Arab Gulf Journal of Scientific Research, 19 (3), pp 137-149 (2001)

ORIGINAL PAPER

M Almulla, M Al-Haddad and H Loeper

An Ada-based Preprocessor Language for Concurrent Object Oriented Programming

Abstract: In this paper, implementation issues of concurrent object-oriented programming using Ada 95 are addressed. Ada is not a pure object-oriented language; in order to make it so, a uniform template for structuring object classes is proposed. The template constitutes a basis for an Ada-based preprocessor language that handles concurrent object-oriented programming. The preprocessor accepts Ada-like object-oriented programs (object classes, subclasses, and main program) as input and produces Ada 95 concurrent object-oriented program units as output. The preprocessor language has the advantage of adding a new component to the class specification called the protocol, which specifies the order for requesting methods of an object. The preprocessor also touches on the extensibility of object classes issue. It supports defining class hierarchies by inheritance and aggregation. In addition, the preprocessor language supports the re-use of Ada packages, which are not necessarily written according to the object-oriented approach. The paper also investigates the definition of circular dependent object classes and proposes a solution for introducing a collection of classes.

Keywords: Preprocessor Language, Ada 95, Concurrency, Object-oriented programming

معالج لغوي بإعتماد لغة Ada للبرمجة المتزامنة والمكيفة وفقا الأشياء .

محمد الملا ، محمد الحداد و هانز لوير

المستخلص:تناقش ورقة البحث قضايا تطبيق البرمجة المتزامنة والمكيفة وفقا للأشياء والمصاغة بلغة . Ada 95 لا تعد لغة Ada 95 لغة مخصصة فقط للبرمجة المكيفة وفقا للأشياء . هذا وللحصول على وحدات برامج مكيفة وفقا للأشياء منظمة وبشكل جيد ، يقترح البحث على المبرمج الأمتثال إلى الهيكل الجديد المقترح لتركيبة نماذج الأشياء . كما تسمح نماذج الأشياء المصاغة ضمن الهيكل المقترح بإشتقاق نماذج أشياء فرعية مستنبطة من النماذج الأساسية وبموجب وحدات برامج ذات معطيات متعددة الأنواع . يشكل الهيكل المقترح قاعدةً أساسية لمعالج لغة مبدئي يعتمد على لغة Ada ، ويعالج برامج متزامنة مكيفة وفقا للأشياء . يدعم المعالج المبدئي بكفاءة تصميم برمجيات متزامنة مكيفة وفقا للأشياء . وذلك عن طريق إخفاء تفاصيل تنفيذ هذه البرامج عن المبرمج ، بالاضافة إلى تبسيط صياغة البرامج . يستقبل المعالج برامج مكيفة وفقا للأشياء مصاغة بلغة قريبة من Ada 95 كمعطيات ، وينتج بصورة أتوماتيكية برامج مصاغة بلغة Ada 95 كمخرجات . هذا كما يضيف معالج اللغة المبدئي عنصر جديد لوصف نموذج الأشياء يدعى بالبروتوكول الذي ينظم بدورة ترتيب استدعاء وحدات البرنامج . تحسن هذه الصفة الجديدة باللغة من أداء عملية وراثة الصفات كما يمكن تعريف تدرجات لنماذج الأشياء بواسطة عمليتي الوراثة والتجمع . كما ينادي معالج اللغة بإعادة استخدام رزم لغة Ada والتي لا يشترط أن تكون مصاغة بأسلوب البرمجة المكيفة وفقا للأشياء . وأخيراً ، تبحث الورقة في موضوع تعريف نماذج أشياء تعتمد على بعضها البعض ، بالاضافة إلى أنها تقترح حل لمشكلة تقديم مجموعة من النماذج .

كلمات مدخلية: الحاسوب، معالج لغوي، برمجة متزامنة، لغة Ada 95

Introduction

Ada 95 offers tools for defining object classes based on abstract data types. In object-oriented programming, the concept of abstract data types is augmented with the features of inheritance, polymorphism, and dynamic binding to derive new object classes from parent object classes. Using these tools, programming with Ada 95 does not automatically result in well-structured object-oriented programs, because Ada 95 is not a pure object-oriented programming language. It is the programmer's task to comply with certain rules in order to maintain well-structured object classes in the Ada 95 software design. In addition, class extensibility, which means deriving a new concurrent object class with new methods from a given superclass, is an important feature of concurrent object-oriented programming. However, Ada 95 does not integrate smoothly object-orientedness and concurrency. Neither the tasks nor protected objects of Ada 95 are extensible (Wellings, 2000). This drawback of Ada 95 was already realized in Loeper (1997) and Loeper (1998). These papers deal with a proposal for

M. Almulla, M. Al-Haddad, and H. Loeper
Department of Mathematics and Computer Science
Kuwait University
Tel:4811188 - Fax:4817201
e-mail:almulla@mcs.sci.kuniv.edu.kw

implementing concurrent object classes in a general well-structured manner, which in turn allows the definition of subclasses by extensions based on inheritance. In this work, new methods may be added to an object class by attaching a new task to it that implements the methods of the derived class. It should be noticed that a supplementary thread of control is added to the subclass, if the superclass is already defined as an active concurrent object class. Normally, active concurrent object classes are at the leaves of inheritance hierarchies when a concurrent system is designed. Therefore, in such cases the superclass is an abstract tagged type and no problem will arise when applying the suggested approach. Otherwise, in general, implementing concurrent object subclasses by extending the parent tagged record with a new task that implements the methods of the derived class may lead to synchronization problems.

Wellings (2000) is also concerned with the integration of concurrency and object-oriented programming in Ada 95. It studies mainly Ada's protected type mechanism and works out a proposal for making Ada 95's protected type mechanism extensible, which causes both syntactic and semantic variations to the Ada 95 language.

The intention of the authors in this work is to avoid revising the current Ada 95 standard. They introduce a preprocessor language as an additional layer to the software design process with the aim of overcoming the drawback of the explained extensibility issue, as well as to support the well-structured object-oriented software design. In this paper, the authors investigate mainly the design and implementation of a preprocessor language based on Ada 95 for concurrent object classes, which will be roughly described. The preprocessor accepts Ada-like object-oriented program units (classes, subclasses, and a main program), and produces Ada 95 concurrent object-oriented code, which is based on a proposed Ada 95 target code template of concurrent object classes.

The preprocessor language allows creating hierarchies of object classes by either inheritance or aggregation. Object-oriented programming languages support inheritance; yet, the preprocessor language enhances the inheritance mechanism by allowing a subclass to revoke unnecessary methods while allowing its child classes to inherit those methods from their grandparent class. Aggregation means to compose an object class by other object classes. In object-oriented design methodology, class aggregation, as an important design principle, is often neglected or only indirectly provided. The preprocessor language provides a direct support for aggregation by allowing a class to have object classes as part of its state. An object in Ada 95 target code is represented as a single task regardless if it is an instant of a class or a subclass in case of inheritance, while aggregate objects have their own tasks.

For the sake of well-structured object-oriented problem decomposition, it is desirable that each object class definition represents one compilable program unit of the preprocessor language. However, more than one object class must be defined together, if the object class definitions are recursively dependent. Therefore, the preprocessor language also provides "collection", which allows the programmer to write circular dependent classes in one compile-able program unit.

The remainder of the paper is organized as follows. Section 2 introduces the general template for concurrent object-oriented programs, which is the basis for this investigation in designing and implementing a preprocessor language, as well as the layout of the Ada 95 target code produced by the preprocessor, which is discussed in Section 3. The concurrent object-oriented language definition that was called before the preprocessor language is described in Section 4. The beginning of the section describes object classes. Two separate constructs define classes, subclasses, and collections: the specification, which includes the necessary information for using the specified item, and the body, which contains the internal implementation of the methods. The section also describes the subclass definition in the preprocessor language. Subclasses inherit the state attributes and methods from their superclass. Subclasses have specification and body like classes but only additional attributes are specified in the state of the subclass, while the superclass attributes are automatically inherited. Methods that have different behavior in the subclass other than that of the superclass are called overridden methods. Only new and overridden methods have declarations and bodies in the subclass. The preprocessor language also provides a facility to revoke unneeded inherited methods. The preprocessor language adds an important new component to the class specification called the protocol, which specifies the order in which methods of a class may be requested. Defining class hierarchies by inheritance and aggregation and adding a new method needs to specify the appropriate entry point in the order in which the methods may be requested. This task is achieved by the protocol. Therefore, all methods including the inherited ones must be specified in the protocol, unless they are intended to be revoked. In addition, the preprocessor language allows defining part-objects as part of the state of an object class specification. Part-objects are objects of a specific class or a subclass, which may themselves have part-objects as well. At the end of Section 4, the collection concept in the preprocessor language is introduced. The preprocessor language target code translation is discussed in Section 5. The section starts with a brief description of the preprocessor implementation and then focuses on the target code translation for object classes, subclasses, part-objects, and collections into Ada 95.

Section 6 discusses concurrent object-oriented programming using the preprocessor language for various examples. These examples demonstrate some of the features of the language and prove its expressiveness and solidity. The preprocessor was fully implemented using the GNU Ada 95 Compiler Version 3.10 (GNAT 1997) running on PC with Windows 97.

Concurrent Object Classes

The object class is the conceptual modeling tool for generating objects (Khattab, 1997). An object encapsulates a set of data (the state) and a set of methods that operate on these data (Krakowiak, 1990). The notion of object orientation is mainly based on the concepts of objects, object classes and class inheritance, combined with polymorphism and dynamic binding (Wegner, 1987). Composition (also referred to as aggregation) is a mechanism of forming an object class as a whole using other object classes as its parts. It reduces the complexity by treating many object classes as one (Odell, 1994).

An object has a state, which holds all properties of the object. Moreover, methods (procedures or functions) are associated with an object, which describe the meaningful operations on the object.

An object class is a set of all objects having the same structure and behavior (Booch, 1991). The general structure of an object class according to Loeper (1998) is represented in Figure 1, with the following remarks:

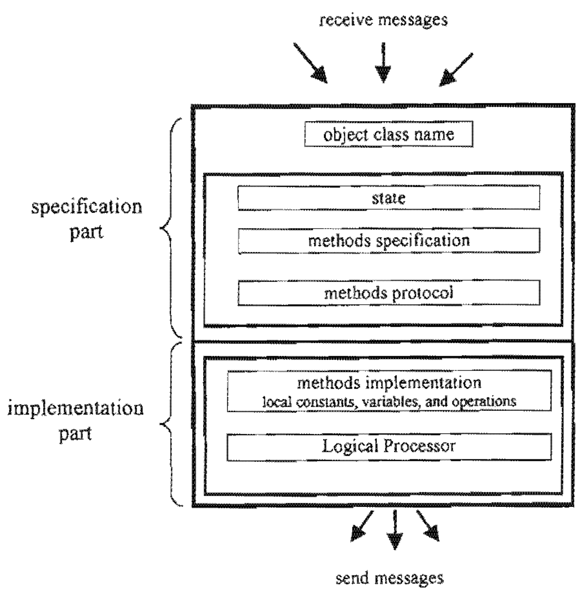

Figure 1: Structure of an object class.

- Memory space is associated with each object of the class for its local constants and variables. These constants and variables determine the state (properties, attributes) of the object at any moment.
- A protocol for receiving messages through the object methods. The protocol specifies the order in which the object's methods are arranged to accept incoming messages. Each object is able to receive messages in accordance with its method's protocol. The messages are queued in the order of arrival to the object. Therefore, the order in which the object responds to the received messages depends uniquely on the protocol's queue.
- Each object may request service from other objects by specifying the following:
  1. The server object name.
  2. The requested service or method name.
  3. The parameters of the method to be passed.
- Objects are considered as concurrent entities that implement autonomous processes. Each object has an independent logical thread of control. Execution of a method performed on the object's own (logical) processor may change the state of the object.

An object class may be seen as an abstract data type when augmented with the capability of defining new object classes based on a parent class by inheritance. In this context, objects of an object class are considered as concurrent units; hence the equation:

Concurrent Object Class =
Abstract Data Type + Inheritance + Concurrency.

A Template Design for Concurrent Object-Oriented Programming

Task type is the Ada 95 implementation for concurrency. The proposed template design, Figure 2, has a single task for each object regardless whether it is a subclass object or a superclass object. Inherited, new, and overridden methods are implemented by a single task of the subclass.

```
with ...;
use ...;
package Classname is
      task type Classname_Class is
             entry Initialize(...);
      entry Method1(X: in A_Type; ...);
      entry Method2(X: in out A_Type; ...);
      ...
end Classname_Class

type Classname_Class_Pointer is access
Classname_Class;
Type T_State_Classname is tagged private;

private
      type T_State_Classname is tagged record
             State_Component : A_Type;
             ...
      end record;
      procedure Initialize(State: in out
T_State_Classname; ...);
      procedure Method1(State: in out
T_State_Classname;
                           X: in A_Type; ...);
      procedure Method2(State: in out
T_State_Classname;
                           X: in out A_Type; ...);
      ...
end Classname;
```

Figure 2: The template of concurrent object classes - its specification.

The state of the object is defined inside the task and the type of the state is defined in the package private part. Methods of the object are the task entries. Encapsulation is automatically provided since the state is part of the task implementation, which is hidden from the calling object. Furthermore, the "accept" statement of a method, which only modifies the state, may have only statements for copying the in-parameters from the calling object. In this case, a higher degree of concurrency is provided. If a method *Method1* has only *in* parameters, then a *Method1's* accept statement may be written as follows:

```
accept Method1(A: in A_Type);
Aa ::= A;
End Method1;
Method1(Aa);
```

The template specifies that the only interface to objects of an object class is through their task entries. Outside objects cannot access internal procedures, because they are defined in the private part. Moreover, outside objects cannot access the state of the object, although the state type T_State_*Classname* is defined in the non-private part of the package specification. However, the state itself is defined as an instance of T_State_*Class* inside the task body. Therefore, the state of the object is encapsulated.

The concurrent object-oriented template defines object classes as task types. Declaring the state as a tagged record supports inheritance. Because an object may be accessed only through its task entries, the new template provides the ability to revoke some methods for a new subclass. This could be accomplished by not specifying task entries for such methods. Hence, the outside objects cannot perform these methods. However, new grandchild classes may inherit these methods from their grandparent class, although their parent class did not inherit these methods.

The Concurrent Object-Oriented Preprocessor Language Definition

In Section 3, a design of an Ada 95 template for representing concurrent object classes was discussed. In this section, a concurrent object-oriented programming language suitable as source language of a preprocessor producing Ada 95 code is introduced. The preprocessor acts like a compiler; it accepts Ada-like object-oriented code as input and produces Ada 95 concurrent object-oriented packages and programs. The produced target code has the structure of the template that was suggested earlier. The preprocessor language forces the use of the object-oriented paradigm. It comprises the definition and the use of program units such as the main program, classes, and subclasses. In addition, the Ada-like object-oriented preprocessor language not only accepts object-oriented code, but also allows the reuse of the existing Ada 95 code.

An object class in the preprocessor language consists of two parts: specification and body. The specification includes the state, methods and the methods' protocol. In Ada 95, task entries are specified in the package specification. However, without referring to the package body it does not give a complete picture of the defined order to access these entries. In the preprocessor language, the protocol specifies the proper order for calling the methods of an object class. Therefore, the user of an object class has all necessary information for using objects of this class from the class specification. The preprocessor uses the protocol to generate the target code for task specification and body. Therefore, the class body only specifies the details of the methods implementation.

The preprocessor allows separate definitions for object classes, subclasses, and main programs. Moreover, definitions of circular dependent object classes have to be put into a single unit called the *collection*. The preprocessor language supports two kinds of class hierarchies, namely inheritance and aggregation. Inheritance provides the use of classes in subclass definitions. Aggregation allows defining a new object class as an aggregation of already defined object classes. In addition, the preprocessor language provides collections of object classes, which have circular dependency on each other.

Object Classes

The object class of the preprocessor is securing its objects such that any access to an object's components is only permitted through its protocol. An object class is a set of objects. An object class consists of two parts: the specification, which provides the visible information, and the body, which provides the hidden details.

```
Syntax:
class_specification ::=
    class classname is
            [type_declarations(3.2.1)]
            [state_declaration]
            methods_declaration
            protocol
    end [classname];
```

The state of an object class is defined in the class specification, but the preprocessor does not provide means to directly access the state from outside as mentioned above. The state declaration is optional, but if the reserved word **state** is specified then at least one declaration must exist. If it is intended to have an empty state, then the state declaration should be omitted. Static and dynamic attributes of the state are declared inside the state specification as variables and/or constants.

```
Syntax:
state_declaration ::=
  state
     variables_and_constants_declarations
  end state;
variables_and_constants_declarations ::=
  var_and_const_declaration;
  {var_and_const_declaration;}
var_and_const_declaration ::=
  object_declaration(3.3.1)number_declaration(3.3.2)
```

An object provides services to other objects through its methods. The methods' declaration of an object class is the list of specifications of the methods that will be defined in the class body.

```
Syntax:
methods_declaration ::=
     {method_specification;}
method_specification ::=
     method method_name parameter_profile(6.1)
```

The methods' protocol, or simply called protocol, is necessary for concurrent objects. It specifies the order in which the methods of an object may be requested.

```
Syntax:
protocol ::=
     protocol
       sequence_of_controls
     end protocol;
```

The protocol consists of a sequence of controls in the preprocessor language. The preprocessor has four kinds of controls: method_specification, exit, select, and loop control. Method_specification control is used to accept methods; it has the same effect as the *accept statement* of Ada 95. Exit control is used to indicate terminating points. Select control provides alternative controls to be selected. Loop control is used for repetitive controls.

```
Syntax:
sequence_of_controls ::=
     control {control}
control ::=
     simple_control | compound_control
simple_control ::=
     exit; | method_specification;
compound_control ::=
     select_control | loop_control
select_control ::=
     select
            sequence_of_controls
            {or sequence_of_controls }
     end select;
loop_control ::=
     loop
            sequence_of_controls
     end loop;
```

The preprocessor language supports generic classes in the same syntax and semantics as the Ada 95 generic packages. For each object class, the preprocessor language automatically defines the types *Classname_***Class** and *Classname_***Class_Pointer**. These types are used to declare objects of the object class or pointers to objects of the object class, respectively. These types are described in more detail in Section 4.2. Moreover, the preprocessor language supports two kinds of service requests: requests of objects' methods, and requests of procedures and functions written in Ada 95.

The Main Program

A complete program in the preprocessor language is conceived as a parameter-less method (subprogram), which calls upon the services of objects through their methods. A method is a program unit or intrinsic operation whose execution is invoked by a method call. A method call is a statement. The definition of a subprogram can be given in two parts: a method declaration defining its interface, and a method body defining its execution.

```
Syntax:
method_declaration ::=
   method_specification;
method_specification ::=
   method method_name parameter_profile(6.1)
parameter_profile ::=
   [formal_part(6.1)]
formal_part ::=
   (parameter_specification(6.1)
   {; parameter_specification} )

method_body::=
   method_specification is
   [variables_and_constants_declarations]
   begin
      sequence_of_statements(5.1)
   end [method_name];
```

Methods in the preprocessor language have the same semantics as Ada 95 procedures. The main difference in the syntax of a method is replacing the reserved word **procedure** by **method**.

Subclasses and Inheritance

Subclasses like object classes have two parts: subclass specification and subclass body.

```
Syntax:
subclass_specification ::=
   subclass subclass_name of subclass_indication is
      [type_declarations(3.2.1)]
      [state_declaration]
      methods_declaration
      protocol
   end [subclass_name];
subclass_indication ::=
   class_name | subclass_name
```

Inheritance is a mechanism that allows the definition of a subclass by extension of a class with new methods and data, while retaining the methods and data of the present class. It provides the ability to reuse and override methods and properties belonging to a parent class (Wong, 1995, Miller, 1998). Most object-oriented programming languages, including Ada 95 do not allow deleting either a property or a method of the new derived class (Loeper, 1998). The preprocessor language adds a new feature to the object-oriented paradigm. It allows revoking some methods, if needed, for new subclasses.

Overriding a method in the preprocessor language is not accomplished by only having the same method name. A method is overridden by a new method only if the new method has the same name and parameter profile. For example, if Y is a subclass of X, and X has the method

```
method Method1(A: in Integer);,
and Y has the method

method Method1(A: in Float);.
```

Then Y may have two methods called *Method1:* one with an *Integer* parameter and the other with a *Float* parameter. The protocol of subclasses has the same syntax and semantics as the class protocol. The only difference between them is that the class protocol refers only to methods that are declared inside the class itself, while the subclass protocol refers to methods of the superclass as well as the methods that are defined in the subclass. The protocol specifies the order for requesting methods of object classes. Therefore, the specification part contains all necessary information for using the object class, while the class body contains the implementation details. Moreover, two new features augment the class inheritance of the object-oriented paradigm, they are abstract methods and selective revoke of methods' inheritance. For example, an object class, say X, may have the following class specification:

```
class X is
  state
    Var1: A_Type;
  end state;
  Method1(A: in P1_Type);
  Method2(B: in P2_Type);
  protocol
    Method1(A: in P1_Type);
    Method2(B: in P2_Type);
  end protocol;
end X;
```

```
subclass Y of X is
 state
Var2:A_type;
 end state;
 Method3(C: in P3_Type);
 protocol
   Method3(C: in P3_Type);
   Method2(B: in P2_Type);
 end protocol;
end Y
```

```
subclass Z of Y is
 protocol
   Method1(A: in P1_Type);
   Method2(B: in P2_Type);
   Method3(B: in P3_Type);
 end protocol;
end Z
```

The subclass Y, which is a child class of X, may revoke *Methodl* by not specifying it as part of the protocol. Yet Z, which is a subclass of Y, may inherit *Methodl* from X.

Aggregation

Object-oriented programming languages have different levels of support for whole/part and generalization/specialization hierarchies. Most object-oriented programming languages, including C++, have not defined special language support for whole/part relationships. Nevertheless, whole/part hierarchies are essential for most object-oriented designs (Wampler, 1998). The preprocessor language supports directly whole/part hierarchies. The aggregation property in the preprocessor language is not simply including a list of parts to provide a new part. It is actually defining a new object class based on existing classes, but the new object class may have its own behavior. For example, an *Automobile* object is not a collection of *Engine, Gear, Body*, and *Tires* only. It has some behaviors, which control these parts to provide the final *Automobile* behavior. In modern automobiles, when the driver requests shifting the gear from park, the *Automobile* object performs two requests to its parts, i.e. shift gear, which is sent to the Gear object, and lock doors, which is sent to the *Body* object.

Whole/part implementation in the preprocessor language is provided by including the definition of an existing object class inside the state of the new object class. Therefore, object class and subclass definitions may contain the other object classes. Hence, both whole/part and specialization/generalization hierarchies may be used together in the preprocessor language at the same time. The preprocessor allows classes to use types defined in their parent classes and/or their part-classes. The *Automobile* object class, for example, uses the type *Gear_Shift*, which is defined in the *Gear* object class. The preprocessor language does not allow an object to access its part-objects directly.

Collections

The preprocessor collection provides the feature of defining a set of object classes in one collection. The collection is a description of a list of related object classes. Objects may be interrelated; for example males and females are related object classes in the following sense. Each object of these object classes has a reference to a father and mother object. Therefore, the female object class is dependent on the definition of the male object class since each female object has a father, which is an object of the male object class. Also the male object class is dependent on the definition of the female object class since each male object has a mother, which is an object of the female object class.

The collection in the preprocessor language differs slightly from the Ada 95 package in the sense that it contains only classes. A library unit may be a collection of related classes, a single class, or a main method. The preprocessor language collections and classes are transformed into Ada packages as target code. Collections in the preprocessor language are composed of two parts: the specification, which gives the general information to the outside world, and the body, which gives the hidden details.

```
Syntax:
collection_specification ::=
   collection collection_name is
      {class_specification |subclass_specification}
   end[collection_name];
collection_body ::=
   collection body collection_name is
   {class_body | subclass_body}
   end [collection_name];
```

Although a collection may define many classes, yet each class is encapsulated

A GCD Example Written in the Preprocessor Language

In this section, a well-known classical algorithm and its concurrent object-oriented coding using the preprocessor language will be discussed. This example is concerned with the concurrent solution for the greatest common divisor (GCD) of N natural numbers specified in (Mattern, 1989). To determine $GCD(x_1, x_2, \ldots, x_{n-1}, x_n)$ where:

$$GCD(x_1, x_2, \ldots, x_{n-1}, x_n) = GCD(x_1, GCD(x_2, \ldots, GCD(x_{n-1}, x_n) \ldots))$$

A ring of N concurrent processes for *Gcd* objects with initial states $x_1, x_2, \ldots, x_N$, where $x_i \in N$ and $N > 1$ is created. Each object sends a message to its left and right neighbor objects to report its current state y. If the receiving object is in state x, then the use of the equation:

$$GCD(x,y) \begin{cases} x & \text{if } x=y \\ GCD\ (y,((-1) \bmod y) + 1) & \text{else} \end{cases}$$

leads to the following pattern of behavior of the objects being in the state x and receiving a message y:

- The message will be accepted if x > y.
- The new state of the receiving object is determined by (x-1) mod y + 1.
- The new state is sent to the neighbor objects.

Figure 3 represents a ring of three *Gcd* objects determining the common divisor *GCD* (210, 105, 231).

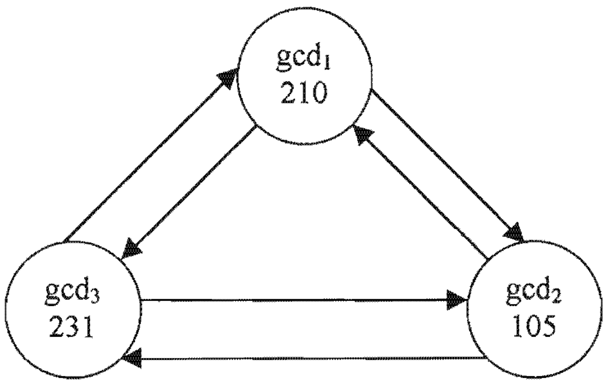

Figure 3: Ring of three *Gcd* objects

A possible process for state transitions of this ring of *Gcd* objects is shown in Figure 4. It is assumed that bold arrows represent those messages that arrive at the neighbor first and/or cause a state transition. As one can see, the system ends up in a kind of fixed point. That is, when all objects have the same state, but they still are sending messages to their neighbors. Although the process of sending messages does not terminate, the stable state is the greatest common divisor. A proof can be found in (Best, 1995).

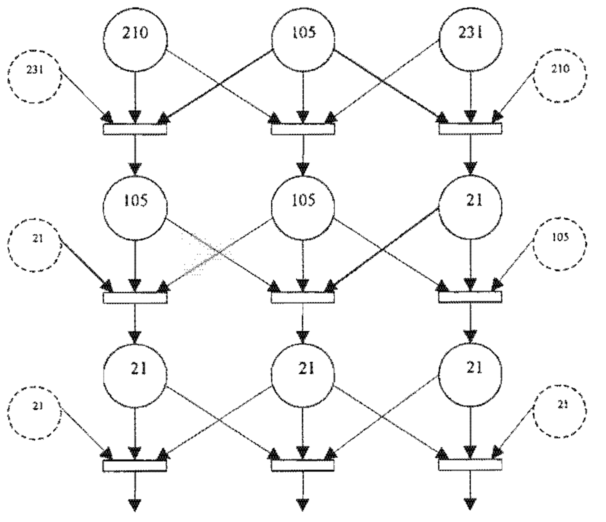

Figure 4: State transitions in the ring of GCD objects

The investigation will concentrate on implementing this example as concurrent object classes. Since the Ada rendezvous is synchronous and asymmetric, the implementation of the *Gcd* example needs more detailed analysis of the communication model. Figure 3 shows how neighboring objects communicate with each other via messages. Each object behaves according to the same cyclic pattern:

- Sending the object's state to its neighbors as messages.
- Receiving the states of the object's neighbors as messages.
- Calculating the new state of the object.

Assume two neighboring *Gcd* objects are sending messages of their states to each other. This leads to a deadlock situation where one *Gcd* object is waiting for accepting the entry call from the other, while the other *Gcd* object is waiting for the acknowledgment signal. Both *Gcd* objects are blocking each other by circular waiting. To prevent the deadlock situation, an additional object called *Dst* is attached to each *Gcd* object (see Figure 5). *Dst* is responsible for receiving the updated state of the *Gcd* neighbors and providing these states to the *Gcd* object.

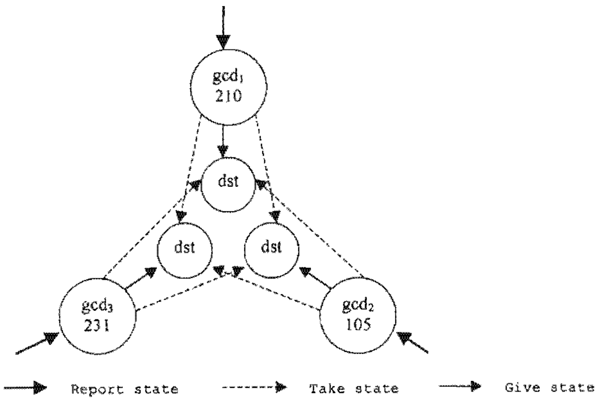

Figure 5: Object-oriented solution for the *Gcd* problem

The definitions of the *Gcd* and *Dst* object classes are combined together in one collection. It is not necessary in this case to have both object classes in one collection since there is no cyclic dependency between *Gcd* and *Dst* object classes. However, the solution is presented in this way to show the collection feature of the preprocessor language. Figure 6 represents the collection specification, which contains the specifications for the *Gcd* and *Dst* object classes. The implementation of these object classes is included in the collection body, which is given in Figure 7.

```
with Text_Io, Int_Io;
use Text_Io, Int_Io;
collection Gcd_Dst is

class Dst is
   state
      Left_Gcd_State : Natural;
      Right_Gcd_State : Natural;
   end state;
   method Take_Left(S: in Natural);
   method Take_Right(S: in Natural);
   method Give_Left(S: out Natural);
   method Give_Right(S: out Natural);

   protocol
      select
      method Take_Left(S: in Natural);
      method Take_Right(S: in Natural);
   or
      method Take_Right(S: in Natural);
      method Take_Left(S: in Natural);
   end select;
   loop
      select
         method Take_Left(S: in Natural);
      or
         method Take_Right(S: in Natural);
      or
         method Give_Left(S: out Natural);
      or
         method Give_Right(S: out Natural);
      or
         exit;
      end select;
   end loop;
   end protocol;
end Dst;

class Gcd is
   state
      Current_State : Natural;
      Dst : Dst_Class_Pointer;
      Left_Dst : Dst_Class_Pointer;
      Right_Dst : Dst_Class_Pointer;
   end state;
method Initialize(S: in Natural;L:in
      Dst_Class_Pointer; R:Dst_Class_Pointer;
      D:Dst_Class_Pointer);
   method Calc_Gcd;
   method Report_State(X: out Natural);
   protocol
method Initialize(S: in Natural; L: in
      Dst_Class_Pointer; R: Dst_Class_Pointer;
         D: Dst_Class_Pointer);
      method Calc_Gcd;
      method Report_State(X: out Natural);
   end protocol;
end Gcd;
end Gcd_Dst;
```

Figure 6: Collection specification for the GCD and DST object classes

The *Gcd* object class has four attributes as part of its state: a natural number to store the current state, and three pointers to *Dst* objects. The first pointer is to the owned *Dst* object, while the other two pointers are to the *Dst* objects of the neighboring *Gcd* objects. The *Gcd* object class has three methods: *Initialize, Calc_Gcd,* and *Report_State. Initialize* is responsible for initializing the Gcd objects and linking each *Gcd* object with the proper *Dst* objects. *Calc_Gcd* is responsible for getting the neighbors' current states from the owned *Dst* object, then calculating the new greatest common divisor based on the neighbors' states until the current state matches both neighbors state. The method *Report_State* is responsible for reporting the current state to the main program.

```
State.Right_Gcd_State := S;
      end Take_Right;
      method Give_Left(S: out Natural) is
      begin
         S := State.Left_Gcd_State;
      end Give_Left;
      method Give_Right(S: out Natural) is
      begin
         S := State.Right_Gcd_State;
      end Give_Right;
   end Dst;
class body Gcd is
      method Initialize(S: in Natural; L: in
                        Dst_Class_Pointer;
collection body Gcd_Dst is
   class body Dst is
      method Take_Left(S: in Natural) is
      begin
         State.Left_Gcd_State := S;
      end Take_Left;
      method Take_Right(S: in Natural) is
      begin
                    R: in Dst_Class_Pointer;
                 D: in Dst_Class_Pointer) is
      begin
         State.Current_State := s;
         State.Left_Dst := l;
         State.Right_Dst := r;
         State.Dst := d;
         State.Left_Dst.Take_Left(s);
         State.Right_Dst.Take_Right(s);
      end Initialize;
   method Gcd(X: in Natural; Y: in out Natural) is
      Z : Natural;
   begin
      if X /= Y then
         Z := Y;
         Y := (x-1) mod z + 1;
         Gcd(State, Z, Y);
      end if;
   end Gcd;

   method Calc_Gcd is
      Neighbor_State : Natural;
      Left_Ok, Right_Ok : Boolean := False;
   begin
      while not Left_Ok or not Right_Ok loop
         Left_Ok := False;
         Right_Ok := False;
         State.Dst.Give_Left(Neighbor_State);
         if Neighbor_State = StateCurrent_State then
            Left_Ok := True;
         else
            Gcd(State,Neighbor_State,State.
               Current_State); end if;
State.Dst.Give_Right(Neighbor_State);
if Neighbor_State = State.Current_State then
Right_Ok := True;
         else
Gcd(State,Neighbor_State,
State.Current_State);
end if;
State.Left_Dst.Take_Right(State.Current_State);
State.Right_Dst.Take_Left(State.Current_State);
         end loop;
      end Calc_Gcd;
      method Report_State(X: out Natural) is
      begin
         x := State.Current_State;
      end Report_State;
      end Gcd;
end Gcd_Dst;
```

Figure 7: The collection body for the GCD and DST object classes

The state of the *Dst* object class has two attributes for storing the states of the *Gcd* objects. It has four methods to take the state of the neighboring *Gcd* objects and to give these states to the owner *Gcd* object based on its request. The protocol specifies that the *Dst* object must take the state of both neighboring *Gcd* objects before providing any result to its *Gcd* object. This confirms that the *Gcd* object will not receive non-initialized values. The main method *Gcd_Example*, given in Figure 8, uses the collection of the *Gcd* and the *Dst* object classes. The target code for the collection and the main method is shown in Appendix A. Note the difference in size and complexity between the source and target code. The program test results are provided in the same appendix.

```
with Text_Io, Int_Io, Gcd_Dst;
use Text_Io, Int_Io, Gcd_Dst;

method Gcd_Example is
  G1, G2, G3  : Gcd_Class_Pointer := new
Gcd_Class;
  D1, D2, D3: Dst_Class_Pointer  := new
Dst_Class;
  A1, A2, A3  : Natural;
  Final, Result  : Natural;
  Finish  : Boolean := False;
begin
  New_Line;
  Put("enter three numbers for greater common
divisor:");
  Get(A1); Get(A2); get(A3);
  G1.Initialize(A1,D3,D2,D1);
  G2.Initialize(A2,D1,D3,D2);
  G3.Initialize(A3,D2,D1,D3);
  G1.Calc_Gcd;
  G2.Calc_Gcd;
  G3.Calc_Gcd;
  New_Line;
  Put("The Greatest Common divisor for: ");
  Put(A1,4); Put(A2,4); Put(A3,4);
  while not Finish loop
     Finish := True;
     G1.Report_State(Final);
     G2.Report_State(Result);
     if Final /= Result then Finish := False;
     end if;
     G3.Report_State(Result);
     if Final /= Result then Finish := False;
     end if;
end loop;
  Put(" is: ");Put(final,4);
end Gcd_Example;
```

Figure 8: The main method Gcd_Example

Conclusion

Although Ada 95 has all the features for building object-oriented programs, the user is not forced to use the object-oriented methodology with Ada 95. In this paper, the authors have proposed a uniform template for the structure of concurrent object classes. The paper has also investigated the design and implementation of a preprocessor language based on Ada 95 for concurrent object classes. The preprocessor accepts Ada-like object-oriented program units (classes, subclasses, and a main program) written in the object-oriented preprocessor language and produces Ada 95 concurrent object-oriented code, which is based on the proposed template.

It has been shown that thè preprocessor language supports the object-oriented design process of concurrent programs by keeping the programmer away from details necessary for implementing concurrent object-oriented units in Ada 95. This is done mechanically by generating the concurrent program units by the preprocessor. The object-oriented preprocessor language has an advantage over other object-oriented languages by adding a new component to the class specification called the protocol. The protocol specifies the order for requesting methods of object classes. Therefore, the specification part contains all necessary information for using the object class, while the class body contains the implementation details. In addition, the preprocessor language supports two different ways for defining class hierarchies: class inheritance and aggregation. Moreover, two new features to augment the class inheritance of the object-oriented paradigm have been discussed: abstract methods and the selective revoke of methods inheritance. The research also investigated the definition of circular dependent object classes and proposed a solution by introducing the collection of classes.

The comprehensiveness and solidity of the preprocessor language and its implementation have been demonstrated in three distinct examples. The examples have shown the direct and simple conversion of the problem analysis to the program coding in the preprocessor language. The preprocessor language supports the re-use of Ada packages, which in turn are not necessarily written according to the object-oriented approach as noticed in the examples. Finally, the difference in size and complexity is noticeable between the source code of the preprocessor language and its Ada-95 target code. These examples are the greatest common devisor of n natural numbers (called *Gcd*), the prime number sieve of Eratosthenes (called Prime), and, last but not least a simulation program for customers entering a bank, standing in line, being served by tellers, and leaving the bank (called Bank). The following table shows the noticeable difference in size and complexity between the source code written in the preprocessor language and the Ada 95 target code produced by the preprocessor for these examples. These results point out that a much higher degree of compactness combined with clarity in the structure of the program units and their interrelations can be reached in describing object-oriented problem solutions using the preprocessor language than that of Ada 95.

Example	Source Code Size	Target Code Size	Ratio Source Code to Target Code
GCD	5012	8951	56%
Prime	1598	2633	61%
Bank	9081	15141	60%
Total	15691	26725	59%

Appendix A: The Target Code for the GCD Example

The Target code for the Gcd_Dst collection specification:

```
with Text_Io , Int_Io ;
use Text_Io , Int_Io ;

package Gcd_Dst is type Dst_Class ;
  type Dst_Class_Pointer is access Dst_Class ;
  type Gcd_Class ;
  type Gcd_Class_Pointer is access Gcd_Class ;
  task type Dst_Class is
    entry Take_Left ( S : in Natural ) ;
    entry Take_Right ( S : in Natural ) ;
    entry Give_Left ( S : out Natural ) ;
    entry Give_Right ( S : out Natural ) ;
  end Dst_Class ;

  type T_State_Dst is tagged private ;

  task type Gcd_Class is
    entry Initialize ( S : in Natural ;
      L : in Dst_Class_Pointer ;
      R : Dst_Class_Pointer ;
      D : Dst_Class_Pointer ) ;
    entry Calc_Gcd ;
    entry Report_State ( X : out Natural ) ;
  end Gcd_Class ;
  type T_State_Gcd is tagged private ;
private

  type T_State_Dst is tagged record
    Left_Gcd_State : Natural ;
    Right_Gcd_State : Natural ;
  end record ;

  procedure Take_Left (State : in out
T_State_Dst; S : in Natural);
  procedure Take_Right (State : in out
T_State_Dst; S : in Natural);
  procedure Give_Left (State : in out
T_State_Dst; S : out Natural);
  procedure Give_Right (State : in out
T_State_Dst; S : out Natural);
  type T_State_Gcd is tagged record
    Current_State : Natural ;
    Dst : Dst_Class_Pointer ;
    Left_Dst : Dst_Class_Pointer ;
    Right_Dst : Dst_Class_Pointer ;
  end record ;

  procedure Initialize ( State : in out
T_State_Gcd ;
      S : in Natural ;
      L : in Dst_Class_Pointer ;
      R : Dst_Class_Pointer ;
      D : Dst_Class_Pointer ) ;
  procedure Calc_Gcd ( State : in out
T_State_Gcd ) ;
  procedure Report_State ( State : in out
T_State_Gcd ;
      X : out Natural ) ;
end Gcd_Dst ;
```

The Target code for the Gcd_Dst package body:

```
package body Gcd_Dst is
  procedure Take_Left ( State : in out
T_State_Dst ;
S : in Natural ) is
  begin
    State . Left_Gcd_State := S ;
  end Take_Left ;
  procedure Take_Right ( State : in out
T_State_Dst ;
      S : in Natural ) is
  begin
    State . Right_Gcd_State := S ;
  end Take_Right ;
  procedure Give_Left ( State : in out
T_State_Dst ;
S : out Natural ) is
  begin
    S := State . Left_Gcd_State ;
  end Give_Left ;
  procedure Give_Right ( State : in out
T_State_Dst ; S : out Natural ) is
  begin
    S := State . Right_Gcd_State ;
  end Give_Right ;
  task body Dst_Class is
    State : T_State_Dst ;
    Reserved01 : Natural ;
    Reserved02 : Natural ;
    Reserved03 : Natural ;
    Reserved04 : Natural ;
    Reserved05 : Natural ;
    Reserved06 : Natural ;
  begin
    select
      accept Take_Left ( S : in Natural ) do
Reserved01 := S ;
      end Take_Left ;
      Take_Left ( State , Reserved01 ) ;
      accept Take_Right ( S : in Natural ) do
Reserved02 := S ;
      end Take_Right ;
      Take_Right ( State , Reserved02 ) ;
    or
      accept Take_Right ( S : in Natural ) do
Reserved03 := S ;
      end Take_Right ;
      Take_Right ( State , Reserved03 ) ;
      accept Take_Left ( S : in Natural ) do
Reserved04 := S ;
      end Take_Left ;
      Take_Left ( State , Reserved04 ) ;
    end select ;
  loop
```

```
    select
      accept Take_Left ( S : in Natural ) do
Reserved05 := S ;
      end Take_Left ;
      Take_Left ( State , Reserved05 ) ;
    or
      accept Take_Right ( S : in Natural ) do
Reserved06 := S ;
      end Take_Right ;
    Take_Right ( State , Reserved06 ) ;
    or
      accept Give_Left ( S : out Natural ) do
Give_Left ( State , S ) ;
      end Give_Left ;
    or
      accept Give_Right ( S : out Natural ) do
Give_Right ( State , S ) ;
      end Give_Right ;
    or
      terminate ;
    end select ;
  end loop ;
end Dst_Class ;

  procedure Initialize ( State : in out
T_State_Gcd ;
      S : in Natural ;
      L : in Dst_Class_Pointer ;
      R : in Dst_Class_Pointer ;
      D : in Dst_Class_Pointer ) is
  begin
    State . Current_State := S ;
    State . Left_Dst := L ;
    State . Right_Dst := R ;
    State . Dst := D ;
    State . Left_Dst . Take_Left ( S ) ;
    State . Right_Dst . Take_Right ( S ) ;
  end Initialize ;

  procedure Gcd ( State : in out T_State_Gcd ;
      X : in Natural ;
      Y : in out Natural ) is
      Z : Natural ;
  begin
    if ( X /= Y ) then
      Z := Y ;
      Y := ( ( X - 1 ) mod Z ) + 1 ;
      Gcd ( State , Z , Y ) ;
    end if ;
  end Gcd ;

  procedure Calc_Gcd ( State : in out
T_State_Gcd ) is
    Neighbor_State : Natural ;
    Left_Ok , Right_Ok : Boolean := False ;
  begin
  while ( not Left_Ok or not Right_Ok ) loop
    Left_Ok := False ;
    Right_Ok := False ;
    State . Dst . Give_Left ( Neighbor_State ) ;
    if Neighbor_State = State . Current_State then
      Left_Ok := True ;
    else
      Gcd ( State , Neighbor_State , State .
Current_State ) ;
    end if ;
    State . Dst . Give_Right ( Neighbor_State ) ;
    if Neighbor_State = State . Current_State then
      Right_Ok := True ;
    else
      Gcd ( State , Neighbor_State , State .
Current_State ) ;
    end if ;
    State . Left_Dst . Take_Right ( State .
Current_State ) ;
    State . Right_Dst . Take_Left ( State .
Current_State ) ;
    end loop ;
  end Calc_Gcd ;

  procedure Report_State ( State : in out
T_State_Gcd ; X : out Natural ) is
  begin
    X := State . Current_State ;
  end Report_State ;

  task body Gcd_Class is
    State : T_State_Gcd ;
    Reserved09 : Natural ;
    Reserved10 : Dst_Class_Pointer ;
    Reserved11 : Dst_Class_Pointer ;
    Reserved12 : Dst_Class_Pointer ;
  begin
    accept Initialize ( S : in Natural ;
        L : in Dst_Class_Pointer ;
        R : Dst_Class_Pointer ;
        D : Dst_Class_Pointer ) do
      Reserved09 := S ;
      Reserved10 := L ;
      Reserved11 := R ;
      Reserved12 := D ;
    end Initialize ;
    Initialize(State, Reserved09, Reserved10,
Reserved11, Reserved12);
    accept Calc_Gcd ;
    Calc_Gcd ( State ) ;
    accept Report_State ( X : out Natural ) do
Report_State ( State , X ) ;
    end Report_State ;
  end Gcd_Class ;
end Gcd_Dst ;
```

The Target code for the Main Method:

```
with Text_Io , Int_Io , Gcd_Dst ;
use Text_Io , Int_Io , Gcd_Dst ;
procedure Gcd_Example is
  G1 , G2 , G3 : Gcd_Class_Pointer := new
Gcd_Class ;
  D1 , D2 , D3 : Dst_Class_Pointer := new
Dst_Class ;
  A1 , A2 , A3 : Natural ;
  Final , Result : Natural ;
  Finish : Boolean := False ;
begin
  New_Line ;
  Put ( "enter three numbers for greater
common divisor:" ) ;
  Get ( A1 ) ;
  Get ( A2 ) ;
  Get ( A3 ) ;
  G1 . Initialize ( A1 , D3 , D2 , D1 ) ;
  G2 . Initialize ( A2 , D1 , D3 , D2 ) ;
  G3 . Initialize ( A3 , D2 , D1 , D3 ) ;
  G1 . Calc_Gcd ;
  G2 . Calc_Gcd ;
  G3 . Calc_Gcd ;
  New_Line ;
  Put ( "The Greatest Common divisor for: " ) ;
  Put ( A1 , 4 ) ;
  Put ( A2 , 4 ) ;
  Put ( A3 , 4 ) ;
  while not Finish loop
    Finish := True ;
    G1 . Report_State ( Final ) ;
    G2 . Report_State ( Result ) ;
    if Final /= Result then Finish := False ;
    end if ;
    G3 . Report_State ( Result ) ;
    if Final /= Result then
      Finish := False ;
    end if ;
  end loop ;
  Put ( " is: " ) ;
  Put ( Final , 4 ) ;
end Gcd_Example ;
```

Test experiment:

Enter three numbers for greater common divisor:

The greatest common divisor for: 105 210 231 is: 21

References

Best, E. (1995) Semantik - Theorie sequntieller and paralleler Programmierung. Vieweg Verlag, Braunschweig.

Brosgol, B. (1997) A Comparison of the Object-Oriented Features of Ada 95 and Java. Proceedings of the Conference on TRI - Ada '97, pp. 213-229.

Booch, G. (1991) Object-Oriented Design with applications. Benjamin/Cummings Publishing Company.

Khattab, A. (1997) Concurrent Object-Oriented Methodology Based on Ada 95. Master Thesis in Computer Science, Kuwait University.

Krakowiak, S., *et al.* (1990) Design and implementation of an Object-Oriented, Strongly Typed Language for Distributed Applications. Journal of Object-Oriented Programming, September/October, 1990, pp. 11-220.

Loeper, H., Khattab, A., and Neubert, P. (1997) Concurrent Objects in Ada 95. ACM Ada Letters **17** (6): 47-64.

Loeper, H., Khattab, A., Neubert, P., and El-Ghabali, M. (1998) An Object-Oriented programming paradigm based on Ada 95. Kuwait Journal of Science and Engineering **25** (2): 275-296.

Mattern, F. (1989) Verteilte Basisalgorithmen. Informatik Fachberichte 226, Springer - Verlag.

Miller, Mark (1998) Reuse through Inheritance and Object Composition Good Class Design in Delphi. http://www.eagle-software.com/inherita.htm .

Odell, J. (1994) Six Different Kinds of Composition. Journal of Object-Oriented Programming **5** (8): 10-15.

Wampler, Bruce E. (1998) The Essence of Object-Oriented Programming (see). http://www.objectcentral.com/oobook/webpref.html

Wegner, P. (1987) Dimensions of Object-Based Language Design. Proceeding Oopsla'87 Acm Signplan Notices 22: 168-181.

Wellings, A., ***et al*** (2000) Integrating Object-Oriented Programming and Protected Objectsin Ada 95. ACM Transactions on Programming languages and Systems **22** (3): 506-539.

Wong, Ken (1995) Inheritance and Code Reuse. http://www.geog.ubc.ca/numeric/labs/c++/c++/node11.html

Received 08/03/2000, in revised form 03/07/2001