Emerald: A General-Purpose Programming Language

Rajendra K. Raj, Ewan Tempero, Henry M. Levy
Department of Computer Science and Engineering, University of Washington, Seattle, WA 98195
[rkr,ewan,levy]@cs.washington.edu

Andrew P. Black
Digital Equipment Corporation, Cambridge Research Laboratory
One Kendall Square, Bldg. 700, Cambridge, MA 02139
black@crl.dec.com

Norman C. Hutchinson
Department of Computer Science, University of Arizona, Tucson, AZ 85721
norm@cs.arizona.edu

Eric Jul
DIKU, University of Copenhagen, Universitetsparken 1, DK-2100, Copenhagen, Denmark
eric@diku.dk

SUMMARY

Emerald is a *general-purpose* language with aspects of traditional object-oriented languages, such as Smalltalk, and abstract data type languages, such as Modula-2 and Ada. It is strongly-typed with a non-traditional object model and type system that emphasize abstract types, allow separation of typing and implementation, and provide the flexibility of polymorphism and subtyping with compile-time checking. This paper describes the Emerald language and its programming methodology. We give examples that demonstrate Emerald's features, and compare and contrast the Emerald approach to programming with the approaches used in other similar languages.

KEYWORDS: Programming languages, Programming methodology, Object-oriented programming, Abstract data types, Inheritance, Object-based concurrency

1. INTRODUCTION

Emerald[1, 2] is a strongly-typed programming language that supports an atypical variant of the object-oriented paradigm. Although originally developed to simplify the construction of efficient distributed applications,[3, 4] Emerald provides a general-purpose programming model. This paper describes the main features of Emerald, comparing and contrasting them with similar features of other languages.

Although Emerald may be bracketed with module- or object-based languages, it differs in several ways from other such languages. Unlike Ada[5] or Modula-2,[6] the sole unit of programming is the *object*, which behaves like a dynamic instance of a Modula-2 *module* or Ada *package*. Unlike C++[7] or Smalltalk,[8] Emerald has no notion of *class*; Emerald provides *object constructors* for run-time creation of objects and *abstract typing* for object classification and comparison. Access to Emerald objects is provided via *operations* that are invoked by other objects.

Past experience has suggested that language designers must choose between the security and efficiency of static typing and the flexibility of dynamic typing. For example, Smalltalk and Self [9] emphasize flexibility by delaying type checking until run-time, while Modula-2 and Ada are statically typed and preclude the safe interaction of existing programs with entities of a new type. Emerald provides both static typing and flexibility using a simple and efficiently-implemented notion of *type* that allows compile-time checking, polymorphism, encapsulated types, and subtyping. A type in Emerald is a collection of operations and their signatures; significantly, a type is itself an object. The Emerald language includes block structure and nesting as found in Simula-67[10] and Beta[11] but noticeably missing from Smalltalk.

Emerald supports the programming of distributed applications by making the invocation of objects independent of their location. Such location independent object invocation allows the application programmer to treat the distributed system as a single machine. Thus, the Emerald programming model is for the most part non-distributed. This paper focuses on Emerald's programming model; we will mention Emerald's distributed upbringing only when motivating the design of some of its features. Several aspects of the distributed nature of Emerald are discussed in the companion paper in this publication.[12]

This paper discusses, with examples, the main features of Emerald that help in supporting general-purpose programming. In the next section we describe the Emerald notion of objects, and how they are constructed and used. Section 3 discusses the Emerald abstract type concept and its usage. We then examine the Emerald approach to object-based concurrency, and outline other interesting features of Emerald. Finally, we indicate the current status of our work and summarize its major contributions.

2. EMERALD OBJECTS

The Emerald *object* is an abstraction for the notions of data and type. Emerald objects consist of private data, private operations, and public operations. An object is entirely responsible for its own behavior and so must contain everything necessary to support that behavior; we call this property *autonomy*. Object autonomy, originally motivated by the need for object mobility in a distributed environment, also enforces the modular approach espoused in software engineering.[13] An object may be accessed only by its public operations; this is done in Emerald by the *invocation*. An Emerald object may invoke some operation defined in another object, passing arguments to the invocation and receiving results. Invocations are semantically similar to procedure calls in traditional languages and to messages in Smalltalk.

Object Creation

In most object-based languages, an object is created by invoking an operation on a *class* object; Smalltalk and Eiffel[14] are examples of this model. The class object has multiple functions: it defines the structure, interface, and behavior of all its instances; it defines the place of those instances in the graph of all classes; and it responds to *new* invocations to create new instances.[15] In prototype-based languages such as Self, objects are typically cloned from existing objects that act as templates.

In contrast, object creation in Emerald is done via the *object constructor*, an Emerald expression that, when evaluated, creates a new object. This expression defines the object's representation, and its public and private operations. The syntax of an object constructor is:

```
object anObject
  % private state declarations
  % operation declarations
end anObject
```

where *anObject* is a local name whose scope is confined to the object constructor itself.

Figure 1 illustrates the creation of an object with an example taken from the Emerald implementation of the Emerald compiler. As part of its execution, the compiler creates a parse tree whose nodes represent the various constructs in an Emerald program. The compiler creates such nodes as it discovers each construct, annotates them with attributes relevant to the construct, and later generates code associated with their semantics. In this example, the object represents an integer literal whose only attribute is the value of the literal. The object constructor has the name *IntegerLiteral*, and the object it creates has public operations *getValue*, *setValue*, and *generate*. Its private state consists of a single integer variable named *value* that is initialized to *val*, which may be regarded as a literal constant for the present. The monitor construct, to be discussed in the concurrency section, is used to serialize concurrent invocations of the object.

When this object constructor expression is evaluated (i.e., executed), it creates the described object, which is named by the identifier *anIntegerNode*. By virtue of the **const** binding, *anIntegerNode* always refers to this object, whose state, however, may be changed by invocations of its *setValue* operation. Objects may also be declared **immutable**, which asserts that their abstract state does not change. The compiler may use such information for optimization, but the language does not attempt to restrict the operations that can be performed on the concrete state. This allows an object to reorganize its concrete state—for example, to make future calls more efficient—so long as doing so does not change the result or effect of the future calls. The **export** clause lists the operations that can be called from outside the object; invocation of one of these operations is the only way in which a client can examine or modify the object's state. Thus, objects

```
const anIntegerNode ←
    object IntegerLiteral
        export getValue, setValue, generate
        monitor
            var value : Integer ← val

            operation getValue → [v : Integer]
                v ← value
            end getValue

            operation setValue[v : Integer]
                value ← v
            end setValue

            operation generate[stream : OutStream]
                stream.printf[“LDI %d\n”, value]
            end generate
        end monitor
    end IntegerLiteral
```

Figure 1: Example use of an Object Constructor

```
% Assume count and nodes have been suitably declared
count ← 5
loop
    exit when count = 0
    count ← count − 1
    nodes(count) ←
        object IntegerLiteral
        % as in Figure 1
        end IntegerLiteral
end loop
```

Figure 2: Creating several integer node objects

in Emerald are fully encapsulated, unlike those of Simula; this makes an Emerald object very similar to a *module* in Modula-2 or a *package* in Ada.

Object constructors perform the following subset of the functions carried out by Smalltalk classes:[15]

1. they generate new objects,
2. they describe the representation of objects, and
3. they define the code that implements operations (methods in Smalltalk terminology).

The essential point to note is that the object constructor is more primitive than the class; the functions of a class can be provided in Emerald by using object constructors in combination with other language features.

The control abstractions available in ordinary languages permit expressions to be executed as desired, i.e., never, once, twice, or as many times as necessary. By treating object constructors as expressions and providing standard control structures, Emerald allows objects to be created as required. Multiple similar

```
const IntegerNodeCreator ←
    immutable object INC
        export new
        const IntegerNodeType ←
            type INType
                function getValue → [Integer]
                operation setValue[Integer]
                operation generate[OutStream]
            end INType
        operation new[val : Integer] → [aNode : IntegerNodeType]
            aNode ←
                object IntegerLiteral
                    % as in Figure 1
                end IntegerLiteral
        end new
    end INC
```

Figure 3: An object that creates integer nodes

objects may be created by placing the constructor in a repetitive construct such as a loop or the body of an operation of another object. Figure 2 illustrates how a loop may be used to fill an array with integer literals. Figure 3 presents an example of the usefulness of placing an object constructor within an operation. In this example, the object named *IntegerNodeCreator* exports an operation *new* that returns the object created by executing the object constructor *IntegerLiteral*. This object constructor is the same as that in Figure 1, but the initialization of its *value* variable is now determined by the argument *val* of *new*. Thus, every time *new* is invoked on *IntegerNodeCreator*, a new node is created. In other words, the *IntegerNodeCreator* object performs the instance creation function of a class.

There is nothing magical about the use of the operation name *new* — any other name would be equally valid. Note that *IntegerNodeCreator* has been declared *immutable*, an assertion that the object's local state, which in this case consists only of the object named by *IntegerNodeType*, will never change. Note also that *IntegerNodeType* actually names a *type* object; types are discussed in the next section.

The approach used in Emerald to create class-like objects can be compared and contrasted with the approach used in a more traditional language such as Ada. An Ada implementation of *IntegerNodeCreator* could be very similar to the Emerald version, but there will be two notable differences. First, because Ada is not object-oriented, the data to be manipulated must be passed to each function; second, the Ada package that implements the creator defines both the operations that create instances and the operations that manipulate those instances.

One-of-a-kind Objects

The absence of a class mechanism from Emerald illustrates the flexibility of the object constructor. When an object constructor is executed only once, a "one-of-a-kind" object is created. To our knowledge, the only other languages that support such a feature are prototype-based languages such as Self.[9]

```
const idServer ←
    object IDS
        export getNextId
        monitor
            var nextId : Integer ← 0

            operation getNextId → [newId : Integer]
                newId ← nextId
                nextId ← nextId + 1
            end getNextId
        end monitor
    end IDS
```

Figure 4: A Unique Id Server

Class-based languages are suited to situations where many objects with the same behavior are needed, but they are unnecessarily verbose when only one instance of an object is needed. For example, consider the Boolean objects *true* and *false*. In Smalltalk, where one-of-a-kind objects cannot be directly defined, one needs to define a class for each object and then instantiate it. Two classes, *True* and *False*, are defined as subclasses of *Boolean*, with the objects *true* and *false* as their only instances. However, this arrangement does not guarantee that only a single instance of class *True* is created; this is rather awkward in Smalltalk. For example, this can be done by defining a class for the required object, along with a tailored *new* class method that allows only a single instantiation, and then instantiating the required object. In languages such as Simula, where not all entities are objects and there is no support for metaclasses, one-of-a-kind objects cannot be created at all.

In practice, it is not just primitive types such as Boolean that use one-of-a-kind objects. Figure 4 depicts the creation of an object that provides unique identifiers (integers) used in naming the objects produced by the Emerald compiler. Different invocations of the *nextId* operation return different identifiers. Only one such server is required in the compilation environment. Indeed, having more than one server managing the same identifier space would be disastrous. As another example, in a typical workstation there is exactly one mouse object, exactly one keyboard object, and perhaps only one console window object. The class abstraction is redundant for any of these situations when exactly one object with a given behavior is needed. The more primitive Emerald object constructor provides programmers with the flexibility to create exactly as many objects as are required in each situation.

No Metaclasses

The fact that Emerald objects are modeled as containing their own operations removes the need for the multi-level instance/class/metaclass hierarchy found in Smalltalk. Figure 5 illustrates the difference. In class-based languages, each object depends upon its class object to describe its structure and behavior. In the Smalltalk hierarchy, each *idServer* object is described by its class *IdServer*, which is described by its class (metaclass of class *IdServer*), which in turn is described by its class (*Metaclass*).

Object constructors provide an ability to create not just class-like objects (creators), but also creators of

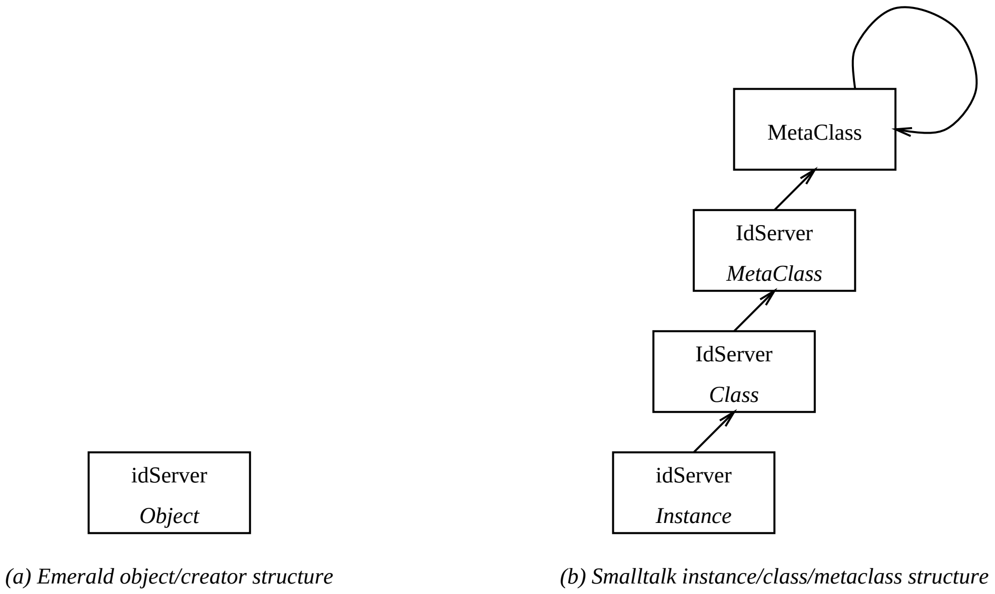

idServer
Object
(a) Emerald object/creator structure
MetaClass
IdServer
MetaClass
IdServer
Class
idServer
Instance
(b) Smalltalk instance/class/metaclass structure

Figure 5: "Metaclass hierarchies" in Emerald and Smalltalk

creators, creators of creators of creators, and so on; multiple levels are indeed useful, as will be illustrated in the section on Polymorphism. In contrast, the metaclass concept and the termination of metaregress in Smalltalk have been found to be one of the biggest hurdles to be overcome in learning Smalltalk.[16] Like Self, Emerald avoids this metaregress by making each object completely self-describing.

3. ABSTRACT TYPES

The notion of type serves several purposes in conventional programming languages.[17, 18] Types facilitate the representation independence required for portability and maintainability of programs. Traditionally, types describe implementations, and the creation of data instances is controlled by their type. When types describe implementation, having type information available at compile time also permits the generation of better code. Finally, types enable the early detection of errors, and permit more meaningful error reporting.

In Emerald, these functions are performed by specialized constructs. The *object* provides representation independence. The object constructor provides for object creation and encapsulates implementation information. What in other languages are type-dependent optimizations are performed by the Emerald compiler based on how objects are used. Since it is freed from these other concerns, the Emerald type system can concentrate on the single task of object classification, thus preventing incorrect usage.

Emerald's type system reflects its design goal of being used for the development of software in constantly running distributed systems. In such systems, objects may be developed and implemented separately and differently on different machines at different times. Furthermore, to accommodate situations where the types of the objects to be bound to an identifier are not known at compile-time, the Emerald type system does not distinguish between objects based on their implementation.

```
const IntegerNodeType ←
    type INType
        function getValue → [Integer]
        operation setValue[Integer]
        operation generate[OutStream]
    end INTType
```

Figure 6: An example of a type object

Emerald allows normal type checking to be performed entirely at compile-time. The only occasion on which types must be checked at run-time is when the programmer explicitly requests that an object should be treated as if it had a type that is stronger than the type that could be attributed to it at compile time. For example, a programmer could state that an object retrieved from a list of arbitrary objects is an integer; this claim can only be verified at run-time.

Type Definition

An Emerald type is a collection of operation *signatures*, where a signature consists of the operation name and the types of the operation's arguments and results. Note that a type contains no information about implementation; it describes only an interface. For this reason, the term *abstract type* is often used to refer to an Emerald type. Abstract types allow for the complete separation of typing and implementation, and mean that multiple implementations of the same type can co-exist and interoperate.

Each identifier in an Emerald program—this includes the names of constants, variables, arguments and results—has a declared type, which must be evaluable at compile time; this is called the *syntactic type* of the name. Type checking is the process of ensuring that the object to which a name is bound always satisfies the syntactic type of the name. For example, the argument declaration for *val* in the *new* operation in Figure 3, and the variable declaration for *value* in the monitor in Figure 1, cause both of these names to be given the syntactic type *Integer*. An attempt to assign a string to *value*, or to pass a file as an argument to *new*, is detected as a type error at compile time.

Each Emerald type is an object and as such can be manipulated by the ordinary facilities of the Emerald language, such as assignment, constant binding, and parameter passing. Type objects may be passed as parameters to implement polymorphism or inspected at run-time to implement run-time type checking.

An example of an Emerald type is *IntegerNodeType*, which has been extracted from Figure 3 and shown in Figure 6. This code fragment binds the name *IntegerNodeType* to the result of evaluating a type constructor, which is delimited by **type** *INType* ... **end** *INType*. The type constructor generates an Emerald object whose type is *AbstractType*, that is, a type object. Objects of this type must export the three operations *getValue*, *setValue*, and *generate* with the indicated argument and result types. *IntegerNodeType*, being an Emerald object, may be passed as a parameter, assigned to a variable, or invoked at run-time.

Type Conformity

The idea of an object "satisfying" the syntactic type of the name to which it is bound, as discussed above, is made precise in Emerald by the *conformity* relation. If an object O is bound to a name I, then the type of O must conform to the syntactic type of I.

The motivation behind Emerald's definition of conformity is the notion of substitutability. Informally, a type S conforms to a type T (written S o> T) if an object of type S can always be *substituted* for one of type T, that is, the object of type S can always be used where one of type T is expected. For S to be substitutable for T in this way requires that:

1. S provides at least the operations of T (S may have more operations).
2. For each operation in T, the corresponding operation in S has the same number of arguments and results.
3. The types of the results of S's operations conform to the types of the results of T's operations.
4. The types of the arguments of T's operations conform to the types of the arguments of S's operations. (Notice the reversal in the order of conformity for arguments.)

Properties 1 and 2 are fairly straightforward and do not need further discussion. Why conformity needs property 3 is illustrated by these types:

```
type JunkDeliverer
        operation Deliver → [Any]
end JunkDeliverer
type PizzaDeliverer
        operation Deliver → [Pizza]
end PizzaDeliverer
```

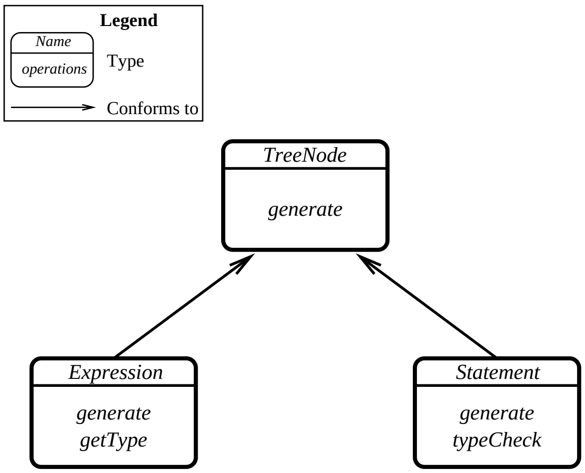
The keyword *Any* denotes the type with no operations; any object may be assigned to an identifier that has type *Any*. These types describe delivery services that may deliver anything and those that deliver only pizzas. Whatever one expects to do with what is delivered by a *JunkDeliverer* can also be done with the pizza delivered by a *PizzaDeliverer*. That is, a *PizzaDeliverer* can be substituted for a *JunkDeliverer*, but not vice versa. Formally stated, *PizzaDeliverer* conforms to *JunkDeliverer*, but *JunkDeliverer* does not conform to *PizzaDeliverer*.

As motivation for property 4, consider the following types:

```
type JunkHeap
        operation Deposit[Any]
end JunkHeap
type Bank
        operation Deposit[Money]
end Bank
```

```
const TreeNode ←
   type TN
      operation generate[OutStream]
   end TN
const Expression ←
   type E
      function getType → [Expression]
      operation generate[OutStream]
   end E
const Statement ←
   type S
      operation typeCheck
      operation generate[OutStream]
   end S
```

(a) Type definitions



(b) Relationship between types

Figure 7: Some compiler types

These types define entities where arbitrary objects and money can be deposited (without retrieval!) respectively. Intuitively, one knows that where one can deposit anything, one should be able to deposit money, and conversely, where one deposits only money, one cannot possibly deposit anything else. That is, *JunkHeap* can be substituted for *Bank* but not vice versa. More formally, *JunkHeap* conforms to *Bank*, but *Bank* does not conform to *JunkHeap*.

The above properties 1–4 characterize a number of relations on types. Conformity is defined as the *largest* relation that satisfies these rules.[2] The object type matching rules in Modula-3,[19] for example, define a smaller relation. Modula-3 requires that the argument and result types of operations be identical, not merely that they conform appropriately.

Property 4 is known as contravariance. In mathematics, a function from A to B is also considered to be a function from A' to B' if A' is a subset of A and B is a subset of B'; this is the contravariance in the argument of the function space operator.[20] In programming, contravariance means that an operation on objects of a given type should also work on objects of subtypes of that type. Applied to the *deposit* operation, contravariance results in *Bank* not conforming to *JunkHeap*. Although such a consequence is sometimes considered to be a limitation of conformity,[21, 22] property 4 is included in Emerald not just to make the type system safe and consistent—although this would be reason enough—but also because contravariance helps to enforce correct program design. In other words, contravariance is a feature rather than a limitation.

The utility of contravariance may be illustrated by additional objects from the Emerald compiler. Nodes corresponding to integer literals were discussed earlier; additional nodes of other types are used to represent declarations, control statements, and expressions. The type *TreeNode* characterizes those properties that are common to all the elements of the language; consequently the operations in type *TreeNode* must be exported by all nodes. Both *Expression* nodes and *Statement* nodes export more operations than do *TreeNodes*, but each has operations that are different from those of the others; these types and the relationships between them are depicted in Figure 7. Type *Statement* does not conform to type *Expression* because *Statement* does not implement the *getType* operation, and *Expression* does not conform to *Statement* because *Expression* does not implement the *typeCheck* operation. However, they both conform to *TreeNode*.

Now consider the objects *codeImprover* and *expressionImprover*:

```
const codeImprover ←
  object CI
    operation optimize[root: TreeNode] → [result: TreeNode]
    ...
  end CI
const expressionImprover ←
  object EI
    operation optimize[root: Expression] → [result: Expression]
    ...
    operation factor[root: Expression] → [result: Expression]
    ...
  end EI
```

Should the type of *expressionImprover* conform to that of *codeImprover*? The former clearly has more operations, and this could be reason enough to declare that it conforms. However, *expressionImprover* is clearly not substitutable for *codeImprover* in every context. An invocation of *codeImprover.optimize* may legally be presented with a *TreeNode* as an argument, but an attempt to present the same argument to *expressionImprover.optimize* is a type error. This is why Emerald's conformity rules clearly state that *expressionImprover* does *not* conform to *codeImprover*.

Before leaving this example, let us look a little more closely at what would happen if Emerald were more lenient in its type checking. Then, it would be possible for *expressionImprover.optimize* to be invoked with a *TreeNode* object as its argument, which would then be bound to the name *root* within the *optimize* operation, even though the syntactic type of *root* is *Expression*. Now there are two situations to consider within the body of *expressionImprover.optimize* (denoted by the ellipsis above). If the operation body invokes only *generate* operations on *root* and never *getType* operations, the body would execute successfully. If the operation body *does* invoke the *getType* operation on *root*, the invocation would fail because *getType* is not implemented on *treeNodes*.

It is of course exactly this sort of run-time failure that Emerald's type system prevents. At first glance, it may seem that the cost of this security is that a legal and useful program—the first case in which *getType* is never invoked—cannot be written in Emerald. But a little thought shows that this is not so. If the body of *expressionImprover.optimize* never invokes *getType* on its argument, then the argument should not be declared to be of type *Expression*: it need only be of type *TreeNode*.

```
const expressionImprover ←
  object EI
    operation optimize[root: TreeNode] → [result: TreeNode]
    ...
    operation factor[root: Expression] → [result: Expression]
    ...
  end EI
```

With this change to the definition of *expressionImprover*, the type of *expressionImprover* conforms to that of *codeImprover*. The type system now permits *expressionImprover* to be substituted for *codeImprover*; the "legal and useful" program mentioned above *can* be written in Emerald.

The lesson here is that the Emerald programmer should strive to characterize argument and result types as accurately as possible, listing exactly what is expected of arguments, and describing results as fully as possible. This is similar to the discipline that must be imposed when writing a program using formal specifications: the declared precondition of an procedure should characterize the properties of the arguments that are *required* for the procedure to work. The precondition should *not* characterize everything that happens to be true of the argument in all of the uses of the operation that have been conceived of to date, for doing so severely limits the reusability of the procedure.

This discipline is dubbed as the use of *best-fitting* types. Best-fitting types provide good documentation and encourage good design, as well as permitting the maximum reusability of code. There is a cost, however: it is harder to reimplement the same operation using a different algorithm. This is because the new algorithm may want to use some operation of an argument that is not possessed by the syntactic type of the argument, even though it is available on the actual argument in all real invocations.

Using best-fitting types works only because argument and result conformity can extend to an arbitrary depth. This makes the Emerald's programming style different from that used in languages without conformity, where perfect matching of argument and result types is required. Here, the programmer has to decide *a priori* on "envelope types" that are weak enough to work correctly in all applications, current and future.

Implicit versus explicit conformity. Implicit conformity can sometimes lead to "mistaken" type-matches. In Figure 8, since *insert* and *remove* have identical signatures in all three objects, Emerald would regard *Stack*, *Queue*, and *Stack2* as having the same type. This allows the unintended use of the *Queue* object as a *Stack*. On the other hand, because Emerald is an open system where objects can be compiled and added at run-time, *Stack* can be replaced by *Stack2* (which might be a newer version of *Stack*). In order to allow the substitution of *Stack2* for *Stack*, but prevent the substitution of *Queue* for *Stack*, the type system would either require the programmer to explicitly declare the conformity relation between types, or need information about the semantics of the operations.

The subtyping notion of Trellis[23] is similar to that of Emerald conformity, but any subtype relationship between types has to be explicitly declared. The use of such explicit declaration prevents meaningless substitutions of objects. In the above example, the Trellis programmer would declare *Stack2* as a subtype of *Stack* but not establish any subtype relationship between *Queue* and either *Stack* or *Stack2*. The price paid in Trellis is that a type *Stack3* defined elsewhere in the type hierarchy cannot be a subtype of *Stack* unless that relationship is explicitly stated. Moreover, *Stack* cannot also be a subtype of *Stack2* or *Stack3*. Eiffel[14] also requires an explicit declaration of type conformance (although the use of covariance in Eiffel's notion of conformance has been found to make the type system unsafe[24]). In contrast, the implicit nature of Emerald conformity allows the types of these three stack objects conform as desired.

Languages such as Algol 68 that use structural equivalence and conventional types systems have similar

```
const Stack ←
   object S
      export insert, remove
      operation insert[Element] → [ ]
         % Suitable body
      operation remove → [Element]
         % Suitable body
   end S
const Queue ←
   object Q
      export insert, remove
      operation insert[Element] → [ ]
         % Suitable body
      operation remove → [Element]
         % Suitable body
   end Q
const Stack2 ←
   object S2
      export insert, remove
      operation insert[Element] → [ ]
         % Suitable body
      operation remove → [Element]
         % Suitable body
   end S2
```

Figure 8: Three objects with identical types

problems. For example, stacks and queues, implemented using the same singly-linked list and the field names, have the same structure, and hence the same type. Modula-3,[19] which uses structure equivalence, recognizes this problem and provides the ability to *brand* types in a unique way. A branded type cannot match another type, no matter what its structure is, thus providing a name equivalence option in a system based primarily on structure equivalence. In Emerald, "mistaken" type-matches may be reduced by including operations that are specific to the type defined. For example, a *Queue* object (as in Figure 8) supplemented by a *queueLength* operation cannot be mistaken for a stack.

Conformity versus inheritance. It is useful to contrast the Emerald notion of type conformity with that of inheritance provided in other object-oriented languages. In short, conformity is a mechanism for object substitution, and inheritance is a mechanism for code (and representation) sharing among objects. Although conformity and inheritance are fairly distinct concepts, many object-oriented languages such as C++ have not distinguished clearly between the two concepts, and the use of the same language structure for providing both conformity and inheritance has caused considerable confusion.

Emerald supports type conformity (and hence object substitution), but it does not support inheritance: code cannot be shared among object implementations. The absence of code sharing is due to Emerald's use in distributed systems, where object mobility is facilitated by self-contained objects and hampered by dependencies on other objects. Since code sharing promotes software reuse, the lack of inheritance is sometimes considered to be a deficiency of Emerald.

However, several aspects of inheritance actually impede software reuse. Inheritance compromises object encapsulation to varying degrees.[25] The use of a single explicit structure for both object substitution and code sharing among objects is often restrictive and sometimes unsafe. For example, Trellis restricts code sharing to its conformity hierarchy, and Eiffel forces inheritance and conformance to be the same with disastrous consequences on type safety.[24] Inheritance has other conceptual problems. It violates the principle of locality,[26] thus preventing the use of local reasoning in understanding and implementing an object. Adequate control is not provided over the granularity of code sharing; the basic inheritance paradigm requires inheriting objects to share *all* their ancestors' implementations. The need for explicit removal via overriding to prevent undesired parts from being shared shows that there is no easy way of inheriting only the needed parts of the ancestors' implementations.

To provide inheritance-like software reuse, we have prototyped an Emerald extension called Jade[27] in which object substitution is provided via conformity, and code sharing among objects via *composition*. The basic building blocks in Jade are self-describing code components, such as operations and objects, which may be combined to form larger objects. Since different objects may use the same components, this leads to conceptual sharing of code among objects but without several shortcomings of inheritance. The noteworthy point about Jade is the complete orthogonality of the code sharing (composition) and object substitution (conformity) mechanisms. A detailed discussion of the Jade language and environment is beyond the scope of this paper, and may be found elsewhere.[27, 28]

Relationship between Types and Objects

Each Emerald object may belong to several types because an object O belongs to a type T when

typeof O ⊳ T

The application of **typeof** to an object returns its *maximal* type, that is, the largest Emerald type that the object can belong to. Because this permits objects to be classified by their types, it now becomes possible to relate implementations and types via conformity. The *anIntegerNode* of Figure 1 names an object whose type conforms to the type definition given in Figure 6; we can thus write

typeof *anIntegerNode* ⊳ *IntegerNodeType*

A type is any object whose type conforms to:

```
immutable type AbstractType
   function getSignature → [Signature]
end AbstractType
```

In other words, any object that exports a *getSignature* operation that returns a *Signature* is a type. Objects of type *Signature* are created by the type constructor syntax (**type** x ... **end** x). A *Signature* is a built-in implementation of an *AbstractType* that can be generated only by the compiler. A signature object

```
const IntegerNode ←
    immutable object INC
        export getSignature, new
        const IntegerNodeType ←
            type INType
                function getValue → [Integer]
                operation setValue[Integer]
                operation generate[OutStream]
            end INType
        function getSignature → [sig : Signature]
            sig ← IntegerNodeType
        end getSignature
        operation new[val : Integer] → [aNode : IntegerNodeType]
            aNode ←
                object IntegerLiteral
                    % as in Figure 1
                end IntegerLiteral
        end new
    end INC
```

Figure 9: An integer node creator with getSignature

exports a *getSignature* operation (returning itself) so it conforms to *AbstractType*. It also exports several secret operations that enable the Emerald implementation to determine the operations provided by the type, and the signatures of these operations. Because the names of these operations are secret, no programmer defined objects will ever have types that conform to *Signature*; all signature objects must stem from a type constructor expression in some Emerald program. Consequently, we can guarantee that the type checker is able to get adequate and consistent information about a type.

The type value denoted by a type object is the result of its *getSignature* function. Consider the *IntegerNode* in Figure 9. This object was constructed by the addition of a *getSignature* function to the *IntegerNodeCreator* of Figure 3. The code for *IntegerNode* reveals that **typeof** IntegerNode is:

```
type IntegerNodeCreatorType
    function getSignature → [Signature]
    operation new[Integer] → [IntegerNodeType]
end IntegerNodeCreatorType
```

When *IntegerNode* is treated as an object, the operations that can be invoked on it are described by *IntegerNodeCreatorType*. However, when *IntegerNode* is treated as a type, its value is *IntegerNodeType*, the result of the *getSignature* function.

Separation of Type and Implementation

The three features discussed above—object constructors for object creation, abstract types to characterize interfaces, and conformity for type comparison—allow Emerald to elegantly separate implementations from types. Emerald allows several different implementations to be used for the same type within a single program.

```
const anotherIdServer ←
    object IDS
        export getNextId
        monitor
            operation getNextId → [newId : Integer]
                var inF : InStream
                var outF : OutStream
                inF ← InStream.FromUnix["/usr/emerald/ID"]
                newId ← inF.getInteger
                inF.close
                outF ← OutStream.ToUnix["/usr/emerald/ID"]
                outF.putInteger[newId + 1]
                outF.close
            end getNextId
        end monitor
    end IDS
```

Figure 10: An alternative implementation of an Id Server

Much of the flexibility found in untyped languages such as Smalltalk is thus available within the framework of the strongly-typed Emerald language. Not only does Emerald's separation of types and implementations simplify separate compilation of programs, but it also enables newly defined objects to be added to a running system, and take the place of previously defined objects, so long as they have conforming types and semantics.

For example, the reliability of the *idServer* defined in Figure 4 may be enhanced by implementing a new version, shown in Figure 10, that stores its state in a file in order to survive crashes. The type of this new server conforms to the type of the original and so may be used anywhere the original was expected. Note that this substitution can be made into an application that is already executing. Such flexibility is not normally available in strongly-typed languages.

It is worth noting that the clean separation of type and implementation found in Emerald is not possible in languages whose objects are not so strongly encapsulated. In CLU or C++, for example, the implementation of an operation in a class is allowed to access the internals of all the objects that are the same type as the receiver: these languages assume that 'same type' implies 'same class'. In Emerald, as in Smalltalk, the implementation of each operation on an object has access to the internals of that object alone.

In Modula-2, separation between abstract typing and implementation is not provided because `DEFINITION` and `IMPLEMENTATION MODULE`s are actually treated as two parts of one module.[6] The coupling of the definition and implementation modules is done during linking, which precludes multiple implementations at run-time. In Ada, each specification corresponds to an implementation in the library. When a different implementation for a specification is needed, the program needs to be recompiled, substituting the new one for the old one, and then relinking the new implementation. Thus, Ada does not allow multiple implementations of the same abstract type to coexist.

Object-oriented languages such as C++ and Smalltalk support a programming style that allows multiple implementations of the same abstraction. This is done via the use of *abstract* classes, which describe only object interfaces, but may be extended into *concrete* subclasses for instantiating multiple implementations. There are some obvious similarities between such abstract classes and Emerald's abstract types, but there are important differences. First, the former represents a conventional use of the class system, while the latter is a language feature. Second, abstract classes permit the definition of default behavior. Although such definitions are occasionally useful, they violate the complete separation of a type from its implementations, as well as the separation of multiple implementations. Finally, the use of abstract classes requires each concrete implementation to be explicitly declared a subclass of the abstract class. In other words, the abstract class must exist before multiple implementations are created. In Emerald, there is no such restriction since the relationship between implementations and abstract types is deduced by the type system.

Support for Polymorphism

At least three broad varieties of polymorphism have been identified in the literature.[17, 21, 29, 30]

Inclusion polymorphism refers to the situation where an object may belong to many different types that need not be disjoint, that is, a type may include one or more of the other types. Subtyping is a good example of this kind of polymorphism. Objects belonging to a type are manipulable as belonging not only to that type, but also to its supertypes. In implementation terms, object representations are chosen so that operations can work uniformly on instances of subtypes and supertypes.

Parametric polymorphism allows a function to have an either implicit or explicit type parameter that determines the type of argument required for each of its applications. This is perhaps the purest form of polymorphism, permitting the *same* operation to be applied to arguments of different types. Implicit type parameters are used in ML[29], and explicit type parameters appear in Russell.[30]

Ad-hoc polymorphism refers to situations where a procedure works, or appears to work, on several types. It is usually not considered to be true polymorphism because these types are not required to exhibit any common structure; neither are the results of the procedure required to be similar. Ad-hoc polymorphism covers the notions of *overloading*, where the same name denotes different functions in different contexts (e.g., the operator "+" in most programming languages), and *coercion*, where values are (automatically) converted to the type expected by a function (e.g., in $2 + 3.1$, the integer 2 is coerced to the real 2.0); the difference between these two notions often disappears in untyped languages.

Relating traditional discussions of polymorphism to object-oriented languages has to be done carefully. Inclusion polymorphism fits naturally into object-oriented programming. However, consider the following standard example of a polymorphic function:

function *length*[*t*: *someType*]

This kind of polymorphism is unnatural in object-oriented languages because functions are not first-class entities in such languages. That is, rather than applying functions to objects, operations are invoked on objects. To obtain the length of an object, one invokes the *length* operation of that object since the object

```
const SimplePolyTester ←
    object SPT
        const Printable ←
            type aPrintableType
                function asString → [String]
            end aPrintableType

        operation PrintIt[p: Printable]
            stdout.PutString[p.asString || "\n"]
        end PrintIt

        process
            SPT.PrintIt[0]
            SPT.PrintIt[1.1]
            SPT.PrintIt[true]
            SPT.PrintIt["This is rather trivial."]
        end process
    end SPT
```

Figure 11: A polymorphic operation

obviously knows its own type and does not need the type as an argument. Instead of applying a *length* function to a list to find its length, the list itself is asked what its length is. In this setting there is no need for *length* to be polymorphic. If one implementation of *length* is shared by many classes, it is polymorphic. If *length* is separately defined for every object, it is not polymorphic. One cannot tell from the interface which case applies.

Although Emerald provides an efficient, practical form of polymorphism, its emphasis on type conformity and object autonomy makes the above classification scheme inappropriate. Based on the role played by the different polymorphic entities, an Emerald-oriented classification of polymorphism is proposed below.

1. *polymorphic operations*, which work "correctly" regardless of the types of their arguments,
2. *operations that return types*, and
3. *polymorphic objects* (*values*), which can be used in situations requiring different types.

Notice that 1 and 3 result from Emerald's notion of conformity which is a form of inclusion polymorphism.

Polymorphic Operations. All operations are naturally polymorphic because objects (of different types) may be used as arguments provided they conform to the types declared for the formal parameters of the operation.

Figure 11 illustrates Emerald's support for polymorphic operations using the function *PrintIt*. This function takes a parameter of type *Printable*, and displays its string form on the standard output. Any object that understands the *asString* operation conforms to the type *Printable*, thus permitting any such object to be passed to *PrintIt*. Notice the use of *SPT* to represent the target of the invocation of *PrintIt*.

```
const List ←
    immutable object ListCreator
        export of

        function of[eType: AbstractType] → [result: ListCreator]
            where
                ListCreator ←
                    immutable type LC
                        function getSignature → [Signature]
                        operation new → [ListType]
                    end LC

                ListType ←
                    type LT
                        operation AddAsNth[eType, Integer]
                        operation DeleteNth[Integer]
                        function GetNth[Integer] → [eType]
                        function Length → [Integer]
                    end LT
            end where

            result ←
                immutable object NewListCreator
                    export getSignature, new

                    function getSignature → [r: Signature]
                        r ← ListType
                    end getSignature

                    operation new → [result: ListType]
                        result ←
                            object theList
                                export AddAsNth, DeleteNth, GetNth, Length

                                % Implementations of the various operations

                            end theList
                    end new
                end NewListCreator
        end of
    end ListCreator
```

Figure 12: A List type generator

Operations that return types. Most programming languages support the *array* type constructor, which is usually a built-in operator that takes a type as an argument and returns a new type as a result, for example, Array[Integer] and Array[Boolean].

While languages such as Pascal support this kind of polymorphism for arrays and similar built-in types, such support is not usual for user-defined types. For example, the programmer cannot describe a *List* type without also specifying the type of the elements. Some languages such as Modula-2+[31] allow elements of any type to be used, but ensure the homogeneity of the List through a required user-specified run-time check. Other languages such as ML[29] allow the specification of polymorphic operations through implicit *type variables* that are instantiated when type-checking is performed.

Emerald supports user-defined operations that return types subject only to the constraint that the types of variables, constants, and formal parameters be evaluable at compile-time. Since types are objects, types may be passed as arguments to functions that create types. As an example, consider the polymorphic list creator in Figure 12. *List* is an object that exports a single function, *of*. This function takes a type *eType* as an argument, and returns a *List creator* for lists whose elements are of type *eType*. This resulting class-like object can be used to create homogeneous lists. For example, a list of integers can be declared and created as follows:

var *iList* : *List.of*[*Integer*] ← *List.of*[*Integer*].*new*

The above example reveals the expressiveness of Emerald invocations. *List* is an object that exports the operation *of*, whose single argument must be an abstract type. The invocation

List.of[*Integer*]

returns an object that is used in two ways. First, when used to specify the abstract type of *iList*, the denoted type is obtained by invoking *getSignature* on *List.of*[*Integer*]. Second, when used to initialize *iList*, the *new* invocation yields an object that is a list of integers. What is unusual about the *of* invocation is that it is a compile-time invocation, that is, the Emerald compiler itself performs this invocation and creates the resulting object. This is possible because *List* is immutable, *of* is a function, and the argument to the invocation, *Integer*, is an immutable compile-time constant. As a result, *iList* is just as efficient as a list of integers that is directly defined; in fact, exactly the same code is generated in either case.

It is interesting to examine the various Emerald features that went into the construction of *List* in Figure 12. The function *of* uses the fact that types are objects to specify a type parameter. The separation of type and implementation allows restriction of the behavior of the elements in the list without constraining their implementation. Finally Emerald's object constructor makes this entire description possible in a self-contained way. Note that there are three levels of constructor, which is not possible in any other object-oriented language. Conformity allows *List.of*[*Integer*] to act both as a type and as a creator.

The result type of the operation *List.of* depends on the type of its argument. Result types of Emerald operations may also depend on the *value* of their arguments. For example, the Emerald compiler uses *TreeNodes* to represent different constructs of the language. Since constructs are identified by different keywords, the result type of the compiler operation that creates *TreeNodes* depends on the value of the input keyword.

Polymorphic Objects. The notion of *value* in traditional languages is subsumed by the notion of *object* in object-oriented languages. Traditional languages with typed pointers such as Pascal usually support the notion of **nil**, which may be assigned to any pointer variable regardless of its type. This is an example of a *polymorphic value*.

Emerald supports polymorphic objects. An object may be assigned to any identifier provided that the object's type conform to that declared for the identifier. A direct analogue of Pascal's **nil** pointer value is the

```
const PolyValue ←
   object pv
      const Printable ←
         type Printable
            function asString → [String]
         end Printable

      process
         var aStringObj: String
         var aPrintableObj: Printable
         var anyObj: Any

         aStringObj ← "Emeralds are green"
         aPrintableObj ← "Emeralds are green"
         anyObj ← "Emeralds are green"
      end process
   end pv
```

Figure 13: Some polymorphic objects

object **nil** in Emerald. *None*, the type of **nil**, conforms to every Emerald type. Any identifier in Emerald, regardless of its declared type may be assigned the object **nil**. Figure 13 illustrates a more restricted example. Here, the identifier *aPrintableObj* may name any object that understands the *asString* function, and the same string object *"Emeralds are green"* may be assigned to *aStringObj*, *aPrintableObj*, and *anyObj* because its type conforms to the types of all three identifiers.

This kind of polymorphism is naturally available in most class-based languages because each object belongs to its class as well as the class's superclasses. However, in these languages, all the classes that an object can belong to are known before the instantiation of the object. In Emerald, this determination is made dynamically. As a consequence, Emerald objects are also polymorphic not merely with respect to existing types, but also to hitherto unthought of types to which they conform.

4. CONCURRENCY

Emerald's support for concurrency takes the form of *active* objects. An active object contains a *process* that is started after the object is initialized, and executes in parallel with invocations of that object's operations and other active objects. This process continues to execute its specified instructions until it terminates. An object with a simple process is shown in Figure 14. The process in this object performs the repetitive actions of a philosopher in the well-known dining philosophers' problem.[32]

An Emerald process represents an independent thread of control. New threads may be created dynamically simply by creating new objects. For example, in Figure 15, five instances of the philosopher object are created. Since the object assigned to the elements of *philoArray* export no operations, the object constructor is used here to achieve an effect similar to the use of the fork procedure.[33]

In Emerald, operation invocation is synchronous. Thus, a thread of control can be thought of as passing through other objects when it invokes operations on those objects, that is, the invocation of an operation

```
const aPhilosopher ←
    object P
        % Assume suitable objects Table and Node are defined
        % Assume integer objects eatingTime and thinkingTime have been defined
        process
            loop
                Table.pickupForks[p]
                Node.sleep[eatingTime]
                Table.putdownForks[p]
                Node.sleep[thinkingTime]
            end loop
        end process
    end P
```

Figure 14: A dining philosopher

```
const anOrganizer ←
    object Organizer
        process
            var count : Integer ← 5
            const philoArray ← Array.of[Philosopher].create[5]
            loop
                philoArray[count] ←
                    object P
                        % as in Figure 14
                    end P
                count ← count − 1
                exit when count = 0
            end loop
        end process
    end Organizer
```

Figure 15: An organizer object

```
const Clock ←
  object C
    export getTimeOfDay, setTimeOfDay
    monitor
      var theTime: Integer ← 0

      operation IncTime
        theTime ← theTime + 1
      end IncTime

      function getTime → [r: Integer]
        r ← theTime
      end getTime

      operation setTime[r: Integer]
        theTime ← r
      end setTime
    end monitor

    operation setTimeOfDay[newTime: String]
      var t: Integer
      % store newTime in some internal form
      setTime[t]
    end setTimeOfDay

    operation getTimeOfDay → [currentTime: String]
      var t: Integer ← getTime
      % return the String form of t
    end getTimeOfDay

    process
      loop
        System.Tock
        IncTime
      end loop
    end process
  end C
```

(a) A clock object

```
const System ←
  object S
    monitor
      const timing ← Condition.Create

      % Tick is invoked by a hardware clock
      operation Tick
        signal timing
      end Tick

      operation Tock
        wait timing
      end Tock
    end monitor
  end S
```

(b) A system object

Figure 16: Concurrent programming in Emerald

provides the thread of execution for that operation. This means there can be multiple simultaneous invocations of an operation in the same object. Each invocation can proceed independently, providing fine-grained parallelism. In other distributed object-oriented languages, such as EPL,[34] each object has a message queue for incoming requests and a set of threads of control to respond to those requests.

Emerald uses monitors to regulate access to an object's local state that is shared by the object's operations and process, and synchronization is achieved using system-defined *condition* objects.[35] The Emerald monitor is similar to the monitor in Concurrent Pascal[36] or Concurrent Euclid,[37] but is completely enclosed within an object. An object's process commences execution outside the monitor, but can enter the monitor by invoking monitored operations when it needs access to shared state.

The choice of the monitor mechanism in Emerald for synchronization reflects designer prejudice and familiarity rather than its clear superiority over other mechanisms. However, monitors were found to provide a natural protection mechanism within the object-oriented paradigm, and they are also efficient for synchronization within an object especially when there is no contention—in this case, monitor entry and exit are performed by a few in-line machine instructions.

Figure 16 illustrates the various aspects of Emerald's concurrent programming model with a naive implementation of a clock. The clock object uses a simple internal representation (*theTime*) that is protected by a monitor and can be manipulated by several monitored operations. The internal representation is constantly updated by the clock object's process, which synchronizes with a timing pulse provided by the system object. The expensive operations for converting between the internal and external representations are defined outside the monitor, thus allowing multiple simultaneous invocations of them to proceed concurrently. The *System* object demonstrates a simple use of Emerald condition objects to provide buffering of timing pulses.

5. OTHER FEATURES

In this section we discuss those remaining features of Emerald that contribute to its programming model.

Block Structure and Nesting

The use of object constructors makes block structure and nested object definitions natural. We have already seen several examples of such nested object constructor definitions, for example, in Figure 12, the object constructor *theList* nested inside *NewListCreator* which in turn is nested inside *aListCreator*.

Emerald also supports the notion of blocks. Emerald blocks, similar to those in Algol-60, define a new scope for identifiers. The scoping of Emerald identifiers is traditional, with the one exception that an identifier is visible throughout the scope in which it is declared, that is, both before and after the declaration. Examples of constructs that open new scopes for identifiers are operation definitions, object and type constructors, monitors, or loop statement bodies. Non-local identifiers are implicitly imported into nested scopes unless they are re-defined.

Since object constructors and type constructors create new objects that persist beyond the end of the enclosing block, identifiers imported into these constructs are treated specially. Each imported identifier is evaluated when the constructor is executed and within the constructor the value is bound to an implicitly declared constant with the same name.

Consider the full form of the integer literal node creator shown in Figure 17. All objects created by invocations of *IntegerNode.new* are identical except for the binding of the identifier *value*, which is initialized from the argument *val* to each invocation. Once an object is created, it cannot see changes to the argument *val*, nor is it bothered when that identifier ceases to exist upon return of the *new* invocation.

Emerald blocks therefore have several similarities with those in Algol-60, and hence Simula and Beta. Smalltalk abandoned block-structure in favor of the single-level declaration of classes and global access to class definitions. Multi-level class definitions are not permitted; the Smalltalk programmer cannot declare a class inside another class.

```
const IntegerNode ←
    immutable object INC
        export new

        const IntegerNodeType ←
            type INType
                function getValue → [Integer]
                operation setValue[Integer]
                operation generate[OutStream]
            end INType

        operation new[val : Integer] → [aNode : IntegerNodeType]
            aNode ←
                object IntegerLiteral
                    export getValue, setValue, generate
                    monitor
                        var value : Integer ← val

                        operation getValue → [v : Integer]
                            v ← value
                        end getValue

                        operation setValue[v : Integer]
                            value ← v
                        end setValue

                        operation generate[stream : OutStream]
                            stream.putString[“LDI %d\n”, value]
                        end generate
                    end monitor
                end IntegerLiteral
        end new
    end INC
```

Figure 17: Block structure and object constructors

There has been considerable discussion in the literature both supporting and opposing the use of block structure in programming languages.[38, 39, 40] By ensuring locality, block structure permits the seclusion of object constructors to the context in which they make sense, thus emphasizing good programming style. On the other hand, a nested object constructor has dependencies on enclosing objects, thus making it difficult for it to be reused in different contexts. Jade, the Emerald extension mentioned in Section 3, improves the reusability of nested object constructors by parameterizing contextual dependencies.[27]

Syntax

In recent years, as parsing techniques have become better understood, the syntax of a language has usually come to represent the prejudices of its designers rather than any significant improvement. This has meant that issues of syntax are generally considered minor and unworthy of comment. Nevertheless, some aspects of the Emerald syntax are interesting enough to merit a brief discussion.

The reader may have been pleasantly surprised to find the absence of semicolons in Emerald — semicolons are needed neither to terminate statements nor to separate statements. Emerald uses square brackets uniformly for invocations and operation definitions. Note that arrays and vectors, although system-defined, are objects that follow the standard invocation paradigm, and do not need any special syntax. However, sugared shorthand notations for common operations (such as subscripting) are provided. The shorthands are available uniformly for both user- and system-defined objects.

Emerald uses the ← symbol uniformly for all instances of the binding operation: assignment, constant declaration, and collecting the results of an invocation. As part of its distributed programming heritage, Emerald clearly distinguishes between the use of input parameters and output parameters. Both invocations that return multiple results such as:

```
var point :
   type T
      operation getCoordinates → [Real, Real, Real]
   end T
var x, y, z : Real
...
x, y, z ← point.getCoordinates
```

and multiple assignments such as:

```
var a, b : Integer
a, b ← b, a
```

are conveniently expressed.

6. FINAL REMARKS

Current Status

An Emerald prototype is implemented under UNIX* on SUN-3† and VAX‡ processors. The compiler generates native code, which is dynamically loaded into a pre-existing Emerald object world. Currently, Emerald does not have a sophisticated programming environment. The success of Smalltalk (and Trellis to a lesser extent) clearly demonstrates that an excellent programming environment is a great boon to its user community, while shortcomings in the programming environment are discouraging to a programmer. The lack of a proper programming environment is not a criticism of the Emerald language as such; it reveals that the provision of a complete programming system is a must for getting the most from the language. Despite this handicap, the development of various applications (such as a prototype mail system, a multi-user calendar system, and the Emerald compiler itself) reveals the usefulness of the Emerald language features.

The focus of our current work is on improving implementation sharing and software reuse in Emerald. We are also working on a new, more portable, implementation of the language based on the reimplementation of the compiler in Emerald.

Conclusions

We have presented and discussed a number of examples that bring out the flavor of the Emerald approach to programming. Emerald is not just the sum of its parts: it is also the *interaction* of those parts. Emerald's object constructor and nesting allow the simulation of Smalltalk classes. The separation of typing and implementation, types as objects, and Emerald's notion of conformity all combine to allow a very natural form of polymorphism.

A number of factors contribute to the differences between Emerald and several other object-based languages. These include its emphasis on abstract typing and compile-time type-checking, the clear separation of types and implementations, and the use of simple mechanisms for each identifiable paradigm. For example, Emerald has the *object constructor* for object creation, the *abstract types* and *conformity* for the classification of objects, and objects themselves contain their own methods. This *unbundling* of the functions of Smalltalk classes has been found appealing by other researchers.[21]

Our experience with Emerald has shown that it is an interesting and useful general-purpose programming language. Indeed, we are now convinced that the Emerald model of objects is a good foundation on which

*UNIX is a trademark of AT&T Bell Laboratories.
†SUN-3 is a trademark of Sun Microsystems.
‡VAX is a trademark of Digital Equipment Corporation.

future object-oriented systems can be built.

ACKNOWLEDGEMENTS

We thank Alan Borning, Jeff Chase, Bjorn Freeman-Benson, Bill Griswold, Niels Christian Juul, John Maloney, and the referees for their feedback and comments. Rajendra Raj, Ewan Tempero, and Henry Levy were supported in part by the U.S. National Science Foundation under Grants No. CCR-8619663, CCR-8700106, and CCR-8907666, and by the Digital Equipment Corporation Systems Research Center and External Research Program. Norman Hutchinson was supported in part by the U.S. National Science Foundation under Grant CCR-8701516. Eric Jul was supported in part by the Danish Natural Science Research Council under Grant No. 5.17.5.1.35.

REFERENCES

[1] A. Black, N. Hutchinson, E. Jul, and H. Levy. Object Structure in the Emerald System. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 78–86, October 1986. In SIGPLAN Notices, 21(11), November 1986.

[2] A. Black, N. Hutchinson, E. Jul, H. Levy, and L. Carter. Distribution and Abstract Types in Emerald. *IEEE Transactions on Software Engineering*, 13(1):65–76, January 1987.

[3] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained Mobility in the Emerald System. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[4] E. Jul. *Object Mobility in a Distributed Object-Oriented System.* PhD thesis, TR 88-12-06, Department of Computer Science, University of Washington, Seattle, December 1988.

[5] US Department of Defense, MIL-STD-1815, Washington, D.C. *Reference Manual for the Ada Programming Language*, January 1983.

[6] N. Wirth. *Programming in Modula-2.* Springer-Verlag, Berlin, 1983.

[7] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, Reading, Massachusetts, March 1986.

[8] A. Goldberg and D. Robson. *Smalltalk-80: The Language And Its Implementation.* Addison-Wesley, Reading, Massachusetts, 1983.

[9] D. Ungar and R. B. Smith. Self: The Power of Simplicity. In *Proceedings of the Second ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 227–241, October 1987. In SIGPLAN Notices, 22(12), December 1987.

[10] G. Birtwistle, O.-J. Dahl, B. Myrhaug, and K. Nygaard. *SIMULA Begin.* Petrocelli/Charter, New York, 1973.

[11] B. B. Kristensen, O. L. Madsen, B. Moller-Pedersen, and K. Nygaard. The BETA Programming Language. In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 7–48. The MIT Press, Cambridge, Massachusetts, July 1987.

[12] H. M. Levy and E. D. Tempero. Modules, Objects, and Distributed Programming: Issues in RPC and Remote Object Invocation. *Software Practice and Experience*, 1990. This issue.

[13] D. L. Parnas. On the Criteria to Be Used in Decompositing Systems into Modules. *Communications of the ACM*, 5(12):1053–1058, December 1972.

[14] B. Meyer. *Object-Oriented Software Construction.* Prentice-Hall International, New York, 1988.

[15] A. H. Borning. Classes Versus Prototypes In Object-Oriented Languages. In *ACM/IEEE Fall Joint Computer Conference*, November 1986.

[16] A. H. Borning and T. O'Shea. Deltatalk: An Empirically and Aesthetically Motivated Simplification of the Smalltalk Language. In *Proceedings of the European Conference on Object-oriented Programming*, June 1987. In Lecture Notes in Computer Science, Vol. 276, Springer-Verlag, Berlin, 1987.

[17] L. Cardelli and P. Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *Computing Surveys*, 17(4):471–522, December 1985.

[18] N. C. Hutchinson. *Emerald: An Object-Based Language for Distributed Programming.* PhD thesis, TR 87-01-01, Department of Computer Science, University of Washington, Seattle, January 1987.

[19] L. Cardelli, J. Donahue, L. Glassman, M. Jordan, B. Kalsow, and G. Nelson. Modula-3 report (revised). Technical Report #52, Systems Research Center, Digital Equipment Corporation, Palo Alto, California, November 1989.

[20] L. Cardelli. Typeful Programming. Technical Report #45, Systems Research Center, Digital Equipment Corporation, Palo Alto, California, May 1989.

[21] S. Danforth and C. Tomlinson. Type Theories and Object-Oriented Programming. *Computing Surveys*, 20(1):29–72, March 1988.

[22] B. Meyer. Static Typing for Eiffel. Technical Report TR-EI-18/ST, Interactive Software Engineering, Inc., Santa Barbara, California, July 1989. See Rationale for the Eiffel Rules.

[23] C. Schaffert, T. Cooper, B. Billis, M. F. Kilian, and C. Wilpolt. An Introduction to Trellis/Owl. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 9–16, Portland, Oregon, September 1986. In SIGPLAN Notices, 21(11), November 1986.

[24] W. R. Cook. A Proposal for Making Eiffel Type-safe. *The Computer Journal*, 32(4):305–311, August 1989.

[25] A. Snyder. Inheritance and the Development of Encapsulated Software Components. In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 165–188. The MIT Press, Cambridge, Massachusetts, July 1987.

[26] B. Liskov. Data Abstraction and Hierarchy. In *OOPSLA '87, Addendum to the Proceedings*, pages 17–34, October 1987. In SIGPLAN Notices (23)5, 1988.

[27] R. K. Raj and H. M. Levy. A Compositional Model for Software Reuse. *The Computer Journal*, 32(4):312–322, August 1989.

[28] R. K. Raj. *Composition and Reuse in Object-Oriented Languages.* PhD thesis, University of Washington, Seattle, 1990.

[29] R. Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[30] J. Donahue and A. Demers. Data Types are Values. *ACM Transactions on Programming Languages and Systems*, 7(3):426–445, July 1985.

[31] P. Rovner, R. Levin, and J. Wick. On Extending Modula-2 For Building Large, Integrated Systems. Technical Report # 3, Systems Research Center, Digital Equipment Corporation, Palo Alto, California, January 1985.

[32] E. W. Dijkstra. Cooperating Sequential Processes. In F. Genuys, editor, *Programming Languages.* Academic Press, 1968.

[33] G. Agha and C. Hewitt. Concurrent Programming Using Actors. In A. Yonezawa and M. Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 37–54. The MIT Press, 1987.

[34] A. P. Black. The Eden Programming Language. Technical Report 85-09-01, Department of Computer Science, University of Washington, Seattle, September 1985.

[35] C. A. R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557, October 1974.

[36] P. Brinch Hansen. The Programming Language Concurrent Pascal. *IEEE Transactions on Software Engineering*, 1(2):199–207, June 1975.

[37] R. C. Holt. *Concurrent Euclid, The Unix System, and Tunis.* Addison-Wesley, Reading, Massachusetts, 1983.

[38] D. R. Hanson. Is Block Structure Necessary? *Software Practice and Experience*, 11:853–866, 1981.

[39] R. D. Tennent. Two Examples of Block Structure. *Software Practice and Experience*, 12:385–392, 1982.

[40] O. L. Madsen. Block Structure and Object-Oriented Languages. In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 113–128. The MIT Press, Cambridge, Massachusetts, July 1987.