# Parameterized Specifications for Software Reuse

Jingwen Cheng
Department of Computer Science
Monash University, Clayton, Victoria, Australia, 3168
E-mail: jim@bruce.cs.monash.edu.au

May 19, 1992

**Abstract**

Software reusability is believed to be the key to improving software development productivity, while specification plays an important part in software reuse. From a modern object oriented view, the reusable software components can be grouped into three categories: Procedural components, abstract data types and concrete data objects. Both procedural components and abstract data types can be parameterized in many Object-Oriented languages. Specifications for reuse of these components are discussed in detail in this paper. The reusable software components and their specifications will form a reusable software library, and the information extracted from the specifications forms a knowledge base. Based on the knowledge base and the reusable software library, a new software development paradigm with software reusability can be realized.

## 1 Introduction

Many programming languages support polymorphism through type parameterization. That is, types can be parameters of a procedure or an abstract data type. In Ada [9] and Eiffel [6], this mechanism is called **genericity**; In Smalltalk [7], it is called **polymorphism**; And in CLU [5], it is called **type parameterization**. C++ [8] does not support this mechanism directly, but it supports function name and operator **overloading** by which we still can achieve polymorphism in the sense of doing different things with the same operation on different objects.

The concept of polymorphism or overloading is not new. Many built-in data types and procedures or functions in traditional languages embody the idea of polymorphism or overloading. For example, the *array* data type in many languages (including those not supporting type parameterization) is a parameterized data type. We can define variables of **array[int]**, **array[real]**, etc. The procedure **print**(...) (the name may be different in different languages) is a parameterized or overloaded procedure. We can use this procedure to print out different types of objects, such as print(5), print(3.6), print("hello"), etc. Almost all operators , such as "+", "−", in all languages are overloaded. When applied on different types of objects, they will perform different operations (generate different machine codes). What is unusual is that the languages supporting type parameterization allow users to design their own parameterized data types and procedures, while other languages do not.

One of the advantages of type parameterization is enhancing software reusability. A procedure or an abstract data type with type parameters will be reused more frequently than nonparameterized one.

In order for a software component to be reused by other users, the provider of the component must provide a specification of the component so that the user can understand what the software does and how to use it without knowing its implementation details. The reusable software components can be grouped into three categories: Procedural (procedure or function) components, abstract data types and concrete data objects. The first two kinds of components can be either parameterized or nonparameterized. We have discussed the specifications for nonparameterized reusable software components elsewhere[1]. In this paper, we will discuss the specifications for parameterized components. Section 2 discusses the parameterized procedural specifications; Section 3 describes the specifications for overloaded procedures or functions; Section 4 gives the parameterized abstract data type specifications; Section 5 explains the

specifications for concrete data objects created through parameterized abstract data types; And finally, Section 6 summarizes our discussions.

# 2 Specifications for Parameterized Procedures

A parameterized procedure usually has more constraints to be specified than a nonparameterized one because some operations on some types will not be meaningful. For example, a procedure performing *sorting* function requires the type of the object to be sorted is totally ordered. So, when a procedure is parameterized, any constraints on type parameters must be specified in the specification. Figure 1 gives the specification for parameterized procedure *maximum*.

```
maximum=proc[t:type](a,b:t)- > t
    preconditions: t has operation ''>="(t,t)- >bool
    behaviours: returns the maximum of a and b.
    keywords: max, maximum, bigger_number.
end maximum
```

Figure 1: A parameterized specification

For comparison, figure 2 gives a specification for nonparameterized procedure *IntMax*.

```
IntMax=proc(a,b:int)- > int
    behaviours: returns the maximum of a and b.
    keywords: max, maximum, bigger_number.
end IntMax
```

Figure 2: A nonparameterized specification

Now let us give some remarks on the parameterized specification:

- The parameterized specification differs in form from the nonparameterized one only in the header, which now has an extra part that defines the type parameter(s).
- In *preconditions* clause, any constraints on type parameter(s) are stated in addition to other constraints. If there are no constraints, the *preconditions* can be omitted as usual.
- The preconditions required on type parameters can be satisfied in two ways when the procedure is used:
  1. Only use the types which have the required built-in operation. For example, types of **int** and **real** all have the operation ">=".
  2. Define the required operation on user defined types or built-in types having no the required operation first, then use these types in the parameterized procedure. This, of course, requires that the language support operator overloading (such as Ada and C++).

# 3 Specifications for Overloaded Procedures

In the languages which support procedure or function name overloading, we can define many different procedures or functions with the same name as long as there is at least one different parameter with each other. Usually all the different procedures or functions do the same kind of thing except on different types of objects. If not, we should have used different names rather than overloaded the different things on the same name. So it is not necessary to write a specification for each procedure or function, instead, we can write one specification for all the procedures or functions. This is, of course, not always convenient. In the following, we will discuss in what cases this is convenient.

## 3.1 All procedures or functions have the same number, but different types of parameters

This is the most case of overloading. In this case, we can write a specification for all the procedures or functions easily. For example, there are three procedures overloaded on the same name *max*.

max=**proc**(a,b:**int**)– >**int**
max=**proc**(a,b:**real**)– >**real**
max=**proc**(a,b:**string**)– >**string**

The specification for all these three procedures is shown in figure 3.

```
max=proc[t:type](a,b:t)– > t
    preconditions: t may be int, real, string.
    behaviours: returns the maximum of a and b.
    keywords: max, maximum, bigger_number.
end max
```

Figure 3: A specification for overloaded procedures

What are the differences between this specification and the specification of parameterized procedure shown in figure 1?

First, the constraint on type parameter *t* in this specification is specified by enumerating all the allowed types, while in parameterized specification, it is specified by giving the required operations which the types should have. Therefore, only those types enumerated in the *precondition* clause can be used when calling the overloaded procedures. Whereas for parameterized procedures, any type, as long as it has the required operation, can be used.

Second, the constraint on type parameters in overloaded specification must be explicitly given in *precondition* clause, whreas it is optional in parameterized specification.

Third, an overloaded specification specifies several procedures overloaded on the same name, while a parameterized specification only specifies one procedure which can be instantiated with different types.

And last, the procedures specified by an overloaded specification can be directly called. For example,

x, y, z: **int**;
x:=5; y:=10;
z:=max(x, y);

will call procedure max=**proc**(a,b:**int**)– >**int**.

While a parameterized procedure cannot be called directly. It must be instantiated with concrete types first, then, the instances can be called. For example, we cannot use the following statement to call the parameterized procedure *maximum*:

z:=maximum(x,y);

Instead, we should use the following statements:

**procedure** IntMax:=maximum[**int**];
z:=IntMax(x,y);

In some languages, such as CLU, these two steps can be combined together, as below:

z:=maximum[**int**](x,y);

## 3.2 Overloaded procedures have different number of parameters

This case sometimes occurs, but not very often. Suppose we have the following procedures overloaded on the same name *sort*:

sort=**proc**(array[**int**])
sort=**proc**(array[**real**])
sort=**proc**(array[**string**])
sort=**proc**(head:list,key:**int**)

sort=**proc**(head:list,key:**string**)

In the last two procedure,

list=**record**
name:**string**
IDnumber:**int**
next:list
**end record**

The specification for these procedures is shown in figure 4.

```
sort=proc[t1,t2,t3:type](a:array[t1] | b:t2,c:t3)
    preconditions: t1 may be int, real, string;
                   t2=record
                       name:string)
                       IDnumber:int
                       next:t2
                       end record
                   t3 may be int, string
    side_effects: modifies a or b.
    behaviours: sorts the array or list in ascending order.  If t3=int, sorts the list by
        field IDnumber; If t3=string, sorts the list by field name.
end sort
```

Figure 4: Another overloaded specification

In this specification, there is something different from other specifications. The input parameter list is separated by a vertical bar. The separated parts represent different forms of the procedures being specified. From this specification, we know that there are two different forms of procedures being specified: One with one input parameter, the other with two input parameters.

Similarly, if the procedures have different number (0 or 1) or types of the return values, we can use the same notation to specify the correspondences between different input forms and return types. For example,

MinElement=**proc**(**array**[t1]|b:t2,c:t3)− >t1|t3

implies that the procedures being specified have the following two forms:

MinElement=**proc**(**array**[t1])− >t1

and

MinElement=**proc**(b:t2,c:t3)− >t3

Other cases of procedure overloading rarely occur, so we do not discuss further. If the overloaded procedures have many differences in parameters with each other, writing one specification for them may not be easier than writing a specification for every procedure.

# 4 Specifications for Parameterized Abstract Data Types

Many languages, such as Ada, CLU, Eiffel, etc. support parameterized abstract data type. An abstract data type is an abstraction of a class of data structures which have some common features. A parameterized abstract data type is an abstraction of a set of abstract data types with types as their parameters. When we apply a concrete type to a parameterized abstract data type, we get a specific abstract data type.

Just as in the case for parameterized procedures, any constraints on type parameters must be specified in the *preconditions* clause of the specification. However, In this case, the type constraints can be placed either on the abstract data type as a whole or on individual operations where the constraints are required.

Figure 5 gives a specification for parameterized *set*, in which the elements are of some arbitrary type. Because there are no duplicate elements in sets, an *equal* ("=") operation is needed in most operations on sets. So the element types are constrained to only those having equal operation, and this constraint is placed on the set as a whole.

In the specification, we use s(+) to represent the state of s after the operation terminates.

```
set=adt[t:type]- > create, insert, delete, member, size, choose, empty
 preconditions: t has an operation "="(t,t)- >bool
 overview: Sets are unbounded mathematical sets of elements of some type.
 operations:
      create=proc( )- >set[t]
          behaviours: returns a new empty set.
      end create
      insert=proc( s: set[t], e: t)
          side_effects: modifies s.
          behaviours: adds e to s. i.e.
              s(+)=s ∪ {e}
      end insert
      delete=proc( s: set[t], e:t)
          side_effects: modifies s.
          behaviours: removes e from s. i.e.
              s(+)=s-{e}
      end delete
      member=proc( s: set[t], e:t) - > bool
          behaviours: If e is an element of s, returns true; otherwise returns false. i.e.
              returns e ∈ s
      end member
      size=proc( s: set[t]) - > int
          behaviours: returns the number of elements in s. e.g.
              size(create())=0
      end size
      choose=proc( s: set[t]) - > t
          preconditions: s is not empty.
          behaviours: returns an arbitrary element of s.
      end choose
      empty=proc( s: set[t]) - > bool
          behaviours: If s is empty, returns true; otherwise returns false. e.g.
              empty(create())=true
      end empty
end set
```

Figure 5: A specification for parameterized set

Another example of specifications for parameterized abstract data types is shown in figure 6, which is the specification for parameterized queue. In this specification, only in procedure *IsIn* is the constraint on type placed, while all the other procedures have no constraint on type. So if a user does not intend to use the operation *IsIn*, he can use any type in this abstract data type. If he intend to use *IsIn* operation to check whether or not an element is already in the queue, then he can use only those types having the required operation "=" (equal). This gives the user more flexibility than placing the constraint on

the abstract data type as a whole.

```
queue=adt[t:type]- >create, append, remove, empty, length, IsIn
 overview: A queue is a first-in-first-out list of elements of type t. elements are appended to the end of the queue
     and removed from the front.
 operations:
      create=proc( )- >queue[t]
          behaviours: returns a new empty queue[t].
      end create
      append=proc( q: queue[t], e: t)
          side_effects: modifies q.
          behaviours: adds e at the end of queue.
      end append
      remove=proc( q: queue[t]) - > t
          preconditions: q is not empty.
          side_effects: modifies q.
          behaviours: removes the front element from the queue and returns it.
      end remove
      empty=proc( q: queue[t])- >bool
          behaviours: returns true if q is empty; otherwise returns false. e.g.
              empty(create())=true.
      end empty
      length=proc( q: queue[t]) - > int
          behaviours: returns the length (the number of elements) of the queue. e.g.
              length( creat())=0
      end length
      IsIn=proc( q: queue[t], e:t)- >bool
          preconditions: t has an operation "="(t,t)- >bool
          behaviours: returns true if e is in q otherwise returns false.
      end IsIn
end queue
```

Figure 6: Specification for parameterized queue

# 5 Specifications for Data Objects of Parameterized Abstract Data Types

In traditional programming environments, data objects created by a program at run time exist only during the execution of the program. It is sometimes necessary to keep them with their structures for a longer time so that other programs or the same program at other run times can reuse the objects directly instead of creating them from scratch every time. Persistent programming provides such a support for storage and retrieval of data objects with their entire structures. However, for these objects to be reused by other people, the programmer must provide their specifications in which the types or structures and the contents of the objects are specified. The specifications for data objects of ordinary types and nonparameterized abstract data types have been discussed elsewhere[1]. Here we discuss the specifications for data objects of parameterized abstract data types.

Because a parameterized abstract data type contains type parameters, while an object of the abstract data type must be of some specific type. For example, If an abstract data type is:

queue[t:type]
the objects of this type may be of
queue[**int**]
or
queue[**string**]
The specifications for such objects must specify the types of the parameters as well as the abstract data type.

Figure 7 gives a specification for an object of parameterized abstract data type *queue* (whose specification is shown in figure 6).

```
StudentsApplyingScholarships=data
    type: adt queue[t:type]
          t=int
    contents: A list of ID numbers of students who apply for scholarships.
end StudentsApplyingScholarships
```

Figure 7: A specification for a data object

To reuse this object, the user must retrieve the object and the abstract data type *queue* first, then get an instance of the abstract data type by applying type **int** to it, then use the operations provided by the abstract data type to access the object.

# 6 Summaries

We have discussed the specifications for parameterized procedures and abstract data types, overloaded procedures, and the objects of parameterized abstract data types for reuse of these kinds of software components. The specifications given in this paper are not complete. For example, for the purposes of search and retrieval, the *keywords, implementation, identifier* and other items should be specified. Here we just want to highlight the new features of the parameterized specifications. The complete specifications for each kind of software components can be found in [1]. The specifications and the knowledge extracted from them will form a knowledge base of reusable software components. Based on the knowledge base, an effective search and retrieval mechanism can be designed and an automatic selection of reusable software components from user's query can be realized.

# References

[1] Cheng, J. W. and Hurst, J. "Specifications in Software Development", in *Proceedings of the Seventh Annual University at Buffalo Graduate Conference on Computer Science*, 1992, pp. 63-72.

[2] Gorlen, K. E., Orlow, S. M. and Plexico, P. S. "Data Abstraction and Object-Oriented Programming in C++", John Wiley & Sons, 1990.

[3] Kirkerud, B. "Object-Oriented Programming with Simula", Addison-Wesley Publishing Company, 1989.

[4] Liskov, B. and Guttag, J. "Abstraction and Specification in Program Development", The MIT Press, McGraw-Hill Book Company, 1986.

[5] Liskov, B., Atkinson R., Bloom, T., Moss, E., Schaffert, J. C., Scheifler, R. and Snyder, A. "CLU Reference Manual", Springer-Verlag, 1981.

[6] Meyer, B. "Object-Oriented Software Construction", Prentice Hall, 1988.

[7] Pinson, L. J. and Wiener, R. S. "An Introduction to Object-Oriented Programming and Smalltalk", Addison-Wesley Publishing Company, 1988.

[8] Stroustrup, B. "The C++ Programming Language", Addison-Wesley Publishing Company, 1987.

[9] Wegner, P. "Programming with Ada: An Introduction by Means of Graduated Examples", Prientice Hall, Inc. , Englewood Cliffs, New Jersey, 1980.