Distributed Eiffel: A Language for Programming Multi-Granular Distributed Objects on the Clouds Operating System *

L. Gunaseelan and Richard J. LeBlanc, Jr.
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280

Abstract

This paper presents the design and implementation of Distributed Eiffel, *a language designed and implemented for distributed programming, on top of the Clouds operating system by extending the object oriented language Eiffel. The language presents a programming paradigm based on objects of* multiple granularity. *While large-granular persistent objects serve as units of distribution, fine-grained objects are used to describe and manipulate entities within these units. The language design makes it possible to implement both* shared-memory *and* message-passing *models of parallel programming within a single programming paradigm. The language provides features with which the programmer can declaratively fine-tune synchronization at any desired object granularity and maximize concurrency. With the primitives provided, it is possible to combine and control both* data migration *and* computation migration *effectively, at the language level. The design addresses a number of issues such as parameter passing, asynchronous invocations and result claiming, and concurrency control.*

1 Introduction

Distributed computing involves a set of physically distributed, heterogeneous processors working together towards achieving a common computation goal. It is, however quite difficult to program such systems. Anyone wishing to write a distributed application has to be concerned with issues such as: how to partition and structure such inherently complex applications, how to exploit concurrency, and how to distribute the computation.

In recent years *objects* have been recognized as a powerful modeling technique because such an abstraction allows one to concentrate on writing one object at a time with a well-defined interface specification. Objects are suitable for distributed programming because objects can be *distributed.* It is possible to exploit concurrency between and within objects. The success of a programming language lies in how effective it is in providing abstractions that are both powerful and natural to the programmer.

*This work is funded by NSF grant CCR-8619886.

In this paper, we discuss the design and implementation of *Distributed Eiffel,* a language that allows distributed applications to be modeled as a collection of large-grained and fine-grained objects on top of the *Clouds* operating system. *Clouds*[5] is a distributed operating system developed at Georgia Tech which provides direct operating system support for programming large-grained objects. *Eiffel*[1] [12] is a modern object-oriented language that emphasizes construction of reliable and reusable software components as "structured collections of abstract data type implementations". We discuss extensions we have made to the Eiffel language to address the issues of programming distributed systems. Specifically, the extended language presents a programming paradigm based on objects of *multiple-granularity*:

- While large-grained objects serve as higher-level abstractions as well as units of distribution, fine-grained objects allow lower-level entities to be defined and manipulated.
- Objects of multiple granularity serve as natural synchronization boundaries. It is possible to lock and synchronize at any desired granularity and thus extract as much concurrency as possible.
- Very large-grained objects allow both *shared memory* and *message-passing* models of programming to be combined.
- With large-granular objects, it is possible to integrate both data migration (Distributed Shared Memory) and computation migration (Remote Procedure Call) effectively.

Towards this end, we have extended the Eiffel language to provide support for programming large-grained persistent objects and to allow expression of distribution and concurrency. The design addresses issues such as parameter passing, synchronization and control of the computation modes. The choice of Eiffel as a base language has enabled extensions to be incorporated with relative ease. For example,

[1] Eiffel is a trademark of Non-profit International Consortium for Eiffel (NICE).

- Eiffel's notion of *"classes as the only types"* has enabled the extension of the programming model in a clean and natural way.
- Eiffel's lack of pointer variables and its uniform *reference semantics* have enabled passing of arbitrarily complex fine-grained objects as parameters during invocation between large-grained objects.
- Eiffel's lack of *public* variables has enabled incorporation of concurrency control mechanism at the object level.
- Eiffel's separation of declaration and storage-allocation for instance variables has enabled the addition of a basic operation (`bind`) for large-grained persistent objects.

We have developed a number of applications using Distributed Eiffel that have convinced us of the simplicity and expressive power of the Distributed Eiffel language, as well as of the viability of the Clouds approach to distributed computing.

In section 2 of this paper, we describe the distributed computing paradigm of the Clouds operating system and the support it provides for programming objects. Section 3 briefly introduces the Eiffel language and the Eiffel computation model. Section 4 describes Distributed Eiffel, the set of extensions which we have added to the original Eiffel language to enable programming distributed objects of multiple granularity. Section 5 discusses parameter passing issues. Section 6 discusses the support available for programming with the DSM & RPC modes of computation. In section 7 we discuss concurrency and synchronization support provided by Distributed Eiffel. Section 8 discusses our current implementation, which is running on top of the Clouds OS, and the programming experiences to date. Section 9 gives a programming example. In section 10, we discuss related work and compare it with Distributed Eiffel. In Section 11, we present conclusions and future work.

2 Clouds Distributed Computing Paradigm

The Clouds distributed computing paradigm is based on two basic notions: *objects* and *threads.* While the traditional meaning of objects is to regard them as pieces of memory, Clouds advances the notion of objects to entire program address spaces. The program address spaces are, more over, persistent in Clouds. A distributed application is structured around a collection of such persistent address spaces, possibly residing on different nodes. We call such persistent address spaces large-grained objects. We use the terms Clouds objects and large-grained objects interchangeably. Thus, Clouds objects are passive containers of code and data and the operating system provides support for location-independent invocation of the code in the objects.

A thread corresponds to a flow of control that executes the code within objects. A computation may be viewed as starting a thread in one of the objects. The thread may move across a set of objects through a chain of invocations to produce the desired results. An active thread within an object may invoke another object, synchronously or asynchronously. An application can have more than one thread, some of which may be active within the same object, while some of them are active in different objects. Also, several different applications may be concurrently active within the same object. The operating system enforces strict encapsulation, that is, only the code within an object can manipulate the associated data.

Since large-grained Clouds objects correspond to persistent address spaces, the size of the objects may be of the order of gigabytes. As the real world also consists of entities best modeled as fine-grained objects such as points, triangles, stacks, trees, linked-lists etc., the notion of treating everything as a large-grained object is not satisfactory. This leaves the language system with responsibility of providing support for fine-grained objects. The persistent nature of objects means that they retain their states across invocations. The operating system provides this view of the long-lived nature of objects by maintaining suitable stable stores transparently. An object, once created, lives forever and is always available for invocation, as long as it is not explicitly deleted. Persistence has a bearing on the language design, as the language should provide facilities for accessing pre-existing objects for which storage has been allocated in some previous execution or even by another program.

Thus, at the language level we deal with two different kinds of objects: *large-grained* and *fine-grained.* Large-grained objects are separate address spaces that serve as units of distribution. Fine-grained objects are internal to large-grained objects. Both large-grained and fine-grained objects provide a well-defined interface to the programmer. The large-grained objects interact with each other by means of *invocations,* while the interaction between fine-grained objects is by local *procedure-calls.* Both invocations and procedure-calls can be either *synchronous* or *asynchronous.* In the case of synchronous invocation/procedure-call the calling thread blocks until completion, while in the case of asynchronous invocation/procedure-call, the call proceeds in parallel as a new thread along with the calling thread.

3 Eiffel Computation Model

Distributed Eiffel supports programming with the following kinds of objects:

- Value objects such as integers, reals, characters, booleans etc.
- Fine-grained objects such as points, strings, stacks, complex numbers etc.
- Large-grained Clouds objects that are units of distribution.

The Eiffel language defines objects of type 1 and 2 above. Eiffel is described in detail in references[12, 8]. To introduce the Eiffel computation model, we will consider the class BOOK below, which can be instantiated to create objects of type BOOK.

```
class BOOK
export issue_to, title, author, ...
feature
        title:STRING;
        accession_no:INTEGER;
        author:PERSON;
        issued:BOOLEAN;
        ...
        issue_to (member:  PERSON) is
                do
                        ...
                end;
        ...
end -- class BOOK

class PERSON
export pname, age ...
feature
        pname:STRING;
        age:INTEGER;
        Create(nm:STRING;ag:INTEGER) is
                do
                        ...
                end;
end -- class PERSON
```

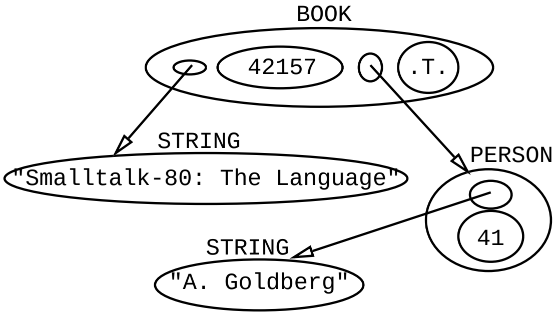

Figure 1: Eiffel Computation Model

Eiffel defines the notions of *basic types* and *references*. Specifically a variable x in a program may be either one of the four basic types (integer, real, character and boolean) or a reference to another Eiffel object. In the example given above, attributes accession_no and issued are of types integer and boolean respectively. Attributes author and title are references to other Eiffel objects (Figure 1).

Thus, the Eiffel computation model consists of a set of fine-grained objects each possibly having a set of references to other fine-grained objects. The model is very dynamic because these objects may be created on the fly and may quickly become garbage.

The Eiffel language defines five basic operations for object creation and instantiation. They are Create, Forget, Equal, Clone and Void. For instance, the operations for author are:

1. author.Create creates an object of type PERSON and makes author refer to it.
2. author.Forget frees the storage allocated to author and makes it void.
3. author1.equal(author2) checks whether individual fields of author1 and author2 are equal.
4. author1.Clone(author2) creates an object author1 and makes the contents (attributes) of author1 same as that of author2.
5. author.Void checks whether author is void (i.e. NULL).

All reference objects (non-basic types) need to be created by an explicit call to Create. This means a declaration of the form author:PERSON does not allocate storage for the person object. An explicit call of the form author.Create(...) is required to allocate the necessary storage.

4 Distributed Eiffel

Having presented the basic Eiffel computation model, we now describe how Eiffel has been extended to provide for programming both large-grained and fine-grained objects.

In Distributed Eiffel, a Clouds object is defined to contain

- Value objects like integers, reals etc.
- References to fine-grained (Eiffel) objects
- References to other Clouds objects. A Reference to a Clouds object is really the sysname of the object, as seen by the Clouds OS. The sysnames are capabilities that provide an ability to invoke the corresponding Clouds object.

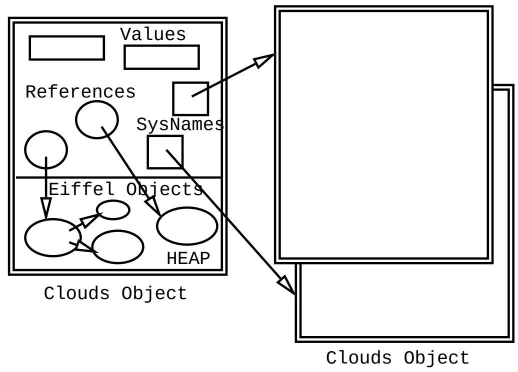

Figure 2: Distributed Eiffel Computation Model

Thus an application may be viewed as consisting of a collection of Clouds objects, each of which may contain a number of fine-grained Eiffel objects internal to them (Figure 2). The Clouds object itself provides a set of entry-points (routines) which are invocable via the OS. Typically, a programmer writes a number of Clouds class descriptions, a set of Eiffel class descriptions (for all the fine-grained objects referred to in the Clouds class descriptions) and then assembles the application system. At run-time, there may be one or more instances of the Clouds class objects, each of which may contain instances of any of the fine-grained objects. The Clouds OS starts a thread in one of the Clouds objects, as desired by the programmer, and the thread may span across any number of Clouds objects.

Our extensions to standard Eiffel allow a programmer to define and use large-grained objects. As the large-grained Clouds objects are persistent, the programmer can create a new large-grained(LG) object or use an existing one. For this purpose, we have added a new basic operation, called *bind*, which allows the programmer to bind a program variable to a preexisting large-grained object and invoke it. The following example describes how a Clouds object is declared and used.

Definition:
```
Clouds class MAILER
export send_mail, deliver, ...
feature
        ...
end
```
Declaration:
```
m1::MAILER; -- Note the use of "::"
```
Usage:
```
send_mail(receiver:USER; msg:MESSAGE) is
do
        ...
        -- find receiver host's MAILER instance
        m1.bind("..");
        m1.deliver(msg); -- Invocation
        ...
end;
deliver(msg:MESSAGE) is
do
        -- put msg in users mail box
end;
```

In the above example, an instance of class MAILER is a persistent Clouds object. It is assumed there will be one such instance for every node. A particular instance, on receiving a request to `send_mail` to a remote user, finds the instance corresponding to the destination host (possibly by a table lookup), binds the variable `m1` (of type MAILER) to the name of the remote instance and invokes it. The `m1.deliver(..)` call is an invocation, which results in network activity. The remote instance puts the message in the local user's mailbox when invoked with operation `deliver`.

As seen in the above example, large-grained object definitions start with the keyword `Clouds`. An LG object variable is declared with a double colon (::), as in `m1::MAILER`, and any operation on a LG object results in an invocation. It is also possible to create a new LG object using the predefined operation `Create` as in:

```
m1::MAILER;
..
m1.create("..");
```

Since it is possible to invoke a preexisting LG object, there is a name server running on the Clouds OS which remembers the sysname of the objects. The create operation also implicitly registers the newly created object with the name server, with its user-level name as the key. The bind operation is essentially a name-server lookup to translate the user-level name into the sysname of the object.

An invocation is a remote procedure call that blocks until the remote procedure is executed and the results are returned. It is also possible to do a non-blocking invocation on remote object that allows a thread to proceed concurrently with the calling thread. We call such invocations asynchronous invocations. Asynchronous invocations have the following syntax:

```
m1::MAILER;
...
m1.bind("..");
m1!deliver(receiver,msg);
...
```

There are basically two ways by which multiple threads can become active within a large-grained object:

- first, multiple threads entering from outside (threads coming from another address space as the result of invocations) and executing within the large-grained object.
- second, threads created as the result of asynchronous procedure-calls on fine-grained objects. These threads execute within the address space of the containing large-grained object.

5 Parameter Passing and Object Mobility

There are a number of issues that need to be addressed with regard to parameter passing when an invocation is made from one Clouds object to another. A programmer must be able to pass objects of any kind, be they of basic type, Eiffel objects or Clouds objects.

The objects of basic types such as `INTEGER`, `REAL` etc. can be passed as values. Similarly, the Clouds objects are passed simply by passing their sysnames. Since no Clouds object contains another, but only has references to others, these references can be easily passed around. The problem comes when we want to pass the fine-grained Eiffel objects between Clouds objects during invocations. Since Eiffel objects are actually references to some place in the heap, it is meaningless to pass such a reference across address spaces to another Clouds object.

There are basically two approaches to solving this problem. One is to make a copy of the object and all its referenced objects and send the object structure to the other node. The other is to retain the object in the calling object and send a unique object-id to the other object, with a tag signifying that it is a reference to an internal object in the calling Clouds object. When the called object needs to access the parameter object, it must do so with the assistance of the calling object.

No mechanism can be regarded as 'the best', since when choosing a mechanism, one has to consider factors such as the size of the object being passed, the invocation frequency on the passed object and whether single copy semantics is required or not. Thus, this choice should not be made by a language designer. The following three parameter passing modes are supported in Distributed Eiffel. It is up to the programmer to choose the appropriate mode for the parameters supplied to an invocation. If the programmer does not specify a mode, the language chooses the `COPY` mode by default.

`COPY` (->) In this mode, the dynamic graph structure of the object (with all the referenced objects, taking into account any circularity) is flattened and passed as a single array of bytes. In the receiving object, it is unflattened, the object is reconstructed in the local dynamic memory and the entry point is called with a reference to the local object.
e.g. `spooler.printfile(->fileobj);`

`COPY OUT-COPY IN` (<->) This scheme is basically similar to the above scheme except that it allows for copying back of the passed objects. This gives the user an ability to get any changes made to the passed object to be reflected in the calling object.
e.g. `customer.update(<->creditDB);`

`PIN` (><) In this scheme, as explained before, the object is retained in (or pinned to) the calling site and only a reference is passed to the other site. This scheme allows the *single copy* semantics to be maintained when it is needed. Also if the object structure is very large and the invocation frequency on it is low, then this mechanism may prove to be more efficient.
e.g. `airportDB.update_yourself(><flightDB);`

It may be noted that here, unlike conventional programming languages, the parameter passing modes are specified at the calling point rather than the declaration point. That is, the caller, rather than the callee, determines the parameter passing mode (this decision was influenced by a desire to simplify our preprocessor implementation). Our current implementation supports the first two modes, viz. `COPY` and `COPY OUT-COPY IN`. We are yet to implement the `PIN` mode. The implementation of the PIN mode will necessitate handling distributed garbage in the system.

6 DSM & RPC

Distributed shared memory(DSM)[9] and remote procedure calls(RPC) [2] are two alternative approaches to distributed computing. In the DSM approach, when a large-grained object is invoked, if it is not available in the current node, it is paged in via the network from a remote machine. In the RPC approach, when a large-grained object is invoked the thread migrates to the node where the object is currently residing. While DSM corresponds to object (data) migration, RPC corresponds to thread (computation) migration. Both the schemes have their relative merits. The Distributed Shared Memory Server(DSMS) of the Clouds OS can make the object available on different nodes simultaneously by a form of network paging, while at the same time maintaining coherence.

When a large-grained object is invoked the programmer can optionally specify the node where the computation should be performed. The exact syntax is as follows:

`Object.routine(..)@(<node_id>);`

Examples:

`Object.routine(..);` DSM mode, when node is not specified. That is, object should be brought to the current node and invoked (*Object migration*).

`Object.routine(..)@(0);` *node_id* 0 implies RPC mode. Object should be invoked in the node where it is currently residing (*Thread migration*).

`Object.routine(..)@(i);` Perform the computation on the virtual processor i. The object will be brought to virtual processor i and invoked (possibly both *Object and Thread migration*).

As shown in the example above, the programmer may also optionally distribute the computation among virtual processors. Virtual processor numbers start at 1 and may be arbitrarily large. The runtime system performs the mapping between the virtual nodes and the actual processors (which may be a many to one mapping, depending upon available number of processors).

In order to get several threads executing within the same LG object on different processors, the programmer can write:

```
from
        i:=1
until
        i>5
loop
        object!operation(..)@(i);
        i:=i+1
end
```

The object `object` may be either fine-grained or large-grained. If `object` is fine-grained all the new threads proceed within the same address space (of the of the containing large-grained object). If `object` is large-grained, all new threads start execution in the remote address space of `object`.

It is possible to program using both *shared memory* and *message passing* models of programming in Distributed Eiffel/Clouds.

- The programmer, by creating several threads running on different nodes within the same LG object, can get the *shared memory* model of programming. The different threads share the same address space of the LG object, but execute on different processors.

- The programmer can also program using the *message passing model* of programming by partitioning the computation into separate objects on different nodes and letting the objects invoke each other.

In fact, the programming model presented by Clouds does not distinguish between the two approaches, and is presented as a collection of large-grained objects and threads, all of which can be located in different nodes dynamically.

7 Concurrency & Synchronization

There are two forms of concurrency support provided by the language: i) Concurrency between objects ii) concurrency within an object.

7.1 Concurrency between Objects

Since large-grained objects may reside on separate nodes, concurrent threads of the same application may execute on different objects at the same time. The language provides constructs for making asynchronous invocations as well as asynchronous procedure-calls. A thread executing within an object may do an asynchronous(non-blocking) invocation on a remote object. Or it may do an asynchronous procedure-call on a local fine-grained object. While the invocation/procedure-call proceeds as a new thread the current thread may proceed concurrently. The thread may choose to claim the results of such an asynchronous invocation/procedure-call at a later time. For this, we provide a mechanism called *Promises*.

To asynchronously invoke an object we use the following syntax:

```
...:= Object!operation(...);
```

An asynchronous invocation returns a typed promise which can be used to claim the results. A *promise* is a type safe handle that holds a "promise" for the result of an asynchronous invocation. The following example illustrates the use of `promises`:

```
hi:PROMISE[INTEGER]; --typed promise
i:INTEGER;
hl:PROMISE[LINKED_LIST[REAL]]; --typed promise
l:LINKED_LIST[REAL];

routine is
local
     x:EIFFELOBJ;
     y::CLOUDSOBJ;
do
     ..
     hi:=x!doit(...); --async.  Procedure-Call on x
     y.bind("...");
     hl:=y!doit(...); --async.  Invocation on y
     ..
     i:= hi.claim; --Claim result (integer)
     l:= hl.claim; --Claim result (linked list of reals)
end
```

The class `PROMISE` is defined in the standard library and provides operations such as `claim` and `ready`. The asynchronous invocation/procedure-call always returns a promise of result-type. A significant point about this promise implementation, compared to promise mechanisms discussed in [10], is that in our implementation, promises can be passed around as parameters to other objects (on other machines) and the original thread that did the asynchronous invocation need not even be alive to claim the result. This means a thread other than the original thread can do the claim. If more than one thread does a claim on the same promise handle, only one of them will succeed (in the current implementation) and the other thread will receive an error code.

Also, to enable programming with a group of asynchronous threads, a PSET (Promise-Set) object has been defined. More details are available in [7].

7.2 Concurrency within an Object

Several threads may concurrently execute within an object. In fact, threads belonging to different applications or to the same application can execute concurrently within the same object. Since the large-grained objects are passive, an object is potentially available for invocation by many applications, possibly on different processors or on the same processor. To address the issue of synchronization among multiple threads that potentially access or update the state information within the object (both large-grained and fine-grained), the language provides a multiple reader/single writer model of synchronization which can be combined with conditional synchronization. A conditional synchronization scheme allows a thread to wait for some condition to become true. The exact syntax is given below:

```
routine accesses|modifies|is
when
      <Boolean_expression>
do
      ...
end;
```

A programmer can optionally tag a routine as a reader (`accesses`) or a writer (`modifies`). He may also provide an optional *when clause* which specifies a boolean expression. A thread invoking the entry point acquires a suitable read or write lock and, if the boolean expression in the *when clause* is true, proceeds to execute the body. If it is not true, it releases the lock and blocks itself until the condition becomes true. When waken up it tries to reacquire the lock and tries the boolean expression again. A null when clause is assumed to correspond to `true`. Following is the definition for the bounded_buffer class.

```
class BOUNDED_BUFFER[T]
export append,remove
feature
     buf:ARRAY[T];
     count:INTEGER; -- initial value 0
     head,tail:INTEGER; -- initial value 0
     size:INTEGER;
     create (sz:INTEGER) is
          do
               size:=sz; buf.create(size)
          end;
     append(element:T) modifies
          when
               count < size
          do
               head:=head mod size + 1;
               buf.put(element,head);
               count:=count+1
          end;
     remove:T modifies
          when
               count > 0
          do
               tail:=tail mod size + 1;
               Result:=buf.item(tail);
               count:=count-1
          end
end
```

A routine may have one of the following tags : `is`, `accesses`, or `modifies`. It may or may not have a *when clause*. Hence the following six combinations are possible. Depending on its tag and whether or not it has a *when clause*, a routine may or may not sleep until some condition becomes true, may acquire read or write locks, and may or may not wake up other waiting threads. For example, a routine that only accesses(reads) the object state need not wake up other threads, as it will not change the object state. However, a routine that modifies state always has to wake up any waiting threads since some condition that they are waiting for may have become true. In the following table, the labels S, Lr, Lw, W indicate what code is generated by the compiler for synchronization.

	is	accesses	modifies
with WHEN	Error	S,Lr	S,Lw,W
without WHEN	–	Lr	Lw,W

S -code to check for condition and sleep after releasing the lock.

L -code to acquire and release lock (Lr-read lock, Lw- Write lock).

W -code to wake up other waiting threads.

It is assumed that a routine with the `is` tag neither accesses nor modifies the state of the object directly. It can access or modify the state only by calling other routines that have a proper reader/writer tag. For example, in a banking application a `do_menu` routine (with `is` tag) displays a menu, takes a request from the user and then calls the appropriate routine to display the balance (`accesses`) or to deposit/withdraw amount (`modifies`).

We provide `LOCK` and `SEMAPHORE` classes in the standard library to allow for custom synchronization. More details are available in [7]. While the conventional *region construct* (for conditional critical sections) specifically locks the named resources, the *when construct* treats the fine-grained object (or the large-grained object), in which the *when clause* is occurring, as a single resource. There is a shift in the abstraction here:

```
class A                 Conventional region statement:
export ...
feature
    v1:TYPE1;           v1:  shared TYPE1;
    v2:TYPE2;           v2:  shared TYPE2;

    routine modifies
    when
        B
    do                  region v1,v2 when B do S end;
        S
    end
end
```

Here, instead of trying to lock the variables v1 and v2 (as it is done in the *region construct*), we lock the fine/large-grained object (i.e. the instance of A) containing the references v1 and v2. Any synchronization needed in accessing v1 and v2 should be done in their class definitions (that is, in TYPE1 and TYPE2). Actually, with Eiffel's reference semantics, it does not make sense to lock the references, as more than one object may have references to any object. The object, not the clients of the object, should be responsible for controlling access to its state. The main idea in object-oriented programming being distribution of such responsibility, the semantics proposed here are consistent with the object-oriented approach. It is possible to push this idea further to get synchronization at any desired granularity.

Figure 3 shows the code for the dining philosophers problem in Distributed Eiffel.

```
class STICK
export pick_up, put_down
feature
     busy:  BOOLEAN; -- initially false
     pick_up modifies
     when
          not busy
     do
          busy:=true
     end;
     put_down modifies
     do
          busy:=false
     end;
end
class PHIL
export doit
feature
     left,right:STICK;
     create(l,r:STICK) is
     do
          left:=l; right:=r
     end;
     doit is
     local
          forever:BOOLEAN
     do
          from -- nothing
          until forever
          loop
               left.pick_up; right.pick_up;
               -- (eat here)
               left.put_down; right.put_down;
               -- think
          end
     end
end
Clouds class DINING
export
     doit
feature
     sticks:array[STICK];
     phils:array[PHIL];

     doit is
     local
          i:INTEGER;
          ph:PHIL;
     do
          -- initialize the sticks and
          -- phils data-structures suitably.
          from
               i:=1
          until
               i>5
          loop
               ph:=phils.item(i);
               ph!doit; -- start a thread
               i:=i+1
          end
     end;
end
```

8 Current Implementation and Application Experience

8.1 Implementation

Presently we have implemented Distributed Eiffel by a combination of a preprocessor and postprocessor technology. User programs in Distributed Eiffel are translated into standard Eiffel. The translated program contains a number of calls to the Distributed Eiffel runtime system and the Clouds kernel. The standard Eiffel code is processed through the Eiffel compiler to generate the equivalent C language program. Finally the object code is post-processed by a postprocessor which, from the *.obj* files, constructs the virtual address space of the Clouds object and writes out the segments. From then on, the DSM (distributed shared memory) server of Clouds OS manages these segments. Within each object, a parameter buffer is allocated on a per-thread basis, where the parameters of the invocation are stored and passed to the invoked object, along with other run-time information such as window handles, parameter types, etc.

Every Clouds object has a built-in garbage collector that collects the fine-grained objects in the heap (actually this garbage collector comes with the standard Eiffel language). We are in the process of fully integrating the Eiffel exception handling scheme into Distributed Eiffel.

8.2 Applications

We have developed several applications using the Distributed Eiffel language. Many small applications including a banking example (discussed in the next section), dining philosophers problem, the traveling salesperson problem, the list-ranking problem, and prime number generation have been developed. One of the large applications developed was a block-diagram editor that allows graphical entities to be described as hierarchical blocks in different layers and enables drawing and editing of these entities on the X-window user interface. When fully developed, the application should allow cooperative editing of graphical entities by different users simultaneously. At present, the application only makes use of the large-granularity and persistence. The graphical entities input by the programmer and their inter-relationships are stored within a large-grained object and persistence avoids storing and loading of information in subsequent editing sessions.

Another large application developed is the simulation of Time-Warp mechanism. This application consists of a set of logical processes (LPs) that exchange time-stamped messages with each other. The simulation scheme uses an optimistic synchronization strategy based on the theory of virtual time, that relies on roll-back and message cancellation.

While the applications like list-ranking, prime-number generation, time-warp simulation and banking employ the message passing model of distributed programming, the applications like dining-philosophers, traveling sales person and the block-diagram editor employ a shared memory approach. One of the interesting things we discovered was that it is possible to incrementally modify programs and smoothly move towards real parallelism. For example,

- The programmer can initially write and test a program using only synchronous invocations and later on, with minor syntactic changes, make them asynchronous invocations. The first case can be the test and debug mode where all executions happen sequentially and the second case may be the real execution mode, where parallelism is employed.
- Similarly, when employing shared memory programming, the programmer can have all the threads executing on the same processor during testing and debugging, and later modify the program to have threads running on different processors.
- Even when objects located on different nodes invoke each other, the programmer can have all the invocations happen in a single node (the DSM mode brings all the objects to the same node), and later on distribute the computation. The point is for testing and debugging we do not need many processors. Because the system provides such a uniform interface, all development and testing can be done on a single processor.

For instance, the Time-Warp application consisted of three classes: `TW` (time-warp), `LP` (logical processor) and `MESSAGE`. We originally structured the application as a single-object(large-grained), single-node concurrent application and then incrementally restructured it as multi-object, multi-node distributed application. We were able incrementally modify the program and run it in three different ways (as given below) during the successive development stages:

- With `TW` as a large-grained object and the `LP`s as a set of fine-grained objects internal to it. The `LP`s asynchronously exchange messages with each other (asynchronous procedure-calls). There was only one large-grained object (`TW`) and all the threads belonging to `LP`s were run on the same processor within the containing large-grained object `TW`.
- Same as above, except that the threads belonging different `LP`s running on different processors, though within the same large-grained object `TW`.
- A multi-object implementation with both `TW` and the set of `LP`s modeled as large-grained objects residing on different nodes and `MESSAGE`s as the fine-grained objects exchanged between the `LP`s. The threads within the `LP`s were run on the nodes wherever the `LP`s were residing.

It was possible to make the above incremental modifications, with very few syntactic changes. Testing and development time was spent only in developing the first version, where algorithmic bugs (those not involving race conditions) were fixed to get the program running correctly on one node. Later restructuring to make it a fully distributed application was one of making few simple syntactic changes.

9 Programming Example

We show below how a banking system can be modeled in Distributed Eiffel. In this example, a `BRANCH` is a large-grained persistent object, that maintains a collection of `ACCOUNT`s. Each branch allows operations like `open`, `deposit`, `withdraw`, etc. on its accounts. The application itself consists of a number of such large-grained branch objects, possibly distributed geographically on different nodes. Each branch object consists of a number of fine-grained objects such as `PERSON`, `ACCOUNT`, `ACCNUM` etc.

The `PERSON` is a fine-grained object that has features such as name, age, address, etc. When an account is opened in a branch, the user gets an account number (`ACCNUM` object) which consists of a branch-id and a serial number.

```
class PERSON
export pname,age,address
feature
    pname:STRING;
    age:INTEGER;
    address:STRING;
    create(nm:STRING;ag:INTEGER;add:STRING) is
        do
            pname:=nm; age:=ag; address:=add
        end
end

class ACCNUM
export branch_id,sno
feature
    branch_id,sno:INTEGER;
    Create(bno,snum:INTEGER) is
        do
            branch_id:=bno;sno:=snum
        end
end
```

The ACCOUNT is also a fine-grained object which has an owner and the current balance. It allows operations such as deposit or withdraw on a particular instance of account.

```
class ACCOUNT
export open, deposit, withdraw, balance, owner
feature
     balance:REAL;
     owner:PERSON;
     open(client:PERSON) modifies
          do
               balance:=0.0; owner:=client
          end;

     deposit(amt:REAL):REAL modifies
          do
               balance:=balance+amt; Result := balance
          end;

     withdraw(amt:REAL): REAL modifies
          do
               balance:=balance-amt; Result := balance
          end
end
```

A BRANCH consists of a collection(ARRAY) of such ACCOUNTs. Also a branch knows about other branches. When a large-grained branch object processes a transaction (a deposit or a withdrawal), it checks the branch-id to see whether the account number belongs to this branch. If so, it does a local procedure call on the corresponding account object in its local collection of accounts. If the account does not belong to the local branch, then the branch forwards the request to the appropriate branch. This forwarding is actually an invocation on the other branch object.

Thus this application consists of the following objects:

1. BRANCH implemented as a large-grained Clouds object.
2. ACCOUNT, PERSON and ACCNUM implemented as fine-grained Eiffel objects.
3. balance, branch_id etc. as value objects.

It may be noted that the accesses/modifies attributes are specified in ACCOUNT and BRANCH objects in such a way that locking occurs at the individual account level (the routines deposit and withdraw have modifies tag in the ACCOUNT class and accesses tag in the BRANCH class). This means two deposit/withdraw transactions can go on concurrently within a branch, as long as they are on different accounts. The modifies tag on open in the BRANCH class gives it exclusive access to the object. Thus, by suitably labeling the operations, the programmer can fine-tune synchronization and achieve as much concurrency as possible.

```
Clouds class BRANCH
export open, deposit, withdraw, transfer , balance
feature
   accounts:ARRAY[ACCOUNT];
   nextaccount:INTEGER;
   thisbranch:INTEGER;
   branches:ARRAY[STRING];
   init(branch_no:INTEGER,branch_info:ARRAY[STRING]) is
   do
      thisbranch:=branch_no; branches:=branch_info;
      accounts.Create(1,25)
   end;
   open(client:PERSON):ACCNUM modifies
   local
      acct:  ACCOUNT
   do
      acct.create; acct.open(client);
      nextaccount := nextaccount+1;
      accounts.put(acct, nextaccount);
      Result.Create(thisbranch,nextaccount);
   end;
   deposit(account_no:ACCNUM,amount:REAL):REAL accesses
   local
      acct:ACCOUNT;
      nextbranch::BRANCH;
      nextname:STRING
   do
      if account_no.branch_id=thisbranch then
         acct:=accounts.item(account_no.sno);
         if not acct.Void then
            Result := acct.deposit(amount)
         end
      else
         nextname:=branches.item(account_no.branch_id);
         nextbranch.bind(nextname);
         Result := nextbranch.deposit(account_no,amount)@(0)
      end
   end;
   withdraw(account_no:ACCNUM,amount:REAL):REAL accesses
   do
      -- similar to deposit
   end;
   balance(acc:ACCNUM): REAL accesses
   do
      ...
   end;
   ...-- code for other routines
end
```

10 Related Work

There are many languages developed for programming distributed systems. Of these Emerald, Orca and Argus are closest to our work.

Emerald[3] is a language and system for programming distributed systems. While Emerald supports both fine-grained and course-grained objects, such a distinction is made by the Emerald compiler rather than by the programmer. Emerald is a class-less language and object creation is by declaration only. Emerald's support for concurrency takes the form of active objects. Each object can have an active process within it. The only way to create many threads is by creating many objects. Synchronization support is provided by means of monitors with *wait* and *signal* operations on condition variables.

Both Argus and Orca use *call-by-value*(COPY) parameter passing mechanism similar to Distributed Eiffel. Emerald uses a novel parameter passing mechanism called *call-by-move*. This allows fine-grained parameter objects to be moved to a remote site, while maintaining a reference in the calling site. This introduces the problem of maintaining forwarding addresses when fine-grained objects move across a series of sites. Object mobility in Clouds takes a different form. For large-grained objects, the DSM system allows the object (or segments of the object) to be demand-paged at the node of invocation. For fine-grained objects, as discussed before, we support the COPY, COPY-OUT

`COPY-IN` and `PIN` modes of parameter passing. In Emerald, the reason for having a *call-by-move* mechanism is to ensure single copy semantics. A mechanism like *call-by-move* for fine-grained objects in the context of Clouds is inappropriate because, in Clouds, objects typically live longer than the applications. Applications(or computations or threads) last only briefly and we do not want one-time applications causing movement of fine-grained objects across a system of large-grained persistent objects. This is the rationale behind the provision of `PIN` parameter passing mode, which ensures single-copy semantics and allows only references to fine-grained objects to be passed. Moreover, this *call-by-move* approach essentially precludes any asynchronous invocations on remote objects, as the fine-grained parameter objects can not be moved simultaneously to two different sites.

Argus[11] is a language and system for fault-tolerant distributed applications. Unlike Clouds, Argus objects are active and applications are modeled as a collection of active servers called guardians. While Argus supports a strict transaction-based approach for preserving consistency of data, the approach being taken by Clouds, called Invocation-based Consistency Control(IBCC)[4] allows a variety of consistency semantics to be enforced, including strict transaction-based semantics. The support system for consistency maintenance of persistent objects in Clouds is being developed and will be integrated into Distributed Eiffel shortly. Interestingly, the Avalon/C++[6] language implemented by extending C++, uses inheritance to provide for recovery and synchronization.

Orca is a language[1] designed to support high-performance parallel applications by providing a shared-memory model of programming on top of non-shared memory architectures. The only way to define a shared object in Orca is to pass it as a parameter to a child process. Our concurrency control mechanism is similar to the Orca mechanism. Orca specifies two syntactic constructs for operations. A programmer wanting conditional synchronization should write the entire routine as a guarded statement. While the Orca language compiler automatically detects whether a routine is a reader or a writer, in Distributed Eiffel, we let the programmer specify the `reader/writer` tag for the operations. This allows the synchronization to be done at any desired finer-granularity (as in the Banking example). This also allows mixing of `is` tags with `reader/writer` tags as mentioned before.

11 Conclusions and Future work

The distributed Eiffel language presents a programming paradigm that allows distributed applications to be modeled as a collection of distributed address spaces and fine-grained objects. Like fine-grained objects, the distributed address spaces too have an object-like interface which, we believe, paves the way for building open systems of the future. Programs can be structured as a set of cooperating objects that invoke each other or as a set of threads that execute within the address space of a large-grained object or as a combination of both. The language provides higher-level control for both computation and data migration, with very few syntactic extensions. The concurrency mechanism is consistent with the object oriented approach, where each object manages its own affairs.

Further extensions to Eiffel will be defined to address issues of consistency and fault tolerance. The support mechanism for consistency of thread and object states, in the presence of possibly failing computations, is being developed for Clouds and the language will integrate features for specifying the degree of global and local consistency desired by the programmer. The Aeolus language[13], designed and implemented to work with an earlier version of Clouds, provides some of the required features. However, Aeolus does not support the dynamic fine-grained objects available in Eiffel.

References

[1] H.E. Bal, A.S. Tanenbaum and M.F. Kaashoek, *Orca: A Language for Distributed Programming*, SIGPLAN Notices, Volume 25, Number 5, May 1990.

[2] A.D. Birrel and B.J. Nelson, *Implementing Remote Procedure Calls*, ACM Transactions on Computer Systems, 2, 1, Feb 1984, pp 39-59.

[3] A. P. Black, N.C. Hutchinson, B.C. McCord and R.K. Raj, *Distribution and Abstract types in Emerald*, IEEE Trans. on Software Engineering, 13, 1, Jan 1987.

[4] Raymond C. Chen and Partha Dasgupta, *Linking Consistency with Object/Thread Semantics: An Approach to Robust Computations*, In proceedings of the 9th Intl. Conf. on Distributed Computing Systems, June 1989.

[5] Dasgupta et. el., *The Design and Implementation of Clouds Distributed Operating Systems, USENIX Computing Systems Journal*, Vol. 3, No. 1, 1990.

[6] David L. Detlefs et. el., *Inheritance of Synchronization and Recovery Properties in Avalon/C++*, IEEE Computer, December 1988.

[7] *Distributed Eiffel: User Document*, Distributed Systems Laboratory, Georgia Tech, November 1990.

[8] Eiffel: The Language, *Manual TR-EI-17/RM*, Interactive Software Engineering Inc., version 2.2B, 1989.

[9] Kai Li and Paul Hudak, *Memory Coherence in Shared Virtual Memory Systems*, ACM Transactions on Computer Systems, 7, 4, Nov 1989, pp 321-359.

[10] B. Liskov and L. Shrira, *Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems*, SIGPLAN 88 Conference on Programming Language Design and Implementation, June 1988.

[11] Barbara Liskov and Robert Scheifler, *Guardians and Actions: Linguistic Support for Robust, Distributed Programs* ACM Transactions on Programming Languages and Systems, Volume 5, Number 3, July 1983.

[12] Bertrand Meyer, *Object Oriented Software Construction*, Prentice Hall, 1988.

[13] C.Thomas Wilkes, Richard J. LeBlanc, *Rationale for the design of Aeolus: A Systems Programming Language for an Action/Object System*, Proceedings of the 1986 International Conference on Computer Languages, Miami Beach, Florida, Oct 1986.