Amadeus Project

Eiffel**: An Implementation of Eiffel on Amadeus, a Persistent, Distributed Applications Support Environment

Colm McHugh and Vinny Cahill
mail:{dcmchugh,vjcahill}@cs.tcd.ie

Distributed Systems Group
Department of Computer Science
University of Dublin
Trinity College, Dublin 2, Ireland.
Fax: +353-1-6772204

Document Status Published
Distribution Public
Document # TCD-CS-93-36
Publication TOOLS Europe '93 Conference Proceedings, pages 47-62

© 1993 University of Dublin

Permission to copy without fee all or part of this material is granted provided that the copyright notice, and the title and authors of the document appear. To otherwise copy or republish requires explicit permission in writing from the University of Dublin.

Eiffel**: An Implementation of Eiffel on Amadeus, a Persistent, Distributed Applications Support Environment

Colm McHugh and Vinny Cahill
mail:{dcmchugh,vjcahill}@cs.tcd.ie

Abstract

*Eiffel** is an implementation of Eiffel which provides support for distribution, persistence, concurrency and transactions. All objects in an Eiffel** system are global (i.e. accessible from nodes other than that at which they are currently located) and persistent (i.e. their lifetimes are not bounded by the duration of the program that created them). Some objects may also be atomic (i.e. accesses to these objects within atomic transactions provide the well-known transactional properties of atomicity, consistency, isolation and durability in the face of concurrent execution and partial failures). Eiffel** is supported by the Amadeus distributed application support platform. In this paper we describe the Eiffel** language and its implementation on Amadeus. We believe that the combination of the Eiffel programming model and the support provided by the Amadeus platform provide a useful environment for the construction of sophisticated distributed applications.*

1 Introduction

Eiffel** is an implementation of Eiffel which provides support for distribution, persistence, concurrency and transactions. Eiffel** is supported by the Amadeus distributed application support platform [Horn 91] [Cahill 91]. Amadeus is intended to support the use of existing (object-oriented) languages for the construction of persistent, distributed applications. To date, Amadeus supports an extended version of C++ [DSG 92] (which we call C**) and Eiffel**. Work is underway to support the E programming language [Richardson 89] for data intensive applications. A prototype of the Guide [Decouchant 88] language for distributed programming is also supported.

Amadeus consists of two main components: the *Generic Runtime* (GRT), which provides common, generic support for the management of distributed and persistent objects, and the *kernel* which provides the underlying support for the GRT including persistent storage, distributed processes, remote invocation, and nested transactions. Rather than forcing each supported language to adopt a common object model and execution structures in order to exploit the platform, the GRT provides the support required in common by a range of language implementations and uses *upcalls* to request language-specific information or actions where these are necessary. *Language-Specific Runtimes* (LRTs) can then be built above the GRT for each language to be supported.

All objects in an Eiffel** system are global (i.e. accessible from nodes other than that at which they are currently located) and persistent (i.e. their lifetimes are not bounded by the duration of the program that created them). Some objects may also be atomic (i.e. accesses to these objects within atomic transactions [Berstein 87] support the well-known transactional properties of atomicity, consistency, isolation and durability in the face of concurrent execution and partial failures). The control of concurrency and transactions is supported through the use of library classes. Overall our approach was motivated by the desire to support reuse of existing software.

The implementation of Eiffel** involved extending the Eiffel runtime (ERT) and providing a preprocessor for Eiffel** programs. The extensions to the ERT were necessary to adapt the support provided by Amadeus to the Eiffel language and object model and to make available the language-specific information required by the Amadeus GRT. The preprocessor is used to generate supplementary code used by the

GRT to trap accesses to remote and atomic objects. The implementation of Eiffel** has been possible without making changes to the Eiffel language, enabling persistence and distribution to be transparent to the programmer. Moreover only minimal changes and extensions to the ERT were necessary. No changes have been made to the Eiffel compiler. This has been made possible by the extent of the run time type information provided by Eiffel, which enables all objects to be treated in a uniform manner.

1.1 Outline

The remainder of the paper is organised as follows. Section 2. describes the GRT and gives an overview of its interface. Section 3. provides some background concerning the ERT which is necessary to appreciate the requirements for supporting the ERT on top of the GRT. Section 4. presents the Eiffel** language, describing the programmer's view of distribution, persistence, concurrency and transactions and giving some simple examples. Section 5 discusses the implementation of Eiffel**. Finally, Section 6. provides a summary and some conclusions.

2 The Amadeus Generic Runtime

The GRT is the part of Amadeus concerned with the management of global, persistent and atomic objects. The GRT supports object creation and naming, mapping (loading) and unmapping (storing) of objects, marshalling and dispatching of invocation requests and garbage collection. It performs these actions in a language independent manner. Whenever language specific information or actions are required, the GRT makes an *upcall* to code supplied by the language. The following sections give a brief overview of the GRT. The interested reader is referred to [Cahill 93] for further details.

2.1 GRT Objects

A GRT object consists of a GRT header and a block of memory. GRT objects can be global, persistent and/or atomic as required. A language may use GRT objects to enclose language level objects. As far as the GRT is concerned, an object is a block of memory which can be uniquely identified. The GRT knows nothing about the internal structure or semantics of an object. Where language objects are enclosed in GRT objects, the layout of the object is governed by the language's runtime system and compiler. The format of object references used to access language objects is dictated by the language and need not be changed from the format usually used by the language to reference objects. This is particularly important since it means that the language's native invocation mechanism can be used to invoke such an object from another object located in the same context (address space).

In Amadeus a global object mapped on a given node may have a reference to it in a context on another node. In this case the reference will actually refer to a *G proxy* for the object. Invoking on this proxy will usually result in a *Remote Procedure Call* (RPC) to the real object being carried out. The code bound to the G proxy is responsible for initiating such RPCs and must be provided by the language. This *proxy class* should provide the same public interface as the real class code.

Persistent GRT objects are grouped together in *clusters* when on secondary storage. A reference to a persistent object which is not currently mapped refers to a *P proxy* (i.e. essentially an area of access-protected memory) for the object's cluster[1] All attempts to access such an object will then result in its cluster being mapped and overlaying its P proxy before the access proceeds.

Accesses to atomic objects must also be reported to the GRT so that concurrency and recovery control operations can be carried out before the access proceeds. For example, the necessary locks must be acquired. This can be achieved by using an alternate version of the class code for atomic instances - the so-called *atomic class code* - which reports the access before carrying out the operation as normal.

Each GRT header contains a *stub* for the object which contains a system-wide unique name for the object, its cluster identifier and infor-

[1] Accesses to objects which are both global and persistent are trapped using G proxies whenever the object is not located in the current context.

mation to allow the creation of a proxy for the object. Also contained in the GRT header is an *upcall* structure which points to the code implementing each of the upcalls for that object.

The GRT maintains a *class descriptor* for every class in an application which contains a copy of the upcall structure to use for instances of the class, as well as the class name, the size of instances of the class and a class identifier. The creation of a class descriptor for each class used in an application takes place in `regclasses`, a language-specific function that is called by the GRT as part of its initialisation. The class descriptor is used when creating new GRT objects.

2.2 The GRT/LRT Interface

The GRT supplies the routine to allocate a new global or persistent object which should be used in place of a language's object allocation mechanism. The GRT also provides a routine to promote a global or persistent object from non-atomic to atomic. Other downcalls are provided to control concurrency and transactions as well as to manage transparency and control the storage and clustering of objects.

In addition to this interface, the LRT must supply a set of *upcalls* for each object which the GRT calls through the object's upcall structure. The full set of upcalls which may be required includes:

- `create()` – Invoked when an object is created. Initialises language-specific bindings for the object (e.g. setting up of the function table).
- `activate()` – Invoked when an object is being made active, i.e. going from being a proxy object to being the real object. It performs this transformation by binding the real class code to the object. Usually invoked when an object is being mapped into a context.
- `deactivate()` – Invoked when an object is being made inactive. It binds the proxy class code to the object. Will be invoked when an object is being unmapped.
- `nextptr()` – Returns a pointer to the n^{th} reference in the object. Invoked when the object is being mapped or unmapped so the GRT can swizzle any pointers contained in the object.
- `norefs()` – Returns the number of pointers in the object. Invoked when the GRT is allocating space for the object in its cluster.
- `onuse()` – Same function as the `create` upcall. Invoked as the final step in making an object active.
- `dispatch()` – Invoked when an incoming RPC for the object is received. This upcall invokes the object specific `dispatch` function (which must be provided by the language) which unbundles the parameters and invokes the target operation.
- `make_atomic()` – Binds the atomic code (responsible for trapping access to atomic objects whether local or remote) to the object. Upcalled by the GRT as part of the process of making an object atomic.

As can be seen a language supporting global objects must provide a G proxy class for each class with global instances and add a dispatch routine to every such class. A language supporting atomic objects must also provide the atomic class code. Depending on whether global, persistent or atomic objects are to be supported some subset of the upcalls `create`, `onuse`, `nextptr`, `norefs`, `dispatch` and `make_atomic` are required for each object. `activate` and `onuse` are also required for G proxies.

3 The Eiffel Runtime

Eiffel is a strongly typed language. Every *entity* is declared to be of a certain type. An entity can be an instance of a *class type* or an instance of an *expanded type*. The run time value of an entity of a class type is a reference to an object of that class. The run time value of an entity of an expanded type is an actual object, rather than a reference to an object. An entity of an expanded type can be of one of the Eiffel *simple types*, or it can be an expanded (i.e. inline) instance of a class type. All objects, except those declared to be of simple types, contain information in the form of an object header which is used by the

ERT to manage the object and which is transparent to the Eiffel programmer. Contained in this information is a number, known as the *Dynamic Type* (DT) of the object, which is the same in all objects of a given class.

3.1 Object Layout

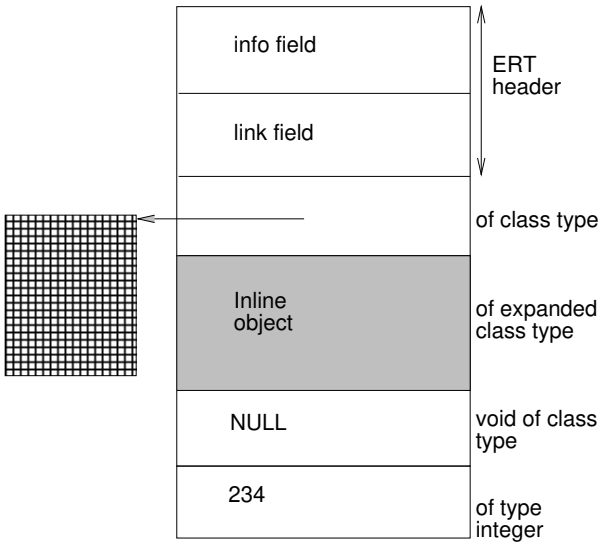


Figure 1: An Eiffel object

Figure 1 shows a snap-shot of an Eiffel object during system execution. Note that there are two fields in the header. The `info` field, contains the DT of the object and, amongst other things, indicates if the object is expanded or is a so-called special object (discussed below). The `link` field points to the last object allocated by the ERT and is used by the Eiffel garbage collector. In Eiffel, an object reference always gives the virtual address to the `info` field of the object being referenced, or is `void`.

The instance data consists of a number of fields which in turn are made up of one or more units called *datums*. A single datum can be a reference to another object, in which case it will contain a virtual address, or can contain an `INTEGER, REAL, BOOLEAN` or `CHARACTER`. A number of datums can go to make up an expanded object or value of type `DOUBLE` or `BITS`. If the expanded object is an instance of a class type, its ERT header will appear in the instance data of the enclosing object. If a class `B` inherits data from a class `A`, then instances of class `B` will be structured so that the ERT header will come first, then the data inherited from `A`, followed by the data for `B`.

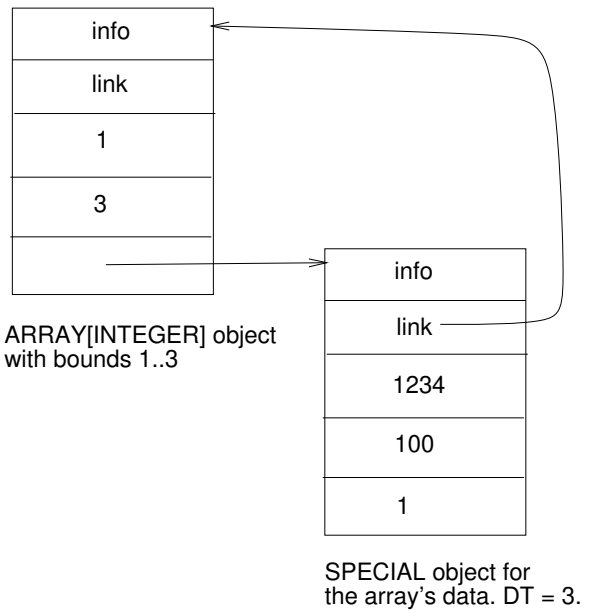


Figure 2: Special object in an Eiffel array

The ERT provides *special* objects, which are used to implement variable sized or generic objects. Figure 2 shows a snap-shot of an object of type `ARRAY[INTEGER]`. The array object has fields for the upper and lower bounds, as well as a pointer to the array elements, contained in a special object. The special object consists of fields for each of the elements in the array. This example shows how Eiffel implements genericity through the use of special objects, as special objects can contain objects of any type. Special objects are transparent to the programmer and are always associated with another object such as a `STRING` or an `ARRAY`.

3.2 ERT Data Structures for A Class

The following list details the data structures that are set up for each class in the system on ERT initialisation. Given the DT of the object to which a reference points, the following information about the object can be obtained:

- `Class_names[DT]` – The name of the object's class.
- `Object_size[DT]` – The number of fields in the object.

- `Num_routines[DT]` – The number of routines in the object's class.
- `Routines[DT]` – Gives a pointer to a function table for the object's routines. Invoking on an object always results in an indirection through this table.
- `Routine_names[DT]` – The names of the object's routines.
- `Attributes[DT]` – The types of the object's attributes. `Attributes[DT][i]` gives the type of the i^{th} field of the object using positive numbers for attributes of class types, giving the DT of the class, and negative numbers for inline objects. Note that in the case of attributes of generic and expanded types, it is necessary to look in the `info` field of the actual object to determine that attribute's DT.
- `Attr_names[DT]` – The names of the object's attributes.
- `Create_Array[DT]` – A pointer to the `Create` function for the object's class.

All classes in the system have entries in these tables. The range of DT goes from 0 to `num_classes` - 1, where `num_classes` is the number of classes in the system. The Eiffel code that the programmer writes for a class compiles into C code with functions for all the routines of the class, plus initialisation functions for the classes entries into the ERT structures described here. Eiffel compilation also generates a C mainline which invokes these initialisation functions before invoking the `Create` routine of the system's root class.

3.3 Object Creation, Access and Invocation

This section describes how object creation, data access and invocation take place at run time.

3.3.1 Object Creation.

At the language level, an Eiffel object is created and initialised as follows (where `x` is an attribute of the current object):

```
x.Create (args) ;
```

At the ERT level the DT of `x` is obtained by looking up the `Attributes` table for the current object's class. The space for the object is then created by passing the DT to `Allocate`. `Allocate` is an ERT function which, given a DT, will return a pointer to an initialised ERT header for the object, as well as the required memory space for the data. The invocation of the object's `Create` function is carried out by indirecting through the `Create_array` using the DT for `x`. The `Create` function will initialise the data space appropriately. If `x` is declared to be of an expanded class type, there is no need to call `Allocate`, as the space has already been allocated when the enclosing object was created. Special objects are allocated using a function called `spAllocate` which is given the number of datums required for the special object rather than the DT.

3.3.2 Object Access.

At the language level, the data members of an Eiffel object are accessed like so :

```
x.y ;
```

The data member `y` in `x` is located at some offset `os` from the start of the instance data for `x` where `os` is determined at compile time. At runtime, this offset is added to the address of the instance data to give the address of the required field which can then be accessed.

3.3.3 Object Invocation.

At the language level, an Eiffel routine is invoked as follows :

```
x.y(args) ;
```

At the ERT level the invocation involves indirecting through the routines table for the target object's class using the DT obtained from its header, to locate the appropriate routines table, and an offset in this table for the required function which is determined at compile time. The routine is passed a reference to the target object as well as the actual arguments.

4 The Eiffel**Language

4.1 Persistence

Eiffel already provides support for persistence through its class library. In particular, object persistence can be obtained in one of two ways, in the current implementation of Eiffel.

Class `STORABLE` offers a simple explicit facility to store an object (and its dependents) to a named file. Any class can make use of this facility simply by inheriting from class `STORABLE`. An instance `x` of such a class is explicitly stored via the call:

```
x.store_by_name("some_file") ;
```

and can subsequently be retrieved via

```
x.retrieve_by_name("some_file") ;
```

where these routines are inherited from `STORABLE`.

Alternatively persistence may be obtained by use of an Eiffel *environment*. An Eiffel environment is a set of objects. Individual objects may be identified by a key with respect to the environment. Such objects, and all their direct and indirect dependents, are the *persistent objects* of the environment. An environment may be opened. All objects created thereafter will belong to the environment (until it is closed). Hence, class `ENVIRONMENT` provides an implicit facility to store a collection of arbitrary objects. The environment as a whole is made to persist between Eiffel sessions, by storing an external representation of all the objects in a named file.

Note that, in both cases, when an object becomes persistent, all of its dependents are also stored or otherwise object references would, on retrieval, be meaningless. Both shared references and cyclic dependencies are handled properly. Object dependency highlights the orthogonality that exists between type and persistence. All classes have potentially persistent instances.

In Amadeus, all persistent GRT objects transitively reachable from a *root* object, an object that has been explicitly registered with the Amadeus environment, will automatically persist across successive program executions.

Three different approaches to exploiting the persistence facilities offered by the Amadeus platform in Eiffel** programs are immediately apparent.

- All Eiffel** objects are created as GRT objects and thus all objects reachable from a root (e.g. the Amadeus name service) automatically persist between different sessions. The support provided by an Amadeus system could allow the explicit object storing provided by Eiffel to become redundant. Any Eiffel** object which remains reachable from a root would then survive across program runs. The Eiffel** programmer would then no longer have to worry about explicitly ensuring that particular instances persist. Coupled with the generic garbage collection facilities provided by Amadeus this offers an elegant approach to long term safe object management.

- Incorporate into the Eiffel** (language-specific) runtime, routines to emulate the per-
  formance of `STORABLE` and `ENVIRONMENT`. Consequently, from the Eiffel** programmer's viewpoint, there would be no difference between Eiffel** and Eiffel with respect to persistence. This has the advantage that normal language semantics are preserved, but the added functionality inherent to Amadeus is not fully exploited.

- A mixture of the above, i.e. provide modified Amadeus implementations of `STORABLE` and `ENVIRONMENT` which essentially mimic the simple naming service that these provide while using the standard Amadeus facilities to provide persistence of individual instances. This would provide the Eiffel** programmer with conventional Eiffel persistence support and permit Eiffel** applications to directly use Amadeus transparent persistence.

In fact the third method outlined above was chosen since it combines the advantages of both of the other alternatives. In Eiffel** all objects are created as persistent GRT objects. This means that an Eiffel** object is *potentially* persistent and will persist if reachable from some root object.

As an example, the following code shows how to create a (potentially) persistent integer in Eiffel**:

```
class EXAMPLE_INT export inc, val
feature
    v : INTEGER ;

    Create (i : INTEGER) is
      do v := i end ;

    inc is
      do v := v + 1 end ;

    val : INTEGER is
      do Result := v end ;
end ; --Class EXAMPLE_INT

class ROOT
inherit
    AMADEUS
feature
    p : EXAMPLE_INT ;

    Create is
      do
        if reset = 1
        then
           p.Create (1) ;
           record ("p.ns", p)
        end ;
        p ?= lookup ("p.ns") ;
        io.putstring("p is ") ;
        io.putint (p.val) ;
        io.newline ;
        p.inc
    end ;
end ; --Class ROOT
```

Note the use of standard routine `reset`, inherited from `AMADEUS`, which tests if the `-reset` option has been passed to the program. By convention `-reset` is used to indicate the first execution of an application which expects to use persistent objects which may not have already been created. `record` and `lookup` are used to register and lookup an object in the Amadeus name service provided by the platform. `record` also designates the specified object as a persistent root.

When this is run several times (with an instance of `ROOT` as the Eiffel system root object) the following output is produced:

```
$root -reset
p is 1
$root
p is 2
$root
p is 3
$root -reset
p is 1
```

No additional code is generated to support persistence. Once a persistent object has been mapped in from secondary storage, there is no additional overhead attached to manipulating the object beyond that of Eiffel. An Eiffel** program that does not make use of any Amadeus facilities incurs no additional overhead beyond the equivalent Eiffel program, except for some extra space taken up by the headers attached to each GRT object in the system.

4.2 Distribution

The approach to distribution adopted allows all objects in an Eiffel** system to be remotely accessible. Thus, distribution, like persistence, is transparent to the Eiffel** programmer. All Eiffel** objects are created as persistent and global GRT objects.

Hence, the class `EXAMPLE_INT` in the previous section not only describes objects which are potentially persistent but which are also remotely accessible. Many users could run the integer program at the same time, possibly on different nodes, and it is completely transparent where the integer object is actually mapped. In this way, the integer object acts as a server capable of handling multiple client requests. Note however, that the Amadeus approach to supporting distribution through the use of RPCs means that it is not possible to access the exported data items of an object remotely. Access to a remote object can only be through the exported routines of its interface.

While distribution is normally transparent, the programmer can also exercise some control over the placement of objects and clusters at run time by specifying a preferred node at which objects and clusters are to be placed using routines exported by the class `AMADEUS`.

4.3 Concurrency

In Amadeus a *job* is a distributed process consisting of a set of *activities*. A job may be thought of as a distributed heavyweight process and an activity as a distributed lightweight process. A job may be executing in several contexts at the same time, on the same or different nodes. An activity may be active in only one context at any point in time.

During the execution of an operation by an activity, an asynchronous invocation may be performed by creating a new activity to carry out the invocation, in parallel with the current activity. The new activity will terminate when the invocation which it was created to carry out completes. A job is created for each application run by a user and terminates when all of the activities created within the job have completed.

The Eiffel** programmer's interface to concurrency is through a set of classes. The user can invoke an operation on an object asynchronously, and at some later stage test for the termination of the invocation, recover the results of the invocation and suspend and resume the call. The following example shows how an activity can be created in Eiffel**. Consider:

```
class EXMPL export calc
 feature
  calc(a,b:INTEGER;c:SOME_CLASS):INTEGER is
     --Do some calculations
  end ; --Class EXMPL
```

The following is a class that includes an instance of an `EXMPL` and invokes its `calc` routine both synchronously, and asynchronously as an activity:

```
class USE_EXMPL
 feature
  act : ACTIVITY ;
  e   : EXMPL ;

  call_calc(A_ref:SOME_CLASS) is
   local
    i : INTEGER ;
   do
    e.Create ; -- Create EXMPL object
    i := e.calc(1, 2, A_ref) ;
      -- Normal synchronous invocation
    act.Create (e, "calc", 1, 2, A_ref) ;
      -- Create activity to do invocation
    i := act.wait_int ;
      -- and wait for the result
   end ;
end ; --Class USE_EXMPL
```

The activity is created by calling the `Create` routine for the `ACTIVITY` class passing it the object on which the routine is to be executed, the name of the routine to be invoked, and finally the argument list for the routine, which is a variable-sized list of arguments, any of which can be a reference to an object.

The full interface of the `ACTIVITY` class includes routines to kill, suspend and resume an activity as well as to wait for results of various types.

The `JOB` class interface is similar: suspending or killing a job suspends or kills all of its activities, while the `wait` routines wait for all of the job's activities to complete before returning the result of the initial activity.

4.4 Atomicity and Transactions

Transactional systems guarantee atomicity, consistency and permanence of effect for operations carried out within the scope of a transaction. Applications which have strong requirements for consistency can use atomic transactions to ensure consistency of data in the presence of concurrency and node or context failures [Berstein 87].

Amadeus provides support for transactions through the use of the RelaX [Kroger 90] [Schumann 89] transaction manager and libraries. In Amadeus a distinction is drawn between *atomic* and *non-atomic* objects. Transactional properties only apply to operations carried out on atomic objects carried out within a transaction. This distinction between atomic and non-atomic instances of a class is motivated by the desire to avoid the additional overheads associated with access to atomic objects for instances of a class for which strong consistency guarantees are not required. The model of atomic objects and transactions supported by Amadeus and RelaX, and its implementation is described in detail in [Mock 92].

Since all Eiffel** objects are global and persistent, any object can be promoted to being atomic at any time using:

```
make_atomic (i) ;
-- inherited from AMADEUS
```

Atomic code is generated by the preprocessor for every class in the system, which is used to trap access to atomic objects. Because access to atomic objects are trapped through functions, a similar restriction to distribution applies in that access to an atomic object should only be through its exported routines.

The Eiffel** programmer's interface for transaction management is provided through the `TRANSACTION` class. The transaction interface is synchronous, but in other respects it is similar to the job and activity interfaces, especially with regards to creating a transaction to perform an invocation on an object.

```
T : TRANSACTION ;
e : EXMPL ;

e.Create();
-- Create EXMPL object

make_atomic(e) ;
-- Make instance of EXMPL atomic

T.Create(e, "calc", 1973, 56, some_ref) ;
-- Create transaction to invoke calc
-- routine from class EXMPL.
```

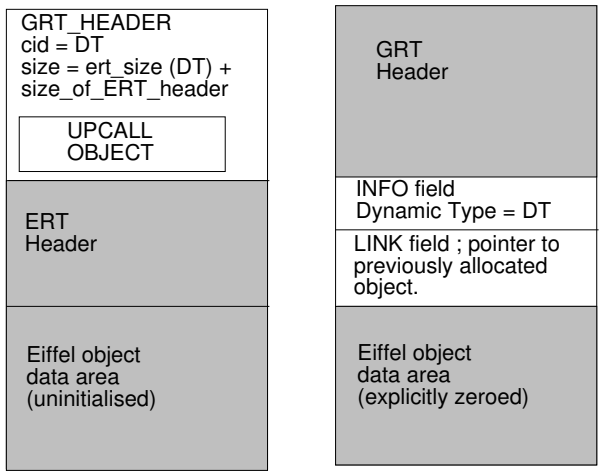
5 Implementation of Eiffel**

The following sections describe the implementation of Eiffel** on Amadeus.

5.1 Persistence

Two matters must be addressed in implementing persistence; supporting object creation so that Eiffel** objects are created as GRT objects, and providing the necessary upcall code for Eiffel** objects.

5.1.1 Object Creation



1. GRT_create is invoked. This allocates space for the object and its GRT header and returns a pointer to the object.

2. ALLOCATE initialises the ERT header, explicitly zeroes the data area and returns a pointer to the info field of the object.

Figure 3: Persistent object creation

To ensure that Eiffel** objects are created with a GRT header, a call to the GRT object creation routine replaces the call to the existing ERT memory allocation function. This function allocates memory for the object and its GRT header, initializes the GRT header and returns a pointer to the uninitialized object space to the ERT. The ERT then initializes the objects ERT header and explicitly zeroes the objects data area before returning a pointer to the object to the calling object. Figure 3 shows the steps involved in persistent Eiffel** object creation.

5.1.2 Implementing the Upcalls

With the support provided by the ERT it has been possible to write generic, class independent version of the upcalls required which are common to all Eiffel** applications; all objects in the system use the same generic upcall code. This contrasts with our C++ implementation which required that the upcalls be generated individually for each class. To support persistence, only the `nextptr` and `norefs` upcalls are required. These functions make calls to the ERT. All the other upcall functions take default actions.

A generic `regclasses` can also be used for all Eiffel** applications which registers class descriptors with the GRT for each class in the application. For every DT in the system, `0..num_classes - 1`, it makes calls to the ERT

to determine the corresponding class name and object size, the information necessary to register a class descriptor with the GRT. The class identifier in the class descriptor is set to the DT of the class. Each class descriptor contains a copy of the generic upcall object.

Eiffel** special objects also have a class descriptor and are allocated as GRT objects.

The following functions have been added to the ERT to provide information from its various data structures which is used to implement the necessary upcalls.

- `int sp_nfields (o)` – Get the number of fields in a special object.
- `char* ert_class_name (o)` – Get the name of an object's class.
- `int ert_DT_by_name(class_name)` – Get the DT assigned to a given class.
- `void ert_set_DT(o, DT)` – Set the DT of an object.
- `char* ert_name_by_DT(DT)` – Get name of class with given DT
- `int ert_size_by_DT(DT)` – Determine the physical size of objects with the given DT when they are created.
- `int ert_rout_index (o, func_name)` – Return the index of the given named function in the given objects class's routines table.
- `int ert_spnorefs(o)` – Determine number of references in a special object.
- `void* ert_spnextptr(pc, o)` – Returns a pointer to the $\texttt{pc}^{th}$ reference of special object.
- `int ert_norefs(o)` – Returns the number of references in an Eiffel object.
- `void* ert_nextptr(pc, o)` – Return a reference to the $\texttt{pc}^{th}$ object pointer in an object.

Given these functions, it is possible to define generic upcall functions and a generic `regclasses` function.

- `void *nextptr(int pc)` – Invoke `ert_next_ptr` on the encapsulated Eiffel object to return a reference to the $\texttt{pc}^{th}$ reference within the Eiffel object.
- `int norefs()` Return number of references in the encapsulated Eiffel objects instance data using `ert_norefs`.

All other upcalls take default actions.

5.1.3 Supporting STORABLE

In Eiffel** `STORABLE` has been altered to use the functionality of Amadeus to implement name storage. An Eiffel** object stored by name becomes an Amadeus persistent root ensuring that it and its dependents will persist. It is important also to note that this has been done only for name storage. `STORABLE` in Eiffel** still permits persistence through file descriptor and FILE storage, but this is unchanged from Eiffel.

5.1.4 Problems in Implementation of Persistence

The problems outlined in this section were encountered during the implementation of persistence, but are common to the implementation of Eiffel** in general.

GRT and ERT initialization. An Amadeus application can be thought of as having two stages; Amadeus initialization and application mainline. The former is invisible to the application programmer, who would view their program as having only the latter stage. The registering of class descriptors with the GRT is called as part of Amadeus initialization. This causes a problem with an Eiffel** application, where ERT initialization, the setting up of all the structures described in Sect. 3.2 for each class, is called in the application mainline stage. The generic `regclasses` function cannot be called until all the ERT information on the classes is available, so `regclasses` is called after ERT initialization. This is an interleaving of the two stages, Amadeus initialization and application mainline.

Special Objects and heterogeneity. A potential problem with special objects is that one can determine whether they contain embedded objects or references, but if they contain embedded objects and these are instances of objects that carry no run-time information (e.g. `INTEGER`) it is not possible to determine the type of the object. This would have serious implications for heterogeneity, one of the goals of the Amadeus project. For transferring and storing an object such as an array of integers on a heterogeneous network, it is necessary to convert it to a machine-independent form. To do so, one must be able to determine the types of its constituent parts. This is not currently supported in Eiffel**.

Garbage Collection. An Eiffel** application must be compiled with Eiffel garbage collection turned off (which is the Eiffel default) so as to avoid interference with GRT garbage collection. Thus the `link` field in an Eiffel** object's ERT header is ignored.

Compilation To make use of all Amadeus facilities, the Eiffel** programmer must use the `AMADEUS` class, an Eiffel class which is basically an interface to the GRT downcalls. An Eiffel** program must first generate a C package. This can be done by editing the Eiffel system description file. The C package consists of C code for all the classes in the system and C code for the ERT. This is necessary because ERT files must be altered in order to interface to Amadeus. The *makefile* produced is also edited to compile and link with Amadeus.

5.2 Distribution

The implementation of distribution requires that a proxy routine is generated for each exported routine in each class, as well as a `dispatch` routine for each class to handle incoming operations that have been initiated remotely. In addition invocations on proxy objects must be directed to the correct code.

A draw-back of this distribution mechanism is that remote access to an Eiffel** objects exported data items is not supported. The dataspace of a proxy object remains uninitialized and therefore undefined. Remote access to an Eiffel** object should be through its exported routines. This makes the type-model of Eiffel** weaker than the Eiffel type model.

5.2.1 Trapping Invocations on Proxy Objects

For each class a routines table for its proxy routines is set up that mirrors the true routines table of the class. The routines array described in Sect. 3.2 is expanded to include entries for the proxy code of all the classes in the system. The DT of the proxy code for a class C can be obtained by the simple rule:

$$DT_{proxy} = DT_C + num_classes$$

giving the position of the proxy code for class C in the routines array.

When an object is already in use in another context, attempting to map the object into another context will result in a *proxy* object being created in that context. The proxy object is the same size as the real object. Its data area is uninitialised, resulting in some memory wastage but making it very easy to overlay the proxy without invalidating existing pointers to the object in that context should the real object become available to be mapped. The DT of the proxy object is set to ensure that invocations on the proxy object will indirect through the correct proxy function table, as shown in fig. 4.

During routine calls between objects in different contexts, parameters are passed by reference, except for instances of the basic types which are passed by value. When passing an object, a *stub* (which contains the global name for the object and information to allow creation of a proxy for the object) for the object is pushed on the transmission block. The passing of arbitrarily large objects between contexts is possible by simply passing the stub for the object.

5.2.2 Proxy Code generation

As already stated, the code for each exported routine of a class `C` must have a proxy routine which is positioned in the proxy routines table at the same offset as the routine it remotely invokes is positioned in the real routines table. All proxy routines are generated from the following proxy routine template:

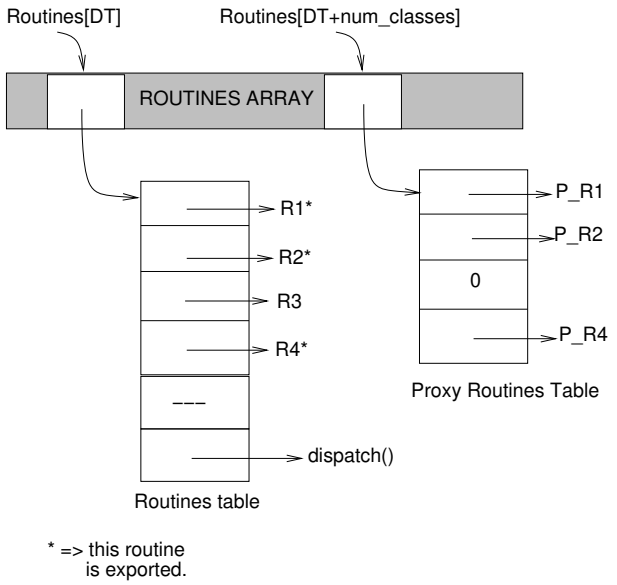


Figure 4: Routines array

```
DATUM <name>(void* curr <arg_list>) {
<return_decl>
int opid = ert_rout_index(curr,FUNCNAME);
aon_oo *t = ((aon_oo *)curr)->hdr();

 if (!amadeus.resolve (t)) {
 marshal aon_m (t,opid,<args>,<stubs>);
   <push>
   if (aon_m.rpc()) {
     <pop>
     return <return_id>;
     }
   }
return
  (*Routines[<DT>][opid])(curr <names>);
}

<name>        -> CLASSNAME_FUNCNAME_pr

<arg_list>    -> nil
              -> ,datum IDENT <arg_list>

<return_decl> -> nil
              -> datum RETURN_NAME;

<args>   -> size of items in <arg_list>
            + size of return item.

<stubs>  -> No. objects in arg_list + 1

<push>   -> nil
         -> aon_m.push(IDENT); <push>

<pop>    -> nil
         -> aon_m.nargs_reset() ;
            RETURN_NAME = aon_m.pop() ;

<DT>     -> ert_dynamic_type (curr)

<return_id>  -> nil
             -> RETURN_NAME

<names>      -> nil
             -> <id_list>

<id_list>    -> IDENT
             -> <id_list>,IDENT
```

For each exported routine `F` in class definition `C` there will be a proxy function `F_C_pr`. The basic mechanism is that if the object cannot be mapped, the parameters plus the stub for the proxy (which will be identical to the stub for the real object, and which enables the GRT at the remote side to locate the object) and the operation identifier are all packaged and an RPC is performed to the remote context where the real object is located. On completion of the call, return data (if any) is popped and returned to the caller.

Parameters are treated in a uniform fashion. A check is made to see if the parameter being pushed is a reference to an object, and if so a stub for the object is pushed and a flag is set to indicate that this parameter is an object reference. When popping parameters at the `dispatch` side is taking place, the `dispatch` will be able to distinguish between parameters that are object references and parameters that are instances of basic types.

5.2.3 Dispatch code generation

Each class's routines table must be augmented with a dispatch routine which is responsible for performing incoming remote invocations.

```
pblock* <disp>(pblock *b, void* curr)  {
marshal aon_m(((aon_oo*)curr)->hdr(),b);
int DT = ert_dynamic_type (curr) ;

 switch(aon_m.which()) {
```

```
            <cases>
            default:;
 } return b ;
}

<disp>          -> CLASSNAME_dispatch

<cases>         -> <case_i>
                   <more_cases>

<more_cases>    -> <cases>
                -> nil

<case_i>        -> case <i>: {
                  <return_decl>
                  <param_decl> <pop>
                  <func_invoke> <push>
                 } break ;

<i>   -> 0..ert_routine_count(curr)-1

<return_decl> -> nil
              -> datum RETURN_NAME ;

<param_decl>  -> nil
              -> datum <id_list>

<pop>         -> nil
              -> IDENT = aon_m.pop() ;
                 <pop>

<func_invoke> -> <func_assign><func>

<func_assign> -> nil
              -> RETURN_NAME =

<func> -> (*Routines[DT][i])(curr,<names>)

<push>        -> nil
              -> aon_m.reset();
                 aon_m.nargs_reset() ;
                 aon_m.push(RETURN_NAME);
```

5.2.4 Upcalls for Proxy Management

This section outlines the upcalls the GRT invokes to ensure the correct code binding for an object. All objects in the system use the same upcall code, while all proxy objects use the same proxy upcall code.

The following upcall functions (in addition to those outlined in Sect. 5.1.1) are required to support distribution:

- `void deactivate ()` Switch class identifier in GRT header to that of the proxy code.
- `void onuse ()` Set the DT of the object to be equal to the class identifier from the GRT header, i.e. the real DT.
- `pblock *dispatch (pblock *t)` Upcall the object's dispatch routine.

Proxies must provide the following upcalls:

- `void activate ()` Switch the class identifier in the GRT to that of the true code.
- `void onuse ()` Set the DT of the object to be the class identifier from the GRT header, i.e. the proxy DT.

All proxy objects will have a copy of this upcall structure in their GRT headers. `activate` is called when the object is being mapped in, while `onuse` is called after an attempt to access the object has resulted in a proxy being created, so the DT in the ERT header must be set to that of the proxy code, ensuring that subsequent invocations on this object will result in an indirection through the proxy routines table.

5.2.5 Problems in Distribution

The following section outlines some problems that have arisen in relation to implementing distribution:

Heterogeneity The treatment of parameters at the ERT level means it is impossible to distinguish between basic type instances, i.e. if a given datum is an `INTEGER` or a `FLOAT` or a `CHARACTER` at the proxy code level. This has serious implications for heterogeneity, whereby it is necessary to determine the type of the given datum in order to package it for transferring between heterogeneous nodes. An upcall mechanism to determine the type of a given datum if it is not an object reference would be necessary in order to support heterogeneity.

Dynamic Linking The current scheme for the dynamic typing of proxy code depends on there being a fixed number of classes in the application at compile time. This scheme would be inconsistent if dynamic linking was introduced, whereby classes are linked in during execution of the program, rather than all being linked together at compile time which is currently what happens with Eiffel and Eiffel**.

5.3 Concurrency

The implementation of concurrency in Eiffel** is straightforward: the classes `JOB` and `ACTIVITY` simply make calls to the appropriate routines implemented by the Amadeus kernel.

One complication in Eiffel** is the lack of variable sized parameter lists. The `Create` functions of the concurrent classes should be able to take a variable number of parameters. The Eiffel** preprocessor captures all occurrences of `JOB` and `ACTIVITY` creation in an Eiffel** file and replaces the variable parameter list in the `Create` call with an argument object reference. The argument object is declared immediately after the `JOB` or `ACTIVITY` declaration, and code produced to push each of the parameters onto the argument object. For example, after preprocessing, the code:

```
act : ACTIVITY ;
-- Somewhere in the code of some class
-- a reference to an ACTIVITY object

act.Create (e, "calc", 1, 2, A_ref) ;
-- Create activity to do invocation.
-- invoking routine calc of object e.
```

becomes:

```
 act : ACTIVITY ;

 a : ARG_OBJ ;
 -- An argument object for storing
 -- the parameters to an asynchronous
 -- object invocation

 a.PushINTEGER (1) ;
 a.PushINTEGER (2) ;
 a.PushOBJECT (A_ref) ;
 -- Insert each of the parameters
 -- into the argument object.

 act.Create (e, calc, a) ;
  -- Variable sized argument
  -- list stored in a
```

This is basically a translation from incorrect Eiffel code to Eiffel code that will be acceptable to the Eiffel compiler.

5.4 Atomicity and Transactions

At the language level supporting transactions involves generating atomic code for each class, and at execution time, when an object is made atomic, switching the binding in the object so that subsequent invocations on the object will use the atomic code.

Atomic code is generated by the preprocessor for every class in the system. As in the case of distribution, the Eiffel** preprocessor generates an atomic routine for every exported routine in a class's definition, and sets up a routine table for these atomic routines. This table will be accessed at run time by a DT for the atomic code. Making an object atomic involves switching its DT to the DT for the atomic code, so subsequent invocations will indirect through the atomic routines table.

When an invocation is made on an atomic object, the atomic routine locates the actual routine (by calculating the true DT), invokes it and returns results to the caller.

Distribution and transactions imply that a class will have three dynamic types, one for accessing the class methods (i.e. the real dynamic type as assigned by by the Eiffel runtime), one for the proxy code and one for the atomic code. The dynamic type of the Atomic code can be obtained by the following simple rule:

$$DT_{atomic_code} = DT + 2 * num_classes$$

The consequence of this is that in the Eiffel** run time, the `Routines` table is three times as big as the equivalent one in Eiffel, due to the fact that each class in Eiffel** has proxy code to support distribution and atomic code to support transactions.

One addition upcall is required from atomic objects:

- `void make_atomic` – Set the DT of the encapsulated Eiffel object to the DT for the atomic code.

which is upcalled by the GRT as part of the `make_atomic` downcall. It sets the DT of the object to the DT for the atomic code to ensure that further invocations on the object will indirect through the atomic routines table.

The means of starting a transaction on an object is similar to that in starting an activity on the object, and the same complication exists as with the `JOB` and `ACTIVITY` classes with the need to be able to pass variable sized parameter lists to the transaction's `Create` feature. Transaction creations are caught by the preprocessor and edited the same way `ACTIVITY` and `JOB` creations are.

6 Conclusion

Our goals in undertaking this work were to provide a programming environment for the construction of sophisticated distributed applications using the Eiffel language as well as to evaluate the ease with which a new language could be supported above Amadeus while maintaining that language's syntax and object model. We intended that existing Eiffel programs and software components could be reused in our environment without any alteration, a feature that is consistent with Eiffel's philosophy of software reusability and modularity. We believe that our goals have been largely achieved.

Eiffel** provides an Eiffel programming environment enhanced with support for distributed and persistent programming including concurrency and transactions. This environment provides a high degree of transparency for the programmer and, in particular, its implementation required no changes to the syntax of the Eiffel language. Moreover, since the usual Eiffel run time mechanisms are maintained for access to objects located in the current context, there is no time overhead incurred for accesses to objects which are not remote, stored or being accessed within a transaction. Additional overhead is incurred at compilation time - to generate proxy and atomic class code - and in terms of the space used by the headers of GRT objects. The main restriction imposed is the requirement that global and atomic objects only be accessed via exported routines. In the future, Amadeus may support use of distributed shared memory allowing this restriction to be relaxed.

The implementation of Eiffel** on Amadeus was facilitated by the structure of the ERT and, in particular, the amount of run time type information available. The main effort was devoted to the implementation of the generic - class independent - upcalls and the implementation of the preprocessor. A number of library classes were also provided to allow the programmer to interact with the underlying Amadeus environment.

Our current work is concerned with the implementation of a number of distributed applications using Eiffel** in order to more fully evaluate the environment. In future we expect to work on interworking between the C++ and Eiffel** environments supported by Amadeus in order to allow invocations between objects programmed using these different languages.

References

[Berstein 87] P.A. Berstein, V. Hadzilacos and N. Goodman; *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.

[Cahill 91] V. Cahill, C. Horn and G. Starovic; *Towards Generic Support for Distributed Information Systems.* In *Proceedings of the International Workshop on Object-Orientation in Operating Systems,* Palo Alto, October 1991.

[Cahill 93] V. Cahill, S. Baker, C. Horn and G. Starovic; *The Amadeus GRT - Generic Support for Distributed persistent Programming* Submitted for publication, January 1993.

[Chase] J. Chase, F. Aamdor, E. Lazowaka, H. Levy and R. Littlefield ; *The Amber System: Parallel Programming on a network of multiprocessors.* Proceedings of the 12th ACM Symposium on Operating Systems Principles, pp211-223.

[Decouchant 88] D. Decouchant et al; *Guide: an Implementation of the Comandos Object-Oriented System.* In *Proceedings of the EUUG Autumn Conference,* October 1988

[DSG 92] Distributed Systems Group, Dept. of Computer Science, Trinity College Dublin; *C** Programmers Guide* 1992.

[Horn 91] C. Horn and V. Cahill; *Supporting Distributed Applications in the Amadeus Environment.* Computer Communications, July-August 1991.

[ISE 89] Interactive Software Engineering Inc; *Eiffel : The Language.* Interactive Software Engineering Inc.

[Kroger 90] R. Kroger et al; *The RelaX Transactional Object Management System.* In *Proceedings of the International Workshop on Computer Architecture to support Security and Persistence of Information.* Springer-Valeg, May 1990.

[Liskov 83] B. Liskov and R. Scheifler; *Guardians and Actions: Linguistic support for robust, Distributed Programs.* ACM Transactions on Programming Languages and Systems, July 1983, vol. 5, No 3, pp381-404.

[Meyer 88] B. Meyer; *Object-oriented Software Construction.* Prentice Hall, 1988.

[Mock 92] M. Mock, R. Kroeger and V. Cahill; *Implementing Atomic Objects with the RelaX Transactional Facility.* Computing Systems vol 5, no. 3, USENIX, 1992.

[Richardson 89] J.E. Richardson and M.J. Carey; *Persistence in the E Language: Issues and Implementation*, SWPE, vol. 19 no. 12, December 1992.

[Schumann 89] R. Schumann et al; *Recovery management in the RelaX Distributed Transaction layer.* In *Proceedings; 8^{th} Symposium on Reliable Distributed Systems,* pages 21-28. IEEE, October 1989.