Bunches for Object-Oriented, Concurrent, and Real-Time Specification

Richard F. Paige[1] and Eric C.R. Hehner[2]

[1] Dept. of Computer Science, York University, Canada
paige@cs.yorku.ca
[2] Dept. of Computer Science, University of Toronto, Canada
hehner@cs.utoronto.ca

Abstract. We show how a collection of object-oriented concepts can be directly expressed in predicative programming [6]. We demonstrate how these features can be used in cooperation with the existing real-time and concurrency features of predicative programming in several examples, thus providing a simple integration of object-orientation, real-time, and concurrency.

1 Introduction

Formal methods—like Object-Z [3], VDM++ [7], and others—have been developed for rigorously specifying and proving properties about object-oriented (OO) systems. Similarly, methods have been developed for specifying and reasoning about real-time and concurrent systems, e.g., CSP, CCS, and the various real-time refinement calculi. There has been much recent interest in integrating these different paradigms. Work on combining CSP and Object-Z [13], Timed CSP and Object-Z (TCOZ) [8], VDM++ (which integrates VDM with concepts from Ada and process algebras), has aimed at producing notations that combine OO, concurrent, and real-time features.

The thesis of this paper is that integrating notations is not necessary to be able to write specifications that combine OO, real-time, and concurrency. We justify this claim by showing how predicative programming [6] and its type system can be used, without modification, for specifying and reasoning about simple OO systems. This would not be a particularly novel contribution by itself. However, predicative programming currently provides a wealth of support for real-time, concurrency, and communication. By showing how the method can also be used for OO, we can immediately begin to use OO with real-time, concurrency, and communication techniques.

Our aim in this paper is to introduce predicative programming and to show how it can be used without modification to specify and reason about a collection of OO concepts in cooperation with its existing real-time and concurrent features. We do not intend to extend or generalize existing OO theories, e.g., those from [1]. Instead, we show how to use predicative notation to specify a core collection of object-oriented techniques, like classes, inheritance, redefinition, and dynamic binding, and show how these features can be used with concurrency and real-time.

J. Wing, J. Woodcock, J. Davies (Eds.): FM'99, Vol. I, LNCS 1708, pp. 530–550, 1999.

1.1 The Paper

We commence with an overview of predicative programming [6]. We summarize the predicative type system, which is based on *bunches* [5], and which will be used to specify OO concepts. We then show how to specify classes and class interfaces, single and multiple inheritance, and explain how to deal with redefinition of feature semantics under inheritance. Section 4 contains examples that demonstrate the techniques, including examples that integrate OO and real-time, as well as OO and concurrency. Finally, in Section 5, we discuss limitations, and suggest directions for future work.

2 Predicative Programming

Predicative programming [6] is a program design calculus in which programs are specifications. In this approach, programs and specifications are predicates on pre- and post-state (as in Z, final values of variables are annotated with a prime; initial values of variables are undecorated). The weakest predicate specification is $\top$ ("true"), and the strongest specification is $\bot$ ("false"). Refinement is just boolean implication.

Definition 1. *A specification P on prestate σ and poststate σ' is refined by a specification Q if $\forall\, \sigma, \sigma' \cdot (P \Leftarrow Q)$.*

The refinement relation enjoys various properties that allow specifications to be refined by parts, steps, and cases. As well, specifications can be combined using the familiar operators of boolean theory, along with all the usual program combinators, as well as combinators for parallelism and communication through channels.

Predicative programming can be used to specify objects and classes. To do so, we need to introduce the predicative notation for types, namely bunches.

2.1 Bunches and Types

Bunches were introduced in [5], and are used in [6] as a type system. They are applied in [11] in formalizing selected static diagrams of UML. A bunch is a collection of values, and can be written as in this example: $2, 3, 5$. A bunch consisting of a single element is identical to the element. Some bunches are worth naming, such as *null* (the empty bunch), *nat* (the natural numbers), *int* (the integers), *real* (the bunch of reals), *char* (the bunch of characters) and so on. More interesting bunches can be written with the aid of the solution quantifier $\S$, pronounced "those", as in the example $\S i : int \cdot i^2 = 4$. The colon, :, is the subbunch operator; in general, $A : B$ is a boolean expression saying that A is a subbunch of B. For example,

$$2 : nat \qquad\qquad nat : int$$

We use the asymmetric notation $m, ..n$ for $\S i : int \cdot m \leq i < n$.

Bunches can be used as the contents of sets, as in

$$\{2, 3, 5\} \qquad\qquad \{\S i : int \cdot i^2 = 4\}$$

though we might choose not to write § in the latter example. Bunches can also be used as a type system, as in the declaration **var** $x : nat$ (perhaps with restrictions for easy implementation). Any bunch, including the empty bunch *null*, can be used as a type. For example, the declaration **var** $x : 1$ says that x can take on one value, 1.

Bunches can also be used in arithmetic expressions, where the arithmetic operators distribute over bunch union (comma):

$$nat = 0, nat + 1$$

We write functions in a standard way, as in the example $\lambda\, n : nat \cdot n + 1$. Function application is by juxtaposing the function name and its arguments, e.g., $f\ x$. The domain of a function is obtained by the Δ operator. If the function body does not use its variable, we may write just the domain and body with an arrow between. For example, $2 \rightarrow 3$ is a function that maps 2 to 3, which we could have written $\lambda\, n : 2 \cdot 3$ with n unused.

When the domain of a function is an initial segment of the natural numbers, we sometimes use a list notation, as in $[3;\ 5;\ 2;\ 5]$. The empty list is $[nil]$ (*nil* without square parentheses is the empty string). We also use the asymmetric notation $[m; ..n]$ for a list of integers starting with m and ending before n. List length is $\#$, and catenation is $^{+}$ (raised plus). A list of characters, such as "*abc*" can be written within quotes.

All functions we use in this paper apply to elements, and thus application of a function f distributes over bunch union, i.e.,

$$f\ null = null \qquad f\ (A, B) = f\ A,\ f\ B$$

A union of functions applied to an argument gives the union of the results, i.e., $(f, g)\ x = fx, gx$. A function f is included in a function g according to the *function inclusion law*.

$$(f : g) = ((\Delta g : \Delta f) \wedge (\forall\, x : \Delta g \cdot fx : gx))$$

Thus we can prove $(f : A \rightarrow B) = ((A : \Delta f) \wedge (\forall\, a : A \cdot fa : B))$. Using inclusion both ways round, we find function equality is as usual.

$$(f = g) = ((\Delta f = \Delta g) \wedge (\forall\, x : \Delta f \cdot fx = gx))$$

list T consists of all lists with items of type T. By defining *list* as $list = \lambda\, T : \Delta list \cdot 0, ..\#(list\ T) \rightarrow T$, *list T* can be used as a type.

The selective union $f \mid g$ of functions f and g is a function that behaves like f when applied to an argument in the domain of f, and otherwise behaves like g. It is similar to Z's function extension.

$$\Delta(f \mid g) = \Delta f, \Delta g$$
$$(f \mid g)x = \textbf{if}\ x : \Delta f\ \textbf{then}\ f\ x\ \textbf{else}\ g\ x$$

One of the uses of selective union is to write a selective list update. For example, if $L = [2;\ 5;\ 3;\ 4]$ then $2 \rightarrow 6 \mid L = [2;\ 5;\ 6;\ 4]$. Another use is to create a record structure. Define *PERSON* as follows.

$$PERSON = \text{“}name\text{”} \rightarrow list\ char \mid \text{“}age\text{”} \rightarrow nat$$

Declare variable p of type *PERSON* and assign p as follows.

$$p := \text{“}name\text{”} \rightarrow \text{“}Smith\text{”} \mid \text{“}age\text{”} \rightarrow 33$$

We can access the name field of p by dereferencing: $p\text{“}name\text{”}$.

2.2 Functional Refinement

A *refinement* relation can also be applied to functions. A function P is refined by a function S if and only if all results that are satisfactory according to S are also satisfactory according to P. Formally, this is just bunch inclusion, $S : P$. When writing refinements, we prefer to write the problem, P, on the left, and the solution, S, on the right. Thus, we write $P :\!\cdot\ S$ (informally read as “P is refined by S”), which means $S : P$.

2.3 Real-Time and Concurrency

Predicative programming is well-suited to specifying and reasoning about real-time, concurrent, and communicating systems. To talk about time, a time variable t is used; the theory need not be changed at all. The interpretation of t as time is justified by how it is used. t is used as the initial time (where execution starts), and t' for final time (where execution ends). To allow for nontermination, the domain of time is a number system extended with an infinite number ∞. The number system can be naturals, reals, et cetera. The following example says that the final value of variable h should be the index of the first occurrence of x in list L, and that any program satisfying the specification must provide an execution time that is linear in the length of L.

$$(\neg\, x : L(0,..h')) \wedge (Lh' = x \vee h' = \#L)) \wedge t' \leq t + \#L$$

Predicative programming includes notations for concurrent specification and for communication. We will not use the communication notations explicitly herein, but we will use concurrency; we direct the reader to [6] for details on communication.

The independent composition operator $\|$ applied to specifications P and Q is defined so that $P \parallel Q$ (pronounced “P parallel Q”) is satisfied by a machine that behaves according to P and at the same time, in parallel, according to Q. The formal meaning of $\|$ is as follows. Let the variables used by P and Q be denoted by σ (σ may be any number of variables, but it does not include t).

$$\begin{aligned} P \parallel Q = \exists\, & \sigma_P, \sigma_Q, t_P, t_Q \cdot \\ & P[\sigma_P/\sigma', t_P/t'] \wedge Q[\sigma_Q, t_Q/\sigma', t'] \wedge \\ & (\sigma_P = \sigma \Rightarrow \sigma' = \sigma_Q) \wedge (\sigma_Q = \sigma \Rightarrow \sigma' = \sigma_P) \wedge t' = max\, t_P\ t_Q \end{aligned}$$

($P[a/b]$ means “substitute a for b in P”.) Informally, if P leaves a variable unchanged, then Q determines the final value, while if Q leaves a value unchanged, P determines its final value. If both processes change the value, then the final value is undetermined (unless the processes agree on the final value).

We define $\|_{i:0,..k}\, P(i)$ to be $P(0) \parallel \ldots \parallel P(k-1)$ for any specification P on i.

3 Using Bunches for Object-Oriented Concepts

We now outline how bunches and predicative notation can be used to specify a core collection of OO concepts, including classes, objects, features, inheritance, and redefinition of feature semantics. Our intent is not to present a new OO theory; rather, it is a step towards being able to use OO, real-time, and concurrency together.

3.1 Specifying Classes and Objects

Several different definitions of the notion of a class have been presented in the literature. The definition of a class that we use is adapted from [9].

Definition 2. *A* **class** *is an abstract data type equipped with a possibly partial behavioural specification.*

A class consists of a number of features, which are *attributes* (representing state) or *routines* (representing computations). Routines may be further subdivided into *functions* (which return a value) and *procedures* (which can change state). No routine is both function and procedure. A class specification has three parts:

- a *class interface*, which declares all the attributes and functions of the class and gives their signatures (our convention is that class interface names end in *Int*).
- a *class definition*, which defines all the functions (our convention is that class definitions will always be in upper case).
- zero or more procedure definitions.

A separation of a class into interface and definition is useful, because it lets us define inheritance in terms of each (the concepts coincide when the interface possesses no functions). Note that our notion of interface is more general than that in Java, since we allow attributes in an interface, and the definition of some, but not necessarily all, functions. In this last respect, our notion of interface is closer to the Eiffel concept of *deferred class* [9].

We illustrate these mechanisms with a simple example: a stack of integers. The stack has one attribute, *contents*, which is a list of integers. It also has three routines, *push*, *pop*, and *top*. The interface specification of the stack, *StackInt*, declares the attributes and functions, and gives their signatures.

$$StackInt = \text{“}contents\text{”} \rightarrow list\ int \mid \text{“}top\text{”} \rightarrow int$$

A specific behavior is required for the parameterless function *top*. The definition of *top* is given in terms of *contents*, and is specified in the class definition *STACK*. (In the definition, recall that $s\text{“}top\text{”}$ is the record dereference syntax.)

$$STACK = \S s : StackInt \cdot s\text{“}top\text{”} = s\text{“}contents\text{”}(\#s\text{“}contents\text{”} - 1) \tag{1}$$

STACK is the bunch of all elements of *StackInt* that satisfy the definition of *top*: *top* is the last element of the list *contents*. (We could, in fact, write a generic *STACK* class, by replacing the *int* type for elements by a generic parameter T.)

For procedures we use a different approach, which is described in Section 3.2. In the interim, we turn to objects, which are instances of classes.

Definition 3. *An* **object** *is a variable with a class definition for its type.*

To declare an object of class *STACK*, we can write **var** s : *STACK*, and can access the *contents* field of object s by dereferencing s, written s“*contents*”. A dereferenced field may be any function or attribute. To assign a value to field *contents*, we just carry out a record field assignment, written either as s“*contents*” $:=$ *value*, or (as a selective union), as $s :=$ “*contents*” $\rightarrow$ *value* $\mid s$. This approach does not support any notion of information hiding; visibility of features is enforced only by specifier discipline.

3.2 Specifying Procedures

The formalization of classes is sufficient for specifying attributes and functions of classes, but is insufficient for capturing procedures, i.e., routines that change the state of an invoking object.

Each procedure of a class is a predicative function that takes an instance of the class as argument, and returns a changed, new instance of the class. Suppose f is to be a procedure of class C. We define a (possibly nondeterministic) function $f : C \rightarrow C$. To use f applied to an object c of class C, we write $c.f$ which is sugar for the assignment $c := f(c)$. The syntax $c.f$ allows specifiers to use procedures in a syntax similar to what is found in languages like C++ or Java. This function does not have side effects; it maintains the command/query separation suggested in [9].

Returning to the stack example, the procedure *pop* would be specified as

$$pop = \lambda\, s : STACK \cdot \text{``}contents\text{''} \rightarrow s\text{``}contents\text{''}[0;\ ..\#s\text{``}contents\text{''} - 1] \mid s$$

The method to push integer x to a *STACK* s is

$$push = \lambda\, s : STACK \cdot \lambda\, x : int \cdot \text{``}contents\text{''} \rightarrow s\text{``}contents\text{''}^{+}[x] \mid s$$

push can be used by writing $s.push(x)$, which is sugar for $s := push\ s\ x$. After a *push* or a *pop* has been applied to a stack s, the value of function s“*top*” will have changed. The definition of s“*top*” will not change, only its value.

3.3 Implementation

The preceding formalization of classes and objects is straightforward to structurally transform into an object-oriented programming language, e.g., Eiffel. A class definition T can be transformed into an Eiffel class `T`. Attributes are transformed into objects that are features of the class; for example, array *contents* of class *STACK* could be mapped to an instance of class `ARRAY` in Eiffel. Function definitions are transformed into bodies of functions in Eiffel; for example, the function definition of *top*, given in equation (1), can be easily transliterated into the following Eiffel function of class `STACK`.

```
top : INTEGER is do
  result := current.contents.item(contents.upper-1)
end
```

References to the bound variable s in (1) are replaced with references to the current object, `current`, in the Eiffel program. In general, a simple transliteration of predicative specification to Eiffel program will not be possible, thus refinement may have to take place beforehand.

Functions on objects in predicative notation can be transliterated into procedures of a class; explicit reference in the function to the object that is passed as an argument can be replaced by explicit reference to the current object. For example, *push* could be transliterated into the following Eiffel procedure (`append` is a feature of class `ARRAY`).

```
push( x:INTEGER ) is do
  current.contents.append(x)
end
```

3.4 Single and Multiple Inheritance

We now give a brief overview of inheritance in predicative programming. There are many different definitions and types of inheritance, e.g., see [1, 9]. The definition we use in this paper is one of *subtyping:* if a (child) class B inherits from a (parent) class C, then B can be used everywhere C can be used. We take this approach predominantly because we want to ensure behavioral compatibility between classes related by inheritance.

It is straightforward to determine if a class definition B is derived from class definition C. Since each class is just a type, we can apply bunch inclusion notation directly.

Rule 1 [Inheritance Relation] *Class B inherits from class C if* $B : C$.

This rule is valid if there are functions in the class definitions; we just apply function inclusion. When applying function inclusion, we must take care with function domains and ranges: functions are anti-monotonic in their domains, and monotonic in their ranges (see Section 2.1: function inclusion).

We also need to show how to build one class from another using inheritance. Single class inheritance is expressed in predicative notation by merging the definition or interface of the parent class with any new features that the child class will provide; this produces a definition or interface for the child class.

Definition 4. *Let C be a class definition or interface. If class B singly inherits C, then*

$$B = \text{``}b_1\text{''} \rightarrow T_1 \mid \ldots \mid \text{``}b_i\text{''} \rightarrow T_i \mid \ldots \mid \text{``}b_k\text{''} \rightarrow T_k \mid C$$

where the b_j *are attribute names and* T_1 *through* T_k *are bunches.*

By definition, $B : C$, because every value satisfactory to B is also satisfactory to C. In other words, class C includes all its extensions. This last fact is an artifact of the axiomatic definition of bunches in [6].

The names of attributes and functions of C and $b_1, .., b_k$ can coincide. If b_i is also the name of an attribute of C, then the attribute in C will be replaced by new attribute b_i in B. In order to maintain the subbunch relation of Rule 1, constraints must be placed on the types of the replacements. If a b_i overrides an attribute in C, then the type of the new attribute must be a subbunch of the original. This is the *contravariant* rule [9]. A discussion of the limitations and advantages of contravariance is in [9].

An implication of using selective union to specify inheritance is that in class hierarchies, the order in which features appear in class definitions or interfaces *matters.* Consider B, above: if C had appeared before all the new features b_i, then the features in C could override the new features – which is probably not what the specifier intended. To get around this complication, we follow the convention that, when using single inheritance, the parent class will always appear last in the child class interface or definition. Most OO programming languages enforce this by syntactic means. (We discuss the effect ordering of parent classes will have on multiple inheritance shortly).

Procedures of a parent class are inherited by a child class in the following sense. If there is a procedure $f : C \rightarrow C$, and class B inherits from C, then f can be applied to objects of class B, and type correctness is guaranteed on the use of f, because $B : C$. Therefore, f can be specialized for the methods of class B. New procedures can also be added to child classes. However, arbitrary procedure addition is not possible, because new procedures may falsify constraints specified in a parent class. Thus a new non-vacuous procedure h (i.e., a procedure that does not map everything to the empty bunch) of child class C that inherits from parent B must guarantee, for all $c : C$, that $h(c) : B$.

3.4.1 Overriding and Redefinition We have defined inheritance in terms of selective union, which allows us to override features of a parent class in a child class. In particular, it lets us give different definitions to functions in child classes than are present in parent classes; this allows us to specify a kind of *redefinition*. In a class definition, functions can always be redefined (as is the case with Java and Eiffel, but not C++).

Let C be a class definition with function $f : T$, and possibly some more attributes. Let $BInt$ inherit from C. By construction, $BInt : C$. Redefine function f in the class definition B as follows.

$$B = \S b : BInt \cdot (b\text{“}f\text{”} = body)$$

where *body* is a subbunch of T. Function f in B can therefore have a definition *body* different from that given to f in the definition of C. There are constraints on the redefinition *body*: a definition for f is inherited from C, say P. In the class definition for B, function f is being further constrained. Thus, the new constraint that $b\text{“}f\text{”} = body$ is effectively being conjoined with the original constraint P from class C. Thus, whatever new definition of f is provided must not contradict the original definition. That is, the specification

$$b\text{“}f\text{”} = P \wedge b\text{“}f\text{”} = body$$

must be satisfiable; this can be ensured by making *body* a refinement of the original definition P. This is akin to the correctness constraints on redefinition in Eiffel [9].

Procedure redefinition can be simulated by overloading procedure names; each instance of the procedure is defined on a different class in a hierarchy. The types of arguments to the procedure dictate the instance of the procedure that is to be used. New procedures must satisfy the constraints of the parent class.

Redefinition allows us to support a form of dynamic binding of functions, where the instance of a function that is used in a call is dependent on the dynamic type of an object, rather than its static type. Suppose we have a class A with feature f, and class B inherits from A and redefines f. Declare a list of instances of A, and an instance of B, and set element 3 of a to reference b.

$$\mathbf{var}\ a : list\ A \cdot \mathbf{var}\ b : B \cdot \text{“3”} \rightarrow b \mid a$$

The static type of $a(3)$ is A; its dynamic type is B. A call to $a(3)$“f” will use the B version of f.

3.4.2 Multiple Inheritance Multiple inheritance allows a child class to have more than one parent. It has been suggested as being useful in describing the complex class relationships that occur in domain modeling, as well as for building reusable object-oriented libraries. In predicative programming, we can easily adopt the simple yet powerful Eiffel approach to multiple inheritance. We summarize some details here.

Multiple inheritance, in predicative programming, takes two or more parent class definitions or interfaces, and produces a child class definition or interface (to simplify the discussion, we will refer only to ‘parent’ and ‘child’ classes, which we allow to mean class definitions or class interfaces). We first provide a preliminary definition of multiple inheritance, and then touch on its limitations.

Definition 5. *Let $C_1, .., C_k$ be classes. If B multiply inherits from $C_1, .., C_k$ then*

$$B = C_1 \mid C_2 \mid \ldots \mid C_k$$

B can also add new features and these new features can override attributes or functions in any of $C_1, .., C_k$. The restriction on overriding is that the types of the overriding features must be subtypes of the original features.

3.4.3 Name Clashes Suppose that the name of a feature is declared in two or more parents, and the parents are multiply inherited. Should there be one or two occurrences of the shared name in the derived class? Following [9], we can treat this problem syntactically, and use one of two mechanisms to resolve name clashes.

1. Order the base classes in the definition of the derived class, so as to override those features that we do not want in the derived class. In this way, we can select the reoccurring feature that we want to inherit in the derived class.
   Unlike multiple inheritance in some languages, in predicative programming the order in which base classes are multiply inherited *does* matter, and we can use this to our advantage to resolve name clashes.
2. Apply a renaming to all the commonly named features of the base classes in order to eliminate name clashes. This approach can be used in Eiffel [9]. An example is shown in Fig. 1: attribute a is common to both $C1$ and $C2$. If we need two occurrences of the attribute in the derived class D, we rename the occurrences of a in the definition of D.

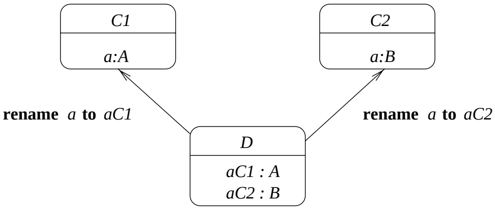

Fig.1. Renaming to avoid name clashes

Renaming in predicative notation is just substitution. The definition of class *D*, from Fig. 1 would be $D = C1[aC1/a] \mid C2[aC2/a]$, where $aC1$ and $aC2$ are fresh names of features. *D* can add new attributes and functions as necessary. We place one restriction on the names of new features like $aC1$: they cannot take on any of the names that are being changed.

If we rename features to avoid clashes in a child, the child is no longer (provably) a subtype of its parents. The proof rule for inheritance involving multiple parents and renaming is therefore slightly more complex.

Rule 2 [Multiple Inheritance Relation] *Let D inherit from both classes* $C1$ *and* $C2$*, and suppose name a is shared between* $C1$ *and* $C2$*. D is derived from* $C1$ *if there exists a substitution* $[a/aC1]$ *such that* $D[a/aC1] : C1$ *(and similarly for* $C2$*).*

Feature renaming must also be applied to the procedures of $C1$ and $C2$ that are inherited by *D*. If a method $f : C1 \to C1$ uses the attribute $a : A$ and $a : B$ is an attribute of class $C2$, then class *D* must have a new procedure, say $Df : D \to D$, with definition

$$Df = f[aC1/a]$$

Multiple inheritance can be expressed and used in predicative notation, but it is not always convenient to use the renaming facility to avoid its problems: the specifier must keep track of all the renamings. For large OO specifications, this will be impractical. Automated support for keeping track of renamings, e.g., as provided by a compiler, is essential for this solution to be feasible.

3.4.4 Repeated Inheritance If a class is a descendent of another through two or more paths, then repeated inheritance has occurred. Under repeated inheritance in bunch notation, a function or attribute from a common ancestor will yield a single method or attribute if it is inherited under a single name (this matches the notion of *virtual base class* in C++). If a renaming is applied to one or more features, a derived class can have two or more instances of a feature; [9] gives examples of when this is useful. The solution that we applied for resolving name clashes can also be used in resolving repeated inheritance (as is the case with Eiffel).

4 Examples

We present several examples of specifying OO systems, as well as combining use of OO and real-time (via a specification of the gas burner) and OO and concurrency (in a specification of a solution to the dining philosopher's problem).

4.1 Sequences and Queues

Our first example simply aims at illustrating the main concepts of the previous sections. We define a *SEQUENCE* class, and derive a *QUEUE* class from it. A sequence consists of the following features: a list *contents* of data elements; an *add* procedure, which puts an element x at position i of the sequence; a *delete* procedure, which removes the element at position i of the sequence; a *get* function, which returns the element at position i, or $-\infty$ if there is no element at i; and, an *empty* function. We first provide a class interface, *SeqInt*, where the sequence is to contain integers. *SeqInt* declares the attributes plus the signatures of *index* and *empty*.

$$\begin{aligned} SeqInt = \ & \text{“}contents\text{”} \to list\ int \\ & \mid \text{“}get\text{”} \to (nat \to int) \mid \text{“}empty\text{”} \to bool \end{aligned}$$

This interface has two functions, *get* and *empty*, which we now define.

$$\begin{aligned} & SEQUENCE = \S s : SeqInt \cdot \\ & \quad s\text{“}empty\text{”} = (\#s\text{“}contents\text{”} = 0) \wedge \\ & \quad s\text{“}get\text{”} = (\lambda\, i : nat \cdot \mathbf{if}\ i < \#s\text{“}contents\text{”}\ \mathbf{then}\ s\text{“}contents\text{”}(i)\ \mathbf{else}\ -\infty) \end{aligned}$$

We next specify the method *add*. If an addition at index i occurs where an entry exists, the entry at index i is overwritten with x; otherwise, catenation occurs.

$$\begin{aligned} & add = \lambda\, s : SEQUENCE \cdot \lambda\, i : nat \cdot \lambda\, x : int \cdot \\ & \qquad \mathbf{if}\ 0 \le i < \#s\text{“}contents\text{”}\ \mathbf{then}\ \text{“}contents\text{”} \to (i \to x \mid s\text{“}contents\text{”}) \mid s \\ & \qquad \mathbf{else}\ \text{“}contents\text{”} \to (s\text{“}contents\text{”}^{+}[x]) \mid s \end{aligned}$$

The *delete* method is defined as follows: to remove an entry that exists, all following entries are shifted left by one; otherwise, the sequence is returned unchanged.

$$\begin{aligned} & delete = \lambda\, s : SEQUENCE \cdot \lambda\, i : nat \cdot \\ & \qquad \mathbf{if}\ (0 \le i < \#s\text{“}contents\text{”})\ \mathbf{then} \\ & \qquad\ \ \text{“}contents\text{”} \to (s\text{“}contents\text{”}[0;\, ..i]^{+} s\text{“}contents\text{”}[i+1;\, ..\#s\text{“}contents\text{”}]) \mid s \\ & \qquad \mathbf{else}\ s \end{aligned}$$

The **then** branch of the *delete* method can be refined using standard predicative techniques. The ability to use standard refinement in developing programs is one benefit of using predicative programming in specifying object-oriented systems. To refine the **then** branch, we introduce a new recursive function, *shift*, which takes three arguments:

a sequence s, a pivot element i (everything to the right of i is shifted left one index), and a counter j. It is recursively defined as follows.

$$\begin{array}{l} \mathit{shift} = \lambda\, s : \mathit{SEQUENCE} \cdot \lambda\, i, j : \mathit{nat} \cdot \\ \qquad \textbf{if } j \geq \#s\text{“}\mathit{contents}\text{”} - 1 \textbf{ then } [\mathit{nil}] \\ \qquad \textbf{else if } j = i \textbf{ then } [s\text{“}\mathit{contents}\text{”}(i+1)]^{+}\mathit{shift}\ s\ i\ (j+2) \\ \qquad\quad \textbf{else } [s\text{“}\mathit{contents}\text{”}(j)]^{+}\mathit{shift}\ s\ i\ (j+1) \end{array}$$

Using the functional refinement laws from [6], it is straightforward to prove that

$$\mathit{delete}\ :\cdot\ \lambda\, s : \mathit{SEQUENCE} \cdot \lambda\, i : \mathit{nat} \cdot \textbf{if } 0 \leq i < \#s\text{“}\mathit{contents}\text{”} \textbf{ then } \mathit{shift}\ s\ i\ 0 \textbf{ else } s$$

The refined specification is implementable in any language that supports lists and recursion.

SEQUENCE can now be used in constructing a *QUEUE* class. *QUEUE* is like a *SEQUENCE*, except it is used in FIFO order. We derive a *QUEUE* class from *SEQUENCE*, adding a new state attribute called *cursor*, which is an index to the front of the *QUEUE*, and a new function called *head*, which gives the element at the head of the queue. First we specify the interface of the new class.

$$\mathit{QueueInt} = \text{“}\mathit{cursor}\text{”} \rightarrow 0 \mid \text{“}\mathit{head}\text{”} \rightarrow \mathit{int} \mid \mathit{SEQUENCE}$$

To define the function *head*, we give a class definition for *QUEUE*.

$$\mathit{QUEUE} = \S q : \mathit{QueueInt} \cdot q\text{“}\mathit{head}\text{”} = q\text{“}\mathit{get}\text{”}\ q\text{“}\mathit{cursor}\text{”}$$

head is the value stored in the *contents* attribute, in entry *cursor*. It follows immediately that *QUEUE* : *SEQUENCE* (since *SEQUENCE* includes all its extensions), and so *QUEUE* is derived from *SEQUENCE*.

We now specify the procedures of *QUEUE*; in doing so, we specialize procedures of *SEQUENCE*. There are two: *enqueue*, which adds an element to the rear of the *QUEUE*, and *dequeue*, which removes the front-most element of the *QUEUE*. To *enqueue* an element, we carry out an *add* in the last position in the sequence. *enqueue* changes only those parts of the queue q that are affected by *add*.

$$\mathit{enqueue} = \lambda\, q : \mathit{QUEUE} \cdot \lambda\, x : \mathit{int} \cdot \mathit{add}\ q\ (\#q\text{“}\mathit{contents}\text{”})\ x \mid q$$

add returns a *SEQUENCE*, which is part of a *QUEUE*. The selective union in the body of *enqueue* therefore overrides the *SEQUENCE* fields of q, while not changing the parts of q that are only defined in *QUEUE*.

To *dequeue* an element, we *delete* the element at position *cursor*.

$$\mathit{dequeue} = \lambda\, q : \mathit{QUEUE} \cdot \mathit{delete}\ q\ (q\text{“}\mathit{cursor}\text{”}) \mid q$$

4.2 Quadrilaterals

The quadrilaterals example is described in [15]; it is used to compare several different object-oriented methods based on Z. The example requires specifying different sorts of quadrilaterals which may be used in a drawing system.

The shapes of interest in the system are: a *quadrilateral*, the general four-sided figure; a *parallelogram*, a *quadrilateral* that has parallel opposite sides; a *rhombus*, a *parallelogram* with identical-length sides; a *rectangle*, which is a *parallelogram* with perpendicular sides; and, a *square*, which is both a *rectangle* and a *rhombus*.

We assume the existence of a class *VECTOR*. The usual vector operations, such as addition, are available. *VECTOR* also has a zero. The edges of a four-sided figure are defined first as a list, $\mathit{EdgesInt} = (0,..4) \to \mathit{VECTOR}$. Then, a class definition is provided, ensuring that the edges form a closed figure.

$$\mathit{EDGES} = \S e : \mathit{EdgesInt} \cdot (e0 + e1 + e2 + e3 = 0)$$

A quadrilateral class consists of edges and a position vector, the latter intended to be used in drawing the quadrilateral on the screen. The class definition of *QUAD* is

$$\mathit{QUAD} = \text{``}\mathit{edges}\text{''} \to \mathit{EDGES} \mid \text{``}\mathit{pos}\text{''} \to \mathit{VECTOR}$$

The class hierarchy in the quadrilateral system is depicted in Fig. 2, using BON notation. Each ellipse represents a class in the system, while directed edges indicate inheritance relationships. Inheritance will be defined predominantly on interfaces (though there are many other ways to use inheritance to specify this system).

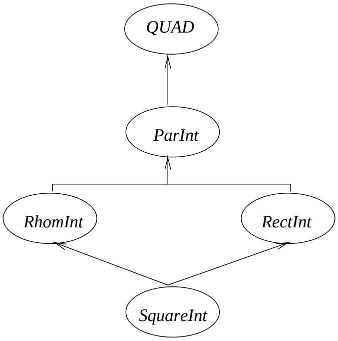

Fig.2. The class hierarchy

We construct the classes in the system by inheritance. In the process, we add a function *angle* to each class, where *angle* is the angle between edge 0 and 1. The hierarchy is described by first specifying class interfaces. Then, class definitions are provided, which give further details on constraints specific to each class.

$$\mathit{ParInt} = \text{``}\mathit{angle}\text{''} \to \mathit{real} \mid \mathit{QUAD} \qquad \mathit{RectInt} = \mathit{ParInt}$$

$$\mathit{RhomInt} = \mathit{ParInt} \qquad \mathit{SquareInt} = \mathit{RhomInt} \mid \mathit{RectInt}$$

Renaming of attributes from *ParInt* in *SquareInt* and *RhomInt* does not have to be done, since we need only one occurrence of each of *ParInt*'s attributes. In *SquareInt*, it is expressed that a square is both a rectangle and a rhombus. However, since both *RectInt* and *RhomInt* have the same class interface, their merge simplifies to *ParInt*.

The derivation hierarchy states that a parallelogram is a quadrilateral, a rhombus is a parallelogram, et cetera. But there are extra constraints associated with these special-case quadrilaterals—e.g., that a rectangle is a parallelogram with perpendicular sides. These constraints can be placed in the class definitions.

$$\begin{aligned} SQUARE &= \S s : SquareInt \cdot IsSquare(s\text{“}edges\text{”}) \wedge s\text{“}angle\text{”} = \pi/2 \\ RECTANGLE &= \S r : RectInt \cdot IsRect(r\text{“}edges\text{”}) \wedge r\text{“}angle\text{”} = \pi/2 \\ RHOMBUS &= \S r : RhomInt \cdot IsRhom(r\text{“}edges\text{”}) \wedge r\text{“}angle\text{”} = cos^{-1}(\ldots) \\ PARALLELOGRAM &= \S p : ParInt \cdot IsPar(p\text{“}edges\text{”}) \wedge p\text{“}angle\text{”} = cos^{-1}(\ldots) \end{aligned}$$

We omit the full definitions of the *angle* methods of *RHOMBUS* and *PARALLELOGRAM* (they are in [15]). *IsRect* is *true* if and only if the list of edges forms a rectangle ($\cdot$ in the body of *IsRect* is dot product.)

$$IsRect = \lambda\, e : EDGES \cdot (e0 \cdot e1 = 0 \wedge e0 + e2 = 0)$$

The predicates *IsSquare*, *IsPar*, and *IsRhom* are similar. We next define a procedure to translate a quadrilateral's position by a vector.

$$TranslateQuad = \lambda\, q : QUAD \cdot \lambda\, v : VECTOR \cdot \text{“}pos\text{”} \rightarrow q\text{“}pos\text{”} + v \mid q$$

To build a translation procedure on rhombi, for example, we specialize *TranslateQuad*.

$$TranslateRhom = \lambda\, r : RHOMBUS \cdot \lambda\, v : VECTOR \cdot TranslateQuad\ r\ v \mid r$$

The generic quadrilateral initialization method is as follows. It can be reused in the initializers of the other classes.

$$InitQuad = \lambda\, q : QUAD \cdot \lambda\, e : EDGES \cdot \lambda\, v : VECTOR \cdot \text{“}edges\text{”} \rightarrow e \mid \text{“}pos\text{”} \rightarrow v \mid q$$

4.3 A real-Time Example: Gas Burner

The gas burner problem has been treated by many researchers [14]. The problem is to specify the control of a gas burner. The inputs of the burner come from a sensor, a thermometer, and a thermostat. The inputs are:

- a real *temp*, indicating the actual temperature,
- a real *desired*, indicating the desired temperature,
- a boolean *flame*, indicating whether there is a flame.

The outputs of the burner are

- *gas*, which is set to *on* if the gas is on, or to *off* if the gas is off,
- *spark*, which maintains the gas and causes a spark for the purposes of ignition.

Heat is wanted when the actual temperature falls ϵ below the desired temperature, and is not wanted when the actual temperature rises ϵ above the desired temperature. ϵ is small enough to be unnoticeable, but large enough to prevent rapid oscillation.

To obtain heat, the spark should be applied to the gas for at least 1 second (to give it a chance to ignite and to allow the flame to become stable). A safety regulation states that the gas must not remain on and unlit for more than 3 seconds. Another regulation states that when the gas is shut off, it must not be turned on again for at least 20 seconds to allow any accumulated gas to clear. And finally, the gas burner must respond to its inputs within 1 second.

We formulate an object-oriented, real-time specification. Thus, we will need to talk about time. As discussed in Section 2.3, to talk about time, global time variables are introduced and are manipulated. In a pure OO specification, there are no global variables. In order to talk about real-time, we therefore formulate a simple class definition, *TIME*, which will be used to represent the passage of time over the lifetime of an object. *TIME* has one attribute, *t*, of type *real*.

$$TIME = \text{“}t\text{”} \to real$$

(*TIME* can be used to introduce a local clock. To introduce a system clock, *TIME* can be inherited by the *root* class in our system, from which computation will begin.) We also specify, implicitly, a function *addtime*, which will be used to describe a nondeterministic increase in time. *addtime* takes three real numbers r_1, r_2, r_3, as parameters, and satisfies the following property.

$$r_1 + r_2 \leq addtime\ r_1\ r_2\ r_3 \leq r_1 + r_3$$

The similar specification *takeone*, which takes one real number r_1 as a parameter, will be used to specify a nondeterministic increase in time of *at most* one second.

$$r_1 < takeone\ r_1 < r_1 + 1$$

The gas burner will be specified as a class. We begin by specifying its interface, giving the names of the attributes and functions local to the class.

$$\begin{aligned} BurnerInt = {} & \text{“}temp\text{”} \to real \mid \text{“}desired\text{”} \to real \mid \\ & \text{“}flame\text{”} \to bool \mid \text{“}spark\text{”} \to bool \mid \text{“}gas\text{”} \to status \mid \\ & \text{“}cold\text{”} \to bool \mid \text{“}hot\text{”} \to bool \mid TIME \end{aligned}$$

In its interface, the burner inherits from *TIME*. The bunch *status* is $status = on, off$. Now, we can define the functions of the class.

$$\begin{aligned} BURNER = {} & \S b : BurnerInt \cdot \\ & b\text{“}cold\text{”} = (b\text{“}temp\text{”} < b\text{“}desired\text{”} - \epsilon) \wedge \\ & b\text{“}hot\text{”} = (b\text{“}temp\text{”} \geq b\text{“}desired\text{”} + \epsilon \wedge b\text{“}flame\text{”}) \end{aligned}$$

This completes the specification of the burner's attributes and functions. Now we specify its procedures.

$$gas\_on = \lambda\, b : BURNER \cdot \text{“}gas\text{”} \to on \mid b$$
$$gas\_off = \lambda\, b : BURNER \cdot \text{“}gas\text{”} \to off \mid b$$

gas_on and *gas_off* are used to turn the gas on or off, on request. The next two procedures, *ignite* and *cutoff*, are responsible for igniting the spark of the burner (leaving it on for between 1 and 3 seconds) and for turning the spark off.

$$\begin{aligned} \mathit{ignite} &= \lambda\, b : \mathit{BURNER} \cdot \text{“}\mathit{spark}\text{”} \to \top \mid \text{“}t\text{”} \to (\mathit{addtime}\ b\text{“}t\text{”}\ 1\ 3) \mid b \\ \mathit{cutoff} &= \lambda\, b : \mathit{BURNER} \cdot \text{“}\mathit{spark}\text{”} \to \bot \mid b \end{aligned}$$

Finally, the procedure *wait* causes the burner to wait for 20 to 21 seconds.

$$\mathit{wait} = \lambda\, b : \mathit{BURNER} \cdot \text{“}t\text{”} \to (\mathit{addtime}\ b\text{“}t\text{”}\ 20\ 21) \mid b$$

The behaviour of the burner system can now be specified as two procedures, *too_hot* and *too_cold*. *too_cold* tests if the temperature is too cold; if it is, the gas is turned on, and the spark is ignited for at most three seconds, then it is cut off, and the test is repeated; if it is not too cold, one unit of time is taken, and then the test is repeated.

$$\begin{aligned} \mathit{too\_cold} = \lambda\, b : \mathit{BURNER} \cdot \\ &\mathbf{if}\ b\text{“}\mathit{cold}\text{”}\ \mathbf{then} \\ &\quad \mathit{too\_hot}\ \mathit{cutoff}\ \mathit{ignite}\ \mathit{gas\_on}\ b \\ &\mathbf{else}\ \mathit{too\_cold}\ (\text{“}t\text{”} \to \mathit{takeone}\ b\text{“}t\text{”} \mid b) \end{aligned}$$

too_hot is as follows. If the temperature is too hot, then the gas is shut off and the burner waits for 20 to 21 seconds; then the temperature is tested. If it is not too hot, then one unit of time is taken, and then the test is repeated.

$$\begin{aligned} \mathit{too\_hot} = \lambda\, b : \mathit{BURNER} \cdot \\ &\mathbf{if}\ b\text{“}\mathit{hot}\text{”}\ \mathbf{then} \\ &\quad \mathit{too\_cold}\ \mathit{wait}\ \mathit{gas\_off}\ b \\ &\mathbf{else}\ \mathit{too\_hot}\ (\text{“}t\text{”} \to \mathit{takeone}\ b\text{“}t\text{”} \mid b) \end{aligned}$$

The OO specification of the gas burner is then

$$\mathbf{var}\ b : \mathit{BURNER} \cdot b.\mathit{too\_hot} \quad \vee \quad b.\mathit{too\_cold}$$

4.4 A Concurrent Example: Dining Philosophers

As a final example, we formulate a simple concurrent and object-oriented specification of the dining philosophers synchronization problem. We assume that we have five philosophers who are either thinking, eating, or hungry. The philosophers are sitting at a circular table which is laid with only five chopsticks, placed between neighbouring philosophers. From time to time, philosophers get hungry and try to pick up the two nearest chopsticks. A philosopher can pick up one chopstick at a time, and cannot pick up a chopstick in the hand of a neighbour. When a hungry philosopher has both his chopsticks at the same time, he eats without releasing the chopsticks. When he is finished eating, he puts down both chopsticks and starts to think again.

We commence by assuming that we have a class called *SEMAPHORE*, used to represent the standard synchronization tool. This class has two procedures, *semwait* and *semsignal*. We also assume that we have used *SEMAPHORE* to specify a class called *CONDITION*, which specifies *condition constructs* for critical regions. This class has a queue, for waiting processes, associated with it as well as two procedures: *csignal*, which resumes exactly one suspended process, and *cwait*, which makes the invoking process wait until another invokes *csignal*. Formulations of both semaphores and condition constructs can be found in [12]. We will use these classes to specify the mutual exclusion required in the dining philosophers problem, via a *monitor.*

A monitor allows safe, effective sharing of objects among several concurrent processes. Monitors assure mutual exclusion; only one process at a time can be active within the monitor. A monitor is a class, consisting of two semaphores, *mutex* (used to orchestrate entrance to and exit from the monitor) and *next* (on which signaling processes may suspend themselves), and a counter *next_count*, which keeps track of the number of waiting processes. It also has two procedures, *enter* and *leave*, used by a process to enter and leave the monitor. Here is the class definition.

$$MONITOR = \text{“}mutex\text{”} \rightarrow SEMAPHORE \mid \text{“}next\text{”} \rightarrow SEMAPHORE \mid \text{“}next\_count\text{”} \rightarrow int$$

The *enter* procedure calls *semwait* on the *mutex* semaphore.

$$enter = \lambda\, m : MONITOR \cdot \text{“}mutex\text{”} \rightarrow (semwait\ m\text{“}mutex\text{”}) \mid m$$

Similarly, procedure *leave* exits the invoking process from the monitor. If the number of waiting processes is 0, *semsignal* is called on *mutex*, and the invoking process leaves the monitor; otherwise, *semsignal* is called on *next*.

$$\begin{aligned} leave = \lambda\, m : MONITOR \cdot \\ &\textbf{if}\ m\text{“}next\_count\text{”} > 0\ \textbf{then}\ \text{“}next\text{”} \rightarrow (semsignal\ m\text{“}next\text{”}) \mid m \\ &\textbf{else}\ \text{“}mutex\text{”} \rightarrow (semsignal\ m\text{“}mutex\text{”}) \mid m \end{aligned}$$

We next specify a philosopher as a class definition, *PHIL*. This class has two attributes, *state* (recording whether the philosopher is thinking, hungry, or eating), and *self*, which is a condition construct used for synchronization (it is used to delay a philosopher when he is hungry but unable to obtain the needed chopsticks).

$$PHIL = \text{“}state\text{”} \rightarrow Status \mid \text{“}self\text{”} \rightarrow CONDITION$$

(The bunch *Status* is *thinking*, *hungry*, *eating*.) The procedures for *PHIL* are used to change the state of an invoking object to one of *hungry* or *thinking*.

$$sethungry = \lambda\, p : PHIL \cdot \text{“}state\text{”} \rightarrow hungry \mid p$$
$$setthinking = \lambda\, p : PHIL \cdot \text{“}state\text{”} \rightarrow thinking \mid p$$

A philosopher uses the *eat* procedure to move to the eating state. A move to the eating state requires a call to the *csignal* procedure of class *CONDITION*, which resumes a suspended process. Thus, a call *p.eat* (where *p* is a philosopher) changes the philosopher's state to *eating,* and calls the *csignal* procedure of the philosopher's *self* attribute.

$$eat = \lambda\, p : PHIL \cdot \text{“}self\text{”} \rightarrow (csignal\ p\text{“}self\text{”}) \mid \text{“}state\text{”} \rightarrow eating \mid p$$

The dining philosophers system is specified as a class, *DINING*, which is a *MONITOR* extended with a list of five philosophers (we use the short-hand $[5 * \mathit{PHIL}]$ for a list of five philosophers).

$$\mathit{DINING} = \text{“}\mathit{phils}\text{”} \rightarrow [5 * \mathit{PHIL}] \mid \mathit{MONITOR}$$

The *DINING* system has several procedures. The first procedure we specify, *test*, takes a number k in the range $0 \leq k \leq 4$, moves philosopher k to *eating* status if possible, and signals that change in philosopher status to the system. A philosopher can move to *eating* only if he can obtain both the chopsticks to his sides and he is hungry. We view this procedure as *private*; it will only be used by other procedures in the dining philosopher system, and is not an entry procedure of the monitor.

```
$test = \lambda\, d : DINING \cdot \lambda\, k : 0,..5 \cdot$
    $\textbf{if}\ (d\text{“}phils\text{”}(k - 1\ \textbf{mod}\ 5)\text{“}state\text{”} \neq eating \wedge d\text{“}phils\text{”}(k)\text{“}state\text{”} = hungry \wedge$
        $d\text{“}phils\text{”}(k + 1\ \textbf{mod}\ 5)\text{“}state\text{”} \neq eating)$
    $\textbf{then}$
      $\text{“}phils\text{”} \rightarrow k \rightarrow (eat\ d\text{“}phils\text{”}(k)) \mid d$
    $\textbf{else}\ d$
```

(Informally, this specification reads “if I am hungry, and my neighbours aren’t eating, then I will eat, otherwise, I won’t change.”)

The procedure *putdown* is used when the philosopher is finished eating. The procedure puts philosopher *i* into a *thinking* state (i.e., the chopsticks are dropped). Then, the *test* procedure is applied to both of the neighbours of philosopher *i*, to see if they can start to eat.

```
$putdown = \lambda\, d : DINING \cdot \lambda\, i : 0,..5 \cdot$
    $test$
      $(test\ (\text{“}phils\text{”} \rightarrow i \rightarrow setthinking\ d\text{“}phils\text{”}(i) \mid d)\ \ (i - 1\ \textbf{mod}\ 5))$
    $(i + 1\ \textbf{mod}\ 5)$
```

The first argument of the inner-most *test* call sets philosopher *i* to thinking; *test* is then applied to the neighbours: philosopher $i - 1\ \textbf{mod}\ 5$, then philosopher $i + 1\ \textbf{mod}\ 5$. However, this specification of *putdown* ignores synchronization issues. In order for a call to *putdown* to synchronize with the actions of all other philosophers, *putdown* must be embedded in synchronization primitives; that is, the process must enter the monitor, then it may execute, and then it leaves the monitor. This is expressed in the procedure *entry_putdown*.

$$\mathit{entry\_putdown} = \lambda\, d : \mathit{DINING} \cdot \lambda\, i : 0,..5 \cdot \mathit{leave}\ (\mathit{putdown}\ (\mathit{enter}\ d \mid d)\ i) \mid d$$

The procedure *pickup* sets a philosopher to *hungry*, then attempts to pickup the chopsticks. If the attempt succeeds, he eats, but if he cannot pickup the chopsticks, he suspends himself by a call to the *wait* procedure of class *DINING*.

```
$pickup = \lambda\, d : DINING \cdot \lambda\, i : 0,..5 \cdot$
    $wait\ (test\ (\text{“}phils\text{”} \rightarrow i \rightarrow sethungry\ d\text{“}phils\text{”}(i) \mid d)\ i)\ i$
```

The first argument to *test* sets philosopher *i* to *hungry*, then tests him. Either this call succeeds and the philosopher eats, or it returns and he waits. *wait* is as follows. If philosopher *i* is eating, it does nothing. Otherwise (if the philosopher is thinking or hungry) it calls *cwait* on the philosopher, delaying him.

$$\begin{aligned}
\mathit{wait} = \lambda\, d : \mathit{DINING} \cdot \lambda\, i : 0,..5 \cdot & \\
& \textbf{if } (d\text{“}\mathit{phils}\text{”}(i)\text{“}\mathit{state}\text{”} = \mathit{eating}) \textbf{ then } d \\
& \textbf{else } \text{“}\mathit{phils}\text{”} \rightarrow i \rightarrow \text{“}\mathit{self}\text{”} \rightarrow (\mathit{cwait}\ d\text{“}\mathit{phils}\text{”}(i)\text{“}\mathit{self}\text{”}) \mid d
\end{aligned}$$

As was the case with *putdown*, the specification of *pickup* ignores synchronization. Thus, we must extend *pickup* with synchronization details, i.e., make it an entry procedure of the monitor. This is expressed in procedure *entry_pickup*.

$$\mathit{entry\_pickup} = \lambda\, d : \mathit{DINING} \cdot \lambda\, i : 0,..5 \cdot \mathit{leave}\ (\mathit{pickup}\ (\mathit{enter}\ d \mid d)\ i) \mid d$$

The initialization of the *DINING* class will be to set all philosophers to the *thinking* state, and to initialize the monitor (which amounts to initializing the semaphores).

$$\begin{aligned}
\mathit{init} = \lambda\, d : \mathit{DINING} \cdot\ & \text{“}\mathit{phils}\text{”} \rightarrow 0 \rightarrow \mathit{setthinking}\ d\text{“}\mathit{phils}\text{”}(0) \mid \\
& \text{“}\mathit{phils}\text{”} \rightarrow 1 \rightarrow \mathit{setthinking}\ d\text{“}\mathit{phils}\text{”}(1) \mid \\
& \text{“}\mathit{phils}\text{”} \rightarrow 2 \rightarrow \mathit{setthinking}\ d\text{“}\mathit{phils}\text{”}(2) \mid \\
& \text{“}\mathit{phils}\text{”} \rightarrow 3 \rightarrow \mathit{setthinking}\ d\text{“}\mathit{phils}\text{”}(3) \mid \\
& \text{“}\mathit{phils}\text{”} \rightarrow 4 \rightarrow \mathit{setthinking}\ d\text{“}\mathit{phils}\text{”}(4) \mid \\
& \text{“}\mathit{mutex}\text{”} \rightarrow 1 \mid \text{“}\mathit{next}\text{”} \rightarrow 0 \mid d
\end{aligned}$$

The dining philosophers system can then be specified as follows. We first declare an object, *d*, of type *DINING*. The object must be initialized, and then it will enter an indefinite concurrent iteration.

$$\textbf{var } d : \mathit{DINING} \cdot d.\mathit{init}.\ \ \mathit{iterate}$$

where

$$\mathit{iterate} = (\|_{i:0,..5}\ d.\mathit{entry\_pickup}(i).\ \ \mathit{Eat}.\ \ d.\mathit{entry\_putdown}(i)).\ \ \mathit{iterate}$$

The procedure *Eat* performs the activity of eating the food; we leave it unspecified. This specification will not allow deadlock, nor will it allow two neighbours to eat simultaneously. However, it is possible for a philosopher to starve to death. We leave the amendment of this as an exercise for the reader.

5 Discussion and Conclusions

That the predicative programming notation can be used to directly specify many key object-oriented concepts is not surprising, since the notation is sufficient to model any form of computation. Without having to change the notation, we can express key object concepts and still make use of the standard predicative method and all its features, such as timing, concurrency, and refinement.

Part of the reason for the simplicity of specifying object-oriented concepts is due to the bunch notation for types. In the predicative notation, all types are based upon a bunch representation, including lists and records. Because of this, classes and functions can be developed from bunch notation, and therefore object instantiation can be given its usual interpretation as variable declaration. This differs from the approach in [4], where objects are specified in terms of their effect on a global system state. Furthermore, inheritance can be given an interpretation akin to that which is available in many programming languages. The interpretation, as selective union, is easy to implement in any programming language that has lists, arrays, or records (overriding of a field can be implemented as assignment to the field of a record instance).

The formalization of OO concepts is not without limitations. Visibility and export of features is left entirely up to the discipline of the specifier; there is no equivalent to C++'s `public` or `private` notation, nor Eiffel's **export** clause. Further, it might be useful to be able to include procedures within a class definition (though see Utting [16], who argues that non-encapsulation of procedures is useful), but it is not possible within the existing type system of predicative programming. Encapsulation of procedures is left informal, based on the signatures of the features. However, procedures can be specified, and are associated with objects and classes by type rules: procedures associated with a class are only (consistently) applicable to objects of that class or of a child class. Misusing procedures results in unsatisfiable specifications.

A key benefit of using predicative programming to specify and reason about object-oriented systems, is that all existing predicative theory applies immediately to such specifications. This implies that we can specify and reason about key object-oriented concepts, as well as the real-time, interactive, concurrent, and timing characteristics of systems, using one notation and method, as the examples in Section 4 showed. A heterogeneous notation, in the sense of [10, 13], does not have to be created in order to integrate the concepts of OO, real-time, and concurrency.

In the future, we intend to work on improving and extending the object-oriented theory, and will formulate examples that combine use of OO and predicative programming's communication features.

Acknowledgements. We thank the reviewers for their very detailed comments. We thank NSERC for support.

References

[1] M. Abadi and L. Cardelli. *A Theory of Objects,* Springer-Verlag, 1996.

[2] A. Bunkenburg and J. Morris. Formal Bunch Theory. Draft.

[3] R. Duke, G. Rose, and G. Smith. Object-Z: A Specification Language advocated for the description of standards. *Computer Standards and Interfaces* **17**(5), 1995.

[4] A. Hall. Specifying and Interpreting Class Hierarchies in Z. In *Proc. Eighth Z User Meeting,* Workshops in Computing Series, Springer-Verlag, 1994.

[5] E.C.R. Hehner. Bunch Theory: A Simple Set Theory for Computer Science. *Information Processing Letters* **12**(1), 1981.

[6] E.C.R. Hehner. *A Practical Theory of Programming*, Springer-Verlag, 1993.

[7] K. Lano. *Formal Object-Oriented Development*, Springer-Verlag, 1995.

[8] B. Mahony and J.S. Dong. Blending Object-Z and Timed CSP: an introduction to TCOZ. In *Proc. ICSE '98*, IEEE Press, 1998.

[9] B. Meyer. *Object-Oriented Software Construction,* Second Edition, Prentice-Hall, 1997.

[10] R.F. Paige. Heterogeneous Notations for Pure Formal Method Integration. *Formal Aspects of Computing* **10**(3):233-242, June 1999.

[11] R.F. Paige. Integrating a Program Design Calculus and UML. To appear in *The Computer Journal*, 1999.

[12] A. Silberschatz and P. Galvin. *Operating System Concepts* 5e, Addison-Wesley, 1997.

[13] G. Smith. A Semantic Integration of Object-Z and CSP. In *Proc. FME'97*, LNCS 1313, Springer-Verlag, 1997.

[14] E.V. Sorenson, A.P. Ravn, and H. Rischel. Control Program for a gas burner, Technical Report ID/DTH EVS2, Computer Science Department, Technical University of Denmark, Lyngby, Denmark, 1989.

[15] S. Stepney, R. Barden, and D. Cooper. *Object-Orientation in Z*, Springer-Verlag, 1992.

[16] M. Utting. *An Object-Oriented Refinement Calculus with Modular Reasoning.* PhD Dissertation, University of New South Wales, October 1992.