_om_2, m, in, out) \in choice\_msg(sd) \\
\quad\quad\quad m.msg \in \mathrm{ran}\ sequence \Rightarrow \\
\quad\quad\quad\quad (rem\_om_1, rem\_om_2, m, in, out) \in sequence\_msg(sd) \\
\quad\quad\quad m.msg \in \mathrm{ran}\ hiding \Rightarrow \\
\quad\quad\quad\quad (rem\_om_1, rem\_om_2, m, in, out) \in hiding\_msg(sd) \\
\quad\quad\quad m.msg \in \mathrm{ran}\ rename \Rightarrow \\
\quad\quad\quad\quad (rem\_om_1, rem\_om_2, m, in, out) \in rename\_msg(sd)
\end{array}
$$

Note that we could have used this function in the definition of the *is_an_effect_of* relation, but chose not to because we wanted to reveal more of the structure of the relation in that outline definition.

7.5 The Parallel Message

A parallel message sends two messages at once. According to the semantic description of parallelism in [12], the operator is defined so that the sets of objects affected by each of the two messages must be disjoint. In terms of our model, this means that there must be some partition of the object map (say, om_1 and om_2) so that the first message affects om_1 and the second affects om_2. The second requirement of the parallel operator is that complementary variables from the arguments are merged. That is, if $x?$ is output from the first argument, and $x!$ is input to the second, then the values of $x?$ and $x!$ must be equated. The auxiliary relation *compatible_with*, which was defined in Subsection 7.1, is used for this purpose.

We have:

$$
\begin{array}{|l}
parallel\_msg : Specification\_denotation \rightarrow \mathbb{P}(Msg\_argument) \\
\hline
\forall\, sd : Specification\_denotation \bullet \\
\quad \forall\, rem\_om_1, rem\_om_2 : Map;\ m : Message\_val;\ in, out : Valuation \bullet \\
\quad\quad (rem\_om_1, rem\_om_2, m, in, out) \in parallel\_msg(sd) \Leftrightarrow \\
\quad\quad\quad \mathbf{let}\ m_1 == (\mu\, Message\_val \mid msg = \pi_1(parallel^{\sim}(m.msg)) \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad var\_vals = m.var\_vals); \\
\quad\quad\quad\quad m_2 == (\mu\, Message\_val \mid msg = \pi_2(parallel^{\sim}(m.msg)) \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad var\_vals = m.var\_vals) \bullet \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \exists\, om_1, om_2, om_3, om_4 : Map; \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad in_1, in_2, out_1, out_2 : Valuation \bullet \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad om_1 \cup om_2 = rem\_om_1 \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad om_3 \cup om_4 = rem\_om_2 \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad in_1 \cup in_2 = in \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad out_1 \cup out_2 = out \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad in_1\ compatible\_with\ out_2 \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad in_2\ compatible\_with\ out_1 \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad (om_1, om_3, m_1, in_1, out_1) \in complex\_msg(sd) \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad (om_2, om_4, m_2, in_2, out_2) \in complex\_msg(sd)
\end{array}
$$

7.6 The Choice Message

The effect of the non-deterministic choice operator is the effect of exactly one of the arguments to the operator. Operationally, the choice is angelic. So if only one of the argument messages is possible, it will be chosen. If both are possible then the choice is non-deterministic.

Semantically, this can be represented as a disjunction, where each disjunct defines a (mini-)system evolution. The disjuncts are mutually exclusive. We have:

$$
\begin{array}{|l}
choice\_msg : Specification\_denotation \rightarrow \mathbb{P}(Msg\_argument) \\
\hline
\forall\, sd : Specification\_denotation \bullet \\
\quad \forall\, rem\_om_1, rem\_om_2 : Map;\ m : Message\_val;\ in, out : Valuation \bullet \\
\quad\quad (rem\_om_1, rem\_om_2, m, in, out) \in choice\_msg(sd) \Leftrightarrow \\
\quad\quad\quad \mathbf{let}\ m_1 == (\mu\, Message\_val \mid msg = \pi_1(choice^{\sim}(m.msg)) \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad var\_vals = m.var\_vals); \\
\quad\quad\quad\quad m_2 == (\mu\, Message\_val \mid msg = \pi_2(choice^{\sim}(m.msg)) \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad var\_vals = m.var\_vals) \bullet \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad ((rem\_om_1, rem\_om_2, m_1, in, out) \in complex\_msg(sd) \lor \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad (rem\_om_1, rem\_om_2, m_2, in, out) \in complex\_msg(sd))
\end{array}
$$

7.7 The Sequential Composition Message

The sequential operator involves an interesting idea. That is, at one level we view $Op_1 \mathbin{⨟} Op_2$ as an atomic transition. To decide that this transition is valid however, it is necessary to show the existence of an intermediate (mini-)system value om_1 which results from the application of Op_1, and in which Op_2 can be applied.

In Object-Z, complementary variables in the output of Op_1 and input of Op_2 are equated. We have:

$$
\begin{array}{l}
sequential\_msg : Specification\_denotation \to \mathbb{P}(Msg\_argument) \\
\hline
\forall\, sd : Specification\_denotation \bullet \\
\quad \forall\, rem\_om_1, rem\_om_2 : Map;\ m : Message\_val;\ in, out : Valuation \bullet \\
\quad\quad (rem\_om_1, rem\_om_2, m, in, out) \in sequential\_msg(sd) \Leftrightarrow \\
\quad\quad\quad \mathbf{let}\ m_1 == (\mu\, Message\_val \mid msg = \pi_1(sequence^{\sim}(m.msg)) \\
\quad\quad\quad\quad\quad\quad\quad var\_vals = m.var\_vals); \\
\quad\quad\quad\quad m_2 == (\mu\, Message\_val \mid msg = \pi_2(sequence^{\sim}(m.msg)) \\
\quad\quad\quad\quad\quad\quad\quad var\_vals = m.var\_vals) \bullet \\
\quad\quad\quad\quad\quad\quad \exists\, om_1 : Map; \\
\quad\quad\quad\quad\quad\quad\quad out_1, in_2 : Valuation \bullet \\
\quad\quad\quad\quad\quad\quad\quad\quad in_2\ compatible\_with\ out_1 \\
\quad\quad\quad\quad\quad\quad\quad\quad (rem\_om_1, om_1, m_1, in, out_1) \in complex\_msg(sd) \\
\quad\quad\quad\quad\quad\quad\quad\quad (om_1, rem\_om_2, m_2, in_2, out) \in complex\_msg(sd)
\end{array}
$$

7.8 Hiding

The hiding operator restricts the interface of an operation. Taking a top-down approach, the effect of the hiding operator is to expand the underlying interface. That is, the interface of the underlying operations includes bindings for the variables hidden at the level above. Formally,

$$
\begin{array}{l}
hiding\_msg : Specification\_denotation \to \mathbb{P}(Msg\_argument) \\
\hline
\forall\, sd : Specification\_denotation \bullet \\
\quad \forall\, rem\_om_1, rem\_om_2 : Map;\ m : Message\_val;\ in, out : Valuation \bullet \\
\quad\quad (rem\_om_1, rem\_om_2, m, in, out) \in hiding\_msg(sd) \Leftrightarrow \\
\quad\quad\quad \mathbf{let}\ new\_m == (\mu\, Message\_val \mid msg = \pi_1(hiding^{\sim}(m.msg)) \\
\quad\quad\quad\quad\quad\quad\quad var\_vals = m.var\_vals); \\
\quad\quad\quad\quad hidden\_in == (\pi_2(hiding^{\sim}(m.msg))) \rhd \{in\}; \\
\quad\quad\quad\quad hidden\_out == (\pi_2(hiding^{\sim}(m.msg))) \rhd \{out\} \bullet \\
\quad\quad\quad\quad \exists\, new\_in, new\_out : Valuation \bullet \\
\quad\quad\quad\quad\quad in \subseteq hidden\_in \\
\quad\quad\quad\quad\quad hidden\_in \subseteq \operatorname{dom} new\_in \\
\quad\quad\quad\quad\quad hidden\_in \lhd new\_in = in \\
\quad\quad\quad\quad\quad out \subseteq hidden\_out \\
\quad\quad\quad\quad\quad hidden\_out \subseteq \operatorname{dom} new\_out \\
\quad\quad\quad\quad\quad hidden\_out \lhd new\_out = out \\
\quad\quad\quad\quad\quad (rem\_om_1, rem\_om_2, new\_m, new\_in, new\_out) \in complex\_msg(sd)
\end{array}
$$

7.9 Renaming

Finally, the renaming operator renames the interface of the underlying operation.

$$
\begin{array}{|l}
rename\_msg : Specification\_denotation \rightarrow \mathbb{P}(Msg\_argument) \\
\hline
\forall\, sd : Specification\_denotation \bullet \\
\quad \forall\, rem\_om_1, rem\_om_2 : Map;\ m : Message\_val;\ in, out : Valuation \bullet \\
\quad\quad (rem\_om_1, rem\_om_2, m, in, out) \in rename\_msg(sd) \Leftrightarrow \\
\quad\quad\quad \textbf{let}\ new\_m == (\mu\ Message\_val \mid msg = \pi_1(rename^\sim(m.msg)) \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad var\_vals = m.var\_vals); \\
\quad\quad\quad\quad\quad\quad ren\_fn == \pi_2(rename^\sim(m.msg)) \bullet \\
\quad\quad\quad\quad \textbf{let}\ new\_in == ren\_fn^\sim \,⨾\, in \cup (\mathrm{dom}\, ren\_fn \lhd in); \\
\quad\quad\quad\quad\quad new\_out == ren\_fn^\sim \,⨾\, out \cup (\mathrm{dom}\, ren\_fn \lhd out) \\
\quad\quad\quad\quad\quad\quad (rem\_om_1, rem\_om_2, new\_m, new\_in, new\_out) \in complex\_msg(sd)
\end{array}
$$

8 Conclusion

We have presented a semantic model for Object-Z that extends the existing semantic universe. This model includes denotations, which are mathematical objects that directly describe syntactic components of a specification; and meanings, which are mathematical interpretations of the denotations.

The model incorporates a reference semantics for object identity, and also accommodates object structure and behaviour. We have shown that by adopting a dual notion of object operations one can give a meaning to classes which is independent of their environment and hence modular.

Finally, we have presented a function which composes system structure from class structure, and outlined a function which composes system behaviour from the behaviour of individual objects. The details of that function were also presented: those details represent a full semantics for operation invocation in Object-Z.

8.1 Applications and Future Work

We conclude by outlining some applications of the ideas presented in this paper, and suggest avenues for further research.

The most important aspect of this work is that it provides a means for linking structure and behaviour in a model-oriented specification technique. There are several possible applications of this idea.

Firstly, object compatability has traditionally been formally defined in terms of object histories. By linking object structure with behaviour we permit a characterisation of object compatability in terms of the underlying structure of objects.

Secondly, the concept of inheritance is frequently motivated as a basis for polymorphism, a concept largely reliant on behaviour. In contrast, the model oriented formal specification languages have traditionally stipulated rules for inheritance based on structure. Our model provides a formal framework in which these rules could be justified.

There are structural similarities between the algebras which denote the meaning of a class and the meaning of a specification. It is likely that these similarities might be emphasized if a categorical approach to the semantics was adopted. One obvious result that might flow from such an approach is a generalisation of the composition function to specifications themselves.

Finally, the set of transitions defined by a class or by a system can be used to define a pre-order on object or system values (respectively). The pre-order could then be used to give a trace model for object and system behaviour. This would provide a domain in which to link the semantics of history predicates in Object-Z[3] with the semantics presented in this paper.

9 Acknowledgements

Thanks to Roger Duke and Gordon Rose for reading earlier drafts of this paper. Thanks also to Carroll Morgan for an invigorating discussion about the ideas. Particular thanks to Mark Utting for meticulously reviewing the paper.

10 Appendix

The definition of *ob_id_vals* is as follows:

$$
\begin{array}{|l}
ob\_id\_vals : \mathbb{P}\,\mathcal{V} \rightarrow \mathbb{P}\,\mathbb{O} \\
\hline
ob\_id\_vals = \lambda\, pv : \mathbb{P}\,\mathcal{V} \bullet \\
\qquad \bigcup obs(\!|\; pv \;|\!)
\end{array}
$$

where,

$$
\begin{array}{|l}
obs : \mathcal{V} \rightarrow \mathbb{P}\,\mathbb{O} \\
\hline
\forall\, v : \mathcal{V} \bullet \\
\qquad v \in \bigcup \mathrm{ran}(givenT \lhd carrier) \Rightarrow \\
\qquad\qquad obs(v) = \varnothing \\
\qquad v \in \bigcup \mathrm{ran}(powerT \lhd carrier) \Rightarrow \\
\qquad\qquad obs(v) = \bigcup obs(\!|\; v \;|\!) \\
\qquad v \in \bigcup \mathrm{ran}(cproductT \lhd carrier) \Rightarrow \\
\qquad\qquad obs(v) = \bigcup obs(\!|\; tuple\_to\_set(v) \;|\!) \\
\qquad v \in \bigcup \mathrm{ran}(schemaT \lhd carrier) \Rightarrow \\
\qquad\qquad obs(v) = \bigcup obs(\!|\; \mathrm{ran}\, v \;|\!) \\
\qquad v \in \mathbb{O} \Rightarrow \\
\qquad\qquad obs(v) = \{v\}
\end{array}
$$

In the above definition $givenT$, $powerT$, $cproductT$, $schemaT$, and $\mathbb{O}$ are all as defined in [6].

The function *tuple_to_set* takes an ordered tuple and returns a set of elements. To define this operator one must step outside the language Z and rely on the set-theoretic definition of ordered tuples. This does not represent a change in style of semantic definition; the semantics requires meta-operators (defined in [6]) which are defined using general set theory — not strict Z.

An ordered tuple $(x_1, x_2, ..., x_n)$ is generally represented set-theoretically as $\{\{x_1\}, \{x_1, x_2\}, ..., \{x_1, x_2, ..., x_n\}\}$. The *tuple_to_set* operator, relying on this definition, may be defined as:

$$
\begin{array}{|l}
tuple\_to\_set : \bigcup \operatorname{ran}(cproductT \lhd carrier) \rightarrow \mathbb{P}\,\mathcal{V} \\
\hline
tuple\_to\_set == \lambda\, t : \bigcup \operatorname{ran}(cproductT \lhd carrier) \bullet \bigcup t
\end{array}
$$

This completes the definition of the function *ob_id_vals*.

References

[1] S Brien, T King, J Nicholls, J Woodcock, and J Wordsworth. Z Base Standard — Version 1.0. Technical report, Oxford University Computing Laboratory - Programming Research Group, November 1992.

[2] D Duke. *Object-Oriented Formal Specification.* PhD thesis, University of Queensland, 1991.

[3] D Duke and R Duke. Towards a semantics for Object-Z. In D. Bjørner, C.A.R. Hoare, and H. Langmaack, editors, *VDM'90: VDM and Z!*, volume 428 of *Lecture Notes in Computer Science*, pages 242–262. Springer-Verlag, 1990.

[4] R Duke. Integrating Formal Methods with Object-Oriented Software Engineering. In *Joint Conference on Software Engineering*, 1993.

[5] R Duke and G Rose. Modelling Object Identity. In *Proceedings 16th Australian Computer Science Conference (ACSC-16)*, pages 93–100, February 1993.

[6] A Griffiths and G Rose. A Semantic Foundation for Object Identity in Formal Specification. Technical Report 95-38, Software Verification Research Centre, Dept. of Computer Science, Univ. of Queensland, September 1995.

[7] Richard Holt and Dennis deChampeaux. A Framework for Using Formal Methods In Object-Oriented Software Development. In *Followup Report on ECOOP '91 Workshop W3: Object-Oriented Formal Methods*, 1991.

[8] K Lano. Formal Methods and Object-orientation: A Historical Perspective. *FACS Europe — Series I Vol. 2, No. 1*, pages 13–16, Summer 1995.

[9] K Lano and H Haughton, editors. *Object-Oriented Specification Case Studies.* Prentice-Hall, 1994.

[10] Kevin Lano. Refinement in Object-Oriented Specification Languages. In *6th Annual Refinement Workshop*, 1994.

[11] S Schuman and D Pitt. Object-oriented Subsystem Specification. In *Program Specification and Transformation.* North-Holland, 1987.

[12] G Smith. Ninth International Conference of Z Users. In J P Bowen and M G Hinchey, editors, *Extending $\mathcal{W}$ for Object-Z*, Lecture Notes in Computer Science. Springer-Verlag, 1995.

[13] J M Spivey. *Understanding Z: A specification language and its formal semantics*, volume 3 of *Cambridge Tracts in Theoretical Comput. Sci.* Cambridge University Press, UK, 1988.