(To appear, Software - Practice and Experience)

The Emerald Approach to Programming[*]

Rajendra K. Raj, Ewan Tempero, Henry M. Levy,
Norman C. Hutchinson[†] and Andrew P. Black[‡]

Technical Report 88-11-01
November 1988
(Revised February 1989)

Department of Computer Science and Engineering, FR-35
University of Washington
Seattle, Washington 98195

Abstract

Emerald is an object-oriented programming language developed primarily to simplify the construction of efficient distributed applications. Its design introduces a new object model and includes a type system that emphasizes abstract types, allows separation of behavior and implementation, and provides the flexibility of polymorphism and subtyping, all of which can be checked at compile-time. The result is a *general-purpose* programming language with aspects of traditional object-oriented languages, such as Smalltalk and abstract data type languages such as Modula-2 and Ada.

This paper describes the Emerald language and methodology which we believe provides a flexible basis for general-purpose programming while also yielding efficient programs. We present a variety of Emerald programs, and compare and contrast the Emerald approach to programming with the approaches used in several other languages.

[*]This work was supported in part by the National Science Foundation under Grants MCS-8004111, DCR-8420945 and CCR-8700106, and by the Digital Equipment Corporation External Research Program.

[†]Norman Hutchinson is with the Department of Computer Science, University of Arizona, Tucson, AZ 85721; he is supported in part by the National Science Foundation under Grant CCR-8701516.

[‡]Andrew Black is with Digital Equipment Corporation, 550 King Street, Littleton, MA 01460.

1 Introduction

Emerald is a strongly-typed, object-based programming language for concurrent, distributed programming [Black et al. 86, Black et al. 87, Jul et al. 88]. It differs in several ways from these other languages that are either module or object based. Unlike Ada [Ada 83] or Modula-2 [Wirth 83], the sole unit of programming is the *object*, which behaves like a run-time version of the Modula-2 *module* or the Ada *package*. Unlike C++ [Stroustrup 86] or Smalltalk [Goldberg & Robson 83], it has no notion of *class*, the staple of virtually all object-based languages. Instead, Emerald provides *object constructors* for run-time creation of objects, *abstract typing* for classification, and *conformity* for type comparison. Access to Emerald objects is provided via *operations* that are invoked by other objects.

Past experience has suggested that languages had to choose between strong typing and flexibility, and using one meant sacrificing the other. For example, Self and Smalltalk emphasize flexibility by delaying type checking to run-time, while Modula-2 and Ada are strongly typed at the expense of flexibility. Emerald provides both typing and flexibility using a simple well-developed, and efficiently-implemented notion of *type* that allows compile-time checking, polymorphism, abstract types, and subtyping. A type in Emerald is a collection of operations and their signatures, and more importantly, a type is itself an object. Simula and C++ are typed languages that also provide some of this run-time flexibility, but through the use of *virtual* procedures.

Operations on objects can be executed concurrently both within objects and among objects. Like Simula-67 [Birtwistle et al. 73] and Beta [Kristensen et al. 87], Emerald supports block structure and nesting, a feature not supported by Smalltalk. Failures can be detected via the use of *failure-handlers* that can be attached to each block. Emerald facilitates distributed programming by providing primitives for locating and moving objects in a distributed system. However, programmers may make distribution transparent when desired by ignoring these features [Jul et al. 88].

This paper discusses the usefulness of the Emerald approach by presenting programming examples illustrating its different features, comparing and contrasting similar features of other languages, and analyzing the benefits and drawbacks of the Emerald programming model. The next section discusses Emerald's object paradigm, and Section 3 examines Emerald's type system from the programming perspective. Section 4 presents the other features of Emerald that contribute to its programming paradigm. The current status of the Emerald system is outlined in Section 5. We conclude with a summary of the Emerald approach to programming.

2 Programming with Object Constructors

Programming in Emerald using its object model is presented here, emphasizing the differences from more traditional object-oriented languages such as Simula-67, Smalltalk-80, Trellis/Owl [Schaffert et al. 86], C++, and Self [Ungar & Smith 87]. Emerald's encapsulation and abstraction mechanisms are also compared with those offered by value-oriented languages such as Modula-2, Modula-3 [Cardelli et al. 88], and Ada. A complete description of the Emerald programming language may be found in the Emerald Language Report [Hutchinson et al. 87], while Hutchinson's dissertation [Hutchinson 87] discusses the rationale for its design and implementation.

The Emerald *object* is an abstraction for the notions of data, type, and process. Emerald objects consist of private data, private operations, and public operations. An object is solely responsible for its behavior and so must contain everything necessary to support that behavior. This self-containment allows objects to be moved freely between nodes in a distributed system. To support effective distributed management of objects (should that be desired), an object may also constrain its location. 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 equivalent to procedure calls in traditional languages, and to messages in Smalltalk-like object-oriented languages.

2.1 Object Creation

In most object-based languages, new objects are created by an operation on a *class* object as in C++ or Smalltalk (*type* objects assume this function in Trellis/Owl). This class object has multiple functions: it defines the structure, interface, and behavior of all its *instances*, and it responds to *new* invocations to create new instances. 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, the operations (both public and private), and its optional process. The syntax of an object constructor is:

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

Figure 1 gives an example of the creation of an object. The object constructor has the name *aRep*, and the object it creates has public operations *store* and *contains.* Its private state consists of (1) *name*, which names (or equivalently, references) a *String* object, and (2) *value*, which names an object of any type. Since exported operations may be invoked concurrently, access to the object's private state is regulated by a monitor (see Section 4.1).

The execution of this object constructor results in the creation of the described object, which is named by the identifier *aRepository.* By virtue of the **const** binding, *aRepository* will always refer to this object; the state of this object may however be changed by invocations of its *store* operation. Objects may also be qualified as being **immutable**, asserting that their abstract state does not change with time. The compiler

```
const aRepository ←
    object aRep
        export store, contains
        monitor
            var name: String ← nil
            var value: Any ← nil

            operation store[n : String, v: Any]
                name ← n
                value ← v
            end store

            function contains[n: String] → [v: Any]
                if n = name then
                    v ← value
                else
                    v ← nil
                end if
            end contains
        end monitor
    end aRep
```

Figure 1: Example use of Object Constructor

may use such information for optimization, but the language does not attempt to restrict the operations that may be performed on the concrete state.

The **export** clause describes the public operations of the object. That is, communication with the object is only possible by *invoking* these operations. Note that the object's private state can be modified only by its own operations. Thus, objects in Emerald are fully encapsulated unlike those in, for example, Simula-67; 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 [Borning 86]:

1. they generate new objects,
2. they describe the representation of objects, and
3. they serve as the repositories for operations (methods in Smalltalk terminology).

How other functions of Smalltalk classes are performed in Emerald is discussed later in this paper. The essential point to note is that the object constructor does not have the "baggage" associated with classes, allowing it to be more flexible, as the next few examples demonstrate.

As mentioned above, only one object is created each time an object constructor is executed. To create more than one object, we could naively copy the text of the creator everywhere it is needed, but there are better alternatives. First, multiple objects can be instantiated by placing the constructor in a repetitive

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

Figure 2: Creating several *Repository* objects

```
const RepositoryCreator ←
    immutable object RC
        export new

        const Repository ←
            type RT
                operation Store[n : String, v: Any]
                function Contains[n: String] → [v: Repository]
            end RT

        operation new → [aRepository: Repository]
            aRepository ←
                object aRep
                    % as in Figure 1
                end aRep
        end new
    end RC
```

Figure 3: A Repository Creator

construct such as a loop. An example of this is shown in Figure 2, where an array is filled with *Repositories*. Another possible context for an object creator is the body of some operation of another object. Thus, the function of a class with respect to object creation can be simulated. For example, in Figure 3, the object referred to by *RepositoryCreator* [1] exports an operation *new*. The behavior of this operation is to return a reference to the object created by executing the object constructor *aRep* shown in Figure 1. Thus, every time *new* is invoked on *RepositoryCreator*, a new repository is created, enabling it to perform the object creation function of a class.

Note that there is nothing magical about the use of the operation name *new*—any other name would be equally valid because this is an ordinary operation. Note also that *RepositoryCreator* has been declared *immutable*, an assertion that the object's state, which in this case consists only of the object named by *Repository*, will never change. (*Repository* actually names a type constructor. Types are discussed in the next section; for the present, it is sufficient to note that a type constructor also creates a full-fledged immutable Emerald object.)

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. Figure 4 presents one approach that can be used in Ada to define a package that can be used for repository creation. While the Emerald and Ada versions of repository are very similar, there are differences stemming from the characteristics of their type system, and due to the differences between a value-oriented language and an object-oriented one.

Since Ada is not object-oriented, data to be manipulated must be passed to each function. Ada also provides for the creation of data items at the same level as the operations that manipulate them, whereas object-oriented languages essentially treat data as objects and let the objects perform their own manipulations. Finally, Ada has no notion of a type similar to that of **Any** in Emerald or REFANY in Modula-3. While Ada's generic packages can simulate some of the repository's behavior by parameterizing a construct with a type, they cannot allow arbitrary values to be stored, thus revealing the inflexibility of Ada's type system. Note also that choosing an "undefined" value to return if a repository does not contain a given name requires that the definition of ANY provide one.

While class-like objects are used by Emerald programmers as and when necessary to create identical objects, our experience so far has been mixed as to whether they (classes) are natural for object creation. The use of such class-like objects could have been caused by an *a priori* bias resulting from programming habits developed earlier in class-based languages such as Simula or Smalltalk. However, as discussed next, what is interesting is that Emerald programmers primarily use object constructors for non-class-like objects.

[1] Notice the distinction between *RC* , which is a bound variable that may be used inside the object constructor to refer to the object being created, and *RepositoryCreator* , which is a name bound by virtue of the **const** declaration to the object thus created.

```
package REPOSITORY_CREATOR is
    type REPOSITORY is private;
    -- Since Ada does not have anything like the Emerald type ANY, the following
    -- type has to be user-defined for a particular application.
    type ANY is private;
    function NEW_REP return REPOSITORY;
    procedure STORE(R: in out REPOSITORY; N: in STRING; V: in ANY);
    function CONTAINS(R: in REPOSITORY; N: in STRING) return ANY;
private
    type ANY is whatever
    type REPOS_REC is
        record
            NAME : STRING(1..80);
            VALUE: ANY;
        end record;
    type REPOSITORY is access REPOS_REC;
end REPOSITORY_CREATOR;

package body REPOSITORY_CREATOR is
    function NEW_REP return REPOSITORY is
        begin
            return new REPOS_REC;
        end NEW_REP;
    procedure STORE(R: in out REPOSITORY; N: in STRING; V: in ANY) is
        begin
            R.NAME : = N;
            R.VALUE : = V;
        end STORE;
    function CONTAINS(R: in REPOSITORY; N: in STRING) return ANY is
        begin
            if R.NAME = N then
                return R.VALUE;
            else
                return 0; -- This must be changed for each definition of ANY
            end if;
        end CONTAINS;
end REPOSITORY_CREATOR;
```

Figure 4: The Repository Creator in Ada

2.2 One-of-a-kind Objects

That a class mechanism is not required in Emerald illustrates the flexibility of the object constructor. 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 provides a natural handle on the creation of objects. When such an expression is executed once, a "one-of-a-kind" (also called *singular*) object is created. To our knowledge, the only other language that supports such a feature is Self.

Class-based languages are typically designed for situations where many objects with the same behavior are needed. They prevent a concise definition 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. The simple solution that a class Boolean be created, with two instances, one *true* and the other *false*, cannot be used because *true* and *false* need to respond differently to messages, and hence cannot be instances of the same class. The actual solution used in Smalltalk is more involved, requiring additional classes *True* and *False* to be defined as subclasses of *Boolean*, with the objects *true* and *false* as their instances respectively. Moreover, given these two classes, multiple instances of *true* and *false* can be created.

Creating a class that is guaranteed to have only one instance is 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 all entities are not full-fledged objects and support for metaclasses is non-existent, one-of-a-kind objects cannot be created at all.

The above justification for one-of-a-kind objects is open to criticism because primitive types such as Boolean almost always defy clean incorporation into a programming language. There are several examples that show that one-of-a-kind objects are in fact quite commonly needed in practice. For example, in a typical workstation, there is exactly one mouse object, exactly one keyboard object, perhaps only one console window object, etc. The class abstraction therefore seems redundant for any of these objects when exactly one instance will ever be needed. Thus, Emerald object constructors provide programmers with flexible object creation, from one to as many as required.

2.3 No Metaclasses

The object constructor removes the need for a multi-level instance/class/metaclass hierarchy found in languages such as Smalltalk. As already shown, by nesting object constructors, Emerald reduces the creation of objects to the familiar iteration and procedure constructs available in most programming languages. Figure 5 (a) illustrates that there is only one level of objects in Emerald. Note that the lack of a metaclass hierarchy is orthogonal to the existence of a type hierarchy (see Section 3).

Figure 5: “Metaclass Hierarchies” in Emerald and Smalltalk

In class-based languages, an object depends upon another object (its class) to describe its structure and behavior. In Figure 5 (b), the object *aRepository* is described by its class *Repository*, which is described by its class (metaclass of class *Repository*), 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 creators, creators of creators of creators, and so on. Understanding two levels of nesting, as in Figure 3, makes it easier to visualize further increases in nesting (see Section 3.5). In contrast, the concept of metaclasses, and the termination of infinite metaregress in Smalltalk, has been found to be the biggest hurdle to be overcome in learning Smalltalk [Borning & O’Shea 87]; this difficulty has motivated a report whose sole goal is to facilitate an understanding of Smalltalk metaclasses [Freeman-Benson 87]. Like Self, Emerald avoids this infinite meta-regress by making each object completely self-describing.

3 Programming with the Emerald Type System

Emerald is a strongly typed language that allows compile-time type checking. All identifiers in Emerald are typed abstractly, and the programmer must declare the type of the objects that an identifier may name.

Types serve several purposes in programming languages, as described in [Cardelli & Wegner 85, Hutchinson 87]. Types facilitate the representation independence required for higher-level programming. Types traditionally describe implementations and are responsible for creation of data instances. Because types provide implementation information, compile-time type checking also permits the generation of better code leading to improved performance. Finally, they enable the early detection of errors, and permit more meaningful error reporting.

In Emerald, most of these functions are performed by specialized constructs. The *object* provides representation independence. The object constructor provides for object creation and implementation information. What are type-dependent compiler optimizations in other languages 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, leading to the prevention and notification of incorrect usage.

Emerald is designed for the development of software in distributed and constantly running systems. Here, objects are usually developed and implemented separately and/or differently on different machines. To facilitate this, and to take into account the evolution of software, it is imperative that its type system not distinguish between objects based on their implementation. Furthermore, there are situations where the actual type of an object is not known at compile-time so it must allow dynamic type-checking. For example, when objects are retrieved at run-time from a repository (as in Figure 1), their types are needed for later processing, which can be type-checked only at run-time. Finally, Emerald supports polymorphism to provide flexibility and extensibility.

3.1 Type definition

Two features of the Emerald type system allow it to meet all of these requirements: abstract types, and types as objects. Emerald's type system is based on the *abstract types* of the entities being described; an abstract type is a collection of operation *signatures*, where a signature consists of the operation name and the types of its arguments and results. Note that a type contains neither code nor any information about implementation. This allows for complete separation of typing and implementation, and therefore allows multiple implementations of the same type to peacefully co-exist. Since a type in Emerald is an *object*, this permits types to be passed as parameters and enables run-time type checking. As discussed in Section 3.5, permitting types as parameters, along with type conformity, allows Emerald to support polymorphism.

The type definition *Repository* used in Figure 3 is extracted and shown in Figure 6. It lists the signatures of the various operations supported by the type. Although *Repository*, being a full-fledged object, can be created using an object constructor, we prefer to use the *type constructor* shown here (see [Black & Hutchinson 89] for a description of how this can be done).

3.2 Type Conformity

Type comparison in Emerald is based on the notion of *conformity.* Informally, a type S conforms to a type T (written $S \circ\!\!> T$) if all objects of type S have a signature that does not conflict with that of T's. In particular, $S \circ\!\!> T$ if:

```
const Repository ←
   type RT
      operation Store[n : String, v: Any]
      function Contains[n: String] → [v: Any]
   end RT
```

Figure 6: Type Definition for a Repository.

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.)

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
```

These types define people who may deliver anything and those who deliver only pizzas. Whatever one expects to do with what is delivered by a *JunkDeliverer* can also be done with the thing (pizza) delivered by a *PizzaDeliverer*. That is, *PizzaDeliverer* can be used in place of *JunkDeliverer*. Formally stated, *PizzaDeliverer* conforms to *JunkDeliverer*, but *JunkDeliverer* does not conform to *PizzaDeliverer*.

As motivation for property 4, consider the following types:

```
type AnyBank
      operation Deposit[Any] → []
end AnyBank
type MoneyBank
      operation Deposit[Money] → []
end MoneyBank
```

These types define banks 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, *AnyBank* can be used in place of *MoneyBank* but not vice versa. More formally, *AnyBank* conforms to *MoneyBank*, but *MoneyBank* does not conform to *AnyBank*.

The expressive power of conformity is illustrated using an example that has been adapted from a multi-user, shared calendar application. Here, people and rooms have calendars representing their schedules. Each participating person has a calendar object that conforms to type *PersonCalendar*, each appointment conforms to type *Appointment*, a list of appointments conforms to type *AppointmentList*, and type *PersonCalendar* conforms to type *ReadOnlyCalendar*. The calendar system also uses the two types:

```
const ViewAppointments ←
   type VA
      operation show[ReadOnlyCalendar] → [AppointmentList]
   end VA
const ViewAndFixAppointments ←
   type VAFA
      operation show[PersonCalendar] → [AppointmentList]
      operation schedule[PersonCalendar, Appointment] → []
   end VAFA
```

Objects conforming to type *ViewAppointments* may be used to show appointments, while those conforming to type *ViewAndFixAppointments* may be used to both show and schedule appointments.

Here, type *ViewAndFixAppointments* does not conform to type *ViewAppointments* because the argument to *show* in the two types does not conform as per Property 4. This feature has been pointed out as a limitation of conformity [Danforth & Tomlinson 88], but we believe otherwise — that this is actually an advantageous aspect of conformity, one that properly enforces correct program design. In the above example, operation *show* in *ViewAndFixAppointments* either (a) uses only those features of *PersonCalendar* that are also in *ReadOnlyCalendar*, or (b) also uses some features specific to *PersonCalendar*. In the former case, the programmer should have defined *show* using *ReadOnlyCalendar*. In the latter case, it is indeed true that because *ViewAndFixAppointments*'s *show* cannot be applied to a *ReadOnlyCalendar*, that is, it is correct for conformity to fail.

The lesson here is that the Emerald programmer needs to declare his arguments and results to conform to "best-fitting" types. While requiring an *a priori* use of such best-fitting types may be open to criticism that it is too restrictive, we believe that this not only enforces good design habits, but also permits maximum reusability of code. Using such types works out to be reasonable only because parameter conformity can extend to an arbitrary depth. This makes the Emerald approach 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 using "envelope types" that are large enough to work correctly in all applications, current and future.

Conformity can sometimes lead to "mistaken" type-matches. For example, in Figure 7 since *insert* and *remove* have identical signatures in all three objects, Emerald would regard *Stack*, *Queue* and *Stack2* as having the same type. This could lead to a situation when the *Queue* object may get used as a *Stack* with

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

Figure 7: “Mistaken” Type Matches

disastrous results. On the other hand, since Emerald is an open system where objects can be compiled and added at run-time, *Stack* can be replaced by *Stack2* (which can be a different object or even 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 need semantic information about what the operations do.

The subtyping notion of Trellis/Owl is similar to that of Emerald conformity, but there are differences. Trellis/Owl can prevent the meaningless substitution problem alluded to above because the notion of subtype is explicit, i.e., the programmer has to explicitly declare one type as a subtype of another. In the above example, the Trellis/Owl programmer would declare *Stack2* as a subtype of *Stack*, but not establish any subtype relationship between *Queue* and *Stack*. This prevents the accidental conformity exhibited in Emerald, but the price paid here 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*. Emerald would permit the types of these stack objects to conform as desired.

In the absence of type conformity declarations by the programmer, all approaches used in existing languages have problems. Unless semantics are incorporated with each type, *any* commonly used form of specifying type equivalence will display similar problems. The "commonly used" forms we refer to are:

1. equivalence by structure,
2. equivalence by name, and
3. conformity notion used in Emerald.

Type equivalence in Emerald has features of both 1 and 2. Aspects of structural equivalance are present because two equivalent operations must have conforming signatures, that is, the two signatures must have the same structure. Moreover, since two equivalent operations must necessarily have the same name, there is the flavor of equivalence by name.

Languages based on name-equivalence usually assume that differently named types are semantically different, and same-named types are equivalent. A common problem here is that semantically *equivalent* types, defined in different modules, even with the same name, or defined with different names, are not considered equivalent by the system. For example, *Stack2* and *Stack* will not be considered equivalent in Trellis/Owl. In languages such as Modula-2+ [Rovner et al. 85], programmers may explicitly state equivalances between different names, but this completely violates name-equivalence.

Structural equivalence has been advocated as a way of avoiding this problem. Here, two types are considered equivalent if their structure or representation is the same. An example of a language that uses structural equivalence is Modula-3. This too has its own share of problems. For example, all stacks, queues and deques implemented with the same kind of a singly-linked-list will have the *same* structure, and hence the same type.

Type conformity has other theoretical ramifications that are beyond the scope of this paper (see [Black & Hutchinson 89] for more details).

3.3 Relationship of Types and Objects

Emerald uses an object constructor to create an object, and a type constructor to define a classification; in fact, the object created by the former is an *implementation* of a type.

An object may belong to several types. Informally, we say an object O belongs to a type T when

typeOf (O) $\circ\!\!>$ T.

The application of **typeOf** to an object returns its "maximal" type, that is, the largest Emerald type that the object can belong to. Since this permits objects to be classified by their types, it now becomes possible to relate implementations and types via conformity. The *aRepository* of Figure 1 in Section 2 names an object whose type conforms to the type definition given in Figure 6.

So what exactly is a type? It is any object that conforms to the following definition:

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

The **Signature** represents a built-in object that may be considered to be a primitive abstract type, which is needed for ensuring proper compilation[2]. Thus any object that defines an exported *getSignature* function is an abstract type, and can be used anywhere a type is required. Note that the abstract type value denoted by any object is the result of its *getSignature* function.

Consider the *RepositoryCreator* in Figure 8. This object was constructed by the addition of a *getSignature* function to the version in Figure 3. On examining the code for *RepositoryCreator*, we see that **typeOf**(*RepositoryCreator*) is:

```
type RepositoryCreatorType
    function getSignature → [Signature]
    function new → [Repository]
end RepositoryCreatorType
```

but on examining the result of the *getSignature* function, we see that it returns the type value:

```
type Repository
    operation Store[String, Any]
    function Contains[String] → [Any]
end Repository
```

3.4 Separation of Type and Implementation

Using (1) object constructors for object creation, (2) abstract types based on interface specifications, and (3) conformity for type comparison allows Emerald to elegantly separate implementations from types. Emerald not only allows different implementations to be used for the same type within a single execution of a program,

[2]If there were no primitive abstract types, it would be possible for a programmer to define two types in terms of each other: getSignature on A returns B, and getSignature on B returns A. The use of signatures rules out such definitions.

```
const RepositoryCreator ←
    immutable object RC
        export new, getSignature
        const Repository ←
            type RT
                operation Store[n : String, v : Any]
                function Contains[n: String] → [v: Any]
            end RT
        function getSignature → [r: Signature]
            r ← Repository
        end getSignature
        operation new → [aRepository: Repository]
            aRepository ←
                object aRep
                    % as in Figure 1
                end aRep
        end new
    end RC
```

Figure 8: A Repository Creator with *getSignature*

but also permits conforming implementations to be used. This also helps provide the flexibility found in untyped languages such as Smalltalk, but within the framework of a strongly-typed language. Newly defined objects can be added to a running system, with these new objects used in place of previously defined objects, as long as they conform properly.

For example, the new version of a repository shown in Figure 9 still conforms to type *Repository* in Figure 6 of the original repository in Figure 1 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.

Allowing multiple implementations of the same type could introduce the following problem. The following operation:

```
operation Append[b: File]
```

defined for objects of type *File* has to operate correctly no matter what the implementation of *b*. This problem arises because *b* is also of type *File* but could be implemented differently from the object on which *Append* has been invoked. In Emerald, this problem is circumvented if all operations have been typed properly with best-fitting types (see Section 3.2), and the multiple implementations are truly equivalent, that is, they have exactly the same set of required operations.

Modula-2 appears to provide similar separation between abstract typing and implementation via the use of `DEFINITION` and `IMPLEMENTATION MODULEs`, however, the language description [Wirth 83] treats these as two parts of one module, thus not allowing such separation. The coupling of the definition and implementation

```
const aRepository ←
    object aRep
        export store, contains, empty
        monitor
            var name: String ← nil
            var value: Any ← nil

            operation store[n: String, v: Any]
                name ← n
                value ← v
            end store

            function contains[n: String] → [v: Any]
                if n = name then
                    v ← value
                else
                    v ← nil
                end if
            end contains

            operation empty → [r: Boolean]
                r ← name = nil
            end apply
        end monitor
    end aRep
```

Figure 9: Another Repository

modules is done outside the type system, thus precluding 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 too does not support coexisting multiple implementations of the same abstract type.

Object-oriented languages such as C++ and Smalltalk also allow multiple implementations of the same abstraction via the use of *abstract* classes, which have *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 only an encouraged but not enforced style of programming, while the latter represents a language feature that ensures proper programming style. Second, abstract classes permit the definition of default behaviour; although such definitions are occasionally useful, they also violate the complete separation both among the different multiple implementations, and between the type (i.e., abstract specification) and the implementations.

3.5 Support for Polymorphism

At least three broad varieties of polymorphism have been identified in the literature [Milner 78, Donahue & Demers 85, Cardelli & Wegner 85, Danforth & Tomlinson 88]. We list them below, pointing out that the first two are considered to be *true* or *universal* polymorphism.

inclusion	This refers to the situation where an object can be viewed as belonging to many different classes that need not be disjoint, i.e., a class may include one or more of the other classes. Subtyping is a good example of this kind of polymorphism, with objects belonging to a type being manipulable as belonging not only to that type, but also to its supertypes; objects are thus considered as belonging to several types. In implementation terms, object representations are chosen suitably so that operations can work uniformly on instances of subtypes and supertypes.
parametric	Here, a polymorphic function has either an 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* object or function to be used in different contexts without requiring changes in implementation. Implicit type parameters are used in ML [Milner 78], and are characterized by the uniform behavior of the same function of over arguments of different types; this variety has been called *functional* polymorphism [Danforth & Tomlinson 88]. Explicit type parameters appear in languages such as Russell [Donahue & Demers 85].
ad-hoc	This kind of polymorphism covers those situations where a procedure works, or appears to work, on several types. It is usually not considered to be true polymorphism because

neither are these types required to exhibit any common structure nor 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 for the addition to performed correctly); the difference between these two notions often disappears in untyped languages.

While inclusion polymorphism naturally fits into object-oriented programming, relating other traditional discussions of polymorphism has to be done carefully. For example, consider the following standard example of polymorphic function:

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

This kind of polymorphism is unnatural in object-oriented languages since functions such as this do not exist on their own; we do not think of applying functions to objects, rather we invoke operations on objects. If the length of an object is needed, one would invoke the *length* operation on that object to obtain the result; the object 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, we ask the list what its length is. In this setting there is no need for *length* to be polymorphic. By defining such functions suitably, i.e., high-up in the class hierarchy, and permitting subclass inheritance, several object-oriented languages support polymorphism.

While Emerald provides an efficient, practical form of polymorphism, its approach is unusual stemming from the interaction of these features: (1) complete separation of typing from implementation, (2) type conformity, and (3) treatment of types as objects. Notice also that operations (and other intra-object entities) belong to the objects themselves rather than to types. While these notions are flexible enough to support various forms of polymorphism, they also make it difficult for Emerald to use the above classification scheme directly. We propose our own classification by examining the role played by each polymorphic entity:

1. *polymorphic operations*, which work "correctly" regardless of the actual types of their arguments,
2. *polymorphic types*, which are defined using type variables, and
3. *polymorphic objects (values)*, which can be used in different situations requiring different types.

We show below that Emerald does support all these forms of polymorphism.

3.5.1 Polymorphic Operations

Emerald supports inclusion polymorphic operations as a direct result of the notion of *conformity.* An operation with parameters is naturally polymorphic since objects (of different types) may be used as parameters provided they conform properly to the types of the formal parameters (see Section 3.2).

```
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, eh?"]
      end process
   end SPT
```

Figure 10: A Polymorphic Operation

Figure 10 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. The use of conformity permits any object that understands the *asString* operation to conform to the type *Printable*, thus permitting any such object to be passed to *PrintIt* with the desired result. The body of the object *anObject* shows how objects of very different types: **Integer**, **Real**, **Boolean** and **String**, are printed using the same operation *PrintIt*.

3.5.2 Polymorphic Types

Polymorphic languages are also expected to support incompletely specified types, i.e., they should permit types to be declared (and implemented) without having to completely specify all the type information. A general example from most programming languages is that of the *array* type, which exhibits behavior independent of the type of its elements. An incompletely specified array may (typically) at compile-time and (sometimes) during execution can be *specialized* into becoming an array of integers, or an array of strings, etc.

While languages such as Pascal support this kind of polymorphism for arrays and similar built-in types, such support is not usual for other 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+ [Rovner et al. 85] provide a mechanism that allows elements of any type to be used, but whose *actual* use is guarded by a user-specified run-time check. Other languages such as ML [Milner 78] allow the specification of types using *type* variables that need to be instantiated when type-checking is done.

```
const List ←
    immutable object aListCreator
        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 New, getSignature

                    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 aListCreator
```

Figure 11: A List Creator with an Incompletely Specified Type.

Emerald supports this kind of polymorphism, subject only to the constraint that all type expressions must be evaluated at compile-time. Since types are objects, we can pass types as arguments to functions that create types. As an example, consider the polymorphic list creator in Figure 11. List is an object that exports a single function, *Of*. This function takes a type (*eType*) as a parameter, and returns a *List creator* (similar to the *Repository* creator in Figure 3) for lists whose elements are of type *eType*. This resulting class-like object can be used to create lists of arbitrary *eType*. For example, a list of integers can be declared and created as follows:

var *iList*: *List.Of*[**Integer**] ← *List.Of*[**Integer**].*New*

This statement is a good example of the expressiveness of Emerald invocations. *List* is an object that provides the operation, *Of* , whose single argument must be an abstract type. Invoking *Of* , with **Integer** as the parameter, on *List* results in the unnamed object

List.Of[**Integer**]

that is similar to the *Repository creator* in Figure 8. There are two ways in which this object is used. First, when used to specify the abstract type of *iList* , the denoted type is obtained by performing a *getSignature* invocation on *List.Of*[**Integer**]. Second, when used to initialize *iList*, the *New* invocation is performed yielding an object that is a list of **Integer**s. What is unusual about the *Of* invocation is that it is a compile-time invocation, i.e., the Emerald compiler itself performs this invocation and creates the resulting object. The compile-time nature of such invocations also means that the resulting list *iList* here is just as efficient as one that is directly defined; in fact, exactly the same code would be generated in either case.

It is interesting to examine the various Emerald features that went into the construction of *List* in Figure 11. The function *Of* uses the fact that types are objects to get a type parameter. The separation of type description and implementation allows implicit restriction of the behavior of the elements in the list. 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.

3.5.3 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 (e.g., Pascal) usually support the notion of **nil**, which may be used for any pointer variable regardless of its type. This is an example of a polymorphic value.

Again by virtue of conformity and the clean separation between type and implementation, Emerald supports polymorphic objects. An identifier may name any object that conforms to its declared type, and an object may be assigned to any identifier provided that the object's type conform to that declared for the identifier. For example, in Figure 12 the identifier *aPrintableObj* may name any object that understands

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

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

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

Figure 12: Polymorphic Objects

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. In these languages, all the classes that an object can belong to are known before the instantiation of the object. On the other hand, operations defined for a class should also work correctly for objects belonging to its subclasses. Standard class-based inheritance also creates the need for so-called **virtual** functions of Simula and C++ so that operation lookup can be done at run-time to ensure that the operation closest in the class hierarchy is used. This problem does not arise in Emerald because operations belong to objects, and types are completely separated from the implementation. Finally, Emerald objects are polymorphic not merely with respect to existing entities, but also to hitherto undefined entities (subject of course to the conformity requirement).

3.6 Subtypes and Inheritance in Emerald

Inheritance usually means at least the following: (1) subtype inheritance, and (2) code sharing and reuse. While Emerald has extensive support for the first variety of inheritance, its support for the second form is sketchy at best.

Subtype inheritance in Emerald refers to the sharing of external interface specifications between different types. We usually consider a type S to be a subtype of type T if and only if $S \circ\!> T$. For facilitating comparison with other languages such as Trellis/Owl, we sometimes loosely state that S *inherits* its interface from T, or that T is the parent type of S, but it must be borne in mind that there is no explicit inheritance structure in Emerald, and that we are merely commenting on the structure of the conformity relationship. This contrasts with the traditional view of subclassing, where classes and their subclasses often share part of

Figure 13: Subtype Inheritance in Emerald

their implementations, but not necessarily their interfaces. For example, it would be possible in class-based languages such as Simula for a subclass to implement an operation with a different signature than that in its superclass.

The lack of an explicit subtyping notion means that objects and types only display this "subtyping" after their creation. Figure 13 shows the nature of this subtyping, with the arrows pointing from (parent) types to (child) types. Although this may look similar to the class or type hierarchy in Smalltalk or Trellis/Owl, note that the arrows in the figure merely indicate conformity. The fact that these arrows are dotted emphasizes the lack of an explicit hierarchy here, that is, subtyping is based solely on the conformity of interface specifications. Thus, while conformity does not establish explicit connections between types, it does establish (theoretically) formal connections between them. Conformity, and hence subtyping, in Emerald is organized into a lattice structure; the bottom element is the type *Any*, with no operations, and the top element is the type *None*, which has all (including contradictory) operations (see [Black & Hutchinson 89] for more details).

Emerald does provide a limited form of code sharing between objects of differing types. This is possible because the Emerald system generates only one instance (per machine) of executable code for each object

Figure 14: Code Sharing in Emerald

constructor, regardless how many times it is used to create objects. Additionally, each object implementation can support all the types it conforms to. To obtain sharing, the Emerald programmer needs to first create the most generalized implementation, and then derive different objects as required by type restriction, i.e., reducing the number of visible operations. For example, we can execute the same object constructor (defined to create a *Deque*) thrice to create three objects: one being the *Deque*, one restricted to a *FiFo* (first-in, first-out) object, and one restricted to a *LiFo* (last-in, first-out) object (see Figure 14). Thus, all three objects share the same implementation and exhibit the same behavior for the common operations. Note that LiFos, Fifos, and Deques with the same abstract types (and behavior) may also be implemented without any code-sharing merely by using different object constructors.

This support for code-sharing has been found to be inadequate. Since the reuse of implementations for constructing new objects is beneficial, we are currently working on an Emerald extension that will permit us to overcome this shortcoming [Raj & Levy 89]. However, it is worth noting that such implementation reuse is not without cost, particularly in terms of encapsulation. The complete encapsulation provided by Emerald objects is due in part to its lack of support for code sharing between classes and their subclasses. In Smalltalk, the implementation of an object is protected from external manipulation, but is revealed for manipulation by subclasses.

4 Other Features

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

```
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 15: A Dining Philosopher

4.1 Support for Concurrency

Emerald's support for concurrency is motivated by its intended target environment: distributed systems. Since multiple processors exist, we must expect multiple activities to be executing concurrently. This in turn requires individual objects to be concerned with concurrency since they must be able to cope with simultaneous invocations from objects on other nodes.

An Emerald object may contain a *process* that is started after the object is initialized, and executes in parallel with invocations of that object's operations. This process continues to execute its specified instructions until it terminates. An object with a simple process is shown in Figure 15; the process performs the repetitive actions of a philosopher in the well-known dining philosophers' problem [Dijkstra 68].

While Emerald was not designed with a shared-memory architecture in mind, there is nothing in its design that prohibits it from running on such an architecture. The programming style would be no different than any other concurrent programming language such as Mesa [Lampson & Redell 80] or Modula-3 — whenever an independent thread of control is needed, it can be set up in a process of an Emerald object. For example, in Figure 16, the process represents a new thread that may be used to create the instances of the philosopher objects. Of course, Emerald, as should all object-oriented languages, has non-shared memory semantics.

For high-level support of concurrency, modern programming languages provide processes (alternatively named threads or tasks), and some synchronization mechanism such as *monitors* [Hoare 74], or the Ada *rendezvous*, or the CSP `!` and `?` [Hoare 78]. Emerald uses the former approach, a choice reflecting designer prejudice and familiarity rather than its superiority over the other mechanisms.

The monitor available to Emerald objects permits them to regulate access to the state shared between multiple simultaneous invocations and the object's own process. This construct is similar to the monitor

```
% Assume type Philosopher has been suitably defined.
const anOrganizer ==
   object Organizer
      process
         var count : Integer ← 5
         const philoArray ← Array.of[Philosopher].create[5]
         loop
            philoArray[count] ←
               as in body of Figure 15
            count ← count − 1
            exit when count = 0
         end loop
      end process
   end Organizer
```

Figure 16: The Organizer Object

construct in Concurrent Pascal [Brinch Hansen 79] or Concurrent Euclid [Holt 83], but is completely enclosed within an object. An object's process executes outside the monitor, but can invoke monitored operations when it needs access to shared state. Synchronization is possible via the use of system-defined **Condition** objects. Figure 17 shows the text-book example of how the Emerald monitor can be used to provide exclusive access to a resource.

4.2 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 definitions of objects; for example in Figure 11, the object constructor *theList* is 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 declared within each nested block. The scoping of Emerald identifiers is generally traditional, but one exception to that is an identifier name is visible throughout the scope in which it is declared, not just textually after that declaration. Constructs such as operation definitions, object and type constructors, monitors, loop statement bodies, etc., open new scopes for identifiers, and non-local identifiers are implicitly imported into nested scopes (unless re-defined).

Since object constructors and type constructors create new independent objects that persist beyond the end of the enclosing block, identifiers imported into these constructs are specially treated. All imported identifiers are evaluated at the time the type or object constructor is executed. It is therefore illegal to assign a new name to an imported **var** identifier inside an object constructor.

Consider the example in Figure 18, where the loop creates ten objects identical except that the identifier i names different integers in the different objects. Once the first object (for which i is 0) is created, changes

```
const Resource ==
   object aResource
      export Acquire, Release
      monitor
         const available == Condition.create
         var beingUsed: Boolean ← false
         operation Acquire
            if beingUsed then
               wait available
            end if
            assert (! beingUsed)
            beingUsed ← true
         end Acquire
         operation Release
            beingUsed ← false
            signal available
         end Release
      end monitor
   end aResource
```

Figure 17: Using the Emerald Monitor

```
var i : Integer ← 0
var o : Any
loop
   exit when i >= 10
   o ←
      object trivial
         export getI
         operation getI → [r : Integer]
            r ← i
         end getI
      end trivial
   i ← i + 1
end loop
```

Figure 18: Block Structure and Object Constructors

to the loop control variable i are not visible to it because its i is bound to the object i named when the first trivial object was created. Of course, if the object named by i was mutable, the state of that object may change.

Emerald blocks therefore have several similarities with those in Algol-60, and hence, Simula and Beta. Smalltalk abandoned block-structure in favor of facilitating a single-level declaration of classes and thus allowing global access to class definitions. Since multi-level class definitions are not permitted, the Smalltalk programmer cannot declare a class inside another class. Unlike all these languages, Emerald does not support classes, but it permits object constructors to be nested as deeply as desired by the programmer.

There has been considerable discussion in the literature both supporting and opposing the use of block structure in programming languages [Hanson 81, Tennent 82, Madsen 87]. By ensuring locality, block structure permits the seclusion of objects to the context in which they make sense, thus emphasizing good programming style. This restricted access reduces reusability of objects, which is being addressed in a proposed Emerald extension.

4.3 Syntax

Finally, we cannot resist mentioning the syntactic aspects of Emerald. In recent years with a better understanding of parsing techniques, the choice of syntax has usually come to represent the prejudices of the language designers rather than any significant improvement. This has meant that discussions of syntax are generally considered minor and unworthy of comment. Nevertheless, we find some aspects of the Emerald syntax interesting enough to merit a couple of paragraphs here.

The reader must have been pleasantly surprised (after the semicolon wars fought between the Terminators and the Separators) to find the absence of semicolons in Emerald — semicolons are not needed to terminate statements (as in Ada) or to separate statements from one another (as in Pascal). Of course, it makes practical sense to follow the traditional precepts of indentation and (usually) keeping one statement to a line, as has been done in the examples discussed in this paper. 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 assignment statement in Emerald uses the $\leftarrow$ symbol ($<-$ on the terminal). As part of its distributed programming heritage, Emerald clearly distinguishes between the use of input (value) parameters and output (result) parameters, with the $\rightarrow$ symbol emphasizing this distinction. We believe that this approach should be used in future programming languages. Since the number of results from an invocation can generally be more than one, the $\leftarrow$ naturally permits multiple assignments such as:

$x,\ y,\ z\ \leftarrow\ point.getCoordinates$

$a,\ b\ \leftarrow\ c,\ d$

5 Current Status and Future Research

Currently, Emerald does not have much more support for programming than that provided by Unix. The Emerald system neither supports any kind of object browser nor does it have any full-fledged debugging facility.

The success of Smalltalk (and Trellis/Owl to a lesser extent) clearly demonstrates that an excellent programming environment can overcome the shortcomings, if any, of the programming language. Shortcomings in the programming environment can prove to be burdensome on a programmer. While 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 these handicaps, the rapid development of various applications (such as a prototype mail system and a multi-user calendar system) reveals the usefulness of the Emerald language features.

As already mentioned, the focus of our current work is on improving implementation sharing and software reuse in Emerald, along with a full user interface.

6 Summary

We have presented and discussed a number of examples that bring out the flavor of the Emerald approach. Emerald is not just the sum of the parts, it is also the *interaction* of those parts. A example of this is how the Emerald programmer would build the class-like entity from Section 2. This construction is possible through Emerald's use of the object constructor together with Emerald's nesting and block structure as noted in section 4.2. Another example is how 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 (1) its emphasis on abstract typing and compile-time type-checking, (2) the clear separation of types and implementations, and (3) the use of simple (and unique) mechanisms for each identifiable paradigm, that is, the elimination of extra functionality for each concept. For example, Emerald has the *object constructor* for object creation, the *abstract type* for typing, *conformity* for the classification hierarchy, and the object model itself acts as the repository for methods. This *unbundling* of the functions of Smalltalk classes has been found appealing by other researchers [Danforth & Tomlinson 88].

This paper has compared the programming approaches used in Emerald with those in several other languages. Emerald provides an excellent fusion of modular and object-oriented languages, supports different kinds of polymorphism, provides concurrency, strongly supports typing and subtyping, separates types and implementations excellently, and has an efficient implementation. We therefore believe that Emerald provides an excellent alternative approach to programming.

Acknowledgments

Eric Jul was also part of the original Emerald team at the University of Washington, and his ideas have obviously influenced this work. We thank Alan Borning, Bjorn Freeman-Benson, Bill Griswold, John Maloney and Jeff Chase for their feedback and comments. Thanks also to Rich Pattis and Rick Zucker for their help on Ada.

This work was supported in part by the National Science Foundation under Grants MCS-8004111, DCR-8420945 and CCR-8700106, and by the Digital Equipment Corporation External Research Program. Norman Hutchinson is supported in part by the National Science Foundation under Grant CCR-8701516.

References

[Ada 83] United States Department of Defense. *Reference Manual for the ADA Programming Language*, 1983.

[Birtwistle et al. 73] Birtwistle, G., Dahl, O.-J., Myrhaug, B., and Nygaard, K. *Simula Begin.* Petrocelli/Charter, 1973.

[Black & Hutchinson 89] Black, A. P. and Hutchinson, N. The Lattice of Data Types (Or: Much Ado About Nil). In Preparation, 1989.

[Black et al. 86] Black, A., Hutchinson, N., Jul, E., and Levy, H. Object Structure in the Emerald System. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 78–86. ACM, October 1986.

[Black et al. 87] Black, A., Hutchinson, N., Jul, E., Levy, H., and Carter, L. Distribution and Abstract Types in Emerald. *IEEE Transactions on Software Engineering*, 13(1), January 1987.

[Borning & O'Shea 87] Borning, A. H. and O'Shea, T. Deltatalk: An Empirically and Aesthetically Motivated Simplification of the Smalltalk Language. In *Proceedings of the European Conference on Object-oriented Programming*, June 1987.

[Borning 86] Borning, A. H. Classes Versus Prototypes In Object-Oriented Languages. In *ACM/IEEE Fall Joint Computer Conference*, November 1986.

[Brinch Hansen 79] Brinch Hansen, P. The programming language Concurrent Pascal. *IEEE Transactions on Software Engineering*, 1(5):50–56, May 1979.

[Cardelli & Wegner 85] Cardelli, L. and Wegner, P. On Understanding Types, Data Abstraction, and Polymorphism. *Computing Surveys*, 17(4):471–522, December 1985.

[Cardelli et al. 88] Cardelli, L., Donahue, J., Glassman, L., Jordan, M., Kalsow, B., and Nelson, G. Modula-3 report. Technical Report #31, Digital Equipment Corporation Systems Research Center, Palo Alto, California, August 1988.

[Danforth & Tomlinson 88] Danforth, S. and Tomlinson, C. Type Theories and Object-Oriented Programming. *Computing Surveys*, 20(1):29–72, March 1988.

[Dijkstra 68] Dijkstra, E. W. Cooperating Sequential Processes. In Genuys, F., editor, *Programming Languages.* Academic Press, 1968.

[Donahue & Demers 85] Donahue, J. and Demers, A. Data Types are Values. *ACM Transactions on Programming Languages and Systems*, 7(3):426–445, July 1985.

[Freeman-Benson 87] Freeman-Benson, B. N. Anyone can understand MetaClasses. Technical Report 87-11-02, Department of Computer Science, University of Washington, Seattle, Washington, November 1987.

[Goldberg & Robson 83] Goldberg, A. and Robson, D. *Smalltalk-80: The Language And Its Implementation.* Addison-Wesley Publishing Company, 1983.

[Hanson 81] Hanson, D. R. Is Block Structure Necessary? *Software Practice and Experience*, 11:853–866, 1981.

[Hoare 74] Hoare, C. A. R. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557, October 1974.

[Hoare 78] Hoare, C. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[Holt 83] Holt, R. C. *Concurrent Euclid, The Unix System, and Tunis.* Addison-Wesley, 1983.

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

[Hutchinson et al. 87] Hutchinson, N. C., Raj, R. K., Black, A. P., Levy, H. M., and Jul, E. The Emerald Programming Language Report. Technical Report 87-10-07, Department of Computer Science, University of Washington, Seattle, Washington, October 1987. (Revised August 1988).

[Jul et al. 88] Jul, E., Levy, H., Hutchinson, N., and Black, A. Fine-grained Mobility in the Emerald System. *ACM Transactions on Computer Systems*, February 1988.

[Kristensen et al. 87] Kristensen, B. B., Madsen, O. L., Moller-Pedersen, B., and Nygaard, K. The BETA Programming Language. In Shriver, B. and Wegner, P., editors, *Research Directions in Object-Oriented Programming*, pages 7–48. The MIT Press, Cambridge, Mass., July 1987.

[Lampson & Redell 80] Lampson, B. and Redell, D. Experience with Processes and Monitors in Mesa. *Communications of the ACM*, 23(2):105–117, February 1980.

[Madsen 87] Madsen, O. L. Block Structure and Object-Oriented Languages. In Shriver, B. and Wegner, P., editors, *Research Directions in Object-Oriented Programming*, pages 113–128. The MIT Press, Cambridge, Mass., July 1987.

[Milner 78] Milner, R. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[Raj & Levy 89] Raj, R. K. and Levy, H. M. A Compositional Model for Software Reuse. Technical Report 89-01-04, Department of Computer Science, University of, Washington, Seattle, Washington, January 1989.

[Rovner et al. 85] Rovner, P., Levin, R., and Wick, J. On Extending Modula-2 For Building Large, Integrated Systems. Technical Report # 3, Digital Equipment Corporation Systems Research Center, Palo Alto, California, January 1985.

[Schaffert et al. 86] Schaffert, C. et al. An Introduction to Trellis/Owl. In *Proceedings of the First ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, Portland, Oregon, September 1986.

[Stroustrup 86] Stroustrup, B. *The C++ Programming Language.* Addison Wesley, March 1986.

[Tennent 82] Tennent, R. D. Two Examples of Block Structure. *Software Practice and Experience*, 12:385–392, 1982.

[Ungar & Smith 87] Ungar, D. and Smith, R. B. 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.

[Wirth 83] Wirth, N. *Programming in Modula-2.* Springer-Verlag, Berlin, 1983.