Class Union and Polymorphism

Jin Song Dong **Roger Duke**

Software Verification Research Centre
Department of Computer Science
University of Queensland, Australia
jason@cs.uq.oz.au rduke@cs.uq.oz.au

Abstract

Polymorphism within object-oriented systems has traditionally been restricted to suitable inheritance hierarchies. This restriction arose historically and has persisted for pragmatic reasons despite the fact that it has long been recognized that polymorphism and inheritance are orthogonal concepts. In this paper some examples are considered that suggest a form of polymorphism not related to inheritance. Furthermore, these examples illustrate that polymorphism can have some meaning even in cases where the underling classes are not subtypes and have little, if any, behavioural compatibility. The specification of such systems is facilitated by a class-union construct; this construct is defined and its implications explored[1].

1 Introduction

In all commercial object-oriented programming languages such as Eiffel[Mey88, Mey92] and C++[Str86], object polymorphism (i.e. dynamic binding, as distinct from genericity) is defined within the subclass structure induced by inheritance. The reasons for this are partly historical and partly pragmatic: the mechanism of inheritance allows for code sharing, and this is a good starting point to capture other requirements such as signature compatibility[WZ88] needed for polymorphism. Languages such as Eiffel go further, taking a strict view of polymorphism and imposing a limited form of behavioural compatibility (subtyping) on inheritance hierarchies. As has long been recognized however [CW85, CHC89, WZ88], the marriage of polymorphism and inheritance can result in conflict when the role of inheritance as simple code sharing is contrasted with the requirements for behavioural compatibility.

This paper is concerned with specifying (rather than implementing) object-oriented systems and a very general view of polymorphism is adopted. In Section 2 we describe informally some simple systems that strongly suggest a form of polymorphism that does not easily fit within an inheritance hierarchy. Indeed, these examples suggest that polymorphism can involve classes not related by subtyping.

In Section 3 a class-union construct is defined. This enables the construction of suitable sets of object references where the underlying objects so referenced are not necessarily all of the same class. This construction is applied to the systems informally discussed in Section 2, with class-union enabling the polymorphic declaration of object references. Indeed, as explained in Section 4, the traditional view of polymorphism expressed within an inheritance hierarchy can also be described using the class-union construct. In Section 5, a more complex case study of the application of class-union is presented.

As implementational aspects of polymorphism are not of concern here, throughout this paper polymorphism will be discussed within the context of system specification. To facilitate this discussion, the Object-Z[2][DKRS91] notation will be used, although only the basic features of this notation will be required. It is assumed, however, that the reader is familiar with object-oriented concepts and with the notion of

[1] In Proceedings of the *International Conference on Technology of Object-Oriented Languages and Systems (Tools 12&9)*, 1993, pp.181-190, Prentice Hall.

[2] Object-Z is an extension of the Z[Spi89] specification language to accommodate object-oriented concepts.

polymorphism as it is traditionally understood in object-oriented systems.

2 Polymorphic References

An object reference within (the specification of) a system is said to be *polymorphic* if the class of the object so referenced is not uniquely determined. Polymorphic object references traditionally form the basis for polymorphism in object-oriented systems, both in programming [Mey88] and in formal specification [DKRS91]. In this section, however, three examples are considered where polymorphic object references are applied in a non-traditional context.

2.1 Example: A Power Tool

A power tool can be fitted with either an attachment for drilling or an attachment for sawing, where such attachments are objects of classes *Drill* or *Saw* specified by

$$
\begin{array}{|l}
\textit{Drill} \\ \hline
\textit{DrillState} ::= \textit{still} \mid \textit{drilling} \\[4pt]
\begin{array}{|l}
\hline
\textit{mode} : \textit{DrillState} \\ \hline
\end{array} \\[4pt]
\begin{array}{|l}
\textit{DrillHole} \\ \hline
\Delta(\textit{mode}) \\ \hline
\textit{mode}' = \textit{drilling} \\ \hline
\end{array} \\[4pt]
\begin{array}{|l}
\textit{Stop} \\ \hline
\Delta(\textit{mode}) \\ \hline
\textit{mode}' = \textit{still} \\ \hline
\end{array} \\ \hline
\end{array}
\qquad
\begin{array}{|l}
\textit{Saw} \\ \hline
\textit{SawState} ::= \textit{still} \mid \textit{sawing} \\[4pt]
\begin{array}{|l}
\hline
\textit{mode} : \textit{SawState} \\ \hline
\end{array} \\[4pt]
\begin{array}{|l}
\textit{Cut} \\ \hline
\Delta(\textit{mode}) \\ \hline
\textit{mode}' = \textit{sawing} \\ \hline
\end{array} \\[4pt]
\begin{array}{|l}
\textit{Stop} \\ \hline
\Delta(\textit{mode}) \\ \hline
\textit{mode}' = \textit{still} \\ \hline
\end{array} \\ \hline
\end{array}
$$

In this case the power tool specification would (informally) look something like

$$
\begin{array}{|l}
\textit{PowerTool} \\ \hline
\begin{array}{|l}
\hline
\textit{att} : \textit{Drill}\ \mathbf{or}\ \textit{Saw} \\
\textit{power\_on} : \mathbb{B} \\ \hline
\neg\ \textit{power\_on} \Rightarrow \textit{att.mode} = \textit{still} \\ \hline
\end{array} \\[4pt]
\begin{array}{|l}
\textrm{I}\scriptstyle{\textrm{NIT}} \\ \hline
\neg\ \textit{power\_on} \\ \hline
\end{array} \\[4pt]
\begin{array}{|l}
\textit{SwitchOn} \\ \hline
\Delta(\textit{power\_on}) \\ \hline
\neg\ \textit{power\_on} \\
\textit{power\_on}' \\ \hline
\end{array}
\quad
\begin{array}{|l}
\textit{SwitchOff} \\ \hline
\Delta(\textit{power\_on}) \\ \hline
\textit{power\_on} \\
\neg\ \textit{power\_on}' \\ \hline
\end{array} \\[4pt]
\begin{array}{|l}
\textit{Change} \\ \hline
\Delta(\textit{att}) \\
\textit{at}? : \textit{Drill}\ \mathbf{or}\ \textit{Saw} \\ \hline
\neg\ \textit{power\_on} \wedge \textit{att}' = \textit{at}? \\ \hline
\end{array} \\[4pt]
\textit{DrillHole} \mathrel{\widehat=} [\textit{att} \in \textit{Drill}] \bullet \\
\qquad\qquad \textit{SwitchOn} \wedge \textit{att.DrillHole} \\[4pt]
\textit{Cut} \mathrel{\widehat=} [\textit{att} \in \textit{Saw}] \bullet (\textit{SwitchOn} \wedge \textit{att.Cut}) \\[4pt]
\textit{Stop} \mathrel{\widehat=} \textit{SwitchOff} \wedge \textit{att.Stop} \\ \hline
\end{array}
$$

Without at this stage precisely defining the semantics of the type *Drill* **or** *Saw*, the intention is that the attribute *att* is a polymorphic object reference : its value is a reference to an attached object in either the *Drill* or *Saw* class. The traditional way to express this polymorphism would be to construct a class *Attachment*, say, of which both *Drill* and *Saw* would be subclasses, and then declare

$$\textit{att} : \downarrow \textit{Attachment}.^3$$

The construction of such a super class, however, is somewhat artificial as distinct attachments may have little in common (except that they must interface with the power tool) and may display completely unrelated functionality.

Notice that the operations *DrillHole* and *Cut* are conditional upon the attribute *att* being in the appropriate class. These operations are reminiscent of the type-down-cast construction in C++. The operation *Stop*, on the other hand, is a polymorphic operation since it is defined irrespective of the class of *att*. Similarly, the class invariant $\neg\ \textit{power\_on} \Rightarrow \textit{att.mode} = \textit{still}$ is interpreted polymorphically.

The *Change* operation enables the attachment to be changed provided the power is turned off. The input to this operation, *att*?, is again a polymorphic object reference.

[3] In Object-Z, the notation $a : \downarrow A$ means that a is an identity of an object in class A or in any subclasses (by inheritance) of A.

2.2 Example: Ginger Meggs

Ginger Meggs[4] is a boy whose pocket is often crammed with various bits and pieces such as marbles, string, etc. This treasure is constantly being collected and discarded. Objects can be examined, e.g. marbles can be polished, string can be knotted, and so on. An (informal) specification of Ginger Meggs might look something like:

GingerMeggs

$$pocket : \mathbb{P}(Marble \ \mathbf{or}\ String)$$

Collect

$$\begin{array}{l} \Delta(pocket) \\ x? : Marble \ \mathbf{or}\ String \\ \hline x? \notin pocket \\ pocket' = pocket \cup \{x?\} \end{array}$$

Discard

$$\begin{array}{l} \Delta(pocket) \\ x! : Marble \ \mathbf{or}\ String \\ \hline x! \in pocket \\ pocket' = pocket - \{x!\} \end{array}$$

SelectMarble

$$\begin{array}{l} m? : Marble \\ \hline m? \in pocket \end{array}$$

SelectString

$$\begin{array}{l} s? : String \\ \hline s? \in pocket \end{array}$$

$$PolishMarble \mathrel{\widehat{=}} SelectMarble \bullet m?.Polish$$

$$KnotString \mathrel{\widehat{=}} SelectString \bullet s?.Knot$$

The attribute *pocket* is a set of polymorphic object references: each element of *pocket* is a reference to an object in either the *Marble* or *String* class. The operations *Collect* and *Discard* are specified polymorphically, but the operations *PolishMarble* and *KnotString* can only be applied to objects of the appropriate class.

As specified, Ginger Meggs can only have marbles and string in his pocket. To enable worms and other treasure to be collected the type of *pocket* would need to be extended. Hence any notation for declaring polymorphic object references would need to be easily extendible. Restricting polymorphism to subclass hierarchies has this advantage, as the extendibility of the hierarchy is facilitated by the mechanism of inheritance.

In the case of Ginger Meggs, however, there seems to be no a-priori class structure that would include classes as distinct as *Marble* and *String* and to which classes like *Worm* could be added by inheritance. Certainly any such hierarchy would not be considered polymorphic in the traditional sense. Furthermore, if the type of *pocket* is extended it may well be necessary to extend the operations in the class *GingerMeggs* as well, e.g. specific operations for worms would be needed (although what such operations might be is left to the imagination of the reader). However, the operations *Collect* and *Discard* should remain unaltered despite any extension.

2.3 Example: A Channel

One kind of polymorphism supported by essentially all programming languages is function and operator overloading. For example, in procedural languages the assignment statement is a polymorphic operator because it supports both the assignment of the value of a boolean expression to a boolean variable, and the assignment of the value of an integer expression to a integer variable. The following example is based on this idea.

Classes *Nat* and *Bool* are defined as follows:

Nat

$$n : \mathbb{N}$$

INIT

$$n = 0$$

Update

$$\begin{array}{l} \Delta(n) \\ x? : \mathbb{N} \\ \hline n' = n + x? \end{array}$$

OutVal

$$\begin{array}{l} x! : \mathbb{N} \\ \hline x! = n \end{array}$$

Bool

$$b : \mathbb{B}$$

INIT

$$b$$

Update

$$\begin{array}{l} \Delta(b) \\ x? : \mathbb{B} \\ \hline b' = x? \end{array}$$

OutVal

$$\begin{array}{l} x! : \mathbb{B} \\ \hline x! = b \end{array}$$

The behaviour of objects of the classes *Nat* and *Bool* are completely different: the interface and functionality of even the common-named operations *Update* and *OutVal* are different. However, consider now the class *Channel* specified as follows:

[4] Ginger Meggs is an Australian cartoon character.

$$
\begin{array}{|l}
\hline
\textit{Channel} \\
\hline
\begin{array}{|l}
\hline
\textit{sender}, \textit{receiver} : \textit{Nat} \ \mathbf{or} \ \textit{Bool} \\
\hline
\{\textit{sender}, \textit{receiver}\} \subseteq \textit{Nat} \\
\quad \vee \{\textit{sender}, \textit{receiver}\} \subseteq \textit{Bool} \\
\textit{sender} \neq \textit{receiver} \\
\hline
\end{array} \\
\\
\begin{array}{|l}
\hline
\textit{Switch} \\
\hline
\Delta(\textit{sender}, \textit{receiver}) \\
s?, r? : \textit{Nat} \ \mathbf{or} \ \textit{Bool} \\
\hline
\textit{sender}' = s? \wedge \textit{receiver}' = r? \\
\hline
\end{array} \\
\\
\textit{Trans} \mathrel{\widehat{=}} \textit{sender}.\textit{OutVal} \parallel \textit{receiver}.\textit{Update} \\
\hline
\end{array}
$$

The attributes *sender* and *receiver* are polymorphic object references, but the class invariant ensures that both *sender* and *receiver* are of the same class. Another kind of polymorphism is represented by the *Trans* operation: this operation can be performed on the objects *sender* and *receiver* regardless of whether they are both in class *Nat* or class *Bool*. However, the component operations *OutVal* and *Update* are not polymorphic because the type of the parameters is different. The *Trans* operation essentially sets up a polymorphic channel that enables two objects of *Nat* or two objects of *Bool* to communicate.

3 Class Union

Underlying the semantics of an Object-Z specification is the understanding that each object has a unique persistent identity that distinguishes it from every other object, regardless of class[DR93]. An object's identity is invariant irrespective of changes to its state.

If A is the name of a class, let the identifier A also denote the set of identities of possible objects of class A.[5] In any system being specified, let set $\mathbb{O}$ denote the universe of all object identities, regardless of class, that may be in the system.[6]

If A and B are distinct classes in the system,

$$A \subseteq \mathbb{O}, \ B \subseteq \mathbb{O} \text{ and } A \cap B = \varnothing.$$

[5] In particular cases, it will be clear from the context whether an identifier is being used as a name of a class or to denote the set of identities of objects of that class.

[6] In any system only a finite number of distinct classes will be specified.

At our level of abstraction the structure of $\mathbb{O}$ is left unspecified.

Informally, we do not distinguish between an object and its identity, i.e. if we refer to some *object* in $\mathbb{O}$, it is with the understanding that we are in fact referring to the object whose identity is in $\mathbb{O}$. As object identities uniquely identify objects, no ambiguity arises. Subsets of $\mathbb{O}$ can be formed by the union of sets of object identities, e.g. if A and B are distinct classes, $A \cup B$ is the collection

$$\{obj : \mathbb{O} \mid obj \in A \vee obj \in B\}$$

of objects in $\mathbb{O}$. In this case, if we define $C \mathrel{\widehat{=}} A \cup B$ then C is not itself a class (see Figure 1) but only a set of object identities.

3.1 Polymorphic Core

Associated with any object is a set of operations. Each operation is named and has a parameter list declaring the name and type of each input and output parameter for that operation. Given a collection of objects, possibly from different classes, an operation named *Op*, say, is said to be *polymorphic with respect to the collection* if

- every object in the collection has an associated operation named *Op*;
- for each object in the collection, the parameter list of the operation *Op* is the same, i.e. the input and output parameters are identically named and typed.

This notion of polymorphic operation is similar to signature compatibility[7][WZ88] and is introduced for the same reason: it establishes a minimal condition under which messages to objects declared polymorphically can be interpreted regardless of the actual class to which they belong.

There are differences, however, between this notion of polymorphic operation and the traditional view of signature compatibility in object-oriented systems. We do not insist that an operation common to the objects in the collection must be polymorphic; collections will be permitted where common-named operations have distinct parameter lists. Furthermore, there need

[7] Signature compatibility requires that if two objects have the same named operation, the interface of that operation must be the same in both cases.

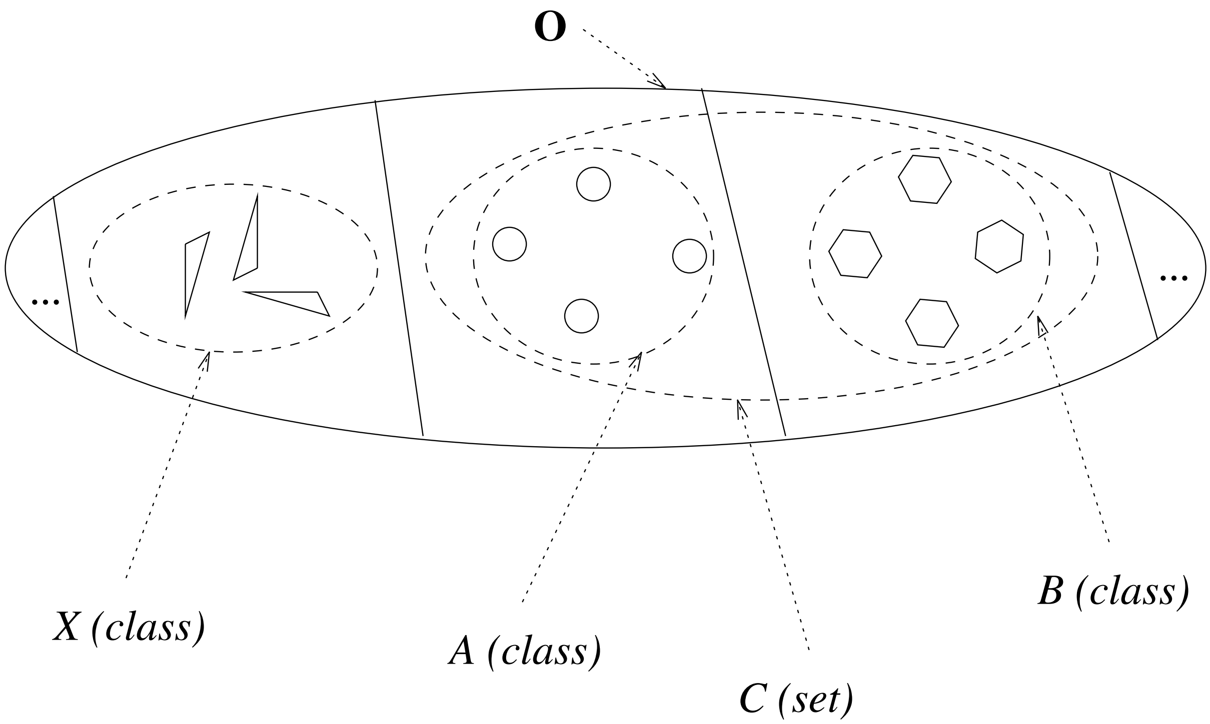

Figure 1: Class Union

not be a distinguished object, or set of objects, in the collection whose operations are each associated polymorphically with all the other objects in the collection; i.e. we shall not suppose the existence of a parent class effectively defining by inheritance the set of polymorphic operations.

Associated with any collection of objects is the set of all operations that are polymorphic with respect to the collection. This set of operations will be called the *polymorphic core* of the collection. It is the set of common operations in the collection that can be applied polymorphically, i.e. without knowing in advance the actual class of an object.

3.2 Examples Revisited

With the notions of class union and polymorphic operations, the Section 2 examples can be revisited and the specification of polymorphic object references formalized.

Power Tool

In the class *PowerTool* the attribute *att* can now be declared as

$$att : Attachment$$

where

$$Attachment \mathrel{\widehat{=}} Drill \cup Saw.$$

The operation *Stop* is in the polymorphic core of the collection *Attachment* and consequently this operation can be used polymorphically in the specification. The other operations, *DrillHole* and *Cut*, must be qualified by a predicate involving the specific type of the attachment. The operation *Change* is an operation to externally manipulate the attachments, rather than to operate upon the attachments themselves; the input *at*? can now also be declared as

$$at? : Attachment.$$

Ginger Meggs

In the class *GingerMeggs* the attribute *pocket* can now be declared as

$$pocket : \mathbb{P}\ Treasure$$

where

$$Treasure \mathrel{\widehat{=}} Marble \cup String.$$

The polymorphic core of *Treasure* is empty so all operations upon the contents of the pocket must be qualified by a predicate involving object type. Similarly, the parameters $x?$ and $x!$ in the operations *Collect* and *Discard* will also have the type *Treasure*.

Communication Channel

In the class *Channel* the attributes *sender* and *receiver* can now be declared as

$$sender, receiver : NatBool$$

where

$$NatBool \mathrel{\widehat{=}} Nat \cup Bool.$$

The inputs $s?$, $r?$ in the operation *Switch* will also have the type *NatBool*. The polymorphic core of *NatBool* is empty despite the fact that the operations *Update* and *OutVal* occur in both *Nat* and *Bool*; the parameter list of these operations differ between the classes. The operation *Trans* effectively qualifies the type of the underlying objects because the class invariant ensures *sender* and *receiver* are the same type.

3.3 Extending Class Unions

The specification of the power tool in Sections 2.1 and 3.2.1 has the property that the value of the attribute *att* is restricted to being a reference to an object in either the *Drill* or *Saw* class. To extend the specification of the power tool to also accept a plane attachment, i.e. an object of class *Plane*, say, it would be necessary to redefine

$$Attachment \mathrel{\widehat{=}} Drill \cup Saw \cup Plane.$$

The existing specification of the class *PowerTool* remains valid under such an extension of the class union type provided the *Plane* class has an operation *Stop* compatible with the same-named operation in the *Drill* and *Saw* classes. If such an operation does exist in the *Plane* class, the polymorphic application of *Stop* in the *PowerTool* class remain valid.

This observation suggests the following rule:

> a class-union type can be extended, without affecting existing specifications using that type, provided the extended type has the same polymorphic core as the original.

That is, extension of class-union types is permitted if the polymorphic core is preserved[8].

In the specification of Ginger Meggs in Sections 2.2 and 3.2.2, the polymorphic core of the class-union *Treasure* is empty. Hence the type *Treasure* can be extended to include class *Worm*, etc. without affecting the existing specification.

4 Class Union and Traditional Polymorphism

The class union construction facilitates the declaration of polymorphic object references. But the traditional notation $\downarrow C$ where C is a class also facilitates such declarations. In this section we compare the two points of view which we see as complementary.

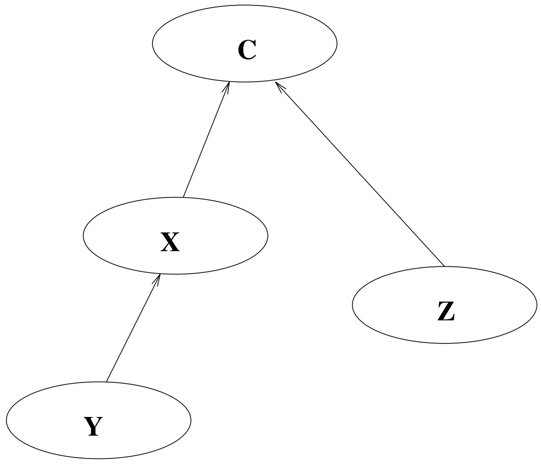

Figure 2: An Inheritance Hierarchy

Consider the inheritance hierarchy in Figure 2 (the actual features of the classes will not be of importance). A declaration $c : \downarrow C$ indicates that the value of c is a polymorphic reference to an object in one of the class C, X, Y or Z. Such an intention could also have been captured by the declaration

$$c : C \cup X \cup Y \cup Z.$$

However, there are two significant distinctions between the way in which these declarations are applied within a specification.

[8] In specific cases, extension may be possible even if the polymorphic core is not preserved, provided those operations not preserved are not applied polymorphically.

Firstly, the notation $\downarrow C$ is used only if the class hierarchy with C at its root satisfies strict signature compatibility[WZ88], namely, all the operations defined in C must be inherited in all the classes below C in the hierarchy, and in each class an operation inherited from C must have an identical parameter list. In other words, the polymorphic core of the union of all the classes in the hierarchy determined by C is precisely the set of operations defined in C. This rule is implicit in the notation $\downarrow C$ but not in the class-union notation which is designed to be applicable in wider contexts.

Secondly, if the declaration $c : \downarrow C$ appears in the specification of a class A, say, it is understood that only operations defined in C (i.e. only operations in the polymorphic core of the hierarchy determined by C) will be applied to c within the specification of A: there will be no qualification of the type of c (i.e. no type-down-casting) so as to apply an operation in X, say, which is not an operation inherited from C. However, declarations such as

$$c : C \cup X \cup Y \cup Z$$

involving the class union allow operations not in C (or more generally, not in the polymorphic core) to be accessed provided the operations are qualified by an explicit predicate that places a suitable restriction on the type of c.

Another aspect of class union can also be illustrated by the inheritance hierarchy in Figure 2. Using class-union it is possible to declare

$$d : C \cup X \cup Y$$

where class Z is deliberately excluded. Such a selection may be made, for example, when only part of the inheritance hierarchy satisfies strict signature compatibility: those parts not conforming to the requirements of operation polymorphism can be excluded. Hence it is possible to take inheritance hierarchies that do not conform to the strict requirements for polymorphism throughout, but can never-the-less be used polymorphically after pruning using the class-union construct. The selection of such substructures using the notation $\downarrow C$ is not possible.

5 Case Study: A Telephone System

In this section class union is applied to the specification of a simple telephone system. This system consists of a collection of telephones that each conform to the core operations of sending and receiving calls, etc. but which can exhibit quite different functionality outside this core. The class-union construct is ideal for specifying such systems.

We take a greatly simplified view of a telephone system, concentrating upon the issue of interphone connection. A more detailed Object-Z specification capturing other aspects of a telephone system is given in[Lee92].

5.1 Informal View of the System

The telephone system specified here consists of a collection of one-line and multi-line telephones, and considers the connections that can be made between them.

A one-line telephone can be connected to at most one other telephone at any one time. When busy, such a telephone cannot send or receive additional calls, i.e. all attempts to dial a busy one-line telephone will fail.

A multi-line telephone, on the other hand, can engage in several calls at once. Even when busy, incoming calls can be accepted and placed on hold. Likewise, outgoing calls can be made after first placing all existing connections on hold.

As a simplification, we shall ignore issues such as ringing telephones only being connected after the receiver is lifted: we shall suppose that connection occurs immediately when a call is successfully sent.

5.2 Specification of the System

At any time a one-line telephone is either idle or busy, while a multi-line telephone is either idle, holding or busy. Hence define

$$\begin{aligned} OnePhState &::= idle \mid busy \\ MultiPhState &::= idle \mid holding \mid busy. \end{aligned}$$

A one-line telephone is an object of the class *OneLinePh*, while a multi-line telephone is an object of the class *MultiLinePh*. In the specification of both these classes it will be necessary to declare polymorphic object references whose value is a reference to objects of either class. Hence define the class union

$$Phone \mathrel{\widehat=} OneLinePh \cup MultLinePh.$$

OneLinePh

$$\begin{array}{l} s : OnePhState \\ current : Phone \\ \hline s = idle \Rightarrow current = self \end{array}$$

$$\textsc{Init}: \quad s = idle$$

Send

$$\begin{array}{l} \Delta(s, current) \\ to!, from! : Phone \\ \hline from! = self \\ to! \neq self \\ s = idle \\ s' = busy \\ current' = to! \end{array}$$

Receive

$$\begin{array}{l} \Delta(s, current) \\ to?, from? : Phone \\ \hline to? = self \\ from? \neq self \\ s = idle \\ s' = busy \\ current' = from? \end{array}$$

Hangup

$$\begin{array}{l} \Delta(s, current) \\ to!, from! : Phone \\ \hline s = busy \\ s' = idle \\ to! = current \\ from! = self \end{array}$$

Cutoff

$$\begin{array}{l} \Delta(s, current) \\ to?, from? : Phone \\ \hline s = busy \\ s' = idle \\ to? = self \\ from? = current \end{array}$$

The attribute *current* denotes the connected telephone, if there is one. When an object of *OneLinePh* (i.e. a one-line telephone) is instantiated, the identifier *self* will denote the identity of the object itself (see [DR93] for a detailed discussion). The input/output parameters of the operations *Send*, *Receive*, *Hangup* and *Cutoff* are object references, so that communication within a telephone system is modeled by the passing of object identities rather than the passing of detailed state information. (The use of the class-union type *Phone* may seem recursive: the specification of *OneLinePh* involves objects of the same class. However, as a type, *Phone* is a set of identities only. A detailed discussion of this issue is given in [DR93].)

MultLinePh

$$\begin{array}{l} max : \mathbb{N} \\ lines : \mathbb{P}\, Phone \\ s : MultiPhState \\ current : Phone \\ \hline \#lines \leq max \\ lines = \varnothing \Rightarrow s = idle \\ s \in \{idle, holding\} \Rightarrow current = self \end{array}$$

$$\textsc{Init}: \quad lines = \varnothing$$

Send

$$\begin{array}{l} \Delta(s, lines, current) \\ to!, from! : Phone \\ \hline from! = self \\ to! \notin lines \\ to! \neq self \\ \#lines < max \\ s \in \{idle, holding\} \\ s' = busy \\ lines' = lines \cup \{to!\} \\ current' = to! \end{array}$$

Receive

$$\begin{array}{l} \Delta(s, lines) \\ to?, from? : Phone \\ \hline to? = self \\ from? \notin lines \\ from? \neq self \\ \#lines < max \\ s \in \{idle, holding\} \\ \quad \Rightarrow s' = holding \\ s = busy \Rightarrow s' = busy \\ lines' = lines \cup \{from?\} \end{array}$$

Retrieve

$$\begin{array}{l} \Delta(s, current) \\ \hline s = holding \\ s' = busy \\ \exists\, p : lines \bullet \\ \quad current' = p \end{array}$$

PutOnHold

$$\begin{array}{l} \Delta(s, current) \\ \hline s = busy \\ s' = holding \end{array}$$

Hangup

$$\begin{array}{l} \Delta(s, lines, current) \\ to!, from! : Phone \\ \hline to! = current \\ from! = self \\ s = busy \\ lines' = lines - \{current\} \\ lines' \neq \varnothing \Rightarrow s' = holding \end{array}$$

Cutoff

$$\begin{array}{l} \Delta(s, lines, current) \\ to?, from? : Phone \\ \hline s \in \{holding, busy\} \\ to? = self \\ from? \in lines \\ lines' = lines - \{from?\} \\ from? = current \wedge lines' \neq \varnothing \\ \qquad \Rightarrow s' = holding \\ from? \neq current \wedge lines' \neq \varnothing \\ \qquad \Rightarrow s' = s \wedge current' = current \end{array}$$

The attribute *lines* denotes the set of telephones currently connected, where the number of such connections is limited by the value of *max*. The attribute *current* denotes the connected telephone not on hold, if there is one.

Compared with a one-line telephone, an additional two operations, *Retrieve* and *PutOnHold*, are defined. Furthermore, the functionality of the operations *Send*, *Receive*, *Answer*, *Hangup* and *Cutoff* is enhanced to reflect the fact that a multi-line telephone can be involved in several connections simultaneously.

A telephone system can now be specified as an aggregation of telephones undergoing connections and disconnections.

PhoneSystem

$$ps : \mathbb{P}\,Phone$$
$$pairs : Phone \leftrightarrow Phone$$
$$\mathrm{dom}\,pairs \cup \mathrm{ran}\,pairs \subseteq ps$$

INIT

$$\forall\, p : ps \bullet p.\mathrm{INIT}$$
$$pairs = \varnothing$$

SelectOne

$$p : Phone$$
$$p \in ps$$

AddTwo

$$p_1, p_2 : Phone$$
$$\neg\ p_1 \ \underline{pairs}\ p_2$$
$$pairs' = pairs \cup \{(p_1, p_2), (p_2, p_1)\}$$

DelTwo

$$p_1, p_2 : Phone$$
$$p_1 \ \underline{pairs}\ p_2$$
$$pairs' = pairs - \{(p_1, p_2), (p_2, p_1)\}$$

$$Connect \mathrel{\widehat=} AddTwo \bullet p_1.Send \parallel p_2.Receive$$

$$Disconnect \mathrel{\widehat=} DelTwo \bullet p_1.Hangup \parallel p_2.Cutoff$$

$$Retrieve \mathrel{\widehat=} SelectOne \bullet [p \in MultLinePh] \wedge p.Retrieve$$

$$PutOnHold \mathrel{\widehat=} SelectOne \bullet [p \in MultLinePh] \wedge p.PutOnHold$$

The state attribute *ps* denotes the set of telephones in the system, while the attribute *pairs* records the (symmetric) connections between the telephones. The operations *Connect* and *Disconnect* establish or terminate the connection between any two telephones, regardless of the type of the telephones, because the operations *Send*, *Receive*, *Hangup* and *Cutoff*, each having the same parameter list in either the class *OneLinePh* or *MultiLinePh*, are all in the polymorphic core of the class union *Phone*. The operations *Retrieve* and *PutOnHold* need to be qualified with a condition that the telephone is multi-line, as these operations are only defined for such telephones.

To complete this section, consider the problem of specifying the telephone system without using class union. One approach would be to define the class *MultiLinePh* by inheriting the class *OneLinePh* and adding the operations *Retrieve* and *PutOnHold*. The class union *Phone* could then be replaced by $\downarrow OneLinePh$. However,

- the *Send* and *Receive* operations of *MultiLinePh* have distinct pre- and post-conditions, so inheritance would only work if arbitrary redefinition of operations were permitted in polymorphic inheritance hierarchies;
- the use of the type $\downarrow OneLinePh$ would disallow the promotion of the operations *Retrieve* and *PutOnHold* from the class *MultLinePh* in the *PhoneSystem* since these operations are not defined in the class *OneLinePh*.

Another approach would be to define *OneLinePh* by inheriting *MultiLinePh* and setting $max = 1$, i.e. consider a one-line telephone to be a multi-line telephone with only one line. Such an inheritance hierarchy is reasonable, but only in hindsight. It would also mean that the operations *Retrieve* and *PutOnHold* would need to be defined for one-line telephones.

A third approach would be to define an abstract *PHONE* superclass with both *OneLinePh* and *MultiLinePh* as subclasses. The skeleton of operations common to the subclasses can be lifted to the superclass, with the detail being left to the subclasses. This illustrates the use of a 'deferred' class (e.g. as in Eiffel) in designing for reusability.

6 Conclusion

There are many systems where the application of polymorphism only to suitable inheritance hierarchies is unnecessarily restrictive. In this paper we considered examples where a polymorphic object reference could be declared whose value referenced an object in one of several classes not related within an inheritance hierarchy. Furthermore, these examples illustrated that even when object references are declared polymorphically, it is often desirable to specify operations that qualify, and hence further restrict, the type of the object in some way.

Our approach to the issues raised by these examples was to define a class-union construct. This construction was then applied, and its implications explored, in several different contexts. Rules for the extension of specifications using class union were formalized which generalized the idea of signature compatibility as usually understood in polymorphic inheritance hierarchies.

The traditional view of object-oriented polymorphism can be seen as a restricted form of class union. As such restrictions are often desirable in practice, the class-union construct is seen as complementary to the marriage of polymorphism with inheritance. Because the class union construct is more widely applicable, however, it enables polymorphism to be applied in situations where the building of an inheritance structure would be cumbersome, or in the case when only part of an inheritance hierarchy satisfies the strict requirements for polymorphism.

This paper shows that the introduction of a class-union construct enhances the flexibility and extendibility of the specification of object-oriented systems. The implications of introducing similar constructs into object-oriented implementation languages, however, need to be studied further.

Acknowledgements

We would like to thank Bernhard Baier, Susan Bray, Alena Griffiths, Wendy Johnston, Andreas Prinz, Gordon Rose, and Margaret Turner for many helpful discussions on the issues raised in this paper.

Thanks also to the anonymous referees for many useful comments and suggestions.

References

[CHC89] W. Cook, W. Hill, and P. Canning. Inheritance is not subtyping. Technical Report STL-89-17, Hewlett-Packard Labs, 1989.

[CW85] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv.*, 17(4):471–522, December 1985.

[DKRS91] R. Duke, P. King, G. Rose, and G. Smith. The Object-Z specification language. In T. Korson, V. Vaishnavi, and B. Meyer, editors, *Technology of Object-Oriented Languages and Systems: TOOLS 5*, pages 465–483. Prentice-Hall, 1991.

[DR93] R. Duke and G. Rose. Modelling object identity. In *Proc. 16th Australian Comput. Sci. Conf. (ACSC-16)*, pages 93–100, February 1993.

[Lee92] A. Lee. *Formal Specification and Analysis of Intelligent Network Services and their Interactions.* PhD thesis, University of Queensland, 1992. (submitted).

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

[Mey92] B. Meyer. *Eiffel: The Language.* Object-Oriented Series. Prentice Hall, 1992.

[Spi89] J.M. Spivey. *The Z Notation: A Reference Manual.* International Series in Computer Science. Prentice-Hall, 1989.

[Str86] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, 1986.

[WZ88] P. Wegner and S. Zdonik. Inheritance as an incremental modification mechanism or what like is and isn't like. In S. Gjessing and K. Nygaard, editors, *Proc. European Conf. on Object-Oriented Programming (ECOOP'88)*, volume 322 of *Lect. Notes in Comput. Sci.*, pages 55–77. Springer-Verlag, 1988.