Abstract

Development of distributed applications is a difficult task. Object Requests Brokers (ORBs) facilitate the development of large and complex distributed applications. However, the programmer still has to deal with a complex framework, which increases the learning curve. In this paper, we present a middleware that tries to facilitate the transition from the design to the implementation. The middleware is made up of a C++ framework and an Agent Definition Language (ADL). The framework is implemented on top of a CORBA ORB and provides the programmer with high-level abstractions, such as “agent” and “FUN” (a special class of agent that delegate part of its functionality into a number of cooperating agents). ADL is a template-like language, which hides the underlying C++ framework from the programmer (including the CORBA ORB). The paper makes special emphasis in describing such a language.

1. Introduction

Object Requests Brokers (ORBs), such as those provided by CORBA ([1]), DCOM ([2]) or Java RMI ([3]) 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, and provides the most interoperable solution. However, the programmer still has to deal with a complex framework, which increases the learning curve. This is specially true for users with no previous experience in developing distributed applications. In this paper we describe a CORBA-based middleware, CABLE, that tries to facilitate the transition from the design to the implementation.

The origin of CABLE is the FUN (Functional UNit) methodology, that allows to break down the functionality of a system into a set of agents and FUNs, a special class of agents that delegate part of their functionality into a number of cooperating agents under the FUN’s coordination. We consider an agent as an autonomous entity that provides the agent community with services. CABLE is built on top of Orbix 2.2 MT ([4]), a CORBA 2.0 ([5]) compliant ORB, and provides 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).

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. [6] provides a detailed description of the project. The demonstrator being developed is made up of a number of graphical facilities that help military users in the decision taking process. All the facilities, and the demonstrator as a whole, have been analysed with the FUN approach, and implemented with CABLE.

The rest of the paper is organized as follows. Section 2 explains the FUN analysis, and section 3 presents an example of its use. Section 4 describes CABLE, making special emphasis in ADL. 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 organization. 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), and (5) the roles that cooperate to provide its services. A FUN, in the same way as a human organization, specifies the roles its members have to fulfil. 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 fulfil a given role, can be selected if the FUN supplies them with the resources and services they lack.

3. An example of use of the FUN analysis

In order to clarify the concepts presented above, this section outlines the FUN analysis 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).

<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 1. FORCE FUN.

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.

From the above description, 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 1 depicts FORCE FUN (the browser). Services S1 and S2 are provided to the end-user and not to other agents.

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 2). 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 \textit{OperateOnOrderOfBattle}, and is used by other agents. As we said in section 2, joining benefits do not have a 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 the FORCE FUNs.</td>
</tr>
<tr>
<td>Joining criteria</td>
<td>Services: \textit{OperateOnOrderOfBattle} (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). Resources: ORBAT ([R1]).</td>
</tr>
<tr>
<td>Joining benefits</td>
<td>Services, resources: None. Service responsibilities: S3.</td>
</tr>
</tbody>
</table>

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{example.png}
\caption{Role 1 for FORCE FUN.}
\end{figure}

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 access to the ORBAT).

4. \textbf{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 closer 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 ([15]) 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.

4.1. \textbf{The Agent Definition Language}

ADL provides a syntax for specifying three types of files, namely, \textit{service specification file} (it specifies a service in a similar way to a CORBA interface), \textit{type specification file} (it specifies, in CORBA IDL, a type used in an entry corresponding to a service), and \textit{agent definition file} (one per agent). In an agent definition file, the programmer specifies the key components of the agent, and provides an implementation for them (currently in C++), embedded into the ADL.

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 to communicate with a CABLE agent. As CABLE is built on top of a CORBA ORB, a CABLE agent can communicate with a CORBA agent.

Among the main components of an agent definition file, we can mention:

\begin{itemize}
\item \textit{Working memory}. Global data to all activities (see below) in the agent.
\item \textit{Initialise section}. Actions to be carried out when the agent comes into existence (for example, for initialising data members of the agent’s working memory).
\item \textit{Finalise section}. Actions to be carried out upon the destruction of the agent (for example, for deleting dynamically allocated data members of the agent’s working memory). Figure 2 shows an excerpt of a finalise section.
\end{itemize}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{example.png}
\caption{An excerpt of a finalise section.}
\end{figure}

\begin{verbatim}
finalise {
  insert C++ {
    MyWorkingMemory.fOperateOnOrderOfBattle->Finish();
    delete MyWorkingMemory.fOperateOnOrderOfBattle;
  }
}
\end{verbatim}

\begin{itemize}
\item \textit{Activities}. A CABLE agent is composed of multiple threads of control, called \textit{activities}, which look like Ada tasks. An activity is made up of a set of \textit{entries}, a \textit{run section} and a private \textit{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 \texttt{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 from the entries and the run section. In the run section, it is possible to specify \texttt{post-rendezvous} code after an \texttt{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. Figure 4 shows an excerpt of the implementation of an activity (service).

There are two types of activities: \textit{internal} and \textit{external ones}. The external activities are also called \textit{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 \texttt{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.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{example.png}
\caption{An excerpt of an activity.}
\end{figure}
Multiple instances of a given service can be running in parallel within an agent.

```
activity OperateOnOrderOfBattle implements service OperateOnOrderOfBattle {
    memory {
        insert C++ {
            bool fItIsTheOwner;
            ... >>
        }
    } << Other entries. >>
    entry GiveUnitComposition {in DontUnitCode aCode, in DontUnitSide aSide, out DontUnitList aUnitList} {
        insert C++ {
            MyWorkingMemory.fOrderOfBattle->
            GetZone(aCode, aSide, out aUnitList);
        }
    }
    run {
        insert C++ {
            bool endOfInteraction = false;
            accept([Start]);
            do {
                select {
                    ... >>
                    accept(GiveUnitInformation);
                    accept(GiveUnitComposition);
                    when (!fItIsTheOwner) {
                        accept(TryToAcquireTheOrderOfBattle) {
                            // Post-rendezvous code.
                            MyWorkingMemory.notifications.
                            NotifyTheOwnerHasChanged();
                        } // accept
                    } // select
                } while (!endOfInteraction);
            } // run
        } // activity
    }
}
```

Figure 4. An excerpt of an activity (service).

An agent definition file can define a FUN instead of a simple agent. Figure 5 shows an example of the definition of FORCE FUN (section 3), which specifies three roles (two of them have not been specified in section 3, for brevity). 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 running.

```
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 (services, internal activities, etc.). >>
}
```

Figure 5. An excerpt of a FUN definition.

4.2. The Manufacturer Agent

There is an agent, called the Manufacturer, running on a machine of the distributed system, whose role is to locate or to launch a providing agent (if it was not running) whenever an agent constructs a proxy for a service. When an agent is no longer required (no other agent is using its services), the Manufacturer kills it. Finally, it is worth noting that applications are launched and killed from the control panel, which is a browser for application configuration files. Such files specify the first agent to be launched for a given application. The rest of agents are launched in subsequent proxy constructions.

5. Related work

[7] also discusses the convenience of providing the programmer with a language, CORRELATE, that supports high-level abstractions 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). 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 ([8]) is an extension provided with Visibroker (a CORBA ORB for Java). It hides CORBA IDL, and its mapping to Java, from the programmer, by allowing him/her to define CORBA interfaces as Java interfaces, and provides pass-by-value semantics for objects. However, most of them will be passed (by the underlying implementation) by using the Java object serialization mechanism ([9]), which is not CORBA-compliant.

Voyager ([10]) is a mobile agent system in Java, and it is integrated with CORBA. With respect to CORBA, it provides a similar functionality to Caffeine, but it goes a step further, because it allows remote-enable third-party classes, but it suffers the same problem as Caffeine with respect to CORBA interoperability.

CABLE has not addressed the problem of making more transparent the mapping of CORBA IDL to the target language, C++ in this case. However, it provides the programmer with high-level abstractions, making possible a closer mapping from the design to the implementation, and hiding the underlying CORBA ORB from the programmer. The transparency in the IDL to C++ mapping could be improved by using the alternative mapping proposed in [11] (based on the C++ Standard Library), on top of the standard mapping.

6. Concluding remarks

EUCLID RTP 6.1 has been running for 4+ years. The functionality of the GRACE demonstrator was broken down into a number of C3I facilities, 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. This agentisation was easy and quick, because CABLE provides a closer mapping from the design to the implementation. The Agent Definition Language allowed to express the key components of agents, and to embed the existing C++ code (developed before CABLE was available). 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.

7. References