An Approach for Developing CORBA-based Multi-Agent Systems

Fernando Bellas*, Ramón Juanes*, Nieves Rodríguez* and Ángel Viña*

*Departamento de Electrónica e Sistemas, Universidade da Coruña, A Coruña, CP/ 15071, Spain.
*Departamento de Electrónica y Sistemas, Universidad Alfonso X El Sabio, Madrid, CP/ 28691, Spain.

Abstract: This paper describes a methodology of analysis and a middleware for developing CORBA-based multi-agent systems. The methodology allows to break down the functionality of a system into a set of agents and FUNs (Functional UNit), a special class of agents that delegate part of their functionality into a number of cooperating agents under the FUN’s coordination. The middleware, CABLE, is built on top of a CORBA ORB, and tries to facilitate the transition from the design to the implementation, by providing the programmer with high-level abstractions, such as, agent and FUN, and is made up of a C++ framework (implementing the core functionality) and a simple Agent Definition Language (ADL) for specifying the key components of an agent. The purpose of ADL is to hide the complexity of the underlying framework from the programmer, who only needs to learn ADL, a small subset of the CABLE framework and CORBA IDL (the programmer need not learn the API of the CORBA ORB). Using a C3I application, a multi-agent browser for land military units, we show an example of use of both the methodology and the middleware. Both of them are being used in the EUCLID RTP 6.1 European research project, undertaken by the GRACE (Grouping for Research into Advanced C3I for Europe) consortium.

Keywords: CORBA, distributed object computing, multi-agent systems, software engineering, distributed systems.

1. Introduction

Object Request Brokers (ORBs), such as those provided by CORBA ([9]), DCOM ([6]) or Java RMI ([13]) are making easier the development of large and complex distributed applications. All of these frameworks allow the programmer to invoke methods of remote objects as if they were local. CORBA is standardized by the OMG, the largest software consortium in the world. CORBA ORBs allow clients to invoke methods on objects without concern for where the objects reside, what language are written in, what OS/hardware platform they run on , or what communication protocols and networks are used to interconnect distributed objects. DCOM is restricted to Microsoft platforms and languages (even though is now starting to be available on other platforms). Java RMI is very easy to learn, but is restricted to the Java language.

Even though ORBs facilitate the development of distributed applications, the programmer still has to deal with a complex framework, which increases the learning curve. This is

---

1 This work is supported by the EUCLID RTP 6.1 programme under contract MAL1a/0262 and partially by CICYT-TIC-96-0982.
specially true for users with no previous experience in developing distributed applications. In this paper we describe a methodology of analysis and a middleware for developing CORBA-based multi-agent systems in an easier way than using a CORBA ORB directly. We consider an agent as an autonomous entity that provides the agent community with services. The methodology allows to break down the functionality of a system into a set of agents and FUNs (Functional UNit), a special class of agents that delegate part of their functionality into a number of cooperating agents under the FUN’s coordination.

The middleware, CABLE, is built on top of Orbix 2.2 MT ([4]), a CORBA 2.0 ([8]) compliant ORB, and tries to facilitate the transition from the design to the implementation, by providing the programmer with high-level abstractions, such as, agent and FUN, and therefore giving direct support to the FUN analysis. CABLE is made up of a C++ framework (implementing the core functionality) and an Agent Definition Language (ADL). The purpose of ADL is to hide the complexity of the underlying framework from the programmer, who only needs to learn ADL, a small subset of the CABLE framework and CORBA IDL (it is not necessary for the programmer to learn the API of the CORBA ORB). ADL provides the programmer with a very simple language for specifying the key components of an agent (for example, the services it provides), in a similar way to a template, in which he/she will embed C++ (for example, for the implementation of the services the agent provides). The ADL compiler parses ADL files into C++ files. These files, together with the C++ files provided by the programmer, when linked, provide the agent’s implementation. The ADL compiler also allows to generate code in order to allow a CORBA agent communicate with a CABLE agent. As CABLE is built on top of a CORBA ORB, a CABLE agent can communicate with a CORBA agent.

Therefore, we have adopted a language based approach, instead of a framework based approach. Proponents of frameworks argue that frameworks are more open to developers, provide greater flexibility, and obviate the need to learn yet another language. However the type of language based approach adopted by CABLE does offer some advantages over the framework based approach, namely, (1) ease of use (there is no need to learn a complex framework), which is specially true for inexperienced users, and (2) a closer mapping from agent design to implementation (provides high-level abstractions such as agent and FUN).

Both the methodology and the middleware are being used in the EUCLID RTP 6.1 European research project, undertaken by the GRACE (Grouping for Research into Advanced C3I for Europe) consortium. The work is to accelerate the application of advanced software engineering and user interface techniques to Command, Control, Communications and Intelligence (C3I) systems. Such systems are an increasingly important part of battlefield, naval and air operations, but also have civilian applications in areas such as air traffic control and emergency services, to mention just a few. [5] provides a detailed description of the project.

The GRACE demonstrator is being designed and implemented as a multi-agent system. It is made up of a number of graphical facilities, which communicate one another, and help military users in the decision taking process. All the facilities, and the GRACE demonstrator as a whole, have been analysed by following the FUN approach. Following this approach, the functionality of a system is broken down into a number of agents that will cooperate to implement it. CABLE is being used for implementing the agents. We believe that both the methodology and the middleware can be reused for other projects.

The rest of the paper is organized as follows. Section 2 explains the FUN analysis (it is out of the scope of this paper to describe the design phase). Section 3 describes CABLE. Taking as an example a C3I application, a multi-agent browser for land military units, section
4 illustrates the use of the methodology and the middleware. Section 5 compares our work with other alternatives. Section 6 finishes with concluding remarks.

2. FUN analysis

The FUN analysis proposes an analogy between a system and a human organisation. Following the FUN analysis, a system is broken down into a set of agents and FUNs, which provide other agents/FUNs with services. Under this methodology, an agent is an autonomous entity that provides the agent community with services. A FUN is a special class of agent that delegates part of its functionality into a number of cooperating agents, under the FUN's coordination.

A FUN is specified by giving the following information: (1) its objectives, (2) the strategy for meeting its objectives, (3) the services it provides to its clients (who could be an end-user or another FUN/agent), (4) the resources it provides and can make use of (a resource is basically a synchronization mechanism as we will see in section 4), and (5) the roles that cooperate to provide its services. A FUN, in the same way as a human organisation, specifies the roles its members have to fulfill. This way, a FUN delegates part of its work into its members.

A role describes the part to be played by an agent (that may itself be a FUN) within a FUN in terms of: (1) the role title, (2) the cardinality (the required number of members taking on this role), (3) its joining benefits, and (4) its joining criteria. Joining criteria are specified in terms of services and resources. An agent must satisfy the joining criteria in order to take on a role in a FUN, which will make use of these services and resources in order to carry out its own services. Joining criteria allow plug and play of agents, as the FUN need not specify the name of the agent fulfilling the role, but just what needs from it. Joining benefits are specified in terms of services, resources and service responsibilities (FUN’s services delegated to the agents playing this role), that an agent gains as a consequence of joining the FUN in this role. Joining benefits allow software reuse, i.e., agents that offer most of what is needed to fulfill a given role, can be selected if the FUN supplies them with the resources and services they lack. In practise, this would involve the joining agent dynamically linking software to manage the joining benefits. As this is very complex to implement, this feature does not have direct mapping into our software, but it is still useful to model.

The FUN analysis is applied in a top-down fashion. It starts by defining a high-level FUN modelling the system as a whole. Next, for each role, it is necessary to consider if it can be fulfilled by a simple agent or another FUN. If new FUNs have been identified, the process continues analysing them in a recursive manner until no further FUNs can be identified. Not all FUNs are actual software, but instead they can represent high-level abstractions in the system analysis. Architectural patterns ([2]) can be helpful for doing this breakdown. In the design phase, design patterns ([1]) can be used for refining each agent/FUN.

As an example, figure 1 depicts a FUN (the ellipse) representing an enterprise, made up of three sub-FUNs. Agents are drawn as small circles and FUNs as double circles, with their roles shown as black segments. Arrows indicate provision of service. The role segments in FUNs implicitly represent arrow heads, indicating that the FUN is being provided with services via its roles. A FUN carries out its services by using the services provided by the agents taking on its roles. Services should be as generic as possible for maximizing software reuse. This way, a given agent can be playing the same role (or several) in a number of FUNs, as agent 5 in the figure.
It is worth noting the difference between taking on a role in a FUN and providing a FUN with a service. When an agent plays a role in a FUN is because the FUN itself has been broken down into smaller parts (agents/FUNs playing roles for it), that is, the agent playing a role in a FUN is a piece of software very related to the FUN, even though if it is designed with generality in mind, it will be loosely coupled with its FUN, and it will be able to play the same role (or other roles) in other FUNs, or providing other agents (which is not so closely related to) with services. This is the case of agent 5 in the figure, which apart from taking on roles in FUNs 1 and 3, provides agent 4 with services.

3. CABLE

CABLE is a CORBA-based middleware being developed by the GRACE consortium, which gives support to the FUN approach and tries to facilitate the development of CORBA-based multi-agent systems, making possible a close mapping of the design to the implementation. It provides a C++ framework for developing this kind of systems, and an Agent Definition Language (ADL) for facilitating its use, that enables the programmer to specify the main components of an agent. The current implementation runs on top of a CORBA 2.0 ([8]) compliant ORB, Orbix 2.2 MT ([4]). The hardware platform is a network of SPARC and Intel computers, running Solaris 2.5.1 and Windows NT 4.0, respectively.

3.1 Architecture of a CABLE agent

A CABLE agent (figure 2) is composed of multiple threads of control, called activities, which look like Ada tasks. An activity is made up of a set of entries, a run section and a private working memory. Entries are like methods in normal objects, which other activities can call on. Whenever an activity is created, the underlying thread executes the run section, and controls the order the entries can be called on, by means of accept-select statements, in a similar way to Ada, i.e., a calling activity is blocked until the called activity accepts a call to that entry. The private working memory is a placeholder for variables that can be referred both from the entries and the run section. In the run section, it is possible to specify post-rendezvous code after an accept statement for an entry. When the execution of the entry completes, the calling activity resumes its execution, and the called activity executes the post-rendezvous code. This maximizes the parallelism.
There are two types of activities: *internal* and *external ones*. The external activities are also called *services*, and its entries can be called from another agent. In order for a client agent can access a remote service, it needs to construct a proxy for it, which enables access to the remote entries. Services are identified by hierarchical names, and CABLE allows to build a proxy for a service from its name, which allows *plug and play* of agents. Whenever a proxy for a service is created, a thread is created in the server agent for executing such a service. Multiple instances of a same service can be running in parallel within an agent.

Unlike services, which are created in a reactive fashion, internal activities are created autonomously by the agent. They can be used for carrying out internal computations, monitoring, etc. There is a special type of internal activity, the *core* activity, which gets executed automatically after the *initialise* section (see below). Both internal and external activities can access global data by means of the *agent’s working memory*, a placeholder for global data.

An activity can subscribe to the events generated by another agent. The subscribing activity need not know the name of the publishing agent (having a proxy for one of its services suffices), which decouples publishers and subscribers. The subscribing activity can specify the data which is interested in for a particular event. ADL provides a syntax for subscribing activities to specify a *notification handler* (the piece of the code to be executed when a notification is received), and *notification criteria* in publishing agents (a piece of code for the publishing agent can access the data which remote subscribing activities are interested in, and send the requested data, if appropriate).

Even though the programmer is not aware of it, each agent has a thread called *main thread*, which receives CORBA requests. If the request is a notification, stores it into the notifications list (in FIFO order). If it is a subscription, it stores it into the subscription data table. When a subscribing activity wishes to accept notifications, its notification handler is executed on all the notifications sent to it. When an activity in a publishing agent publishes an event, the agent’s notification criteria section is executed on all the data in the subscription data table.
When an agent comes into life, it executes its initialise section, which is a piece of code that can be used, for example, for initialising data members of the agent’s working memory or launching internal activities. An agent also has a finalise section, which gets executed upon the destruction of the agent.

In order to give more support to the FUN methodology, CABLE also allows to define FUNs, a special type of agents that are used to create and manage organisations of agents. FUNs have all the functionality of agents, but in addition have a number of associated roles. The roles of a FUN are played by agents which may themselves be other FUNs. Currently it is only possible to specify roles in terms of joining criteria.

3.2 The Agent Definition Language

ADL provides a syntax for specifying three types of files, namely, service specification file (it specifies a service in a similar way to a CORBA interface), type specification file (it specifies, in CORBA IDL, a type used in an entry corresponding to a service) and agent definition file (one per agent). The purpose of agent definition files is to hide the complexity of the underlying C++ framework from the programmer. An agent definition file allows the programmer to specify the main components of an agent (working memory, activities, notification criteria, etc.) and provide an implementation for them (currently in C++). As an example of code, figure 3 shows an excerpt of the implementation of a service.

ADL gives support for agent inheritance. An agent can be defined by deriving it from another one. Derived agents inherit the base agent’s activities, its core activity, service proxies, notification criteria and working memory. The derived agent can extend the functionality of the base agent and override the implementation of the inherited activities. It also possible to define abstract agents (similar concept to abstract classes), i.e., agents that can not be compiled to get an executable, but to derive from them.

An agent definition file can define a FUN instead of a simple agent. Figure 4 shows an example of the definition of Force FUN, which specifies three roles. Each role requires one agent (cardinality 1) providing the corresponding service. Such agents can be used by other FUNs/agents (sharing all). CABLE searches on run-time available agents for fulfilling these roles, launching them in the event they are not already running.

When the ADL compiler processes an agent definition file generates two C++ files (header and implementation), that basically contain the code generated by the IDL compiler (which is automatically called by the ADL compiler), and concrete classes for the agent/FUN, services, internal activities, proxies and working memory. The concrete classes for services, internal activities and proxies are embedded into the definition of the concrete agent/FUN class. A derived agent inherits from the concrete base agent class, and therefore inherits all its embedded classes. The working memory derives from the base agent's working memory.

3.3 The Manufacturer Agent

To achieve service location transparency, one instance of the Manufacturer agent runs on a machine of the distributed system. The main purpose of this agent is to locate service providing agents, and launch and kill them when necessary. When a client agent constructs a proxy for a remote service, the implementation of the proxy contacts the Manufacturer for finding the name of the providing agent. If the agent is not running or can not be shared (CABLE allows to specify several sharing modes), the Manufacturer registers a new instance (using agent configuration files) of the providing agent in the Orbix implementation repository and returns its name (a character string that identifies the agent in the whole distributed system). The implementation of the proxy now contacts the providing agent.
(orbixd will start it if it was not running). The main thread receives the request and creates a CORBA object representing the service (executed by a thread of its own), and returns a reference to that object, that the proxy will use for subsequent invocations of entries in that service.

![Figure 3. An excerpt of the implementation of a service (in an agent definition file).](image)

Finally it is worth noting that applications are launched and killed from the control panel, which is a browser for application configuration files. These files specify the first agent to be launched for a given application. The rest of agents are launched automatically in subsequent proxy constructions. When an application is killed from the control panel, the Manufacturer kills all the agents making up such an application (if they are not being used by other applications). The end-user only makes use of the control panel, and is not aware of the underlying cloud of agents.
3.4 The CABLE framework

Figure 5 is an OMT class association diagram showing the main components of the CABLE framework related with the concept of agent. The TcabAgent class has been highlighted to indicate its centrality. Concrete classes (specific to a particular agent) are not part of the framework, but have been added to illustrate its use, and are drawn with a broken line style. All concrete classes are automatically generated by the ADL compiler. As it can be seen, the framework corresponds to the architecture of a CABLE agent, commented in section 3.1.

The main responsibilities of the TcabAgent class are: to start and end services, terminate the agent, handle notifications and provide an API for FUNs. Several methods of this class, those used for starting/ending services, terminating the agent, notifying events, subscribing/unsubscribing remote activities, etc., are not called directly by the agent programmer, but from other agents. An IDL interface, TcabAgentInterface, specifies those methods and TcabAgent provides an implementation for them (TcabAgentInterface and TcabAgent are associated with a TIE declaration). The methods of this interface are executed by the main thread. TcabAgent also provides other methods, those used for accessing the FUN API, publishing events, subscribing a local activity to the notifications sent by a remote agent, etc., which are called by other parts of the framework, by the code generated by the ADL compiler, and a few of them by the programmer. Concrete agent classes provide a constructor (initialise section in ADL), a destructor (finalise section in ADL) and an implementation for the StartService method (the method that creates an instance of a service whenever a remote agent creates a proxy for it). TcabAgent defines a default implementation of the NotificationCriteria method (notification criteria in ADL), which can be overridden in concrete agent classes.

The TcabThread class encapsulates the underlying operating system thread functionality. It declares a virtual Run method, which is launched when the thread is started. In order to increase the portability of the code, the framework also define classes for encapsulating the underlying synchronization mechanisms (TcabMutex, TcabAutoMutex, TcabSemaphore and TcabConditionVariable).

**Figure 4.** An excerpt of a FUN definition (in an agent definition file).

```plaintext
FUN Force {
    role OrbatHandler with cardinality 1 with sharing all {
        joining criteria {
            service OperateOnOrderOfBattle;
        }
    }
    role SetOfUnitsHandler with cardinality 1 with sharing all {
        joining criteria {
            service HandleSetOfUnits;
        }
    }
    role UnitCommandLevelHandler with cardinality 1 with sharing all {
        joining criteria {
            service HandleUnitCommandLevel;
        }
    }
} << Other components. >>
```
The Ada rendezvous mechanism is implemented by the `TcabRendezvous` class. This class defines methods which allow an activity to say, at specific times, that it is willing to accept calls on a specific entry or on a subset of them. The `TcabActivity` class derives from `TcabRendezvous` and is ultimately derived from `TcabThread`. Hence, each activity runs a single thread which has the ability to rendezvous with other activities. In addition, this class provides methods for notification handling, that is, to subscribe, unsubscribe, accept notifications, etc. The notification mechanism implemented by `TcabActivity` and `TcabAgent` is based on the Observer pattern ([1]). Concrete activity classes define an implementation of the `Run` (run section in ADL) method, along with specific entry methods which are part of the activity public interface, and an empty implementation of the virtual `NotificationHandler` method (notification handler in ADL).

![Figure 5. Main classes of the CABLE framework related with the concept of agent.](image)

`TcabService` is the class which all services provided by an agent derive from. Concrete service instances define its entries. The ADL compiler generates an IDL interface, which is associated with the concrete service class via a TIE declaration, which enables its methods (entries) to be called from other agents. The `TcabProxy` class enables access to remote services. The most important methods of this class, `FindAgentNameForServices` and `ConnectServiceRequester`, contact the local Manufacturer agent for finding the name of the agent that implements a given service (or a set of them), as commented in section 3.3, and obtain a reference to the remote CORBA object implementing such a service, respectively. `ConnectServiceRequester` firstly binds to the remote agent (from its name), and then calls `StartService` on the remote agent interface (playing the role of proxy factory), which apart
from creating an instance of the corresponding service, returns a reference to the underlying CORBA object, that the proxy will use for subsequent remote entry invocations.

The `FindAgentNameForServices` method, in `TcabProxy`, allows the client agent to specify if it is willing to share the server agent with other client agents. There are four possible sharing modes: kShared (the server agent can be shared), kSharedInApplication (the server agent can be shared with any other agent, which were launched as part of the same application), kExclusive (the server agent can not be shared) and kSharedInFUN (the server agent can be shared with other agents which are members of the same FUN as the client agent). If the server agent can not be shared, the Manufacturer agent registers a new instance in the Orbix implementation repository.

Concrete proxy classes define a facade for each remote service method (the implementation calls the corresponding method on the associated CORBA proxy, apart from doing other project-specific tasks, such as logging the request). Concrete proxy classes provide a number of constructors that firstly call on the inherited `FindAgentNameForServices` method, and finally call on `ConnectServiceRequester`. The agent working memory class is a container for the agent's global variables.

`TcabFUN` inherits the normal functionality of an agent from `TcabAgent`. A FUN specifies a number of roles (`TcabRole`). `TcabFUN` provides a number of methods that for example allow a FUN to obtain the name of the agent playing a given role. From this name, the agent developer can construct a proxy for one of its services. It is only possible to specify joining criteria in roles in terms of services. Each role is specified by giving the name of the required service, the cardinality (the number of agents playing such a role) and the sharing mode of the agent taking on the role. This specification makes up the parameters of the constructor for `TcabRole`. The `Populate` method of this class contacts the Manufacturer agent in order for it to locate (or create) an appropriate server agent. Similarly, the `TcabFUN` class provides a similar method that calls `Populate` on all of its roles. The constructor for concrete FUN classes creates the corresponding roles, and calls `Populate`, which in turns calls `Populate` on all of its roles. The FUN API, implemented mostly by `TcabFUN`, `TcabRole` and `TcabAgent`, allows the FUN's programmer to refer to the agents making up such a FUN.

4. FORCE: An example

In order to illustrate our approach for building CORBA-based multi-agent systems, this section outlines the FUN analysis (section 4.1) and the implementation (section 4.2) of a distributed multi-agent browser for land military units (FORCE). The objective of this browser is to allow a number of military users to explore and analyse the ORBAT (Order Of Battle), a tree of military units (divisions, brigades, and so on), representing a future scene of the battlefield (with own and enemy forces).

FORCE allows a user to explore the ORBAT in the same way a file browser allows to explore a file system, evaluate which is the effect of moving a unit to another place in the tree and compare forces. Several users can be running browsers at the same time, but only one of them, the owner, can move a unit to another father, which causes changes to the information stored in some units. The rest of browsers have to update their windows if a change made by the owner affects the information they are displaying.

4.1 FUN analysis

It seems clear that there are two separate responsibilities in FORCE, the handling of the ORBAT and the graphical environment (which allows the user to analyse the ORBAT in a convenient way). Therefore it is quite natural to think of the browser as a FUN that specifies a
role for accessing the ORBAT, implemented by another agent. Figure 6 depicts FORCE FUN (the browser). Services S1 and S2 are provided to the end-user and not to other agents.

<table>
<thead>
<tr>
<th>Objective</th>
<th>To allow a military user to explore and analyse the ORBAT. Several users can be running browsers at the same time (the browser that acquires the resource ORBAT is the owner and can modify the ORBAT until it releases it).</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strategy</td>
<td>All browsers will make use of the same agent handling the ORBAT.</td>
</tr>
<tr>
<td>Services</td>
<td>S1. To allow the military user to explore the ORBAT, modify its structure and compare forces.</td>
</tr>
<tr>
<td></td>
<td>S2. To refresh windows when an owner modifies the ORBAT, if necessary (i.e., if a browser was showing information that has just been modified).</td>
</tr>
<tr>
<td></td>
<td>S3. To allow other agents to access the ORBAT (to retrieve unit information, the sons of a unit, to acquire/release the resource ORBAT, etc.).</td>
</tr>
<tr>
<td>Resources</td>
<td>R1. ORBAT.</td>
</tr>
</tbody>
</table>

**Figure 6.** FORCE FUN.

In order to carry out S1 service, this FUN draws on the *OperateOnOrderOfBattle* service provided by the agent playing the role of *ORBAT handler* (figure 7). All the instances of this FUN make use of such an agent. If the owner modifies the ORBAT, it notifies all FORCE FUNs, in order for them to consider whether or not they need to refresh their windows.

Finally, S3 service is delegated into the agent playing the role of ORBAT handler. S3 service corresponds to *OperateOnOrderOfBattle*, and is used by other agents. As we said in section 2, joining benefits do not have direct mapping into our software; it is only a way to specify that a FUN, in this case FORCE, delegates a service into an agent playing the corresponding role, which actually provides the service.

<table>
<thead>
<tr>
<th>Title</th>
<th>ORBAT handler</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cardinality</td>
<td>Exactly one, and shared by all FORCE FUNs.</td>
</tr>
<tr>
<td>Joining criteria</td>
<td><strong>Services:</strong> <em>OperateOnOrderOfBattle</em> (provides entries to retrieve unit information, the sons of a unit, to acquire/release the resource ORBAT, and to change the father of a unit).</td>
</tr>
<tr>
<td></td>
<td><strong>Resources:</strong> ORBAT ([R1]).</td>
</tr>
<tr>
<td>Joining benefits</td>
<td><strong>Services, resources:</strong> None. <strong>Service responsibilities:</strong> S3.</td>
</tr>
</tbody>
</table>

**Figure 7.** Role 1 for FORCE FUN.

Hence, the functionality of FORCE has been broken down into two agents: the browser and the ORBAT handler, each with its own responsibility. Breaking down the software this way makes it easier to design and implement than a monolithic version of FORCE. Furthermore, part of its functionality, the management of the ORBAT (the agent taking on this role), can be reused by other agents, those needing to access ORBAT information, as in fact it occurs in the GRACE demonstrator. This breakdown corresponds to the application of two architectural patterns: Presentation-Abstraction-Control and Model-View-Controller ([2]).
4.2 Implementation

Figure 4 shows the roles specified in Force FUN. Whenever a browser (Force FUN) is launched from the control panel, the Manufacturer agent checks if there are agents running that can be shared by this FUN, for fulfilling the three roles (two of them have not been commented in the previous section for brevity). Agent configuration files specify that OrbatHandler agent provides those three services and the name of the machine where to launch such an agent. Supposing that OrbatHandler is not running when the first browser is launched, the Manufacturer agent registers it in the Orbix implementation repository. If another user launches another browser, the same active OrbatHandler will be used.

Figure 8 depicts two browsers making use of the service provided by OrbatHandler. The initialise section of the browser creates a proxy for the service OperateOnOrderOfBattle (S3 in the figure, actually called Force.ForceOntology.OperateOnOrderOfBattle) and stores it in the agent’s working memory. Finally it creates an internal activity (A1 in the figure) and subscribes it to the notifications sent by the agent implementing S3 service (notifications will inform about the changes in the ORBAT which the activity is interested in). The internal activity runs the Ilog Views event processing loop (similar to the X Windows main loop) and a timer, which executes every given time period the internal activity’s notification handler to refresh the window, if necessary. The proxy stored in the working memory is used from the Ilog Views callbacks for accessing unit information, trying to acquire the resource ORBAT, releasing it, modifying the tree, etc.

OrbatHandler’s initialise section constructs a proxy for one of the services provided by the Wide Area Picture (WAP) agent, which wraps up a data base, for getting the unit information that makes up the ORBAT. It also creates an internal activity (A2) and subscribes it to the notifications provided by the WAP agent, for updating its own ORBAT when unit information changes as a consequence of other GRACE agents changing military information.
Whenever WAP sends a notification (1a), A2 updates its own ORBAT, and sends the notification (1b and 1c) in another format to OrbatHandler’s clients interested in such data. Whenever an owner browser modifies the ORBAT (2) by using the corresponding entry in S3 service, the thread dispatching it, notifies all browsers (2a).

5. Evaluation and related work

From the previous sections, it should be clear that the programmer need not learn the API of the underlying CORBA ORB. He/she only needs to deal with an easy-to-learn language in order to specify the key components of the agent, in which C++ will be embedded. There is no need to create CORBA objects, nor threads. CORBA objects (representing services) and external activities for executing these services are created automatically when needed. Internal and external activities are higher-level abstractions than threads, because they can rendezvous with other activities, decoupling the caller from the called activity, and therefore maximizing the parallelism. Furthermore, activities can subscribe to and send notifications.

When an agent constructs a proxy for a service, CABLE searches on run-time an appropriate agent providing such a service, launching it if it was not already running or launching a new instance if it can not be shared with the client agent. It is possible to specify different sharing modes for agents (shared, shared in application, shared in FUN and exclusive). Agents are killed when they are no longer used. Furthermore CABLE allows to specify FUNs in an elegant way. Summing up, CABLE tries to automate part of the work when building a distributed application with a CORBA ORB, by providing the programmer with high-level abstractions, in order for him/her to concentrate on the key components of the agent.

The ability to rendezvous with external activities, decoupling called and calling activities, has a similar goal as the Partial Processing ([14]) and Active Object ([10]) patterns. The advantage of the rendezvous mechanism presented here is that it is a higher-level abstraction, in the form of ADL sentences (as in Ada), and therefore very easy to use.

[3] also discusses the convenience of providing the programmer with a language, CORRELATE, that supports high-level abstractions (for example, active objects) in order for the programmer need not learn a complex framework. CORRELATE provides high-level abstractions for low-level concurrent distributed programming (the authors are building a I/O framework that handles networking, persistent storage and visualization). The only point in common (obviously the domain of application is different) with the approach presented here is the use of a language for hiding the underlying framework from the programmer. ADL is a template-like language instead of a complete language like CORRELATE. Among the benefits of using a template-like language based approach, in which the programmer will embed C++, is that there is no need to learn yet another language, but only some new constructs, and the possibility to use the huge amount of commercial software written in C++ (as it is the case of our project).

Caffeine ([15]) is a extension provided with Visibroker for Java (a CORBA ORB for Java). It allows to define CORBA interfaces as Java interfaces (in a similar way to Java RMI), and provides pass-by-value semantics for objects. Some local objects, conforming to a set of rules, will be passed by value by using equivalent IDL structs, and the rest of them (most of them), by using the Java object serialization mechanism ([12]), which is non CORBA-compliant, and therefore other CORBA applications will not be able to communicate with Caffeine applications most of the times.

Voyager ([7]) is a mobile agent system in Java, that apart from moving objects (and agents, a special class of objects) across the network, it allows an object to invoke the
methods of another remote object. Voyager includes a utility, “vcc”, to generates a remote-enabled class from an existing class. The “vcc” utility reads a “.class” or “.java” file and generate a new virtual class. This virtual class contains a superset of the original class methods. A client agent can call on the methods of a remote object by using a reference to the corresponding virtual class (virtual reference). This kind of transparency is even superior to the one provided by Java RMI or Caffeine because third-party classes can be remotely enabled. It also supports future messages, that is, the caller can call on a remote method without being blocked, and retrieve the return value later by polling, blocking or waiting for a callback. However, remote objects can not control when their methods are called in a similar way to the accept-select mechanism provided with CABLE. Voyager is also integrated with CORBA. A CORBA object can communicate with a Voyager object, and vice versa. However it has the same problem as Caffeine with regard to pass-by-value semantics, due to most of the times, other CORBA-compliant applications will not be able to communicate with a Voyager/Caffeine application.

CABLE has not addressed the problem of making more transparent the mapping of IDL to the target language, C++ in this case. Both Caffeine and Voyager can achieve this level of transparency because they rely on the Java object serialization mechanism in order to achieve pass-by-value semantics, which is non CORBA-compliant. In a compiled language, like C++, which lacks a similar mechanism, is not possible to achieve this level of transparency. Furthermore, both in Caffeine and Voyager is still necessary to learn the mapping of IDL to Java when communicating with other CORBA applications developed with other CORBA-compliant ORBs. The objective of CABLE is to provide the programmer with high-level abstractions for developing CORBA-based multi-agent systems, but no effort has been done in improving the transparency in the mapping of IDL to the target language (C++ in this case). [11] proposes an alternative mapping for C++ based on the C++ Standard Library. CABLE could use this mapping on top of the standard mapping and use the pass-by-value semantics for objects when the OMG comes up with a standard solution.

6. Concluding remarks

The multi-agent approach encouraged by the FUN methodology allows to break down the functionality of a complex system into a number of cooperating agents, which should not be complex in terms of design and programming. This approach is also useful in multi-company projects, where agents are developed by different companies; the multi-agent approach facilitates software integration. This is the case of EUCLID RTP 6.1, a large and complex C3I project, which is being carried out with the collaboration of 17 European companies/organisations.

The project has been running for four years. During this time, the functionality of the GRACE demonstrator has been broken down into a number of C3I facilities, which in many cases are also composed of a number of agents (like FORCE). In parallel with the development of CABLE, these facilities were implemented. At the beginning of 1997, the first operative release of CABLE was available, and all the facilities were agentised with CABLE, which enabled them to communicate one another. CABLE allowed to do this transition in a easy and quick way. As CABLE services are executed in different threads of control, some C++ upper-level classes needed to be made thread-safe, however this did not show to be an important problem. The current demonstrator is a true multi-agent system, where military users are not aware of the underlying cloud of agents. Furthermore, agents can be replaced by improved versions at run-time (plug and play of agents).
7. References

1. E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley, 1995.


