Object CHILL - An Object Oriented Language for Systems Implementation

Jürgen F. H. Winkler, Georg Dießl
Siemens AG, Corporate Research and Development
Otto-Hahn-Ring 6, W-8000 Munich 83, Germany

Object orientation is a programming paradigm that has gained considerable attention during the last years, especially as a principle for structuring large software systems that are typical for the telecom field.

This paper reviews object oriented principles and describes the main aspects of Object CHILL, an object oriented extension of CHILL which has been developed and implemented at Siemens. Apart from the elements of object oriented programming i.e. classes and inheritance, Object CHILL contains concurrent classes, generic classes and assertions for methods and classes/objects. Object CHILL is currently being used for the implementation of switching software in an ISDN-project.

1 Introduction

Object orientation is a programming paradigm that has gained considerable attention during the last years. It is a principle not only for programming-in-the-small but - more important - for structuring large software systems in order to control complexity, to facilitate evolution, and to provide a basis for reuse of software components. These topics are vital in the application area of CHILL as well [KKM 91].

Within Siemens the OO paradigm is being used in an ongoing pilot project for the development of an ISDN switching system [GW 91].

For the implementation of this system we use Object CHILL [DSW 90] an extension of the CCITT language CHILL [ITU 8x; Rek 82; Win 86]. Object CHILL contains the essential elements of OOP, and furthermore other elements which support the construction of SW systems with high complexity and reliability. These other elements are generic classes and assertions. Additionally, the concept of concurrency has been fully integrated into the OO framework making Object CHILL a concurrent object oriented language.

CHILL has originated in the telecom field but it is by no means limited to this field. It is a procedural language similar to Ada and can be used for the whole field of system implementation.

The paper is structured as follows. Chapter 2 reviews the essential elements of OOP. Chapter 3 reviews the main elements of CHILL and the limitations of it in the area of OOP. In chapter 4 we describe the new language elements of Object CHILL and how they have been integrated into the framework of CHILL. In the appendix we show the application of the OO features of Object CHILL by using the popular example of simple geometric figures.

2 Object Oriented Principles

The basic idea of OOP is to break up a software system into components or building blocks, which are called objects. An **object** consists of **data** and **methods** to manipulate these data. Like a data abstraction, an object provides a well defined interface but hides implementation details. Objects of the same kind are grouped into **classes**, in the same way as the integer numbers are grouped into the type INTEGER. A class is a pattern used to define **instances** of the class which are the **objects**.

Classes can be constructed incrementally using the notion of **inheritance**: a class derived from another class inherits all of its elements and features. The inheritance mechanism enables the programmer to efficiently reuse software components; he utilizes existing classes as *ancestors* with the possibility to modify and extend the *heirs* and without the necessity to change the ancestors (which might be harmful to other components using ancestor objects and therefore relying on the stability of the ancestor classes).

Objects communicate with each other by exchanging messages: a **message** sent to an object invokes one of the methods provided by this object. Strictly speaking, message passing does not impose a preference on any of the usual communication mechanisms. Due to the fact that many of currently available "object oriented" programming languages are implemented in a procedural environment, message passing has the flavor of procedure calling. We will show that the realization of object oriented programming in a CHILL based environment allows for a variety of **synchronous** and **asynchronous communication mechanisms** to be subsumed under the message passing concept. Thus object oriented principles are naturally translated into a world of **concurrency**.

Many of the principles described are already well supported by the CHILL language, others require some additional language features. Both are discussed in chapter 3.

3 Facilities and Limitations of CHILL

As mentioned in chapter 2 there are mainly two concepts which characterize the difference of object oriented and classical programming languages, namely:

- data abstraction
  (classes are abstract types);
- inheritance
  (subclasses are extensions of superclasses).

3.1 Data Abstraction

In an earlier paper one of the authors investigated the facilities of CHILL for the realization of data abstractions, and presented a proposal for including genericity and assertions for data abstractions into CHILL [Win 84]. This investigation shows that CHILL contains some language constructs for the realization of data abstractions, but that there are also certain limitations.

CHILL allows for data abstraction by information hiding; this can be easily achieved in a module or region by granting (i.e. exporting) the operations but not the data components manipulated by these operations. This kind of abstraction allows the definition of objects but not of classes (abstract types), because a module is rather an object than a type. On the other hand, CHILL types could be a basis for abstract types. Abstraction and information hiding, however, are possible only for *struct types*: granted struct types can be made opaque using the FORBID clause.

Remark: in CHILL types are called "modes". In this paper we use consistently the term "type" because it is more popular in the programming language field.

This shows that a direct modeling of classes is not possible because of limitations of CHILL. Therefore it seems reasonable to consider an extension of CHILL allowing the full definition of abstract types within the language.

3.2 Inheritance

In a technical sense inheritance means the extension / modification of abstract data types. This can be done in two dimensions:

- adding / modifying operations;
- adding data components.

In CHILL operations are realized as procedures where the type in question occurs as the type of one or more parameters and/or the type of the result. It is always possible to define further procedures with arbitrary types. Therefore, addition and modification of operations is possible in CHILL; however, this requires a modification of the module defining the original abstract data type.

Adding new components to an existing type (e.g. a struct type) can only be achieved by modifying the very definition of this type. Thus, the containing module has to be recompiled and, even worse, completely retested. Additionally, this modification will also affect the users of the original type. This situation is quite the same as in other classical programming languages [Wir 88]. In object oriented languages, this kind of data extension is possible in a way that the base type itself as well as the users of the base type are not affected by such extensions. This leads to software components that are both self-contained (closed and stable for clients) and adaptable (open to changes for the programmer via inheritance). For these benefits is seems worthwhile to consider the introduction of inheritance (type extension) into CHILL.

3.3 Concurrency

CHILL is well suited to handle the aspects of concurrency that are typical and critical in the telecom field. Up to now, concurrency has not been fully addressed by object oriented languages. There are first solutions, but there is also an ongoing discussion how object oriented programming and concurrency should be combined [AWY 89; OOP 90]. Therefore, the introduction of object orientation into CHILL is

done in a way that retains the power of CHILL in the area of concurrency.

4 From CHILL to Object CHILL

Object CHILL [DSW 90] has been developed by adding language features to CHILL which support object oriented programming as described above.

The idea to achieve this goal by introducing *module types* and *region types* was considered but rejected due to some problems concerning concurrency and due to lack of inheritance mechanisms for types.

Consequently, *classes* have been introduced as the basic program units, including the definition of *inheritance* relations between classes.

An Object CHILL program contains CLASS definitions and at least one CHILL module which serves as the main program; other modules may be included, thereby facilitating the combination with existing CHILL software. This combination of classes and modules facilitates the migration from CHILL to Object CHILL

4.1 Classes and Inheritance

An Object CHILL *class* includes components which may be CHILL variables or constants, types and methods for manipulating the variables. Methods are very similar to CHILL procedures. Syntactically, a class definition consists of two parts: class specification and class body.

The class *specification* defines the essential interfaces of the class:

- The *export interface* which defines all methods, constants and types accessible to program units that declare and use objects of that class (clients); this interface is defined via GRANT statements.
- The *import interface* which names entities (classes, types, etc.) used within the specified class; it is defined via SEIZE statements.
- The *internal interface* which defines additional methods, variables, constants and types accessible to the class body and to subclasses.
  The export and the internal interfaces where designed to support separate compilation in an efficient manner: a client class / subclass can be compiled as soon as the specifications of all of its server / ancestor classes are available. It does not need (and as a consequence, is not affected by changes to) the bodies of its server / ancestor classes.

The class *body* contains the implementation of methods defined in the specification and may include further procedures, types , and constants that are reserved for internal use.

A class is used as a pattern to define *objects*. In Object CHILL this definition (also called instantiation in the object oriented world) resembles the declaration of a CHILL variable:

```
DCL my_circle CIRCLE (10,-20,20);
```

or the allocation of a variable on the heap :

```
DCL my_ptr REF CIRCLE;
my_ptr := NEW CIRCLE (10,-20,20);
```

Object CHILL provides for *inheritance* as described above. The language supports two kinds of inheritance:

```
CIRCLE: MODULE CLASS IS_A FIGURE
    and
UNIT_CIRCLE: MODULE CLASS INHERITS FIGURE
```

that differ in the properties subclasses inherit from their superclass. Currently Object CHILL does not support multiple inheritance; any class can inherit from at most one ancestor class.

For *internal* use, i.e. for implementing the subclass methods, all the components defined in the superclass specification, i.e. in the export and internal interfaces, are available. Additional components may be defined as needed. This holds for both forms of inheritance.

The *export interface* is defined as follows:

An IS_A-subclass inherits the export interface from the ancestor class; extensions are possible, but deletions are prohibited. As a consequence, a subclass object provides at least the methods provided by a superclass object. This enables the programmer to access subclass objects via superclass pointers without encountering the danger of inconsistencies - an important feature of object oriented programming languages known as *polymorphism*.

For an INHERITS-subclass, the export interface may be defined independently of the ancestor class. Any of the components, inherited from the ancestor class or defined within the subclass, may be made available to clients as appropriate. Objects of INHERITS-subclasses cannot be accessed in a polymorphic fashion.

Methods are invoked by addressing an object, identifying the desired method and providing appropriate actual parameters:

```
my_circle.set_position (MyPoint);
```

The semantics of method invocation with respect to concurrency and synchronization is defined along with different class types that will be described in the following section.

4.2 Concurrency

Object CHILL allows for concurrent execution of objects. Thus one or more objects of the same or different classes can be active at the same time. Communication and synchronization are achieved with the calling of methods.

- *Communication*: Parameters and results of methods are the means for communication.
- *Concurrency*: Different objects can execute in parallel. The activation of an object is achieved by the elaboration of its declaration (DCL or NEW). It is then ready to accept calls of its methods. Whenever a method is called asynchronously the calling object continues its execution without waiting for the termination of the method called. So both objects execute in parallel.
- *Consistency*: Methods of one object operate on the internal data of that object. Inconsistencies are possible if several methods of an object execute in parallel. Therefore, mechanisms are provided to ensure that at most one method is active at the same time.

With respect to concurrency Object CHILL provides three kinds of classes :

- **MODULE** classes whose objects have the properties of CHILL modules:
  `MODULE CLASS` objects are accessed procedurally (no own thread of control), and concurrent method invocations are not synchronized.
- **REGION** classes whose objects have the properties of CHILL regions:
  `REGION CLASS` objects are accessed in a procedural way (no own thread of control), but concurrent method invocations are synchronized. They are a means for providing objects with mutually exclusive access to their data. Therefore, a REGION object behaves very much like a monitor [Hoa 74].
- **CONCURRENT** classes whose objects have the properties of CHILL processes :
  `CONCURRENT CLASS` objects are accessed in an asynchronous way (own thread of control), and concurrent method invocations are synchronized.

4.3 Genericity

Genericity, which is sometimes also called "parametric polymorphism" [CW 85], is a concept for general parameterization of program components where the possible parameters also include types. This concept, the most elaborate version of which has been incorporated into Ada [Ref 83], is especially suited for strongly typed languages [Weg 87].

Genericity is typically applied in situations where a concept can be used with different types. An example is the parameterization of a data abstraction with the type of a component, as e.g. a stack type which may be parameterized with the type for the stack elements. The basic algorithms for the operations on a stack do not depend on the element type. Therefore, the same algorithms can be used for a stack of integers or a stack of structures. In languages without genericity stack operations must be repeated for each different element type. This leads to an unnecessary duplication of code with the well known negative effects on maintainability.

Genericity does not violate strong typing. The concept for Ada and the proposal given in [Win 84] guarantee that generic instantiations of a correct generic construct are strongly typed and type correct.

In Object CHILL genericity is incorporated on the level of the classes: a class may be parameterized by constants, types, and other classes.
A generic class for a general stack may look as follows:

```
StackType: GENERIC
    SYNMODE  ElemType = ANY_ASSIGN;
    SYN      Length   = RANGE (1:255);
REGION CLASS
    GRANT Push, Pop, Empty PREFIXED StackType;
    Push:  PROC (Elem ElemType); END Push;
    Pop:   PROC (); END Pop;
    Empty: PROC (); RETURNS (Bool) END Empty;
    StackType: CONSTR (); END StackType;
 /* Begin of internal part */
    DCL Stack  ARRAY (1:Length) ElemType;
    DCL TopOfStack  RANGE (0:Length);
END  StackType;
```

There are several kinds of generic types: `ANY`, `ANY_ASSIGN`, `ANY_DISCRETE`, and `ANY_INT` . The kind of a formal generic type determines which types can be substituted as actual types, and on the other hand which properties may be assumed inside the generic class. For the kind `ANY_ASSIGN` any type for which the assignment operator is defined can be substituted as actual generic parameter, and inside the generic class the assignment operator can be used for vari-

ables of this formal generic type. Apart from the comparison and result operation, which are typically associated with assignability, no other operations are available for entities of the kind ANY_ASSIGN.

A generic instantiation of a generic class is obtained by substituting the formal generic parameters by actual ones:

```
IntStack : REGION CLASS = StackType
  SYNMODE  ElemType = Integer;
  SYN      Length   = 127:
END IntStack;
```

"IntStack" is a nongeneric class and can therefore be used as follows:

```
DCL MyStack IntStack();
MyStack.Push(10);
MyStack.Push(999);
MyStack.Pop();
```

As observed by Meyer [Mey 86] genericity is an alternative mechanism to obtain extensible and reusable program components. In strongly typed languages like CHILL it is a necessary complement to classes and inheritance. This has been shown in detail in [Mey 86].

4.4 Assertions

Extending the CHILL ASSERT action, Object CHILL supports the definition of *assertions* in the following way.

Invariants are boolean expressions connected to an Object CHILL class; they can be used to express the consistency of objects of that class in terms of internal data values.

Preconditions and *postconditions* are boolean expressions connected to class methods; preconditions can be used to define restrictions on parameter values (possibly depending on the object's state), and postconditions are a means to describe the result of methods and the final state of an obeject after completion of a method call.

Used properly, assertions provide an elegant way to express certain aspects of the semantics of a class and its methods [Mey 88].

In Object CHILL assertions are not only included as "semantic comments" but are optionally transformed into runtime checks:

- invariant and precondition are checked prior to execution of any method;
- postcondition and invariant are checked following the execution of any method.

Every violation of assertions will cause an exception to be raised.

Assertions are taken care of in connection with the inheritance mechanisms described in section 3.1 to make sure that semantic constraints connected to an ancestor class are adequately transferred to subclasses.

The program in the annex contains several examples for the assertion mechanism.

5 Summary

Object CHILL is described as an extension of CHILL to support object oriented programming of large and concurrent software systems. Object CHILL may be roughly characterized by the following "equation":

Object CHILL = CHILL
+ Classes
+ Inheritance
+ Genericity
+ Assertions

The design of Object CHILL has been influenced by Ada [Ref 83], C++ [ES 90] and Eiffel [Mey 88]. Other object oriented extensions of CHILL are proposed in [KKM 91; Sco 90]. Neither of these extensions includes all language elements mentioned above.

An Object CHILL programming environment, including a compiler, a source level debugger, a class library and a testsystem for objects, is currently being developed. Object CHILL is used for the implementation of switching software in an ISDN project [GW 91].

Acknowledgments

We are very grateful to G. Schulz for his collaboration during the first phase of the language development. We thank E.Ritter for very helpful comments on an earlier version of the paper. Many thanks are also due to the people who helped Object CHILL into real existence by building the first compiler: B. Dehm, S. Eichholz, W. Friedel, K. Gieselmann, J.Hauser, S.Itzenplitz, R.Müller, Z.Nagy, J.Peifer, E.Ritter, N.Schlager, G.Schramm, B.Schwarz, E.Sigl, M.Stadel, G.Walter, W.Wölfel, and W.Wolff. We are also very grateful to the first users of Object CHILL, of whom we may especially mention W.Günther and G.Wackerbarth, whose enthusiasm for the application of OOP to switching systems initiated the whole Object CHILL project.

References

AWY 89 Agha, G.; Wegner, P.; Yonezawa, A.: Proceedings of the ACM SIGPLAN Workshop on Object-Based Concurrent Programming. SIGPLAN Notices, 24, 4 (1989).

CW 85 Cardelli, L.; Wegner, P. : On Understanding Types, Data Abstraction, and Polymorphism. Computing Surveys, 17, 4 (1985) 471-522.

DSW 90 Dießl, G.; Schulz, G.; Winkler, J.F.H.: Object CHILL - The Road to Object-Oriented Programming with CHILL. In: Palma, A. (Ed.): Proc. 5th CHILL Conference, Rio 1990. Elsevier 1991,135..142.

ES 90 Ellis, Margaret A.; Stroustrup, Bjarne: The Annotated C++ Reference Manual. Addison-Wesley Publ. Conp., Reading etc., 1990.

GW 91 Günther, Wolfgang; Wackerbarth, Gerd: Vorgehen und Ergebnisse bei der objektorientierten Strukturierung von Vermittlungssoftware. Softwaretechnik-Trends 11,4 (1991) 44 .. 57 (in German).

Hoa 74 Hoare, C.A.R.: Monitors: An Operating System Structuring Concept. CACM 17,10 (1974) 549..557.

ITU 8x International Telecommunication Union (ed.): CCITT High Level Language (CHILL). Recommendation Z.200, Geneva 1980, 1984, 1988.

KKM 91 Maruyama, Katsumi; Watanabe, Nobuyuki; Koyanagi, Keiich; Kai, Toshihiro; Tomita, Shuji: A Concurrent Object-Oriented Switching Program in Chill. IEEE Communications Magazine, Jan. 1991, 60..68.

Mey 86 Meyer, B.: Genericity versus Inheritance. OOPSLA'86, SIGPLAN Notices, Vol. 21, No. 11 (1986) 391-405

Mey 88 Meyer, B.: Object-Oriented Software Construction. Prentice Hall, New York, 1988.

OOP 90 OOPSLA / ECOOP '90 Proceedings. SIGPLAN Notices 25,10 (1990) and Special Issue 1991.

Ref 83 Reference Manual for the Ada Programming Language ANSI / MIL-STD 1815 A, February 1983.

Rek 82 Rekdal, Kristen: CHILL-The Standard Language for Programming SPC Systems. IEEE-COM 30,6 (1982) 1318..1328.

Sco 90 Scortesse, A.: OO_CHILL: Integrating the Object Paradigm Into CHILL. In: Palma, A. (Ed.): Proc. 5th CHILL Conference, Rio 1990. Elsevier 1991, 127..133.

Weg 87 Wegner, P.: Dimensions of Object-Based Language Design. OOPSLA'87, SIGPLAN Notices, 22, 12 (1987) 168-182.

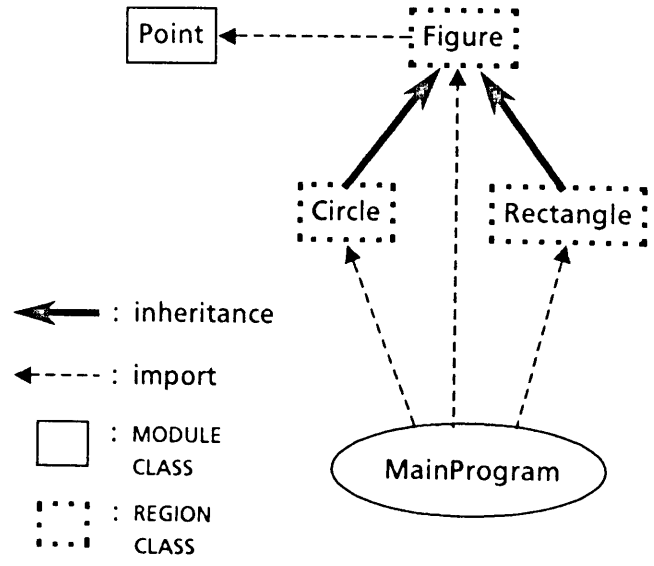
Win 84 Winkler, J.F.H.: The Realization of Data Abstractions in CHILL. Third CHILL Conference, Cambridge, September 23-28, 1984, 175-181

Win 86 Winkler, J.F.H.: The Programming Language CHILL. Automatisierungst. Praxis 28,5 (1986) 252..258 and 28,6(1986) 290..294 (in German).

Wir 88 Wirth, N.: Type Extensions. ACM TOPLAS 10, 2 (1988) 204-214.

Annex: Example

The example given below is part of the well known application to handle graphical objects like circles and triangles. Basic types for the definition of coordinates are included in the class `Point` for convenience and simplicity only; it might be reasonable to collect these declarations in another class. The example has the following structure:



Static Program Structure

Specification of the class Point

```
Point: MODULE CLASS
   GRANT set_pos, check, position_x, position_y,
         move, draw,
         coordinate_value, relative_value,
         lower_bound, upper_bound,
         lower_value, upper_value PREFIXED Point;

   SYN lower_bound = -100;
   SYN upper_bound = 100;
   SYN lower_value = -200;
   SYN upper_value = 200;

   SYNMODE coordinate_value =
           RANGE (lower_bound : upper_bound);
   SYNMODE relative_value =
           RANGE (lower_value : upper_value);

   DCL x coordinate_value;
   DCL y coordinate_value;

   Point : CONSTR (); END Point;
   Point : CONSTR (px, py coordinate_value);
      PRE px >= lower_bound AND
          px <= upper_bound AND
          py >= lower_bound AND
          py <= upper_bound;
   END Point;
```

```
   set_pos : CONSTR (px, py coordinate_value);
      PRE px >= lower_bound AND
          px <= upper_bound AND
          py >= lower_bound AND
          py <= upper_bound
   END set_pos;

   check : PROC (px, py coordinate_value)
           RETURNS (bool); END check;

   position_x : PROC ()
       RETURNS (coordinate_value); END;

   position_y : PROC ()
       RETURNS (coordinate_value); END;

   move : proc (px, py relative_value);
      PRE (x + px >= lower_bound) AND
          (x + px <= upper_bound) AND
          (y + py >= lower_bound) AND
          (y + py <= upper_bound);
   END move;

   draw : PROC (); END draw;
END Point;
```

Body of the class Point

```
Point: MODULE BODY

   Point : CONSTR ();
      x := 0;
      y := 0;
   END Point;

   Point : CONSTR (px, py coordinate_value);
      x := px;
      y := py;
  END Point;

  set_pos : CONSTR (px, py coordinate_value);
    x := px;
    y := py;
  END set_pos;

  check : PROC (px, py coordinate_value)
        RETURNS (bool); END;
    RESULT (px = x AND py = y);
  END check;

  position_x : PROC ()
        RETURNS (coordinate_value);
    RESULT x;
  END position_x;

  position_y : PROC ()
        RETURNS (coordinate_value);
    RESULT y;
  END position_y;

  move : proc (px, py relative_value);
    x := x + px;
    y := y + py;
  END move;

  draw : PROC ();
    /*  some implementations for draw  */
  END draw;
END Point
```

Specification of the class Figure

```
Figure: REGION CLASS VIRTUAL
  SEIZE CLASS Point;

  GRANT position, set_position,
        move, move_to, draw,
        coordinate_value, relative_value, length
        PREFIXED figure;

  SYN lower_bound = point!lower_bound;
  SYN upper_bound = point!upper_bound;

  SYNMODE coordinate_value =
            point!coordinate_value;

  SYNMODE relative_value =
            point!relative_value;

  SYNMODE length = RANGE(1:Point!upper_value);

  DCL p Point;

  Figure: CONSTR () END Figure;
     /*  default value constructor  */

  Figure: CONSTR (px, py coordinate_value)
     /*  constructor overloading !  */
     POST p.position_x() = px AND
          p.position_y() = py
  END Figure;

  position: PROC () RETURNS (Point) INLINE;
     /*  read current position  */
     result p;
  END position;

  set_position: PROC (pp Point);
     POST pp = p;
  END set_position;

  move_to: PROC (pp Point); END;

  move: PROC (px, py relative_value)
     /*  relative motion  */
     PRE p.position_x()+px >= lower_bound AND
         p.position_x()+px <= upper_bound AND
         p.position_y()+py >= lower_bound AND
         p.position_y()+py <= upper_bound;
     POST p.position_x() = OLD(p.position_x())+px
                 AND
          p.position_y() = OLD(p.position_y())+py;
  END move;

  draw: PROC () VIRTUAL; END draw;
     /* display figure: virtual method,
        to be implemented in subclasses  */

END Figure;
```

Body of the class Figure

```
Figure: REGION BODY

   Figure: CONSTR ();
      p.set_pos (0,0);
   END Figure;

   Figure: CONSTR (px, py coordinate_value);
      p.set_pos (px,py);
   END Figure;
```

```
   set_position: PROC (pp Point);
      p := pp;
   END set_position;

   move: PROC (px, py relative_value);
      p.move (px, py);
   END move;

  move_to: PROC (pp Point);
     p.set_pos (pp.position_x(),pp.position_y());
   END move_to;

     /*  no implementation for draw  */

END Figure;
```

Specification of the class Circle

```
Circle: REGION CLASS IS-A Figure
   GRANT diameter, set_diameter, draw
         PREFIXED Circle;

   DCL l_diameter length;

   Circle: CONSTR (px, py coordinate_value,
                   diam length);
      /*  constructor method  */
      POST p.position_x() = px AND
           p.position_y() = py AND
           l_diameter = diam;
   END Circle;

   diameter: PROC () RETURNS (length) INLINE;
      result l_diameter;
      /*  read current diameter  */
   END diameter;

   set_diameter: PROC (diam length);
      POST diameter() = diam;
   END set_diameter;

   draw: PROC () END draw;
      /*  display circle: no longer virtual  */

      /*  Point p inherited from Figure
          is used as center of the circle  */

  END Circle;
```

Body of the class Circle

```
  Circle: REGION BODY

     Circle: CONSTR (px, py coordinate_value,
                     diam   length);
        figure (px, py);
        /* call parent constructor */
        l_diameter := diam;
     END Circle;

     set_diameter: PROC (diam length);
        l_diameter := diam;
     END set_diameter;

     draw: PROC ();
        /*  some implementation  */
        ...
     END draw;

   END Circle;
```

Specification of the class Rectangle

```
Rectangle: REGION CLASS IS-A Figure;

   GRANT width, height,
         set_width, set_height,
         draw          PREFIXED Rectangle;

   Recangle: CONSTR(px, py coordinate_value,
                    width, height length);
      /*  constructor method  */
     PRE p.position_x() + px <= upper_bound AND
         p.position_x() + px >= lower_bound AND
         p.position_y() + py <= upper_bound AND
         p.position_y() + py >= lower_bound ;
    POST p.position_x() = px    AND
         p.position_y() = py    AND
         Rectangle!width() = width     AND
         Rectangle!height() = height;
  END Rectangle;

   width: PROC () RETURNS (length); END width;
      /*  read current horizontal side  */

   height: PROC () RETURNS (length); END height;
      /*  read current vertical side  */

   set_width: PROC (width length);
      POST Rectangle!width() = width;
   END set_width;

   set_height: PROC (heigth length) END;
      POST Rectangle!height() = height;
   END set_height;

   draw: PROC () END draw;
      /* display rectangle: no longer virtual */

   DCL top_right_corner Point;

END Rectangle;
```

Body of the class Rectangle

```
Rectangle: REGION BODY

   Rectangle: CONSTR(px, py coordinate_value,
                     width, height length);
      Figure (px, py);
      /* call parent constructor */
      top_right_corner.set_pos
         (p.position_x() + width,
          p.position_y() + height);
   END Rectangle;

   width: PROC () RETURNS (length);
      RESULT ABS(top_right_corner.position_x() -
                 p.position_x());
   END width;

   height: PROC () RETURNS (length);
      RESULT ABS(top_right_corner.position_y() -
                 p.position_y());
   END height;

   set_width: PROC (width length);
      top_right_corner.setpos
        (p.position_x()+width, p.position_y());
   END set_width;
```

```
   set_height: PROC (height length);
      top_right_corner.setpos
         (p.position_x(), p.position_y()+height);
   END set_height;

   draw: PROC ();
      /*  some implementation  */
   ...
   END draw;

END Rectangle;
```

MainProgram

```
MainProgram: MODULE

   SEIZE CLASS Figure;
   SEIZE CLASS Circle;
   SEIZE CLASS Rectangle;

   DCL my_circle  Circle (1,2,20);
   DCL my_rectangle  Rectangle (0,0,10,30);
    /* create circle and rectangle
       objects with parameters given    */

   DCL my_ref  REF Figure;
   DCL area  INT;

   my_circle.draw ();
     /* display object my_circle */
   my_circle.set_position (0,0);
   my_circle.set_diameter (10);
     /* move and resize object my_circle */
   my_circle.draw ();

   my_ref := ADDR (my_circle);
     /* my_ref refers to object my_circle */
   my_ref->.draw ();
     /* display object referenced by
        my_ref (here: my_circle)  */

   my_ref := ADDR (my_rectangle);
   my_ref->.draw ();
     /* display object referenced by
        my_ref (here: my_rectangle)  */

   area := my_rectangle.width()
           * my_rectangle.height();

 END MainProgram;
```

COMMUNICATIONS

20th ANNUAL
COMPUTER SCIENCE CONFERENCE
March 3-5, 1992

CSC '92

PROCEEDINGS

KANSAS CITY CONVENTION CENTER
KANSAS CITY, MISSOURI

1992 ACM COMPUTER SCIENCE CONFERENCE
March 3 - 5, 1992
Kansas City Convention Center
Kansas City, Missouri

COMMUNICATIONS

PROCEEDINGS

EDITORS

Jagan P. Agrawal, *Program Co-Chair*
University of Missouri-Kansas City

Vijay Kumar, *Proceedings Chair*
University of Missouri-Kansas City

Virgil Wallentine, *Program Co-Chair*
Kansas State University

ASSOCIATION FOR COMPUTING MACHINERY, INC.
1515 Broadway
New York, NY 10036-9998

The Association for Computing Machinery, Inc.
1515 Broadway
New York, NY 10036-9998

ISBN 0-89791-472-4

Additional copies may be ordered prepaid from:

ACM Order Department
P. O. Box 64145
Baltimore, MD 21264

ACM Order Number: 404920