Interfacing a Language to the Comandos Virtual Machine

Colm McHugh and Vinny Cahill

Colm McHugh and Vinny Cahill
Distributed Systems Group,
Department of Computer Science,
Trinity College Dublin,
Ireland
Phone: +353 1 7021795
Fax: +353 1 6772204
Email:{Colm.McHugh,Vinny.Cahill}@dsg.cs.tcd.ie

Abstract

This report describes the implementation of Eiffel**, the extended version of the Eiffel programming language supported by Amadeus.

This report is published as Chapter 11. of *The Comandos Distributed Application Platform* Cahill, V., Balter, R., Harris, N. and Rousset de Pina, X. (Eds.), Springer-Verlag, Berlin, 1993.

Document No. TCD-CS-93-33

This report describes the implementation of Eiffel**, the extended version of the Eiffel programming language supported by Amadeus. Eiffel** provides support for distribution, persistence, concurrency and transactions. All objects in an Eiffel** application are potentially global and persistent, while concurrency and transactions are available through the use of library classes. See Chap. 4. of [Cahill et al. 1993] for further details.

The implementation of Eiffel** required no changes to the Eiffel programming language – persistence and distribution are transparent to the programmer. Moreover only minimal changes and extensions to the Eiffel run-time (ERT) were necessary. No changes have been made to the Eiffel compiler. This has been made possible by the extensive run time type information provided by Eiffel, which enables all objects to be treated in a uniform manner.

The type information provided by the ERT means that there is no need to generate different up-calls for each class. All objects in the system use the same up-call code. Preprocessing is necessary to generate proxy functions for each of the exported functions in a class in order to support remote access to instances of the class. Preprocessing is also necessary to generate the code for each class which is necessary to support atomic instances of the class and also to implement transaction creation and asynchronous invocation of objects.

The next section describes those details of the VMI (the interface between the GRT and a language-specific run-time) which are necessary to understand the remainder of this report . The following section describes the ERT. The implementation of persistence, distribution, concurrency and transaction support are all dealt with in turn in subsequent sections.

1 The Generic Run-time

Comandos objects can be distributed and persistent. An object mapped on a given node may be referenced by an object in a context on another node; in this case this reference will actually refer to a *proxy* for the object. Invoking on this proxy will result in an RPC on the real object if the object cannot be mapped.

A Comandos object consists of a header and space for language objects. Contained in the header is a stub for the object. This consists of the global name of the object, the identifier of the object's cluster and information to allow the creation of a proxy for the object. Also contained in the header is the *up-call* structure which is used by the GRT to obtain language-specific information about the language object such as the number of pointers in the object and their locations.

A GRT up-call structure is stored in another GRT data structure called a *class descriptor*. There is a class descriptor for every class in the application. This descriptor also contains the class name, the size of instances of the class and a class identifier (`cid`). A class descriptor is embedded in its own GRT object, and the creation of a class descriptor object for each class takes place in `regclasses`, a function that is called by the GRT as part of its initialisation, before the application mainline is called. The class descriptor is used to initialise the up-call structure in the GRT header of an object when the object is created.

1.1 Down-calls

This section describes some of the down-calls available to a language to enable it to interact with the underlying platform.

- `grt_create()` – This is used to create a GRT object. The language preprocessor inserts this call whenever an object is created in the language. The GRT allocates space for the GRT header and the language object, initialises some of the fields in the GRT header and returns a pointer to the uninitialised language object as a result back to the language-specific run-time.
- `grt_resolve()` – A check to see if the given object is mapped locally.
- `grt_promote()` – A down-call to promote the given GRT object. This involves assigning a global name to the object.
- `grt_make_atomic()` – A down-call to promote an object from being non-atomic to being atomic.

1.2 Up-calls

Each GRT object has a set of up-calls which can be generated by the language preprocessor and are invoked by the GRT through the up-call structure in the GRT header. The up-calls required are as follows:

- `create()` – Invoked when an object is created. Initialises the 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.
- `deactivate()` – Invoked when an object is being made inactive. It binds the proxy class code to the object.
- `nextptr()` – Returns a pointer to the n^{th} pointer in the object. Invoked when the object is being mapped and unmapped so the GRT can unfold and fold its pointers.
- `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` up-call. Invoked as the final step in making an object active.
- `dispatch()` – Invoked when an incoming RPC for the object is received. This up-call invokes the object specific `dispatch` function (generated by the language preprocessor) which unbundles the parameters and invokes the target operation.
- `make_atomic()` – Binds the atomic code to the object. Up-called by the GRT as part of the process of making an object atomic, a process initiated by the `make_atomic` down-call.

The up-calls `create`, `onuse`, `nextptr`, `norefs`, `dispatch` and `make_atomic` must be provided by each global and persistent object, while `activate` and `onuse` are required for each proxy.

2 The Eiffel Run-time

The ERT provides the execution environment for Eiffel applications. Since multiple inheritance and polymorphism are key features of Eiffel, the run time mechanism for locating a routine when it is invoked is sufficiently generic to be able to handle each of these cases and hence incurs a small penalty over a standard procedure call. Genericity is also a key feature of Eiffel, and the ERT implements genericity with no code duplication. Refer to [Meyer 1989a] for a full description of the Eiffel language.

Eiffel provides a facility for automatic compilation management called `es` which is responsible for compiling and linking all the classes in an application. The compiler uses C as an intermediate language and the ERT is written in C. The execution sequence of an Eiffel application starts when the `Create` routine of a specified *root* object is invoked, and ends when the `Create` routine completes.

2.1 Types and Objects

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 pointer 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 simple types, (`INTEGER, REAL, CHARACTER, DOUBLE, BITS` or `BOOLEAN`) or it can be an expanded (i.e. in-line) instance of a class type. All objects, except those declared to be of simple types, contain information used by the ERT 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. The DT, along with other information, is stored in a header at the start of each object. The use of the DT will be described in the next section on the ERT data structures for object management.

2.2 Object Layout

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, which contains the DT of the object, and the `link` field, which points to the last object allocated by the ERT. The `link` field is used by the Eiffel garbage collector to keep track of objects. In Eiffel, an object reference always points to the `info` field of the object being referenced, or is `void`. The `info` field bits are used by the ERT for the following purposes:

- `bits[0..19]` The object's DT, an integer common to all instances of the same class. If the object is a special object, it will give the number of fields in the data.
- `bits[20]` Expanded flag. This bit is set if the object is an expanded object.
- `bits[21]` Invariant flag. Set if there is a class invariant defined.
- `bits[22]` Mark flag. Used by Eiffel components such as the garbage collector and `STORABLE` (an Eiffel class that provides persistence) to mark objects when traversing an object tree.
- `bits[23]` Special object flag. Set if the object is a special object.
- `bits[24..27]` Generic bits. Bit 27 determines if the object's class is a generic type. In a special object, bit 27 indicates whether or not the special contains in-line data or is made up of references to other objects.
- `bits[28..31]` These bits are used by the ERT garbage collector.

The instance data consists of a number of fields called *datums*. A single datum can be a reference to another object, in which case it will contain a virtual memory address, or can contain an `INTEGER, REAL, BOOLEAN` or

CHARACTER. A number of datums can go to make up an expanded object or a 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. A DOUBLE occupies 2 datums, while an instance of BITS M (bit string of M bits) occupies (M div number of bits in a datum) + 1 datums. If a class B inherits data from a class A, then instances of B will be structured so that the ERT header will come first, then the data inherited from A, followed by the data for B.

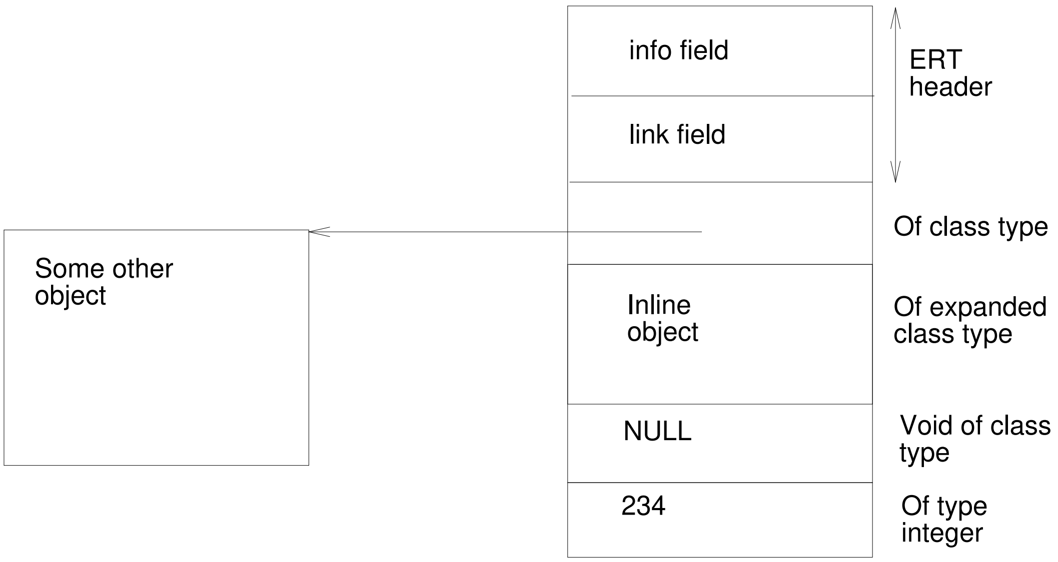

Figure 1: An Eiffel object.

The ERT provides functions and macros with which the programmer can obtain information about an object, given a reference to the object. These functions and macros interrogate the info field of the object, and the ERT data structures for the object's class. For instance, it is possible to get the DT of an object, determine if it is in-line, get the name of its class, get the number of fields in the object, get the type of a field, and get the names of the fields and methods of the object given a reference to the object.

The ERT also provides so-called *special objects*, which are used to implement classes such as STRING and ARRAY, and are basically blocks of memory used to hold the string's characters or the array's elements. Figure 2 show 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. The generic bit in the special object's info field is not set which indicates that its data fields are not object references. Because the special object is allocated immediately after the array object, its link field points to the array object. 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.

2.3 ERT Data Structures for a Class

Figure 3 shows the data structures that are set up for each class in an Eiffel application on ERT initialisation. Given the DT of the object a pointer refers to, the following information about the object can be obtained:

- Class_names[DT] – The name of the object's class.

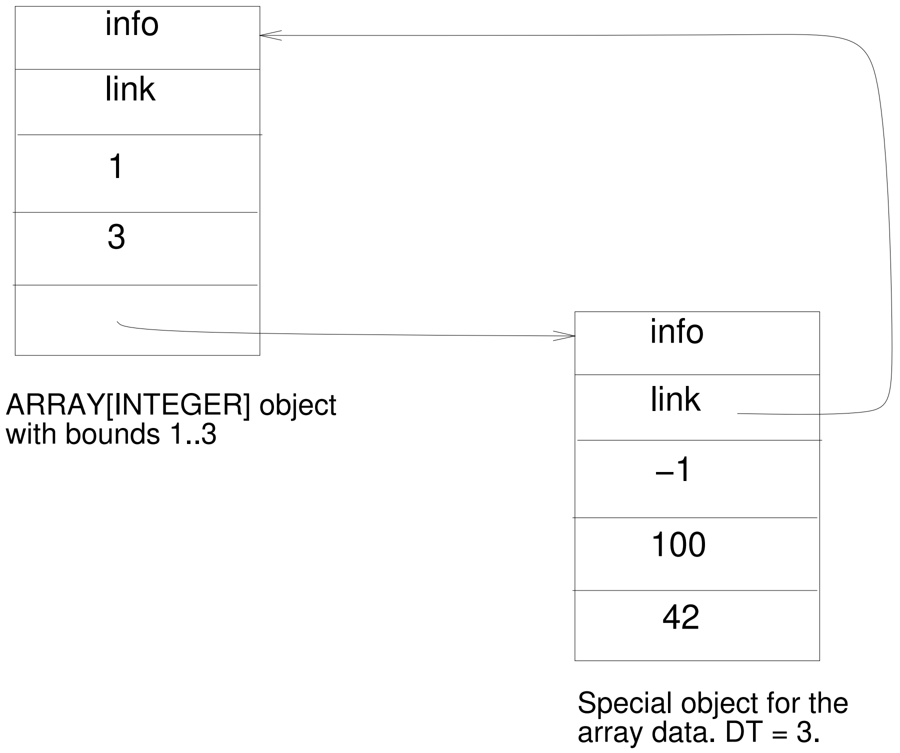

Figure 2: A special object used to implement an Eiffel array.

- `Object_size[DT]` – The number of fields in the object.
- `Num_routines[DT]` – The number of methods in the class.
- `Routines[DT]` – Gives a pointer to a function table for the class routines. Invoking on an object always results in an indirection through this table.
- `Routine_names[DT]` – The names of the object's methods.
- `Attributes[DT]` – The types of the attributes of the object's data, according to the following type-code:
  - greater than or equal 0 : object reference
  - -1 : integer
  - -2 : real
  - -3 : packed booleans
  - -4 : character
  - -9 : generic
  - -340 : expanded type
  - less than -350 : bits M

  This code uses positive numbers for attributes of class types (i.e. where the attribute is a reference to an object) giving the DT of the class, and negative numbers for in-line 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

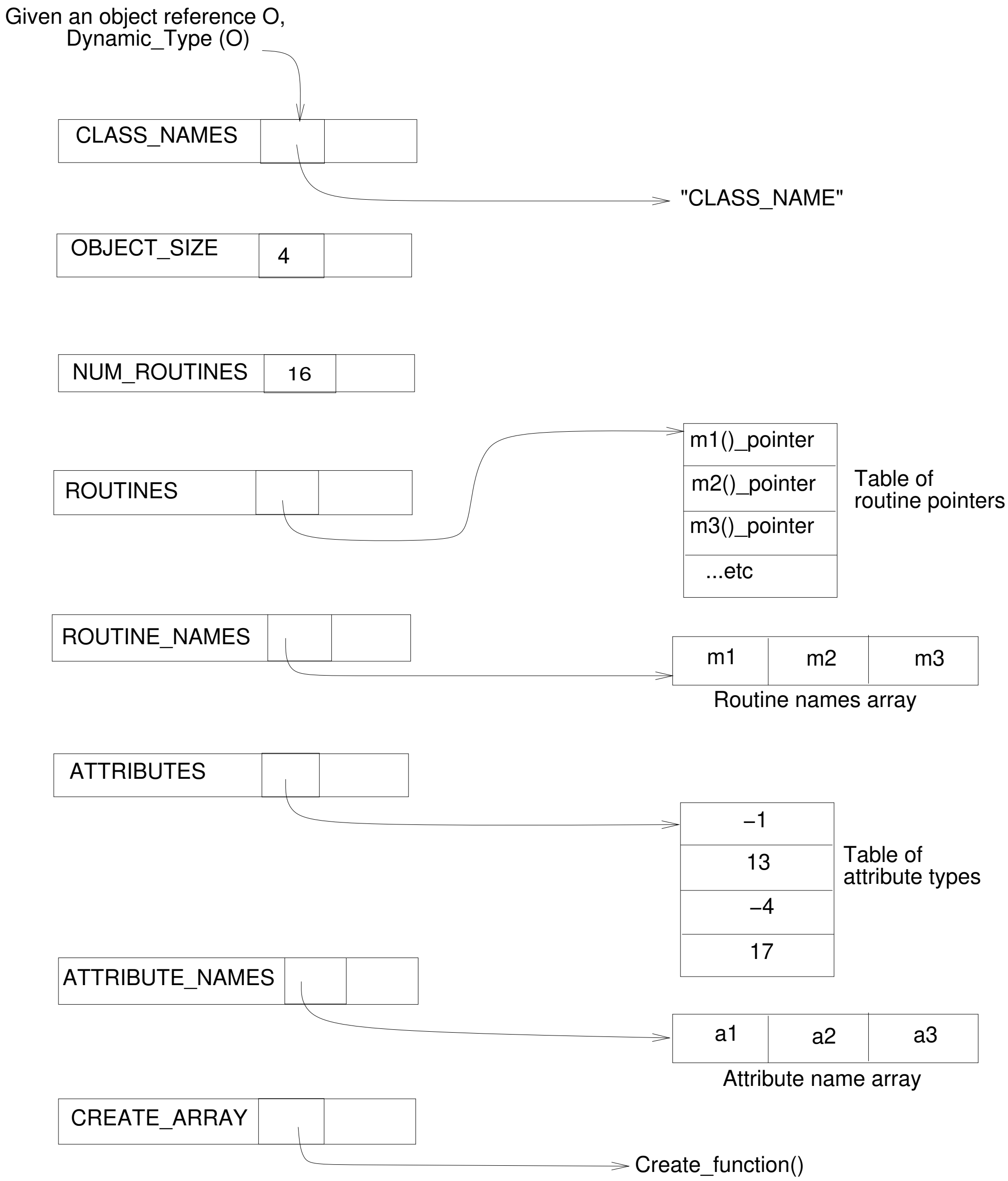

Figure 3: ERT data structures.

determine that attribute's DT. The entry for an entity declared to be of `BITS M` will be set such that $-$`(Attributes[DT][i] - 350)` gives the number of datums taken up by the entity. For example, an entity that is declared to be of `BITS 256` will have an entry equal to `-350 + (256 div bits_in_a_datum) + 1`. If `bits_in_a_datum` is 32, then this will be -342.

- `Attr_names[DT]` – The names of the object's attributes.
- `Create_Array[DT]` – A pointer to the `Create` function for the class.

All classes in the application 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 application. The Eiffel code that the programmer writes for a class compiles into C code with functions for all the methods of the class, plus initialisation functions for the entries in the ERT data structures, shown in Fig. 3, for the class. Eiffel compilation also generates a C mainline, which invokes the initialisation functions for each class to initialise the ERT data structures, before invoking the `Create` function of the application's root class.

To summarise, C programmers might like to think of an Eiffel object as a `struct` containing the data for the object, and a function table for the methods, where, to invoke an object's method, the address of the struct is passed to the function as well as the parameters for the invocation.

2.4 Object Creation, Access and Invocation

This section describes how object creation, data access and invocation take place at run time by illustrating the run time sequence of events using C-like pseudo-code.

2.4.1 Object Creation.

At the language level, an Eiffel object is created and initialised as follows :

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

At the ERT level, the following steps are taken to create and initialise the object to which `x` refers:

```
DTx = Attributes[DT(Current)][i];
DCurrent[i] = Allocate(DTx);
(*Create_Array[DTx])(DCurrent[i], args);
```

where `Current` is a pointer to the current object and `DCurrent` is a pointer to the instance data of the current object. The object reference `x` is located at field `i` in the current object's instance data and its DT is obtained by looking up the `Attributes` table for the current object's class. The space for the object is then created by passing this 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, all explicitly zeroed. The invocation of the object's `Create` function will then initialise this 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 current object was created. The header is marked as being expanded, and the `create` function is invoked. 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.

2.4.2 Object Access.

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

```
x.y;
```

At the ERT level, this is implemented as follows:

```
DCurrent[i][Access(DCurrent[i]) + os];
```

where `DCurrent` is a pointer to the instance data of the current object. `DCurrent[i]` is the run time value of `x`, field `i` in the instance data of the current object. This is a pointer to the object to which `x` refers. The data member `y` in `x` is located at an offset `os` from the start of the instance data for `x`. `Access` is an ERT function which, given a pointer to an object, returns a pointer to that object's instance data. Add `os` to `Access(DCurrent[i])` to obtain the location of `y` in the instance data of `x`. Both `i` and `os` are generated at compile time. If `x` is an expanded object, the access is implemented as follows :

```
*(Access(DCurrent + i) + os);
```

i.e. get a pointer to the instance data of the expanded object `x`, `Access(DCurrent[i])`, get a pointer to the feature `y` in `x`, `Access(DCurrent[i]) + os`, and de-reference it.

2.4.3 Object Invocation.

At the language level, a method of an Eiffel function is invoked as follows :

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

At the ERT level, this becomes:

```
(*Routines[DT(DCurrent[i])][os])(DCurrent[i], args);
```

where `DCurrent` is a pointer to the current object's instance data and `i` is the offset of `x` in the current object's instance data. `Routines[DT(DCurrent[i])]` yields a pointer to the routines table for `x`'s class. A pointer to the routine `y` is located at some index `os` within this table. The routine is invoked by indirecting through the routines table, passing it a reference to `x` as well as the actual arguments.

2.5 Functionality Provided by the ERT

The ERT provides functionality which, at the C code level, allows all Eiffel objects to be treated in a uniform manner. This has implications for the implementation of both persistent and distributed objects, which will become apparent later. This section presents a list of some of the important functions provided by the ERT which are used in implementing both persistence and distribution. (In the following function headers, `o` is a reference to an Eiffel object).

- `boolean is_special(void* o);` Return `TRUE` if `o` is a special object.

- `int c_dynamic_type(void* o);` Get the DT of `o`.
- `int c_size(void* o);` Get size (in bytes) of `o`.
- `int c_fd_nb(void* o);` Return the number of datums in `o`.
- `int c_lfd_nb(void* o);` Return the number of logical entities in `o`.
- `int num_classes;` Constant giving the number of classes in the system.
- `void* c_field(int i, void* o);` Return pointer to field `i` of `o`.
- `boolean c_f2_expanded(int i, void* o);` Return `TRUE` if field `i` of `o` is an expanded object.
- `int c_field_type(void* o, int i);` Get the type-code of field `i` of the given object.
- `boolean field_pointer(void* o, int i);` Return `TRUE` if field `i` of `o` is of a class type.

3 Persistence

This section discusses the implementation of persistence in Eiffel**. To use the support provided by the platform, all Eiffel** objects are created as GRT objects. Each ERT object includes a GRT header. Two matters must be addressed in implementing persistence; providing the necessary up-calls for Eiffel** objects and supporting object creation so that when an object is created, it has an initialised GRT header as well as an ERT header.

3.1 Interfacing the ERT and GRT

Figure 4 shows persistent Eiffel** object creation. The up-call object is the means of providing the GRT with language-specific information about the object. For example the GRT can invoke an object's up-call functions to determine the number of references in the object's instance data and obtain each of these references in a cyclical fashion via the up-call functions `norefs` and `nextptr` respectively.

With the support provided by the ERT it has been possible to write generic, class independent up-calls, the idea being that all objects in the application use the same generic up-call code. The `nextptr` and `norefs` functions of the generic up-call object make use of ERT functionality. All the other up-call functions take default actions.

In Eiffel** the GRT is made aware of the number of classes in the system, and, during `regclasses`, can make up-calls to get a class name and size, given a DT in the range `0..num_classes` − 1, in order to facilitate registering of a class descriptor for this class as part of GRT initialisation. The class identifier for a class descriptor is set to be the DT of the class as assigned by the ERT.

The following functions have been defined as a layer between the ERT and GRT and are used by the up-calls from the GRT to the ERT:

- `int sp_nfields(void* e)` – Get the number of fields in a special object.

```
int sp_nfields(void* e)
 if is_special(e)
   return c_dynamic_type(e);
 else
   return -1;
```

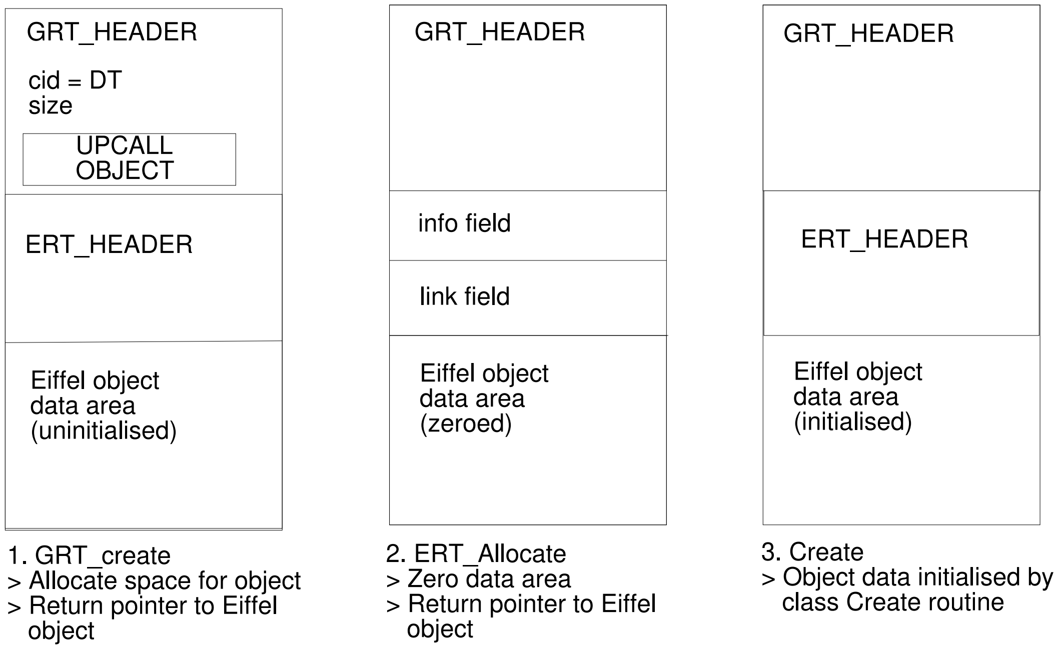

Figure 4: Persistent object creation.

- `char* ert_class_name(void* e)` – Get the name of an object's class. Note that special objects do not have any entry in the `Class_names` array; their DT is really the number of fields in the object.

```
char* ert_class_name(void* e)
 if is_special(e)
   return "special";
 else
   return Class_names[c_dynamic_type(e)];
```

- `void ert_set_DT(void *e, int newDT)` – Set the DT of a given object.

```
void ert_set_DT (void *e, int newDT)
 e->info = e->info + newDT
```

- `int ert_DT_by_name(char* name)` – Get the DT of a given class.

```
int ert_DT_by_name(char* name)
 for each DT in 0..num_classes - 1
   if Class_names[DT] == name
     return DT;
 return -1;
```

- `char* ert_name_by_DT(int DT)` – Get name of class with given DT.

```
char* ert_name_by_DT(int DT)
 return Class_names[DT];
```

- `int ert_size_by_DT(int DT)` – Determine the physical size of objects with the given DT when they are created.

```
int ert_size_by_DT(int DT)
 return Object_size[DT]*bytes_in_datum
                    + bytes_in_ert_header;
```

- `int ert_spnorefs(void* e)` – Determine number of references in a special object. In a special object the generic bit means that the object consists of references to other objects or consists of in-line expanded data.

```
int ert_spnorefs(void* e)
 if generic bit set in e->info
   return  sp_nfields(e);
 else
   return 0;
```

- `void* ert_spnextptr(int n, void* e)` – Returns a pointer to the $\texttt{n}^{th}$ reference of a special object. If the special object does contain references and `n` is less than the number of fields in the object, it suffices to return a reference to the $\texttt{n}^{th}$ field of the object.

```
void* ert_spnextptr(int n, void* e)
 if (generic bit set in e->info)
             AND (n < sp_nfields(e))
   return address of field n of e;
 else
   return NULL;
 end
```

- `int ert_norefs(void* e)` – Returns the number of references in an Eiffel object. Check first if the object is a special object. If not, go through each field of the object. If that field is a reference then increment the `norefs` count, else if it is an expanded object, increment `norefs` by the number of references contained within the field.

```
int ert_norefs(void* e)
 set norefs to 0;
 if is_special(e)
   return ert_spnorefs(e);
 else
   for each field i in e
     if c_f2_expanded(i, e)
       norefs += number of references in c_field(i, e);
     else
       if field_pointer (i, e)
         norefs++;
       end
```

```
    end
  return norefs;
end
```

- `void* ert_nextptr(int n, void* e)` – Return a reference to the $\mathtt{n}^{th}$ object pointer in an object. This includes a check for `e` being a special object. It also includes checking expanded objects for the $\mathtt{n}^{th}$ reference. A counter, `ref_count`, is kept of the number of references encountered as the fields of the object are checked in turn. If a field is an expanded object, then a check is made to see if the $\mathtt{n}^{th}$ reference is in here so the `n` − `ref_count` reference in this field is obtained. If this returns `NULL`, `ref_count` is updated by the number of references within this expanded object. When a field reference is encountered, a check is made to see if this is reference number `n`. If so, that field's address can be returned, else `ref_count` can be incremented and the search continues.

```
void* ert_nextptr(int n, void* e)
 set ref_count to 0; /* Number of references so far */
 if is_special(e)
   return ert_spnextptr(n,e);
 else
   for each field i in e
     if c_f2_expanded(i, e)
       if nextptr = ert_nextptr(n-ref_count, c_field(i,e))
         return nextptr;
       else
         ref_count += ert_norefs(c_field(i,e));
       end
     else
       if field_pointer(i,e) AND n == ref_count
         return address of c_field(i,e);
       else
         ref_count++;
       end
     end
 end
 return NULL;
```

Given these functions, it is relatively straightforward to define generic up-call functions and a generic `regclasses` function. Here are the up-calls of the generic up-call structure that are necessary to implement persistence: (note: `data()` returns a pointer to the Eiffel** object embedded in the GRT object)

- `void *nextptr(int n) {return ert_nextptr(n, data());}`

  Return reference to the $\mathtt{n}^{th}$ pointer in the objects instance data.

- `int norefs() {return ert_norefs(data());}`

  Return number of references in the objects instance data.

The generic regclasses code is presented below:

```
void regclasses() {
 upcall_structure UC; /* Instance of generic up-call class */
 class_descriptor CD; /* Another GRT data structure  */

 CD.upcall = UC;
 for DTindex in 0 to num_classes -1 {
   CD.class_name = ert_name_by_DT(DTindex);
   CD.size = ert_size_by_DT(DTindex);
   CD.cid = DTindex; Register(CD);
 }
 /* Register a class descriptor for the SPECIAL class */
 CD.class_name = "special";
 CD.size = 0; CD.cid = -1;
 Register(CD);
}
```

Every Eiffel** application can use this `regclasses`. The only attributes that have to be different in each class descriptor object are the class name, the instance size and the `cid`.

3.2 Down-calls

The functionality provided by the ERT means that no code generation is necessary for registering class descriptors for each class. A generic `regclasses` can be applied to all applications. It takes every DT in the system, `0..num_classes` $- 1$, and for each one registers a class descriptor object with the GRT. It is necessary to insert the name of the class and the size of objects of the class in the class descriptor. The `cid` of each class descriptor is set to the DT of the class. It also initialises each class descriptor with the same up-call object, so that all objects of all classes will share the same up-call table. This policy of registering class descriptors with the GRT for all Eiffel classes in the system means that all Eiffel objects are potentially persistent. Special objects, which have no entries in the ERT data structures of Fig. 3, are registered by the GRT with a DT of -1, and a name of "special". Their size, which varies, is set to `0` in the class descriptor. When the ERT is allocating a special, it passes in the size rather than the DT, and the GRT can set the correct size in the GRT header using this value.

During creation of a persistent object, the ERT's `Allocate` function makes use of the following down-call:

```
void* persis_create(int sz, int DT, char* class_name);
```

when allocating memory for the object, instead of using `malloc`. This function will return a pointer to a memory chunk of `sz` bytes with a GRT header above it, i.e. a pointer to the data area of a GRT object. So, `Allocate` initialises the ERT header as before, zeroes the data area and returns a pointer to the newly allocated object, which now has a GRT header. The function `persis_create` down-calls a GRT create function to allocate and initialise the GRT header and data area. Figure 4 shows how a persistent object is allocated and initialised.

In Eiffel, persistence is available to the programmer via the `STORABLE` and `ENVIRONMENT` library classes, which allow the conservation of objects in secondary storage. In Eiffel** class `STORABLE` has been altered to use the GRT down-calls to do the storing and retrieving, thus rendering the GRT transparent to an application that uses

`STORABLE`. 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.

3.3 Problems in the 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.

3.3.1 GRT and ERT Initialisation.

An application can be thought of as having two stages; GRT initialisation and application mainline. The former is invisible to application programmers, who would view their programs as having only the latter stage. Note that `regclasses` is called as part of GRT initialisation. This causes a problem with an Eiffel** application, where ERT initialisation, i.e. the setting up of all the structures described in Fig. 3 for each class, is called in the application mainline stage. The generic `regclasses` which Eiffel** uses cannot be called until all the ERT information on the classes is available, so `regclasses` is called after ERT initialisation, as part of the Eiffel mainline. This means an interleaving of the two stages, GRT initialisation and application mainline.

3.3.2 Signals.

The ERT installs signal handlers to deal with exceptions. For instance, to deal with segmentation violation, the signal handler will print a diagnostic attempting to show where the crash occurred, which could be, for example, an attempt to use an uninitialised pointer. The installation of these handlers is done at ERT initialisation. However, in an Eiffel** program, the GRT will have, as part of GRT initialisation, installed its own signal handlers. Thus, the ERT handlers will overwrite those of the GRT. This is because, at the present time, the GRT allows only one handler per signal. The ideal solution is to change the GRT to allow several handlers for a signal. The GRT would then have to be able to determine which handler to invoke, i.e. be aware of what is causing the signal, the ERT or the GRT. A good illustration of this is the case of `SIGALRM`, which the GRT uses to schedule threads. The ERT installs a handler for this which treats it like a segmentation violation, in that it will attempt to print a diagnostic and abort the program. Whenever a thread is scheduled, the program will be aborted. The temporary solution is to prevent the ERT from installing handlers for those signals that the GRT handles. The ideal solution is to allow the ERT install handlers for these signals, and the kernel will be able to determine which handler to call.

3.3.3 Special Objects and Heterogeneity.

Figure 2 shows how special objects are used to implement an array object. The ERT uses a function `spAllocate(size)` to allocate special objects, which, unlike `Allocate` which is given the DT of the desired object, is passed the size required. This function has to be changed to put a GRT header on top of the object returned in order for it to persist. It invokes `persis_create` in order to do so, passing it the size. The `cid` in the special's GRT header is set to -1. Whereas the ERT keeps no data structures for special objects, the GRT registers a class descriptor for them and they must have a GRT class identifier and a global identifier in order to persist.

Another potential problem with special objects is that while it can be determined whether they contain expanded objects or references, if they contain expanded 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. 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**.

3.3.4 Garbage Collection.

Eiffel links all the objects created in a linked list via the `link` field that exists in every object's header (c.f. Fig. 1). An application must be compiled with Eiffel garbage collection turned off (which is the Eiffel default). The `link` is not used and is ignored. At the moment, Eiffel garbage collection is disabled in Eiffel** making the `link` field redundant.

3.4 Compilation

To make use of all the GRT facilities, the Eiffel** programmer must use the `SYSTEM` class, an Eiffel class which is basically an interface to the GRT down-calls. An Eiffel** program must first generate a C package which 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.

4 Distribution

This section discusses the implementation of distribution in Eiffel**. The approach adopted for distribution is similar to that for persistence in that an Eiffel** object can be *potentially* distributed. Distribution is transparent to the programmer. This requires generating a proxy routine for each exported routine in the class definition, as well as generating a `dispatch` routine for each class to handle incoming operations that have been initiated remotely. This means, however, that it is not possible to access the exported data attributes of an object remotely. Access to a remote object can only be through its exported routines.

4.1 Calling Mechanism

For each class a routines table for its proxy routines is set up that mirrors the true routines table of the class. A proxy function attempts to map in the real object or, if this is unsuccessful, to pack the parameters into a transmission block and perform a RPC. In this case the corresponding function will be invoked on the real object remotely, via the real object's `dispatch` routine. The routines table of Fig. 3 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`. This gives the index for the proxy code for the class C in the routines table.

When an object is already mapped into another context, attempting to map the object into a different context will result in a *proxy* object being created. The proxy object is the same size as the real object, but its data space is uninitialised, resulting in some memory wastage but making it very easy to overlay the proxy should the real object it represents become available to be mapped in. The DT of the proxy object is equal to the DT of the true object plus the number of classes in the system ensuring that invocation on the proxy object will result in indirection through the proxy function table, as shown in Fig. 5.

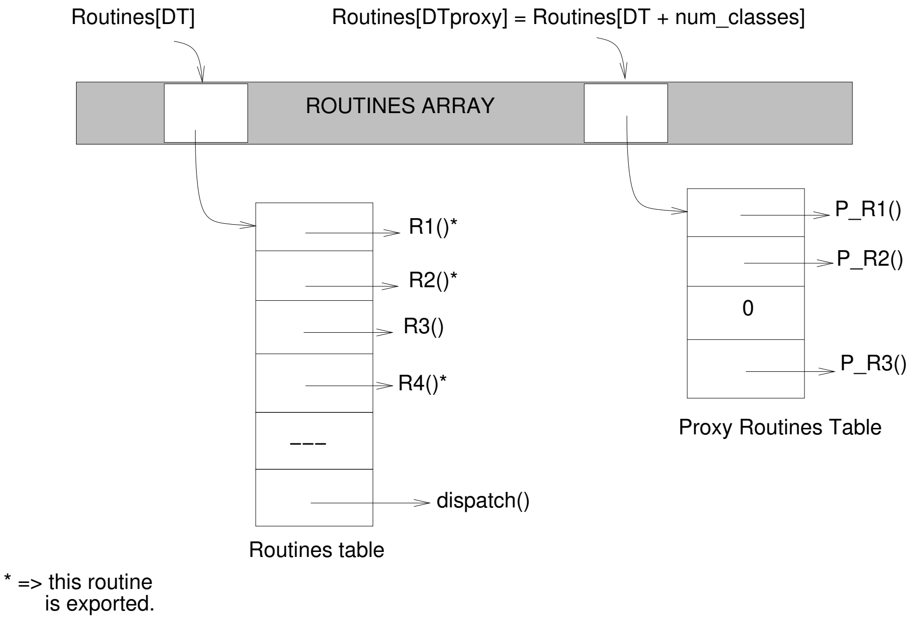

Figure 5: Routines array.

Thus, the call:

```
x.func (args);
```

at the Eiffel** level becomes:

```
(*Routines[Dynamic_type(x)][os])(x, args); ==
(*Routines[DT + num_classes][os])(x, args);
```

at the ERT level.

This will result in the proxy function packing `args` into a transmission block and performing an RPC. In the remote context, where the true object is mapped, the GRT will up-call the true object's `dispatch` up-call, giving it the transmission block which contains the parameters and indicates which function to invoke. This in turn calls the `dispatch` routine which has been added to the class's routines table, which finally invokes the specified routine on the object. Then, possible return data is packed into a transmission block which is returned to the calling context.

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 reference, a *stub* for the object is passed in the transmission block (resulting in the object being promoted if not already so) enabling the remote object which is being invoked to access it. The lack of pointer variables in Eiffel, as well as its reference semantics, enable the passing of arbitrarily large objects between contexts by simply passing the stub for the object in all cases.

A draw-back of distribution is that remote access to an Eiffel** object can only be through its exported routines; direct access to its exported data items is not possible as the data-space of the proxy is uninitialised. This makes the Eiffel** type model weaker than the Eiffel type model in allowing just exported routines to be the only features of an object that are remotely accessible; it is an aim to try and deviate as little as possible from the Eiffel type model for the type model of Eiffel**.

4.2 Code Generation

The following two sections present templates for generating proxy routines and the `dispatch` routine. At the ERT level, parameters are treated in a uniform manner, which simplifies the generation of this code a great deal.

4.2.1 Proxy Code Generation.

The proxy routine for each exported routine of a class `C` is positioned at the same offset in the proxy routines table as the routine it remotely invokes is positioned in the real routines table. All proxy routines are generated from the following proxy routine template:

```
<return_type> CLASSNAME_FUNCNAME_pr(void* current <arg_list>) {
 <return_decl>
 const int opid = ert_rout_index(current, FUNCNAME);
 aon_oo *_t = ((aon_oo *)current)->object();

 if (!amadeus.resolve(_t)) {
   Eiffel_aon_marshal aon_m(_t, opid, <arg_size>, <no_stubs>);
   <push_list>
   if (aon_m.rpc()) {
     <pop>
     return <return_id>;
   }
 }
 return (*Routines[ert_dynamic_type(current)][opid])
                                  (current <arg_names>);
}


<return_type> -> datum | void

<arg_list>    -> nil
              -> ,datum IDENTIFIER <arg_list>

<return_decl> -> nil
              -> datum RETURN_NAME

<arg_size>    -> size_of(datum) * no. of ids. in <arg_list>
                 + size_of(datum) iff <return_decl> not nil
```

```
<no_stubs>     -> no. of object ids. in <arg_list>
                   + 1 iff <return_decl> not nil

<push_list>    -> nil
               -> aon_m.push(IDENTIFIER); <push_list>

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

<return_id>    -> nil
               -> RETURN_NAME

<arg_names>    -> nil
               -> <id_list>

<id_list>      -> IDENTIFIER
               -> <id_list>,IDENTIFIER
```

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 of the proxy (which will be identical to the stub of the real object, and enables the GRT at the remote node to locate the object) and the operation identifier (`opid`) are all packaged and a RPC is performed to the remote context where the true object is located. On completion of the RPC 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 of parameters is taking place at the remote node, the `dispatch` routine will be able to distinguish between parameters that are object references and parameters that are instances of basic types.

4.2.2 Dispatch Code Generation.

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

```
aon_tblock *CLASSNAME_dispatch(aon_tblock* b, void* current) {
 Eiffel_aon_marshal aon_m(((aon_oo*) current)->object(),b);
 int DT = ert_dynamic_type (current);
 switch(aon_m.which()) { <cases>
                         default:; }
 return b;
}

<cases>        -> <case_i><more_cases>
```

```
<more_cases>  -> <cases>
              -> nil

<case_i>      -> case <i>: { aon_UNLOCK l;
                  <return_decl> <param_decl>
                  <pop_list> <func_call> <push>
                 } break;

<i>           -> no. in range 0..ert_routine_count(current)-1

<return_decl> -> nil
              -> datum RETURN_NAME;

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

<pop_list>    -> nil
              -> IDENTIFIER = aon_m.pop(); <pop_list>

<func_call>   -> <func_assign><func_call>

<func_assign> -> nil
              -> RETURN_NAME =

<func_call>   -> (*Routines[DT][<i>])(current, <arg_names>)

<push>        -> nil // => <return_decl> & <func_assign> = nil
              -> aon_m.reset();
                 aon_m.nargs_reset();
                 aon_m.push(RETURN_NAME);
```

This code generation is implemented using the Unix Yacc and Lex compiler development tools.

4.3 Up-calls for Distribution

Now that distribution has been discussed, the remainder of the generic up-call object definition can be presented as well as the up-call object definition for each proxy class. As stated before, all objects in the system contain the same up-call code. All proxy objects contain the same proxy up-call code.

The up-call object functions are as follows (note that `object()` returns a pointer to a GRT object, while `data()` returns a pointer to an Eiffel** object):

- `void deactivate() {object()->cid += num_classes;}`

  Switch `cid` in GRT header to be that of the proxy code.

- `void onuse() {ert_set_DT(data(),object()->cid);}`

  Set DT of the object to be `cid` in GRT header.

- ```
  aon_tblock *dispatch(aon_tblock *t) {
   return ((*Routines[ert_dynamic_type(data())]
                   [ert_routine_count(data())])(b, data()));
  }
  ```

  Up-call the object's dispatch routine.

The proxy up-call object functions are:

- `void activate() {object()->cid -= num_classes;}`

  Switch `cid` in GRT header to be that of the true code.

- `void onuse() {ert_set_DT(data(), object()->cid);}`

  Set DT of the object to be `cid` in GRT header.

Note that `activate` is called when the object is being mapped in, while `onuse` is called after an attempt to map the object in; this 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.

To support distribution, a class descriptor object for the proxy code as well as a class descriptor for the actual class must be registered with the GRT. The following is pseudo-code for the `regproxies` function, called as part of `regclasses` to register class descriptor objects for the proxy code of each class:

```
void regproxies() {
 proxy_upcall_structure PUC;
 class_descriptor       CD;

 CD.upcall = PUC;

 for (DTindex in 0 to num_classes -1) {
   CD.class_name = ert_name_by_DT(DTindex) + "proxy";
   CD.size = ert_size_by_DT (DTindex);
   CD.cid = DTindex + num_classes;
   Register (CD);
 }
}
```

4.4 Problems in Distribution

This section outlines some problems related to heterogeneity and dynamic linking which arose in the implementation of distribution.

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`. 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 up-call mechanism to determine the type of a given datum, if it is not an object reference, would be necessary in order to support heterogeneity.

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 were 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 Concurrency

This section discusses the implementation of concurrency in Eiffel**. The Eiffel** programmer's interface to concurrency is through class interfaces. To exploit concurrency in Eiffel** an object must use an instance of one of the concurrent classes (`ACTIVITY` or `JOB`) or inherit from them. The `ACTIVITY` and `JOB` interfaces provide for asynchronous invocation on an object and allow suspension/termination/resumption of the invocation as well as testing for invocation termination and recovery of invocation results.

5.1 Implementation

The following segment of code shows how an `ACTIVITY` instance is used to perform an asynchronous invocation on an object.

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

act.Create(obj, "op1", 1, 2, A_ref);
  -- Create activity to do invocation:
  -- invoke routine "op1" of object obj

i := act.wait_int;
  -- Wait for the result
```

The invocation is initiated by calling the `Create` function of the `ACTIVITY` object. As arguments to the `ACTIVITY`'s `Create` it must be able to take a reference to the object upon which the invocation is taking place, the name of the method that is being invoked and then the parameters for the invocation.

One complication in Eiffel** is the lack of variable sized parameter lists, and obviously the `Create` function of the concurrent classes must be able to take a variable number of parameters. The Eiffel** preprocessor captures all occurrences of `ACTIVITY` creation in an Eiffel** file and replaces the variable parameter list in the `ACTIVITY` `Create` call with an argument object reference. The argument object will be declared immediately after the `ACTIVITY` declaration, and code is produced to push each of the parameters onto the argument object. After preprocessing, the above code will look like :

```
act : ACTIVITY;

arg : ARG_OBJ;
 -- Create an object to store the parameters
```

```
 -- to an asynchronous object invocation

arg.PushINTEGER(1);
arg.PushINTEGER(2);
arg.PushOBJECT(A_ref);
 -- Insert parameters into argument object

act.Create(obj, "op1", arg);
  -- Variable sized argument list stored in arg
```

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

6 Transactions

This section describes the implementation of the transactional facility in Eiffel**.

Like persistence and distribution, the approach taken for transactions is that all objects in an Eiffel** system are potentially atomic. An object can be promoted to being atomic at any time.

```
make_atomic (i);
 -- Make it atomic by making a down-call to the GRT
 -- This make_atomic routine is inherited from class
 -- AMADEUS which is the Eiffel** interface to the GRT.
```

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.

Because accesses 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. So atomic objects should only be accessed through their class's exported routines.

Distribution and transactions imply that a class will have three DTs, one for accessing the class methods (i.e. the real DT as assigned by the ERT), one for the proxy code and one for the atomic code. The DT 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 for Eiffel, due to the fact that each class in Eiffel** has proxy code to support distribution and atomic code to support transactions.

The generic up-call object definition includes the following function:

```
void make_atomic{ert_set_DT(data(),object()->cid+num_classes*2;}
```

which is up-called by the GRT as part of the `make_atomic` down-call. 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 Eiffel** programmer's interface to using transactions is through the `TRANSACTION` class interface. The transaction interface is synchronous, but in other respects it is similar to the job and activity interface, especially with regards to creating a transaction to perform an invocation on an object.

The means of starting a transaction on an object is identical to that for starting an activity on the object, and the same complication exists as with the `ACTIVITY` class interface 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 filtered out in the same way `ACTIVITY` and `JOB` creations are.

The `TRANSACTION_OPTIONS` class provides programmers with an interface enabling them to alter the transaction state and thus control the way the transaction behaves.

7 Conclusion

Eiffel** provides an object-oriented, persistent, distributed, concurrent and transactional programming language environment with Eiffel as the programming language model. Eiffel** was intended to support all the features of the underlying environment without altering or compromising the Eiffel programming language and model. The extent of information provided by the ERT has enabled this goal to be achieved with minimum violation of the given constraint (c.f. Sects 5.1 and 7). It has been possible to implement Eiffel** without making changes to the Eiffel compiler or language, and with only minimal changes to the ERT.

Existing Eiffel programs and software components can be taken and used in the Eiffel** environment without any alteration, a feature that is consistent with Eiffel's philosophy of software reusability and modularity. The mechanisms of persistence, distribution, concurrency and transactions are all transparent to the Eiffel** programmer. The programmer can interact with the underlying environment using supplied classes. The programming model of Eiffel** remains faithful to that of Eiffel, with the exception of the restriction of non-data access for remote and atomic objects.

References

[Cahill et al. 1993] Cahill, V., Balter, R., Harris, N. and Rousset de Pina, X. (Eds.) *The Comandos Distributed Application Platform.* Springer-Verlag, Berlin, 1993.

[Meyer 1989a] Meyer, B. *Eiffel: The Language.* TR-EI-17/RM, I.S.E., 1989.