The Comandos Supported Programming Languages

Bridget Walsh, Paul Taylor, Colm McHugh, Michel Riveill,
Vinny Cahill and Roland Balter

Bridget Walsh, Paul Taylor, 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:Vinny.Cahill@dsg.cs.tcd.ie

Michel Riveill and Roland Balter
Unite mixte BULL-IMAG,
Gieres,
France
Phone : +33 76 54 49 12
Fax : +33 76 54 76 15
Email:{Michel.Riveill,Roland.Balter}@imag.fr

Abstract

This report describes the three programming languages supported by the Comandos platform: C++; Eiffel; and the Comandos Object-Oriented Language.

This report is published as Chapter 4. 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-93-34

The provision of programming language support was a key aspect of the Comandos project. Two different approaches to the provision of programming language support were considered in the framework of the project: the use of existing languages, and the provision of a new language environment. The first approach allows the facilities of the existing programming environment to be exploited and existing code to be reused where appropriate. However, because of the constraints imposed by using (or modifying) existing compilers, some features of the Comandos model cannot be fully integrated within an existing language. In contrast, defining a new language allows full access to the facilities of the Comandos virtual machine to be provided. Both approaches are, however, complementary in that objects written in various programming languages can be combined within a given application.

This report describes the three programming languages currently supported by the Comandos platform:

- Two existing languages, C++ and Eiffel, which have been extended with various features of the Comandos computational model. The choice of these languages was motivated by their industrial and academic success.
- The newly defined Comandos object-oriented language.

Appendix A. of [Cahill et al. 1993] shows an example program, written using each of the three languages, in order to illustrate how to program an application which makes use of the features of the Comandos platform using each of these languages.

1 C++

C++ is an object-oriented programming language which is now widely used in industrial and academic environments. Refer to [Stroustrup 1987] for an introduction to C++ and [Ellis and Stroustrup 1990] for a complete definition of the C++ language. The following section assumes some familiarity with C++. Any discussion of specific features of the C++ language is based on the AT&T C++ Language System, Release 2.0.

The extended version of C++ supported by the Comandos platform is known as C**. C** provides the application programmer with support for using distribution, persistence, concurrency and transactions. Distribution and persistence are provided through extensions to the C++ language while concurrency and transactions are provided through library classes.

An important feature of C** is that not all C** objects need be potentially persistent or remotely accessible – the programmer is free to choose those classes whose instances are to be potentially persistent or remotely accessible. Instances of other classes behave as normal C++ objects and are not known to the underlying Comandos platform. A complete description of C** is given in [DSG 1992].

1.1 Type Model

C** extends C++ with support for distribution and persistence by the addition of new keywords to indicate those classes whose instances are to be remotely accessible or potentially persistent. This section describes how these changes effect the type model of C++.

New keywords were chosen so as to force the programmer to be explicitly aware of the use of these features given the extra costs incurred. This decision was motivated by the general C++ philosophy of making costs explicit.

In order to support distribution *global classes* and *global objects* were introduced. A global object is an instance of a global class. A class is global when one of the following hold:

- the class contains methods which are qualified by the keyword `global`;
- the class contains member objects which are global;
- or, the class inherits from a global class.

Global objects are the only objects in C** which may legitimately be mapped on a different machine (or in a different context) from those objects which hold pointers to them. Since member objects are never on a different node to their enclosing objects, remote access to an object must always be via a pointer.

C** imposes a number of restrictions on access to global objects. In particular, all accesses to global objects must always be via those methods of the objects' classes which have been qualified as `global`. This applies even if the target object is currently mapped locally, since a global object may migrate to another node at any time.

Similarly, global classes may not have `friends` since direct access to the data of a global object is not allowed. Access between peer objects of the same class is also disallowed; however, there is no way at present of preventing this, using C++ syntax, i.e. making data or methods truly private.

A global class can be derived (inherit) from a non-global class. A class can also inherit from a global class, in which case the global features of the parent class are inherited by the derived class i.e. the derived class is necessarily a global class. Note that allowing global classes to be derived from non-global classes implies that there is a single class hierarchy thus facilitating code reuse, and giving the programmer more flexibility. However, assignment of a pointer to a global object to a variable of one of its non-global super types is not allowed i.e. if a pointer to a local object is assigned as its value a pointer to a global object, then if the global object subsequently migrates, it could no longer be accessed by simply de-referencing the local pointer. The user can override these restrictions on assignment by assigning via a `void*` pointer, but this is not recommended.

In order to support persistence, the concept of a *persistent class* was introduced. Whether a C** object persists or not depends on two conditions – that its class supports persistence and also that it is transitively reachable from a persistent root. There is a certain overhead in supporting persistence, and in accordance with the C++ philosophy of making costs explicit, the user must identify classes which support persistence by using the keyword `permclass` instead of `class` in its definition, or else by deriving the class from an existing persistent class. Thus only instances of designated persistent classes – also known as *persistent objects* – are potentially persistent, and will persist only if they are reachable from a persistent root object.

Instead of maintaining two separate class hierarchies for persistent and non-persistent classes, a single class hierarchy including both normal C++ classes and persistent classes is supported. Thus a persistent class may inherit from a class, and a class may inherit from a persistent class (it is then a persistent class). The keyword `volclass` is used to distinguish classes which are non-persistent, regardless of what they inherit from. Again the main motivation behind allowing the hierarchies to interact is to facilitate code reuse. Note, however, that assignment between persistent objects and ordinary C++ objects is not allowed.

Member objects of persistent objects will implicitly persist since they are part of the data of the enclosing object even if they are not, themselves, instances of persistent classes. Pointers or references in a persistent object which point to other persistent objects will cause the referenced objects (since they are transitively reachable from the root) to persist. Objects of non-persistent classes to which a persistent object has pointers will not persist and hence such pointers are by default re-initialised as null pointers whenever the persistent object is mapped, although the user can override this.

1.2 Distribution

In order to describe how global classes are defined in C** an example of a simple class providing remote access to its instances – from any node at which pointers to such instances exist – is given below.

```
class system_wide_int {
   int value ;
public:
   system_wide_int(int Value=0) { value = Value ; };
   global int operator()() { return value; };
   global void inc() { value++ ; };
};
```

This C** class definition describes integer objects which have remotely invocable functions to return and to increment their values. Note that the global functions are required since remote access to the instance variable `value` is not allowed. Such an integer can be created as normal and a pointer to it can be obtained:

```
system_wide_int *i = new system_wide_int();
```

The pointer to the new object can subsequently be communicated to other nodes in the distributed system. The pointer can then be used to access the object from any node in the distributed system. For example:

```
cout << "i is " << (*i)() << "\n";
```

The use of a pointer to a remote object in C** is syntactically *identical* to the usual use of a pointer in C++: the current location of a (potentially remote) object is not apparent from a pointer to it. Indeed, the location of the object may change at run time, transparently to objects holding pointers to it.

In order to allow the programmer to exercise some control over the placement of objects at run time, C** supports the notion of a *preferred* node for each activity. The preferred node is the node at which objects to be used by the activity will be mapped if possible i.e. the preferred node is used as a hint to the underlying platform. A programmer can also explicitly request clusters to be mapped and unmapped.

The preferred node and the mapping and unmapping of clusters can be controlled using the standard operations listed in Table 1.

1.3 Persistence

As described previously, if the programmer wishes to make instances of a class potentially persistent, then the keyword `permclass` is used instead of `class`, or the class derived (directly or indirectly) from any other persistent class.

Since even an instance of a persistent class will only persist if reachable from a persistent root, some mechanism of specifying the persistent roots in the system is required. The standard operation `record` is used both to specify that an object is to be a root and to associate a symbolic name with the object. A program may subsequently use the standard operation `lookup` to obtain a pointer to the object at any time, even in a separate execution of the code.

For example, to create a (potentially) persistent integer, and ensure that it persists by making it a root, the following code is used:

```
permclass perm_integer {
   int value;
public:
   perm_integer(int Value=0)
           { value = Value ; };
   int operator()()
           { return value ; };
   void inc() { value++ ; };
} ;

main() {
   perm_integer *i;
   if (system.reset() || !(i=system.lookup("myinteger")) ) {
     i = new perm_integer();
     system.record("myinteger", i);
   }

   cout << "i is " << (*i)() << "\n";
   i->inc();
}
```

When this is run several times, it produces the following output (assuming that the program is called `counter`):

```
%counter -reset
i is 0
%counter
i is 1
%counter
i is 2
```

Note the use of the standard operation `reset` which indicates if the flag `-reset` has been passed to the program.

Table 1: Operations to control mapping and unmapping

Operation	Description
int get_nodeid();	Get identifier of current node.
int get_prefnode();	Get identifier of current preferred node.
int set_prefnode(int n);	Set preferred node to be node `n`.
int mapcluster(int c);	Map cluster `c` at the preferred node.
int unmapcluster(int c);	Try to unmap cluster `c`.

The programmer may make use of this feature to allow the user to re-initialise persistent objects.

When a pointer to a persistent object is de-referenced, and the target object is not currently resident in memory, then some overheads are incurred while the target object is being fetched. Subsequently, all uses of that pointer incur no additional overheads beyond that of C++: the pointer refers directly to the target object in the usual way.

As a side effect of fetching one object from the store, the Comandos run-time may fetch further objects (in the same cluster), thus reducing the cost of subsequent object faults on the pre-fetched objects.

1.4 Concurrency

C** provides concurrency based on the model of jobs and activities. In C**, this model is presented through the use of *futures* as in Multilisp [Halstead 1985]. Instead of invoking a member function synchronously, and waiting for its result, a new job or activity can be `forked` to execute the member function in parallel to the caller. At some later stage the caller can test for the termination of the job or activity, and recover the results of the asynchronous function call. Equally, the caller may choose to ignore those results and never synchronise with the job or activity. A job/activity can be suspended or resumed as often as is required. Moreover a job or activity can also be explicitly terminated.

Job and activity support is provided through a number of library classes. The following example shows how jobs and activities can be created. In the example, the operation to be invoked asynchronously is the `calculate` method of the `do_math` class: The code shows both an ordinary (synchronous) invocation of `do_math::calculate` as well as the creation of an activity to carry out the same invocation asynchronously.

```
call_calculate() {   // some function

   do_math *m_obj = new do_math();  // the target object
   glob_obj *obj = new glob_obj();  // some global object
   int result;

   // do normal invocation and get result
   int result = m_obj -> calculate(3, obj, 7);

   // create activity to do it asynchronously
   activity *act = new activity(m_obj, 0, ArgList+3+obj+7);

   // do other things...
   // ...perhaps suspending and resuming the activity
   act -> suspend();
   // ...
   act -> resume();
   // and then wait for the result
   int result = act -> wait_int();
};
```

The activity is created by calling the constructor from the `activity` class as follows:

- The first parameter to the constructor is the object on which the method is to be executed.
- The second is the operation number of the method to be invoked. Operation numbers are assigned to global operations defined in each class (not baseclass) consecutively starting from `0` in the order in which the operations are declared. Currently, there is no way of using the global operations defined in a base class as the initial operation of a job or activity.
- The third parameter is of the format `ArgList + actual parameters in order`. Default parameter values should not be used in the invocation.
- There is an optional fourth parameter, which is the identifier of a preferred node for the activity (as described in Sect. 1.2).

The constructor returns a pointer to an activity object which may be used to kill, suspend, resume and get status information for the activity. Finally, a call of the form `wait_X` on the activity waits for the result. Results may be of the type `void`, in which case `act -> wait()` is used, any of the basic types including `floats`, or a `global void*` pointer, in which case `act -> wait_global_pointer()` is used.

The interface to class `job` is very similar; suspending a job suspends all of its activities. Similarly, killing a job kills all of its activities. A simple `wait()` waits until all of the job's activities have completed – the other wait functions return the result of the job's initial activity, but only when all of the activities have completed.

1.4.1 Synchronisation.

An activity is analogous to a lightweight process or thread. Programmers using activities need to take special care when sharing data between activities or when using UNIX system calls from a C** application.

Sharing data between activities simply requires synchronised access to the data, as outlined below. Standard UNIX libraries such as `clib` do not currently support multi-threaded execution. A number of other difficulties arise in the areas of non re-entrant system calls, the `errno` variable, signals and blocking I/O [Jones 1991]. Current thread packages – on which the implementation of activities is based – such as `Cthreads` or Sun's `lwp` package offer no real library support for multi-threaded programming although the `Pthreads` initiative, a part of the POSIX standardisation activity, intends to recommend solutions for these problems.

Shared Data Areas for Activities. C** provides the `semaphore` class, and a variation on it, called the `acquiring` class, which can be used to synchronise access to shared data. The definition of the `semaphore` class (from the C** library) is as follows:

```
permclass semaphore : public aon_semaphore {
public:
   semaphore():(1) {}          // binary semaphore
   semaphore(int i):(i) {}     // general semaphore

   global void wait();         // wait
   global void condwait();     // conditional wait
```

Table 2: Operations to control storage and clustering

Operation	Description
int setlc(int l);	Make container `l` be the default container.
int getlc();	Get identifier of current default container.
int wherelc(int l);	Get identifier of node where container `l` mounted.
int setcluster(int c);	Make cluster `c` be the default cluster.
int newcluster(int s);	Create cluster of size `s` and return its identifier.
int objcluster(void* o);	Get identifier of cluster where object `o` is stored.

```
      global void signal();      // signal
   };
```

where the usual `wait`, `signal` and `condwait` (conditional wait) operations are defined. Note that such semaphores are both potentially persistent and remotely accessible.

To synchronise access to shared data a user program must `wait` on a semaphore before accessing shared data and `signal` the semaphore afterwards. Generally there is one semaphore per shared data section, however the programmer may opt for a finer granularity e.g. a semaphore per field of a shared data area so as to improve concurrency. The definition of the class `acquiring` is as follows:

```
class acquiring {   // Acquire semaphore for current scope.
   semaphore& s ;
public :
   acquiring(semaphore& S);  // wait on S
   ~acquiring();             // signal S
};
```

`acquiring` is used by simply declaring it at the beginning of a scope, initialising it with a semaphore and allowing the implicit constructors and destructors to operate.

1.5 Storage

The user can choose to be completely unaware of the underlying storage system, or can use the interface to the virtual machine to explicitly control or extract information concerning the storage or clustering of persistent objects.

The default policy for clustering particular groups of objects is determined by the underlying system, and implicitly, new clusters are created as needed.

The user may designate the *default container*, and within it the *default cluster*, which are used, by default, to store newly created persistent objects. However, these are essentially hints to the underlying system, and while in most cases they will be complied with, occasionally the system may override them.

The standard operations available to the C** programmer to control the storage and clustering of persistent objects are listed in Table 2.

1.6 Transactions

This section describes how transactions are supported in C**.

1.6.1 Atomic Objects.

In C**, all instances of persistent classes are potentially atomic objects. Given an instance of such a non-atomic object, the object is promoted to being atomic with the standard operation `make_atomic`. Consider the following example:

```
#include <system.h>

permclass atomic_int {
   int value;
public:
   atomic_int(int);

   global int  read() const;
   global void write(int);
};

main() {
    atomic_int *i = new atomic_int(11);

    system.make_atomic(i);
    // ...
}
```

Object creation and atomic promotion can be combined into a single statement as follows:

```
atomic_int *i = new {ATOMIC} atomic_int(11);
```

Here, promotion is not performed immediately, but only when the object is next mapped. This may be used to create persistent atomic objects during the initialisation stage of a C** program (e.g. when the `-reset` flag is passed to the program).

When an object is created in a transaction and immediately promoted to being an atomic object, the operation `create_atomic` should be used instead of the operation `make_atomic`. For example:

```
// create new object
atomic_int *i = new atomic_int(11);
// promote object to atomic
system.create_atomic(i);
```

The `create_atomic` operation performs atomic promotion and also ensures the permanence of the creation in case the object is not otherwise modified during the transaction. This operation also has the side effect of acquiring a write lock on the object.

The transaction system supports non-strict, two-phase read/write locking. Hence, C** distinguishes between modifying operations and read-only operations. In the example class above, the operation `read` is a read only operation since the declaration of the operation is followed by the keyword `const`. The operation `write` is assumed to be a modifying operation and therefore may only be used inside of a transaction.

C** only guarantees the transactional properties for atomic objects so long as the objects are only accessed through operations which are declared as either `global` or `virtual`. Atomic objects should never be accessed via non-virtual or non-global operations and an atomic object's instance data should never be directly accessed. These restrictions are not enforced by the C** compiler or by the Comandos virtual machine so care should be taken to ensure proper use of atomic objects.

1.6.2 Creating Transactions.

Transactions are created in a way similar to activities and jobs. The greatest difference is that while the activity and job interfaces are asynchronous, the transaction interface is purely synchronous. A transaction is created to perform an invocation on an object and the transaction ends when the invocation has completed successfully or when the transaction is aborted. Thus, there is no explicit operation for ending the transaction. Consider the following example:

```
#include ...

class interface {
public:
   global void start(int);
};

void interface::start(int i) {
   cout << "interface::start() i = " << i << "\n";
}

main() {
   interface *i = new interface();
   transaction T;

   T.begin(i, 0, ArgList+1968);
   cout << "Outcome of transaction is: ";
   switch (T.outcome()) {
    case TRANSACTION_COMMITTED: cout << "committed\n"; break;
    case TRANSACTION_COMPLETED: cout << "completed\n"; break;
    case TRANSACTION_ABORTED: cout << "aborted\n"; break;
    case TRANSACTION_ERROR: cout << "error\n"; break;
```

```
    }
  }
```

As for jobs and activities, there is a special `transaction` class used in creating transactions. The operation for creating a transaction, `begin`, takes an object reference, an operation number and an optional list of arguments. In this example, when the transaction starts it invokes the operation `start` on the object referenced by `i`. Note that to be the initial operation of a transaction, the method `start` must be a `global` method. When the `begin` operation returns, the outcome of the transaction may be determined via the operation `outcome` which has four possible results:

- `TRANSACTION_COMMITTED:` the transaction committed.
- `TRANSACTION_COMPLETED:` the transaction completed.
- `TRANSACTION_ABORTED:` the transaction aborted.
- `TRANSACTION_ERROR:` some unspecified error occurred.

When creating a transaction, a top-level transaction is created if the current activity is not currently running in a transaction, otherwise a nested transaction is created.

1.6.3 Transaction Options.

When creating a transaction, the programmer may effect how the transaction behaves by altering the value of some of the transaction options described in [Cahill et al. 1993] . Each option may be turned on or off or inspected via operations on the `transaction` class.

As an example, suppose a programmer wishes to create a transaction which only accesses committed data and which raises an exception instead of blocking. This may be done as follows:

```
transaction T;

T.set_committed_only();      // access committed only data
T.unset_blocking();          // non-blocking mode
T.begin(i, 0, ArgList+1968); // now create transaction
```

All of these options except for `new_sync_level` may also be turned on or off from within a transaction via the special operations described in the next section.

1.6.4 Transaction Operations.

A number of standard operations are provided for manipulating transactions. The first of these is an operation `transaction_abort` to abort the current transaction. The outcome of an aborted transaction is `TRANSACTION_ABORTED`. Currently there is no way for a programmer to initiate the abort of a transaction other than the current transaction.

Since non-strict two-phase locking is supported, an operation `setlockpoint` is provided to specify that the current transaction has reached its lockpoint. Once a transaction has reached its lockpoint, the transaction may not acquire any more locks. That is, the transaction enters its shrinking phase and may only release previously acquired locks.

C** provides a way of explicitly setting and releasing locks on a given object by inheriting from a special class `atomic_lock`. The operations provided by this class are: `set_read_lock`, `release_read_lock`, `set_write_lock` and `release_write_lock`.

The operations `get_ts_options` and `set_ts_options` allow the programer to alter the behaviour of a transaction after the transaction has been created: The operation `get_ts_options` retrieves the current settings which may be then altered before calling the operation `set_ts_options` to change the current settings. All the available options are described in [Cahill et al. 1993] .

In order to determine if the current activity is currently running inside a transaction, the operation `in_trans` is provided. This operation returns non-zero if the current activity is running inside a transaction.

Given any object, it is possible to determine if the object is an atomic object or not. The operation `is_atomic` takes a reference to an object and returns non-zero if the object is atomic.

1.7 Exceptions

The current method of handling exceptions in C** is very simple. The user can specify one handler (essentially a function address) per activity, by using the standard operation `set_eehandler(handler_func)`.

The user can raise an exception using the operation `raisexc(exception)`. If the activity is executing in other than its initial context when the exception is raised then it exits and returns to its initial context. In any case, the handler originally specified by the user is invoked, passing it the exception identifier specified in the `raisexc` call.

If there is no user-supplied handler, the default system exception handler is invoked, which simply causes the activity to exit having displayed appropriate status information.

1.8 Implementation

C** is supported by a modified version of the GNU G++ version 1.37 compiler.

The compiler generates the up-calls for each persistent class necessary to allow instances of that class to be managed by the virtual machine. The up-calls are produced as the methods of an up-call class corresponding to each persistent class. An instance of the appropriate up-call class in embedded in the header of each Comandos object onto which a persistent object is mapped allowing the virtual machine to locate the up-calls.

The C** compiler supports distribution by generating additional code: this includes an additional method – `dispatch` – and a *proxy class* for each `global` class. An instance of this class, a *proxy*, is used to represent each instance of the class in any context where a pointer to the instance exists but the instance is not currently present. The proxy has the same interface as the real object and contains *stubs* for each of the global methods, to marshal parameters to be delivered to the target object via the virtual machine. At the remote side the virtual machine up-calls the `dispatch` method attached to the real object, which unpacks the parameters and invokes the required method. On return from the real method, `dispatch` marshals any results and returns these, via the virtual machine, to the proxy. This then unmarshals any results, and returns to the calling code.

2 Eiffel

Eiffel is a strongly typed, class based, object-oriented programming language which, together with its extensive class library, facilitates object-oriented design and software reuse.

A good introduction to the Eiffel language is contained in [Meyer 1988]. The complete definition of the Eiffel language is given in [Meyer 1989a] and the standard libraries (including the core `KERNEL` and `SUPPORT` class libraries) are detailed in [Meyer and Nerson 1990] The reader is assumed to be familiar with the Eiffel System (language and environment) and is referred to the above in any case. The discussion of specific language features and the Eiffel run-time environment is based on Version 2.3, level 4.

The extended version of Eiffel which exploits the facilities provided by the Comandos platform is referred to as Eiffel**. Support for using the main features of the Comandos platform – distribution, persistence, concurrency and transactions – is provided in Eiffel**. Unlike C** no changes to the Eiffel language were made to provide support for using persistence and distribution. In fact both persistence and distribution are normally transparent to the Eiffel** programmer – all Eiffel** objects are potentially persistent and distributed. Like C** concurrency and transaction support are provided through library classes. The Eiffel** programmer can interact with the Comandos virtual machine via the `COMANDOS` class interface.

2.1 Type Model

Eiffel is a typed language – every entity or function is declared as being of a certain type. The Eiffel type system is entirely based on the notion of class. An object's (or value's or instance's) behaviour is characterised by its associated class.

The Eiffel** type model is the same as the type model of Eiffel, except that the only features of an object that can be remotely accessed are its exported routines. Exported data items are not remotely accessible.

There have been no changes to the Eiffel language, making distribution and persistence transparent to the Eiffel** programmer. All objects in an Eiffel** system are potentially global and persistent. This decision was partly motivated by the issue of software reuse, a central feature of Eiffel, and partly by the extent of the run time information provided by Eiffel.

The Comandos type model distinguishes the notion of type, that is an abstract interface, and class, that is an implementation of a type. Each Comandos type can have potentially many (implementation) classes. In Eiffel, each type (class) has just one direct implementation.

Consequently, in Eiffel, the sub-typing and inheritance hierarchies are the same. The Eiffel term class applies both to a type and to its implementation. This approach is a restriction in terms of the overall Comandos type model, and hence presents no difficulties, when trying to map the Eiffel model on to the Comandos type model. Note also that the Eiffel information hiding mechanism is orthogonal to the inheritance structure. This allows an heir class not to export a parent's feature.

2.2 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 X can make use of this facility simply by inheriting from class STORABLE. For example, given a definition of class X of the form:

```
class X export store_by_name, retrieve_by_name, ...
inherit
    STORABLE
feature
    <class_features>
end ;  --class X
```

An instance x of X is stored via the call:

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

and can subsequently be retrieved via

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

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, with selective retrieval. A simple naming service is provided for object access. Any class of object may be stored in an environment (i.e. without having to inherit from any particular system class). The collection of stored objects can be queried, to discover e.g. the number of stored objects of a particular type. 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. When an object is retrieved from an environment all type information can be obtained, as this is stored along with the object representation that is written to the file. The reverse assignment statement can then be used to re-attach the object to a normal typed reference.

In both cases, when an object becomes persistent, all of its dependents are also stored since otherwise object references would, on retrieval, be meaningless. Shared references and cyclic dependencies are handled properly.

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

- All Eiffel** objects are created as Comandos objects and thus all objects reachable from a root (e.g. the default Name Service) automatically persist between different sessions. The support provided by Comandos could make the explicit object storage provided by Eiffel 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.
- Incorporate into the Eiffel** (language-specific) run-time, routines to emulate the performance 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 Comandos, is not fully exploited.

- A mixture of the above, i.e. provide modified implementations of `STORABLE` and `ENVIRONMENT` which mimic the simple naming service that they provide while using the standard Comandos 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 Comandos transparent persistence.

In fact the third method outlined above was chosen since it combines the advantages of both of the other alternatives. 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
    COMANDOS
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 operations `reset`, `record` and `lookup` described in Sect. 1.3. 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
```

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 Comandos facilities incurs no additional overhead beyond the equivalent Eiffel program, except for some extra space taken up by the headers attached to each Comandos object in the system.

2.3 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.

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.

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 for each activity and using the standard operations `get_nodeid`, `set_prefnode`, `get_prefnode`, `mapcluster` and `unmapcluster` described in Sect. 1.2.

2.4 Concurrency

Currently Eiffel retains the notion of a *program* which spans the execution of the `Create` operation of an instance of the `ROOT` class created for the program run. A single thread of control performs the sequence of object creations and method calls of the root class's `Create` operation. In effect, this corresponds to a Comandos job with a single activity.

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, through the appropriate class interface. The following example shows how an activity can be created in to call the `calculate` method of some class `CALC`:

```
calculate (a,b : INTEGER ; c : SOME_CLASS) : INTEGER
```

The `call_calculate` method takes an instance of `CACL` and invokes its `calculate` method both synchronously, and asynchronously as an activity:

```
call_calculate (c : CALC ; A_ref : SOME_CLASS) is
 local
   act : ACTIVITY ;
```

```
   i : INTEGER ;
do
  c.Create ; -- Create CALC object
  i := c.calculate (1, 2, A_ref) ;
             -- Normal synchronous invocation
  act.Create (c, "calculate", 1, 2, A_ref) ;
             -- Create activity to do invocation
  i := act.wait_int ;
             -- and wait for the result
end ;
```

The activity is created by calling the `Create` routine for the activity class passing it the object on which the method is to be executed, the name of the routine to be invoked, and finally the argument list for the method, which is a variable-sized list of parameters, 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: as usual suspending or killing a job suspends or kills all of its activities, while the `wait` methods wait for all of the job's activities to complete before returning the result of the initial activity.

2.4.1 Synchronisation.

Eiffel** provides the `SEMAPHORE` class to allow objects to synchronise accesses by multiple activities. The `SEMAPHORE` class provides `wait`, `signal` and `condwait` (conditional wait) operations as in C** (c.f. Sect. 1.4.1).

2.5 Storage

The Eiffel** programmer is by default unaware of the storage and clustering of objects, i.e the identity of an object's cluster or container, what other objects reside in the same cluster or where the cluster is stored.

As in C** the programmer may control the storage and clustering of objects by specifying the default container and default cluster. In particular, the standard operations `setlc`, `getlc`, `wherelc`, `setcluster`, `newcluster` and `objcluster`, described in Sect. 1.5, are also provided to the Eiffel** programmer through the `COMANDOS` class interface.

2.6 Transactions

Like persistence and distribution, the approach taken for transactions is that all objects in an Eiffel** system are potentially atomic. Making an object atomic in Eiffel** is a two step process, involving creating the object and subsequently calling the system to make the object atomic.

```
class USE_ATOMIC
inherit
   COMANDOS
feature
```

```
   i : EXAMPLE_INT

   use_i is
     do
       i.Create (1) ;
       make_atomic (i) ;
     end ;
end ; --Class USE_ATOMIC
```

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 to 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 following example shows how to create a transaction:

```
class EXAMPLE export f1
feature
   f1 (i : INTEGER) : INTEGER
end ;

class USE_TRANSACTIONS
feature
 i : EXAMPLE ;
 T : TRANSACTION ;

 do_transaction is
   local
     r : INTEGER ;
   do
     i.Create (2) ;
     T.Create(i, "f1", 1973) ;
       --Create transaction to invoke method f1 on i
     io.putstring ("The outcome of the transaction is ") ;
     inspect
        T.outcome
     when Transaction_Commited then
        io.putstring ("commited")
     when Transaction_Completed then
        io.putstring ("completed") ;
        r := T.get_integer  --Get result of operation.
     when Transaction_Aborted then
        io.putstring ("aborted")
     when Transaction_Error then
        io.putstring ("error")
     end ;
```

```
        io.newline ;
      end ;
   end ; --class USE_TRANSACTIONS
```

The means of starting a transaction on an object is similar to that of starting an activity to invoke a method on the object.

The `TRANSACTION` class also provides an interface to allow programmers to alter the transaction options described in [Cahill et al. 1993] and thus to control the way in which the transaction behaves.

Finally, the `COMANDOS` class provides a number of methods which are expressly concerned with transaction management including, for example, the `make_atomic` method.

2.7 Exceptions

In Eiffel an exception can be triggered by the user, the system or by hardware signals, traps and interrupts. A single exception model deals with all possible exceptions. Exceptions can be triggered by:

- assertion violation;
- de-reference on void;
- operating system signals;
- failure of a called routine to meet its postcondition;
- programmer defined exceptions.

The exception handler can be associated at the routine or class level with uncaught exceptions propagating back through the activation stack. See Chap. 14 of [Meyer 1989a] for further details. The overall exception model in Eiffel is similar to that in the Comandos Language.

2.8 Implementation

The implementation of the Eiffel** language on top of the Comandos virtual machine is described in detail in [Cahill et al. 1993] .

3 The Comandos Object-Oriented Language

This section describes the facilities provided by the Comandos object-oriented language. This language has been designed to faithfully reflect all the facilities of the Comandos model. The Comandos language provides all the functionality described by the computational models and, in addition, provides extra functionality which is tailored for the construction of distributed object-oriented applications such as groupware. The Comandos object-oriented language is described fully in [Comandos 1991].

3.1 Type Model

Objects are the units of modelling and programming. An object encapsulates a set of data, the state of the object, and a set of operations, or methods, which operate on this data. Every object is an instance of a *type*, where the type describes the behaviour of that object (i.e. all the operations that the object can respond to, and all the attributes of the object that may be accessed by another object). Every object also possesses an *implementation* which describes the representation of that object and also the way in which the operations are carried out.

To create an object, a generic `New` operation is provided by all types. This operation creates a new object and returns a *reference* (i.e. unique system-wide identifier) for the object.

Objects are accessed through *variables*. Variables may be considered as place-holders for object references. More than one variable may be used to access the same object. A variables possesses a type; when accessing an object through a variable, only the interface described by the type of that variable may be used.

3.1.1 Basic Data Types and Type Constructors.

All denotable values possess a type. The basic types of the type model are `Boolean`, `Integer`, `Real` and `Character`. Assignment and comparison are provided for all of these types. In addition each type has a set of type-specific operations, such as the arithmetic operations for `Integer` and `Real`, and the logical operations for type `Boolean`.

There is a special type `Top` which indicates a place-holder for a reference to an arbitrary object.

Complex objects may be defined from the basic data types by using a number of pre-defined *type constructors*: `String`, `List Of`, `Array Of`, `Record` and `Collection Of`.

3.1.2 User-defined Types and Classes.

In the Comandos language, a type consists of three aspects:

- a name;
- an abstract description, which defines the behaviour of objects of the type; and
- one or more implementations of this abstract description.

The abstract description characterises the common interface to all objects of the type and is called an *abstract type* or, where context permits, just *type*. An implementation of this abstract type, a *class*, represents code and storage layout information.

A type specifies a behavior, that is common to all objects of the given type, in terms of the operations applicable to these objects; this is the usual concept of an abstract type. Each operation in a type description is defined by a *signature*, which specifies the name of the method, the type of its parameters, and whether each parameter is an argument (`In`), a result (`Out`), or both (`InOut`).

A type may specify that a component of the state of the object is visible. Such a visible component is also called an *attribute*. If `x` (a variable of type `t`) is an attribute of an object `obj`,two operations are implicitly defined: a function `x-val:t`, such that `obj.x-val` returns the value of `x`, and a procedure `x-set(In v:t)`, such that `obj.x-set(v)` sets the value of `x` to `v`. As a shorthand, these operations are expressed with the usual notations (e.g. `x:=v` for `obj.x-set(v)`), and the declaration of an attribute `x` of type `t` takes the usual form `x:t`. An initial value may be specified for an attribute. If a declaration (with initial value) is prefixed by the keyword `Const`, then the value of the attribute may not be modified (i.e. only `val` is provided).

A class specifies a particular implementation of a type. A class definition is common to all instances of the class. It includes a description of the internal state of the object, as a set of instance variables, and the code of the methods. The attributes declared in the type definition are automatically included as instance variables.

An instance of a class is created by calling the method `New` of the class. The method `New` returns a reference to the created object. Thereafter, methods may be invoked on this object by calls using this reference.

3.1.3 Type Conformance.

Generally, in classical typed languages, objects having different implementations, but which are of the same type as a variable, may be assigned as value to that variable. Object-oriented languages go one step further by allowing assignment of objects of different types to a variable. For instance, a variable in Smalltalk-80 can be assigned any object; however, no type checking is performed, and errors (e.g. non-existent method) can only be detected at run time. A goal of the Comandos language was to allow some of the flexibility of Smalltalk, while performing type checking at compile time as far as possible. This raises the following question: under what conditions may an object be assigned to a variable of a different type? Let `ref` be a variable of type `T1`. Then, an object of type `T2` may be safely assigned to `ref` if `T2` *conforms* to `T1` so that subsequent operations on `ref`, allowed by its type `T1`, will be legal if applied on the object of type `T2`. Conformance is defined as follows:

- For each method defined in the interface of `T1`, there exists a method of the same name in the interface of `T2`.
- Each method of `T2` has the same number of arguments (`In` parameters) and of results (`Out` parameters), and of `InOut` parameters as the corresponding method of `T1`.
- For each method `m`, the type of its arguments in `T1` conforms to the type of its arguments in `T2`.
- For each method `m`, the type of its results in `T2` conforms to the type of its results in `T1`.
- For each method `m`, the type of `InOut` parameters in `T1` and `T2` are the same.
- Every type conforms to itself.

This definition implies that for each attribute in type `T1`, there must exist an attribute of the same name and type in `T2`. This agrees with the notion of type extension [Wirth 1988], that applies to the special case where types `T1` and `T2` are records (i.e. they only specify attributes, and no explicit methods).

Note that conformance is a purely syntactic relationship that involves no semantic equivalence, since it only relates the signatures of the methods. Conformance has therefore no automatic implications regarding the actual behavior of the objects. However, in the special case of records, the only (implicitly defined) methods are `val` and `set`, whose semantics are well-defined; therefore, in this case, syntactic conformance guarantees consistent behavior.

3.1.4 Sub-typing.

Sub-typing is a well-known way of specifying shared behavior between types. Informally, type `T2` is a *sub-type* of type `T1` (and `T1` is a *super-type* of `T2`) if the interface of `T2` provides at least the same operations (including attributes) as the interface of `T1` (it may provide more operations). The sub-typing relationship defines a hierarchy between types. In the current design of the Comandos language, this hierarchy is simple, i.e. a type has only one super-type. Furthermore, it is specified, by definition, that a sub-type must conform to its super-type. As a consequence, methods may be overloaded in sub-types, provided that the conformance conditions are respected. Since the conformance relationship may be statically checked, the language is strongly typed.

Note that conformance and sub-typing are really two different concepts, although they have often been confused in the literature. Specifying that a sub-type must conform to its super-type in a language is a design decision which results from the requirement of statically checked type safety. However, some language designers have decided otherwise. The decision to allow non-conform sub-typing is usually motivated by the restriction imposed by the contravariant behavior [Cook 1989] of method arguments imposed by the conformance rule.

The restrictions imposed by static type checking may be alleviated by dynamic checking. Dynamic type checking is supported in the Comandos language, by the `Typecase` and `Assertype` constructs.

The sub-typing hierarchy is paralleled by a subclassing hierarchy, which allows reuse of physical properties (i.e. attributes and operation implementations) between objects that already share a common behavior. Let `T2` be a sub-type of `T1`. If class `C1` implements `T1`, a class `C2` which implements `T2` may be constructed by inheritance from `C1` as follows.

- `C2` inherits all the instance variable definitions present in `C1`.
- Additional instance variables may be defined in `C2`.
- `C2` inherits all the methods defined in `C1`.
- An inherited method may be redefined in `C2`, as long as the conformance conditions are respected.
- Additional methods may be defined in `C2`, namely those which are part of the interface of `T2` and which are not included in the interface of `T1`.

As usual, when an inherited method is overloaded, the overloaded method is still accessible within the new program (through a pseudo-variable `Super`).

3.1.5 Object Composition.

The construction of complex structured objects is achieved by embedding references to objects within the state of other objects. This mechanism allows composite structures of arbitrary size to be built from component objects, and allows objects or sub-structures to be shared. Since references provide a system-wide naming scheme, composite structures, as opposed to individual objects, may be distributed on several nodes (although this distribution is not apparent at the programming language level).

Object composition is usually done through predefined constructors such as arrays, records, lists and collections which are the main building blocks for the construction of complex structures. For instance, the body of a document may be defined as a list of chapters, each of which is in turn a list of paragraphs, etc. Several documents may share a chapter (or a list of chapters).

3.2 Persistence

Every object in the Comandos language is potentially persistent. However, only objects which are reachable from a persistent root are made persistent by the run-time; other objects are garbage collected.

3.3 Distribution

Distribution is, by default, transparent. However, the language also provides constructs to force the creation or execution of an object to take place at a specified node. However, from the programmer's point of view remote and local invocations are indistinguishable. In particular, the public attributes of remote objects can be accessed in the same way as for local objects (unlike extended C++ and Eiffel which do not support this feature).

3.4 Concurrency

A job is created by an activity of another job using the primitive `Start_Job`. This primitive specifies a method to be carried out and creates an initial activity within the new job to invoke this method.

The execution of concurrent activities within a job may be controlled by the `CoBegin .. CoEnd` construct. Each object invocation enclosed between `CoBegin` and `CoEnd` is executed as a separate activity. The calling activity (i.e. the activity in which the construct is called) is suspended. Its resumption is controlled by the termination condition associated with `CoEnd`, which has the form of a boolean expression over the status (either terminated or not terminated) of the concurrent invocations. By default, if no termination condition is specified, the calling activity is resumed when all concurrent invocations have terminated.

3.4.1 Synchronisation Model.

The Comandos language offers sophisticated facilities for the control of concurrent accesses to shared objects.

Object sharing must be controlled in order to guarantee that shared data remain in a consistent state. The approach taken in the Comandos language is to express synchronisation as a set of conditions associated with objects, rather than as primitives appearing within activities. This is fully consistent with the object-oriented approach, since the specification of the synchronisation constraints is concentrated in the class that describes the object instead of being spread out in methods that use the object. In addition, this synchronisation specification is shared by all instances of the class.

The only way for an activity to access or modify an object is to execute a method of this object. Therefore, synchronisation constraints are specified as a set of *activation conditions*, which form a control clause. Each activation condition is attached to a method and must be satisfied before the execution of this method may start. If no activation condition is attached to a method, then the execution of this method is unconstrained.

An activation condition is a boolean expression which may contain the following parameters: instance variables which represent the internal state of an instance, actual parameters of the method, and *synchronisation counters*.

Synchronisation counters are internal data which give, for each method of a given object, values such as the total number of invocations on the object, the total number of completed invocations and the number of invocations currently pending. These counters are automatically updated by the language. The synchronisation mechanism is fully described in [Decouchant et al. 1991].

The following counters are defined for each method `m`: `invoked(m)`, `started(m)`, `completed(m)`, `current(m)`, `pending(m)`. The three first counters count the number of times that method `m` has been invoked, started or completed respectively, and the last two count the number of activities currently executing `m` and currently waiting to execute `m`. The values of the last two counters can be calculated using the values of the first three.

Synchronisation constraints are illustrated by the following program, which implements a bounded buffer used for communication between activities. The types `ProducerConsumer` and `Element` are assumed to have been introduced elsewhere:

```
Class FixedSizeBuffer Implements ProducerConsumer Is
   Const size = <some constant>;
   buffer: Array [0..size-1] Of Element;
   first, last: Integer = 0;

   Method Put(In m: Element);
   Begin
      buffer[last] := m;
      last := last + 1 MOD size;
   End Put;

   Method Get(Out m: Element);
   Begin
      m := buffer[first];
      first := first + 1 MOD size;
   End Get;

   Control
           Put: (completed(Put) - completed(Get) < size)
                     and current(Put) = 0;
           Get: (completed(Put) > completed(Get))
                     and current(Get) = 0;
End FixedSizeBuffer;
```

It should be noted that activation conditions are expressed using only boolean expressions. Modification of internal variables or more general algorithms are not allowed. As a consequence, some synchronisation schemes cannot be expressed directly, and must involve the use of additional methods.

3.5 Transactions

The Comandos object-oriented language provides a restricted transaction model. A transaction is started by using the `Transaction` primitive. A transaction terminates successfully when the operation specified in the call to `Transaction` completes. A transaction may be terminated unsuccessfully by invoking the primitive `AbortTransaction` when some undesired situation occurs. Transaction properties apply only to atomic objects. In the Comandos language, individual objects may be made atomic at creation time, or atomicity may be a property of a class, in which case all instances of that class will be atomic objects.

3.6 Exceptions

There are basically three kinds of exceptions. *User-defined exceptions* are raised at the language level. *System exceptions* report problems from the system (e.g. timeout during a remote call, abort of a transaction). *Hardware traps* are signalled by the processor (e.g. arithmetic exceptions or segmentation violation).

Although they are raised and reported differently, these various categories of exceptions are viewed and handled uniformly at the language level. In the Comandos object-oriented language, it is natural to express exception handling in terms of object invocation, in which any object invocation may terminate either normally returning a result or abnormally signalling an exception.

The main aspects of exception handling within the Comandos object-oriented language are summarised below:

3.6.1 Exception Declaration.

Exceptions are identified by symbolic names – system exceptions and hardware traps have reserved names.

Exceptions are associated with object methods. The user-defined exceptions, potentially raised by a method, appear in the method's signature.

3.6.2 Exception Raising.

User-defined exceptions may be raised using the `Raise` statement within the code of a method (provided that the corresponding exception is declared in the signature of the method). System exceptions and hardware traps are raised by the system.

A termination model of exception handling has been adopted: the interrupted method is stopped and execution is resumed at the level of the invoking method.

An unexpected exception is transformed into a pre-defined system exception which is propagated to the invoking method.

3.6.3 Exception Handling.

Handlers are associated with method invocations. A handler specifies a specific procedure for handling expected exceptions (including pre-defined and user-defined exceptions). The keyword `All` specifies that a given routine handles all expected exceptions. A handler may also be associated with a class, in which case the specified handler is used for all methods of the class.

The keywords `Replace` and `Retry` specify fault recovery policies as follows:

- `Replace` provides an alternative result for the method which failed. Conformance between the replacement value and the one expected from the method is checked at compile time.
- `Retry` asks for a new execution of the method which raised the exception. To avoid the risk of recursive calls to the same handler, the system ensures that a given handler cannot be called again before terminating.

3.6.4 Restoration.

Exception handling procedures allow alternative or retry policies to be implemented when a method fails. However this does not ensure that the invoked object is in a consistent state after it has raised an exception.

The `Restore` statement allows a restoration block, which is executed whenever a method exits abnormally by raising an exception, to be defined. This block is not executed if the method returns normally. As for handlers, restoration code may be provided at the method and class levels.

3.6.5 Inheritance and Conformance.

Exceptions, being part of the signature of a method, are also part of the sub-types of a given type. Furthermore, the conformance rules between types have been extended to take exceptions into account.

Inheritance is also applied to handling and restoration code. A handler which is associated with a method is inherited as being part of the method; this means that either the subclass inherits the method and then automatically inherits the related handler, or it redefines the method and then must also redefine the handler. Handlers which are associated with a class are automatically inherited by its subclasses. The rule is similar for restoration blocks.

3.6.6 Concurrency.

The `CoBegin .. CoEnd` construct allows concurrent activities to run in parallel. If the failure of a branch prevents the normal termination condition from ever being verified, then the remaining branches are stopped and a `Join_failed` exception raised.

3.7 Implementation

The compiler for the Comandos language is written in C and consists of approximately 30 000 lines of code. The compiler produces C which is in turn compiled by a standard C compiler. This approach allows the Comandos language compiler to be easily ported. The current version is able to generate C code which can be compiled on a wide range of target machines (Sun-3, Sun-4, and Sun386i; Bull DPX/1000 and DPX/2, Bull Zenith i386; DECstation 3100, and 5100)

For each type of compiler, the compiler produces a descriptor which is used during the compilation of other types and classes. For each class, the compiler produces the corresponding C code and a descriptor which is used in the compilation of further classes. Descriptors are stored in the UNIX file system. The output of the C compiler is post-processed to produce a Comandos binary, which is stored in the Comandos storage system.

The compiler runs in five passes:

- Pass 1 performs lexical and syntactical analysis of the source code, generating an appropriate tree structure using `lex` and `yacc` (or `flex` and `bison` from GNU).
- Pass 2 performs semantic analysis. The tree structure generated by pass 1 is decorated appropriately and a descriptor created. This step uses type and class descriptors created already.
- Pass 3 generates C code from the tree structure.
- Pass 4 compiles the C code and resolves external references.
- Pass 5 builds a Comandos binary from the UNIX binary and stores it in the storage system.

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.

[Comandos 1991] Comandos Consortium *Comandos Language Reference Manual.* Deliverable D4-T2.3., Comandos Consortium, 1991.

[Cook 1989] Cook, W.R. A proposal for Making Eiffel Type Safe. *Computer Journal*, 32(4):305–311, 1989.

[Decouchant et al. 1991] Decouchant, D., Le Dot, P., Riveill, M., Roisin, C. and Rousset de Pina, X. A Synchronisation Mechanism for Typed Objects in a Distributed System. In *Proceedings of the* 11^{th} *International Conference on Distributed Computing Systems*, Arlington, TX, USA, 1991. IEEE, Los Alamitos, 1991. pp. 152–159.

[DSG 1992] Distributed Systems Group *C** Programmer's Guide (Amadeus v2.0).* Technical Report TCD-CS-92-03, Department of Computer Science, Trinity College Dublin, 1992.

[Ellis and Stroustrup 1990] Ellis, M.A. and Stroustrup, B. *The Annotated C++ Reference Manual.* Addison-Wesley, Reading, MA, 1990.

[Halstead 1985] Halstead, R. Multilisp: a Language for Concurrent Symbolic Computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, 1985.

[Jones 1991] Jones, M.B. Bringing the C Libraries with Us into a Mutli-Threaded Future. In *Proceedings of the USENIX Winter Conference*, Dallas, TX, USA, 1991. USENIX, Berkeley, 1991. pp. 81–91.

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

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

[Meyer and Nerson 1990] Meyer, B. and Nerson, J.-M. *Eiffel: The Libraries.* TR-EI-7/LI, I.S.E., 1990.

[Stroustrup 1987] Stroustrup, B. *The C++ Programming Language.* Addison-Wesley, Reading, MA, 1987.

[Wirth 1988] Wirth, N. Type Extensions. *ACM Transaction on Programming Languages and Systems*, 10(2):204–214, 1988.