MODEL-BASED DISTRIBUTED OBJECT COMPUTING

by

Antoine F. Elhage

Submitted in partial fulfillment of the
requirements for the degree of

Master In Computer Science

Thesis Advisor

Haidar M. Harmanani

School of Arts and Sciences
LEBANESE AMERICAN UNIVERSITY
Byblos

July 2002

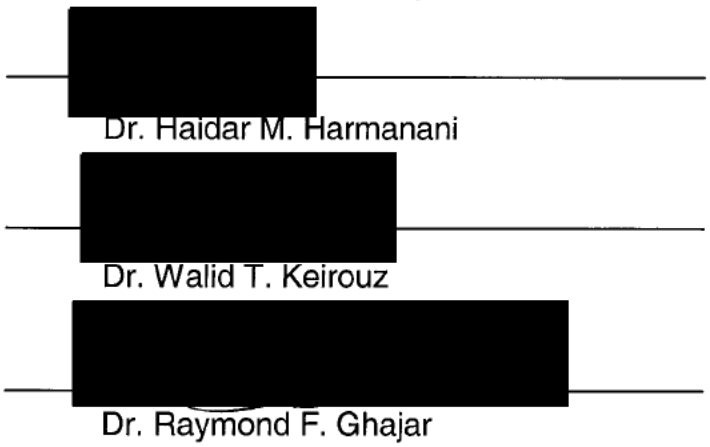
LEBANESE AMERICAN UNIVERSITY

GRADUATE STUDIES

We hereby approve the thesis of

Antoine F. Elhage

Candidate for the Master of Science degree:



Date: August 2, 2002

*We also certify that written approval has been obtained for any proprietary material contained therein.

I grant to the LEBANESE AMERICAN UNIVERSITY the right to use this work, irrespective of any copyright, for the University's own purpose without cost to the University or to its students, agents and employees. I further agree that the University may reproduce and provide single copies of the work, in any format other than in or from microforms, to the public for the cost of reproduction.

To my parents

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGMENTS

The author wishes to express sincere appreciation to Dr. Haidar Harmanani for his assistance in the preparation of this thesis. In addition, special thanks to Dr. Walid Keirouz, whose familiarity with the needs and ideas of the project was helpful during the phases of this undertaking.

Thanks also to the VCE department members of the Volvo Information Technology AB for their valuable input and support. For the type I spent with them in Sweden to accomplish this technology and make out of it a useful product.

I would like to express my sincere gratitude to the Lebanese American University whose financial support during my graduate studies made it all possible.

Finally, I would like to thank my friends and family for their long support.

MODEL-BASE DISTRIBUTED

OBJECT COMPUTING

ABSTRACT

by

Antoine F. Elhage

Four trends are shaping the future of commercial software development. First, the software industry is moving away from *programming* applications from scratch to *integrating* applications using reusable components. Second, there is great demand for *distribution technology* that provides remote method invocation and/or message-oriented middleware to simplify application collaboration. Third, there are increasing efforts to define standard software infrastructure frameworks that permit applications to interwork seamlessly throughout *heterogeneous* environments. Finally, next-generation distributed engineering modeling applications require *quality-of-service* (QoS) guarantees for latency, bandwidth, and reliability.

A key software technology supporting these trends is *distributed object computing (DOC) middlewar*e. DOC middleware facilitates the collaboration of local and remote application components in heterogeneous distributed environments. The goal of DOC middleware is to eliminate many tedious, error-prone, and non-portable aspects of developing and evolving distributed applications and services. At the heart of DOC middleware are *Object Request Brokers* (ORBs), such as CORBA, DCOM, and Java RMI

This thesis describes how the DOC concept was used to create a new *Model-based Distributed Object computing (MDOC)* technology that creates a robustness object oriented environment to optimize and automate modeling through the Intranet / Internet.

In particular this document will describe the architecture, security layers, connectivity, and the client/server application of MDOC.

CHAPTER 1. INTRODUCTION

1.1 General

Traditionally, product design, manufacturing and inspection process planning, finite element modeling and analysis were done independently as a part of the required processes for the design to-production cycle of custom parts. In the last 15 years, computer systems have been extensively used in product design automation, while the manufacturing process design and planning has remained a manual effort with little or no automation. In the area of finite element modeling and FEA, a number of applications have been developed to address these issues. As a result, independent solutions for design (CAD), analysis (FEA), and manufacturing (primarily machining) have been developed and are widely available. Although additional efforts have focused on integrating these individual applications, these efforts have not addressed the issues required to integrate and automate the entire design-to-manufacturing cycle for concurrent engineering. This trend has primarily taken the approach of integrating these applications by using file transfer from one to another. Since a design is evaluated by simply transferring the data from one application to another to execute the different processes, these applications do not share a common underlying part model that enables the dynamic linking of all these processes for bi-directional data exchange. The limitation to this approach is the lack of a unified part model that is the basic requirement for simultaneous engineering to enable the integration of the different disciplines involved in the design-to-production of custom parts. Therefore, no process knowledge for analysis, manufacturing, and inspection is made available at the design level to assist in the product design cycle.

Another shortcoming of current techniques is that they do not provide the direct interpretation of the part design to automatically generate the enhancement to the part geometry required to automate the down-stream processes. To overcome this deficiency, existing tools have focused on assisting the user in remodeling the part to create different representations as required for each downstream process. For example, tools for automatic mesh generation are widely used to assist the analysis engineers in creating the finite element model to be used for analysis. These tools, furthermore, enable the engineer to enhance the mesh attributes for deriving the model required for the analysis process. These tools do not enable the design engineer to enhance the part geometry as an input for analysis, therefore, eliminating the intermediate steps. Similarly, manufacturing and inspection applications require the user to redesign the part in terms of manufacturing and inspection features for process automation rather than directly interpreting the design for inspection and manufacturability. Therefore, existing CAD/CAM/CAE systems are islands of automation that require manual remodeling of the part model to automate the relevant processes such as machining, analysis, and inspection.

1.2 Problem Description

Concurrent engineering requires the engineers from various disciplines to interact with each other for resolving conflicts that may occur because of the different requirements set by the downstream processes. To simulate these interactions, a unified part model to enable the multiple representations and requirements of the downstream processes is required. A unified part model enables engineers from various disciplines to simultaneously interact with the part representation for concurrent engineering. Another requirement of concurrent engineering is a single

underlying framework that seamlessly integrates the different processes required for the multiple engineering disciplines without resorting to creating multiple models (which contradicts the requirements of a unified model).

In addition, open access to the part representation is another important requirement because it enables the engineer to interact dynamically with the unified part model and conduct "what-if": scenarios to evaluate alternative designs and processes.

Unlike existing CAD systems, MDOC uses a unique, single, underlying object-oriented architecture that supports a virtual layer topology assuring seamless integration of the different processes involved in the engineering cycles of custom parts. The part (geometry, features, materials, function, etc.) and process (manufacturing, inspection, analysis) designs are concurrently generated from, and stored in, a single part model. This part model reflects the hierarchy of the design intent. The relations among the various processes and their dependencies on the part geometry are captured with that unified model. Dependencies are automatically tracked and used to trigger the constraints to enforce these relationships.

Another important feature missing in existing engineering systems is knowledge-based modeling. Knowledge or expertise as related to the part geometry cannot be captured into a generic methodology to assist in modeling and production planning of custom parts. To address this deficiency MDOC enables the user to define the part in a modular, object-oriented fashion so that the part components can be used to address a wide spectrum of applications. Furthermore, MDOC objects referencing the process definition (analysis, manufacturing, inspection) and its relationship with the design also capture the knowledge to automate these

processes. Inheritance (as supported by MDOC) enables the user to combine a number of existing objects to form a new part definition, modify its behavior, and deduce its processes through using the inherited knowledge.

JAVA/RMI and Lisp were the chosen technologies to base MDOC on. RMI empowers MDOC to distribute models, in the sense that it has at least two components running on different machines and links transparently the non-distributed applications that are limited in scalability. Thinking of these applications as distributed applications and running the right components in the right places benefits the user and optimizes the use of network and computer resources. Lisp empowers MDOC the multiple inheritance capability, tagging and propagation technique, and the re-generation of parts from the fixed mesh.

VOLVO was the first car company to adopt the MDOC Technology.

It used to take engineers at VOLVO a couple of months to finalize, analyze and test a new design. Whenever they invented a new design, engineers used to take this new design and generate various data format from it. These data were then passed to different applications for the purpose of analyzing and testing. When the output of these applications was collected, they used to check it and tweet the design accordingly; then reformat and pass the data again to other applications until they reach the stable optimized version.

Collaborating with Dr. Haidar Harmanani, VOLVO assigned a team of 7 engineers including myself to take this technology further into production.

MDOC was able to automate the process of importing a new design, link all these applications to the design, and provides graphical capabilities to control the quality of the finite element, repair them, backward propagation, and re-generation of parts out of the fixed mesh.

1.3 Outline

Chapter 2 will cover the existing techniques with a brief description of the architecture of each one plus a comparison between them. Chapter 3 will detail the solution approach and the architecture of MDOC. Chapter 4 will cover the case study of the VOLVO truck Cab plus snapshots. Chapter 5 will conclude the thesis and go over the further studies and research.

CHAPTER 2. REVIEW OF LITERATURE

2.1 Introduction

Distributing an application is not an end in itself. Distributed applications introduce a whole new kind of design and deployment issues. For this added complexity to be worthwhile, there has to be a significant payback.

Some applications are inherently distributed: multi-user games, chat and teleconferencing applications are examples of such applications. For these, the benefits of a robust infrastructure for distributed computing are obvious.

Many other applications are also distributed, in the sense that they have at least two components running on different machines. But because these applications were not designed to be distributed, they are limited in scalability and ease of deployment. Any kind of workflow or groupware application, most client/server applications, and even some desktop productivity applications essentially control the way their users communicate and cooperate. Thinking of these applications as distributed applications and running the right components in the right places benefits the user and optimizes the use of network and computer resources. The application designed with distribution in mind can accommodate different clients with different capabilities by running components on the client side when possible and running them on the server side when necessary.

Designing applications for distribution gives the system manager a great deal of flexibility in deployment.

Distributed applications are also much more scalable than their monolithic counterparts. If all the logic of a complex application is contained in a single module, there is only one-way to increase the throughput without tuning the

application itself: faster hardware (Johnson, 1997). Today's servers and operating systems scale very well but it is often cheaper to buy another identical machine than to upgrade to a server that is twice as fast. With a properly designed distributed application, a single server can start out running all the components. When the load increases, some of the components can be deployed to additional lower-cost machines. Three of the most popular distributed object paradigms are Microsoft's Distributed Component Object Model (DCOM), OMG's Common Object Request Broker Architecture (CORBA) and JavaSoft's Java/Remote Method Invocation (Java/RMI). Let us go over an introduction of the old techniques and examine the differences between the three distributed object paradigms from a programmer's standpoint and an architectural standpoint.

2.2 Client-Server Programming

TCP and IP were developed by a Department of Defense (DOD) research project to connect a number different networks designed by different vendors into a network of networks (the "Internet"). It was initially successful because it delivered a few basic services that everyone needs (file transfer, electronic mail, remote logon) across a very large number of client and server systems. Several computers in a small department can use TCP/IP (along with other protocols) on a single LAN. The IP component provides routing from the department to the enterprise network, then to regional networks, and finally to the global Internet. On the battlefield a communications network will sustain damage, so the DOD designed TCP/IP to be robust and automatically recover from any node or phone line failure. This design allows the construction of very large networks with less central management (Gilbert, 1995). However, because of the automatic recovery, network problems can go undiagnosed and uncorrected for long periods of time

As with all other communications protocol, TCP/IP is composed of layers:

- IP - is responsible for moving packet of data from node to node. IP forwards each packet based on a four-byte destination address (the IP number). The Internet authorities assign ranges of numbers to different organizations. The organizations assign groups of their numbers to departments. IP operates on gateway machines that move data from department to organization to region and then around the world.
- TCP - is responsible for verifying the correct delivery of data from client to server. Data can be lost in the intermediate network. TCP adds support to detect errors or lost data and to trigger retransmission until the data is correctly and completely received.
- Sockets - is a name given to the package of subroutines that provide access to TCP/IP on most systems.

2.3 TCP/IP Levels

There are three levels of TCP/IP knowledge. Those who administer a regional or national network must design a system of long distance phone lines, dedicated routing devices, and very large configuration files. They must know the IP numbers and physical locations of thousands of subscriber networks. They must also have a formal network monitor strategy to detect problems and respond quickly (Gilbert, 1995).

Each large company or university that subscribes to the Internet must have an intermediate level of network organization and expertise. Half dozen routers might be configured to connect several dozen departmental LANs in several buildings. All traffic outside the organization would typically be routed to a single connection to a regional network provider.

However, the end user can install TCP/IP on a personal computer without any knowledge of either the corporate or regional network. Three pieces of information are required:

- The IP address assigned to this personal computer
- The part of the IP address (the subnet mask) that distinguishes other machines on the same LAN (messages can be sent to them directly) from machines in other departments or elsewhere in the world (which are sent to a router machine)
- The IP address of the router machine that connects this LAN to the rest of the world.

In the case of the PCLT server, the IP address is 130.132.59.234. Since the first three bytes designate this department, a "subnet mask" is defined as 255.255.255.0 (255 is the largest byte value and represents the number with all bits turned on). It is a Yale convention (which we recommend to everyone) that the routers for each department have station number 1 within the department network. Thus the PCLT router is 130.132.59.1. Thus the PCLT server is configured with the values:

My IP address: 130.132.59.234

Subnet mask: 255.255.255.0

Default router: 130.132.59.1

The subnet mask tells the server that any other machine with an IP address beginning 130.132.59.* is on the same department LAN, so messages are sent to it directly. Any IP address beginning with a different value is accessed indirectly by sending the message through the router at 130.132.59.1 (which is on the departmental LAN).

2.4 *Remote Procedure Calls*

Remote procedure calls (RPC) are exactly what they sound like: a model in which client code invokes a procedure on a remote server. This is a bit further up the chain of abstraction, whereas sockets are trapped at the lowest common level, data – RPCs look to the calling application like normal function or subroutine call. RPCs put distributed processing into a model familiar to all programmers. An application programmer is able to include a header file and some libraries and make calls to the remote resource (Christopher, 2000).

RPCs are not, of course, immune to the challenge of network applications. They must deal with unexpected loses of connectivity. They must also deal with the problem of differing platform data formats. But where sockets left you largely to you own devices, RPCs build some support into the RPC runtime code. Losses of connectivity are handled with an internal timeout mechanism. If no reply is received in a specific interval, an error is propagated up into the application making the call. This is novel for a programmer new to distributed computing, of course. When was the last time you called a well-tested function in a monolithic program only to have it suddenly fail for reasons outside your control? Almost never, if it is truly well tested. Almost everything in a monolithic application is within you control. Still, programmers trying to move to distributed processing have to get used to some new ideas, no matter how great the support provided by implementing technology.

Rather than map from each supported platform to every other supported platform, RPCs support a wire format common to all platforms. It specifies all the basic data types including their size and byte order. It is an arbitrary format designed to provide common ground for all platforms, which use it. It becomes the

responsibility of each client and each server to translate from their binary format into the common format and vice versa. Each RPC call, then, involves four translations:

- Client to wire format
- Wire to server format
- Server back to wire (to return value)
- Wire format back to client format

Unlike socket programming, RPC programming tools programmers a bit more support. Function interfaces – the parameter list and the return type – are specified in an interface definition language. IDL is a text-based description of interfaces, their methods, parameters, and return values. At build time, an IDL compiler generates proxies and stubs. The client application compiles and links one of these onto its executable image to handle client-side translations, and the server application that implements the body of the RPC compiles and links the other. RPCs are a good model for programmers in as much as every programmer is intimately familiar with the concept of functions and subroutines (Christopher, 2000). The protocol of RPCs is implied by the functional interface: call the subroutine synchronously with a given parameter list and wait until a known return type is sent back from the server. Although synchronous calls have problems in terms of scalability, this is much simpler than having to develop a protocol and implement a state machine. Where good RPC development tools are available for a platform, RPCs are one of the most accessible technologies for doing distributed processing.

There is one area in which RPCs are somewhat lacking, though, and it is the matter of maintaining state. State is essential to a traditional, top-down structured

application that is handled through global data. Obviously, though, when you have a central server supporting multiple clients, the server has to maintain state information for each client, or each client has to maintain state information for itself. In the latter case, the state information must be provided to the server every time an RPC is invoked.

If the server is stateful, something in the functional interface must suggest this. The function call metaphor does not cover the idea of holding information from one call to another, so you typically end up with calls along the lines of SetSomeValue().It works, but there should be a more intuitive way f handling this.

Another factor that diminished the popularity of RPCs is the rise of object-oriented programming and its close cousin, component software. RPCs hark back to the heyday of structured programming – the 1970s. The fashion for nearly twenty years has been objects, so it should not be surprising that we should desire an object metaphor for distributed computing. Happily, there are several such technologies, and they address the state issue.

2.5 Remote Method Invocation

Remote method invocation allows Java developers to invoke object methods, and have them execute on remote Java Virtual Machines (JVMs). Under RMI, entire objects can be passed and returned as parameters, unlike many remote procedure call based mechanisms that require parameters to be either primitive data types, or structures composed of primitive data types. That means that any Java object can be passed as a parameter - even new objects whose class has never been encountered before by the remote virtual machine (Reilly, 1998).

This is an exciting property, because it means that new code can be sent across a network and dynamically loaded at run-time by foreign virtual machines. Java

developers have a greater freedom when designing distributed systems, and the ability to send and receive new classes is an incredible advantage. Developers don't have to work within a fixed codebase - they can submit new classes to foreign virtual machines and have them perform different tasks. When working with remote services, RMI clients can access new versions of Java services as they are made available - there's no need to distribute code to all the clients that might wish to connect. While code can be accessed from a local or remote file-system, it can also be accessed via a web server, making distribution easier. RMI also supports a registry, which allows clients to perform lookups for a particular service. The following diagram shows the interaction between different components of an RMI system. Clients that know about a service can look up its location from a registry and access the service. If a new class is required, it can be downloaded from a web server.

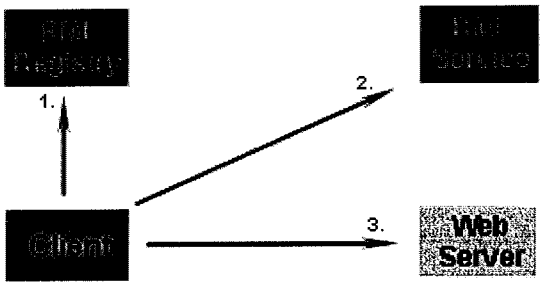


Figure 2-1 Client connects to a registry server, accesses a RMI service, and downloads new code as required from a web server.

Remote method invocation has a lot of potential, from remote processing and load sharing of CPU's to transport mechanisms for higher-level tasks, such as mobile agents that execute on remote machines (Reilly, 1998). Because of the flexibility of remote method invocation, it has become an important tool for Java developers

when writing distributed systems. However, there are many legacy systems written in C/C++, Ada, Fortran, Cobol, and other exotic languages. If legacy systems need to interface with your RMI systems, or your RMI systems need to interface with them, problems can occur. RMI is Java specific, and you'll need to write a bridge between older systems.

2.5.1 The RMI Architecture

The RMI system is designed to provide a direct, simple foundation for distributed object oriented computing. The architecture is designed to allow for future expansion of server and reference types so that RMI can add features in a coherent way (Reilly, 1998).

When a server is exported, its reference type is defined. The references for these objects are appropriate for UnicastRemoteObject servers, which are point-to-point unreplicated servers. Different server types would have different reference semantics. For example, a MulticastRemoteObject would have reference semantics that allowed for a replicated service (Wollrath, Riggs & Waldo, 1996).

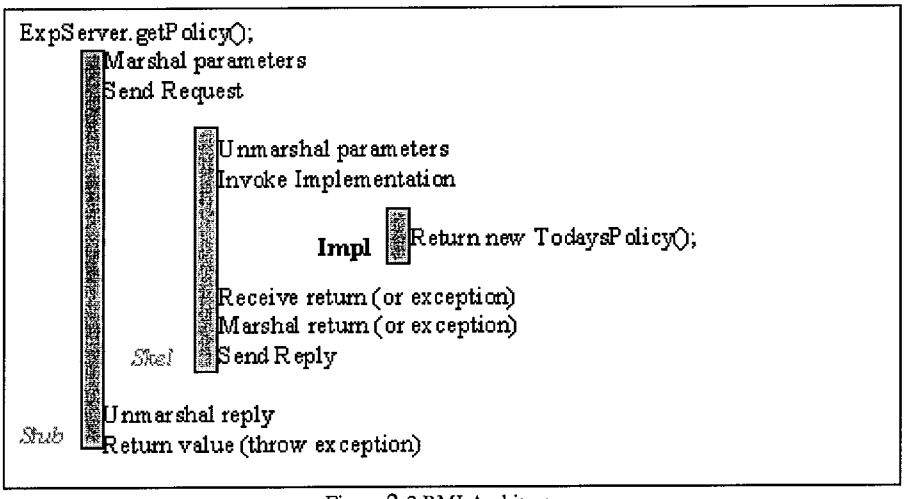


Figure 2-2 RMI Architecture

When a client receives a reference to a server, RMI downloads a stub that translates calls on that reference into remote calls to the server. As shown in Figure 2, the stub marshals the arguments to the method using object serialization, and sends the marshalled invocation across the wire to the server. On the server side the call is received by the RMI system and connected to a skeleton, which is responsible for unmarshalling the arguments and invoking the server's implementation of the method. When the server's implementation completes, either by returning a value or by throwing an exception, the skeleton marshals the result and sends a reply to the client's stub. The stub unmarshals the reply and either returns the value or throws the exception as appropriate. Stubs and skeletons are generated from the server implementation, usually using the program rmic. Stubs use references to talk to the skeleton. This architecture allows the reference to define the behavior of communication. The references used for UnicastRemoteObject servers communicate with a single server object running on a particular host and port. With the stub/reference separation RMI will be able to add new reference types (Reilly, 1998). A reference that dealt with replicated servers would multicast server requests to an appropriate set of replicants, gather in the responses, and return an appropriate result based on those multiple responses. Another reference type could activate the server if it was not already running in a virtual machine. The client would work transparently with any of these reference types.

2.6 Microsoft DCOM

DCOM that is often called '*COM on the wire*' supports remoting objects by running on a protocol called the Object Remote Procedure Call (ORPC). This ORPC layer

is built on top of DCE's RPC and interacts with COM's run-time services. A DCOM server is a body of code that is capable of serving up objects of a particular type at runtime. Each DCOM server object can support multiple interfaces each representing a different behavior of the object. A DCOM client calls into the exposed methods of a DCOM server by acquiring a pointer to one of the server object's interfaces. The client object then starts calling the server object's exposed methods through the acquired interface pointer as if the server object resided in the client's address space. As specified by COM, a server object's memory layout conforms to the C++ vtable layout. Since the COM specification is at the binary level it allows DCOM server components to be written in diverse programming languages like C++, Java, Object Pascal (Delphi), Visual Basic and even COBOL. As long as a platform supports COM services, DCOM can be used on that platform. DCOM is now heavily used on the Windows platform.

2.6.1 The DCOM Architecture

DCOM is an extension of the Component Object Model (COM). COM defines how components and their clients interact. This interaction is defined such that the client and the component can connect without the need of any intermediary system component. The client calls methods in the component without any overhead whatsoever (Box, 1997). Figure 3 illustrates this in the notation of the Component Object Model:

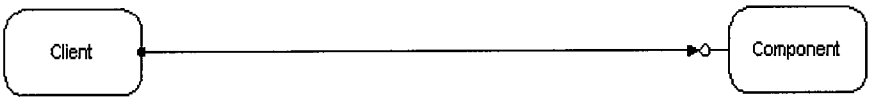


Figure 2-3 COM components in the same process

In today's operating systems, processes are shielded from each other. A client that needs to communicate with a component in another process cannot call the component directly, but has to use some form of interprocess communication provided by the operating system. COM provides this communication in a completely transparent fashion: it intercepts calls from the client and forwards them to the component in another process. Figure 4 illustrates how the COM/DCOM run-time libraries provide the link between client and component.

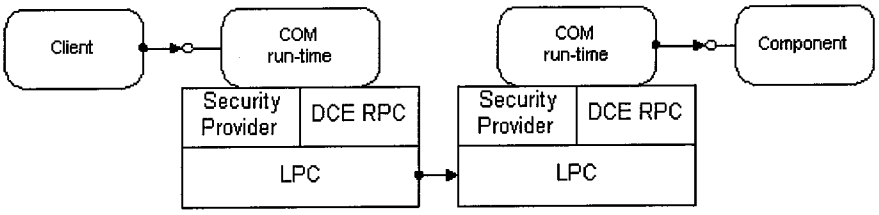


Figure 2-4 COM components in different processes

When client and component reside on different machines, DCOM simply replaces the local interprocess communication with a network protocol. Neither the client nor the component is aware that the wire that connects them has just become a little longer (Box, 1997).

Figure 5 shows the overall DCOM architecture: The COM run-time provides object-oriented services to clients and components and uses RPC and the security provider to generate standard network packets that conform to the DCOM wire-protocol standard.

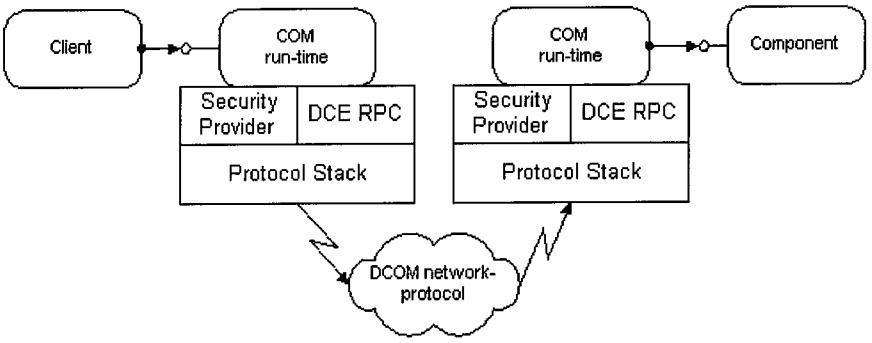


Figure 2-5 DCOM: COM components on different machines

2.7 CORBA

Common Object Request Broker Architecture (CORBA) is a competing distributed systems technology that offers greater portability than remote method invocation. Unlike RMI, CORBA isn't tied to one language, and as such, can integrate with legacy systems of the past written in older languages, as well as future languages that include support for CORBA. CORBA isn't tied to a single platform (a property shared by RMI), and shows great potential for use in the future. That said, for Java developers, CORBA offers less flexibility, because it doesn't allow executable code to be sent to remote systems (Morgan, 1998). CORBA services are described by an interface, written in the Interface Definition Language (IDL). IDL mappings to most popular languages are available, and mappings can be written for languages written in the future that require CORBA support (Reilly, 1998). CORBA allows objects to make requests of remote objects (invoking methods), and allows data to be passed between two remote systems. Remote method invocation, on the other hand, allows Java objects to be passed and returned as parameters. This allows new classes to be passed across virtual

machines for execution (mobile code). CORBA only allows primitive data types, and structures to be passed - not actual code.

Under communication between CORBA clients and CORBA services, method calls are passed to Object Request Brokers (ORBs). These ORBs communicate via the Internet Inter-ORB Protocol (IIOP). IIOP transactions can take place over TCP streams, or via other protocols (such as HTTP), in the event that a client or server is behind a firewall. The following diagram shows a client and a servant communicating.

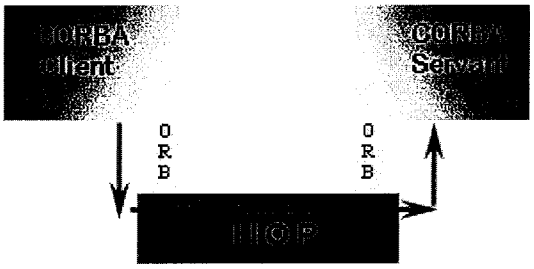


Figure 2-6 CORBA client sends a request through its local ORB to a remote ORB's servant

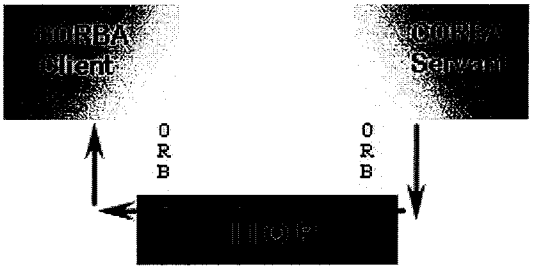


Figure 2-7 CORBA servant sends back a response to a remote ORB

2.7.1 The CORBA ORB Architecture

The following figure illustrates the primary components in the CORBA ORB architecture. Descriptions of these components are available below the figure.

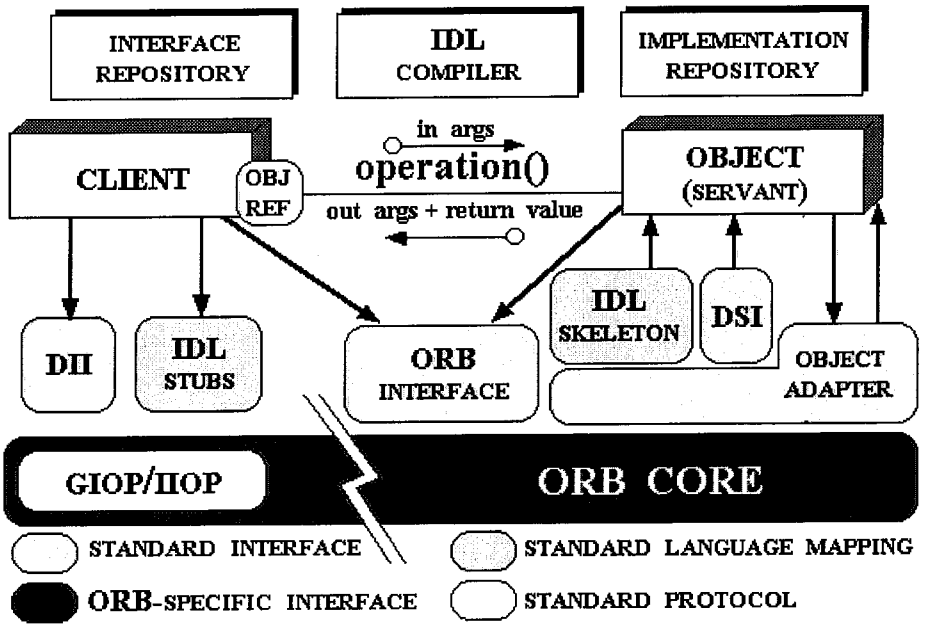


Figure 2-8 CORBA and ORBs Architecture

- Object -- This is a CORBA programming entity that consists of an *identity*, an *interface*, and an *implementation*, which is known as a *Servant*.
- Servant -- This is an implementation programming language entity that defines the operations that support a CORBA IDL interface. Servants can be written in a variety of languages, including C, C++, Java, Smalltalk, and Ada.
- Client -- This is the program entity that invokes an operation on an object implementation. Accessing the services of a remote object should be transparent to the caller. Ideally, it should be as simple as calling a method on an object, i.e., obj->op(args). The remaining components in Figure 8 help to support this level of transparency.
- Object Request Broker (ORB) -- The ORB provides a mechanism for transparently communicating client requests to target object implementations. The ORB simplifies distributed programming by

decoupling the client from the details of the method invocations. This makes client requests appear to be local procedure calls. When a client invokes an operation, the ORB is responsible for finding the object implementation, transparently activating it if necessary, delivering the request to the object, and returning any response to the caller.

- ORB Interface -- An ORB is a logical entity that may be implemented in various ways (such as one or more processes or a set of libraries). To decouple applications from implementation details, the CORBA specification defines an abstract interface for an ORB. This interface provides various helper functions such as converting object references to strings and vice versa, and creating argument lists for requests made through the dynamic invocation interface described below.
- CORBA IDL stubs and skeletons -- CORBA IDL stubs and skeletons serve as the ``glue'' between the client and server applications, respectively, and the ORB. The transformation between CORBA IDL definitions and the target programming language is automated by a CORBA IDL compiler. The use of a compiler reduces the potential for inconsistencies between client stubs and server skeletons and increases opportunities for automated compiler optimizations.
- Dynamic Invocation Interface (DII) -- This interface allows a client to directly access the underlying request mechanisms provided by an ORB. Applications use the DII to dynamically issue requests to objects without requiring IDL interface-specific stubs to be linked in. Unlike IDL stubs (which only allow RPC-style requests), the DII also allows clients to make

non-blocking *deferred synchronous* (separate send and receive operations) and *one-way* (send-only) calls.

- Dynamic Skeleton Interface (DSI) -- This is the server side's analogue to the client side's DII. The DSI allows an ORB to deliver requests to an object implementation that does not have compile-time knowledge of the type of the object it is implementing. The client making the request has no idea whether the implementation is using the type-specific IDL skeletons or is using the dynamic skeletons.
- Object Adapter -- This assists the ORB with delivering requests to the object and with activating the object. More importantly, an object adapter associates object implementations with the ORB. Object adapters can be specialized to provide support for certain object implementation styles (such as OODB object adapters for persistence and library object adapters for non-remote objects).

2.8 Comparison

As mentioned above three of the most popular distributed object paradigms are Microsoft's Distributed Component Object Model (DCOM), OMG's Common Object Request Broker Architecture (CORBA) and JavaSoft's Java/Remote Method Invocation (Java/RMI). Let us examine the differences between these three models from a programmer's standpoint and an architectural standpoint. At the end, you will be able to better appreciate the merits and innards of each of the distributed object paradigms. And you will know what distributed object MDOC is based on.

CORBA relies on a protocol called the Internet Inter-ORB Protocol (IIOP) for remoting objects. Everything in the CORBA architecture depends on an Object

Request Broker (ORB). The ORB acts as a central Object Bus over which each CORBA object interacts transparently with other CORBA objects located either locally or remotely. Each CORBA server object has an interface and exposes a set of methods. To request a service, a CORBA client acquires an object reference to a CORBA server object. The client can now make method calls on the object reference as if the CORBA server object resided in the client's address space. The ORB is responsible for finding a CORBA object's implementation, preparing it to receive requests, communicate requests to it and carry the reply back to the client. A CORBA object interacts with the ORB either through the ORB interface or through an Object Adapter - either a Basic Object Adapter (BOA) or a Portable Object Adapter (POA). Since CORBA is just a specification, it can be used on diverse operating system platforms from mainframes to UNIX boxes to Windows machines to handheld devices as long as there is an ORB implementation for that platform. Major ORB vendors like Inprise have CORBA ORB implementations through their VisiBroker product for Windows, UNIX and mainframe platforms and Iona through their Orbix product (Vinky, Bakken & Schantz, 1997).

DCOM that is often called '*COM on the wire*', supports remoting objects by running on a protocol called the Object Remote Procedure Call (ORPC). This ORPC layer is built on top of DCE's RPC and interacts with COM's run-time services. A DCOM server is a body of code that is capable of serving up objects of a particular type at runtime. Each DCOM server object can support *multiple interfaces* each representing a different behavior of the object. A DCOM client calls into the exposed methods of a DCOM server by acquiring a pointer to one of the server object's interfaces. The client object then starts calling the server object's exposed methods through the acquired interface pointer as if the server object resided in

the client's address space. As specified by COM, a server object's memory layout conforms to the C++ vtable layout. Since the COM specification is at the binary level it allows DCOM server components to be written in diverse programming languages like C++, Java, Object Pascal (Delphi), Visual Basic and even COBOL. As long as a platform supports COM services, DCOM can be used on that platform. DCOM is now heavily used on the Windows platform. Companies like Software AG provide COM service implementations through their Entire X product for UNIX, Linux and mainframe platforms; Digital for the Open VMS platform and Microsoft for Windows and Solaris platforms (Box, 1997).

Java/RMI relies on a protocol called the Java Remote Method Protocol (JRMP). Java relies heavily on Java Object Serialization, which allows objects to be marshaled (or transmitted) as a stream. Since Java Object Serialization is specific to Java, both the Java/RMI server object and the client object have to be written in Java. Each Java/RMI Server object defines an interface, which can be used to access the server object outside of the current Java Virtual Machine (JVM) and on another machine's JVM. The interface exposes a set of methods, which are indicative of the services offered by the server object. For a client to locate a server object for the first time, RMI depends on a naming mechanism called an RMI Registry that runs on the Server machine and holds information about available Server Objects. A Java/RMI client acquires an object reference to a Java/RMI server object by doing a lookup for a Server Object reference and invokes methods on the Server Object as if the Java/RMI server object resided in the client's address space. Java/RMI server objects are named using URLs and for a client to acquire a server object reference, it should specify the URL of the server object as you would with the URL to a HTML page. Since Java/RMI relies

on Java, it can be used on diverse operating system platforms from mainframes to UNIX boxes to Windows machines to handheld devices as long as there is a Java Virtual Machine (JVM) implementation for that platform. In addition to Javasoft and Microsoft, a lot of other companies have announced Java Virtual Machine ports (Reilly, 1998).

2.8.1 Comparing Apples to Apples

CORBA 3.0 will add a middleware component model (much like MTS or EJB) to CORBA. Since it is still in a pre-spec stage, we do not know much about how the CORBA middleware component model is going to look like. As of CORBA 2.x, there is no middleware component model that CORBA defines. There is a lot of comparison going on between COM and EJB. This is entirely wrong. This is like comparing apples to oranges. The competing technologies are MTS and EJB. Hence, the real comparison should be between MTS and EJB (Vinoski, 1997).

2.8.2 A Note on the Terminology

Let us now look at some of the terminology.

- Resource Dispenser (RD)

  The ODBC Resource Dispenser manages a pool of database connections. The Resource Dispenser can also reclaim connections for use by other clients. When a client gets a connection from the Dispenser, it is automatically enlisted in the object's transaction. Resource Dispensers are about making efficient use of a limited pool of resources.

- Resource Manager (RM)

  Resource Managers are the services that provide the resources to be used in transactions. Resource Managers are about managing the effect of transactions

on resources by making sure that the effects of a transaction are either committed or rolled back accordingly.

- Microsoft Distributed Transaction Coordinator (MS-DTC)

  A transaction is an atomic unit of work: either all the actions in a transaction are committed or none of them are. Work can be committed as an atomic transaction even if it spans multiple resource managers, potentially on separate computers. A Transaction Coordinator is a transaction manager that coordinates transactions, which span multiple resource managers. In the MTS environment, a transaction is coordinated by the Distributed Transaction Coordinator (DTC).

- Shared Property Manager (SPM)

  Shared Properties are properties global to the middleware MTS component. Since these properties may be shared with multiple clients, there is a distinct problem of Synchronization that has to be handled. The Shared Property Manager synchronizes access to shared properties.

- Microsoft Message Queue (MSMQ)

  The Microsoft Message Queue Server (MSMQ) guarantees a simple, reliable and scalable means of asynchronous communication freeing up client apps to do other tasks without waiting for a response from the other end. It provides loosely-coupled and reliable network communications services based on a messaging queuing model. MSMQ makes it easy to integrate applications, implement a push-style business event delivery environment between applications, and build reliable applications that work over unreliable but cost-effective networks.

- COM Transaction Integrator (COMTI)

  COM Transaction Integrator (COMTI) enables Microsoft Transaction Server to execute applications running under CICS or IMS on an IBM MVS system. COMTI ships with SNA Server version 4.0

- Java Naming and Directory Interface (JNDI)

  An API for naming-service-independent resource location. This provides Java applications with a unified interface to multiple naming and directory services on the enterprise. JNDI enables seamless connectivity to heterogeneous enterprise naming and directory services. Developers can now build powerful and portable directory-enabled applications using this industry standard.

- Java Transaction Service (JTS)

  This is an API to ensure data integrity across several systems and databases using two-phased commits and rollbacks. The API is compatible with the OMG's Object Transaction Service (OTS). The Java Transaction API (JTA) specifies standard Java interfaces between a transaction manager and the parties involved in a distributed transaction system: the resource manager, the application server, and the transactional applications.

- Java Message Service (JMS)

  JMS provides a reliable, flexible service for the asynchronous exchange of critical business data and events throughout an enterprise. The JMS API adds to this a common API and provider framework that enables the development of portable, message based applications in the Java programming language. The JMS API improves programmer productivity by defining a common set of messaging concepts and programming strategies that will be supported by all JMS technology-compliant messaging systems.

Middleware Component Models take a high level approach to building distributed systems. They free the application developer to concentrate on programming only the business logic, while removing the need to write all the "plumbing" code that is required in any enterprise application development scenario. For example, the enterprise developer no longer needs to write code that handles transactional behavior, security, database connection pooling or threading, because the architecture delegates this task to the server vendor.

When it comes to Middleware Component Models there are several *'de-jure'* and *'de-facto'* Standards in the industry today. In this article, we will look at two of them namely, The Microsoft Transaction Server (MTS) Architecture from Microsoft and JavaSoft's Enterprise JavaBeans (EJB). Let us examine the differences between these models from a programmer's standpoint and an architectural standpoint. At the end, you will be able to better appreciate the merits and innards of each of these Middleware Component Architectures.

MTS, based on the Component Object Model (COM) which is the *middleware component model* for Windows NT, is used for creating scalable, transactional, multi-user and secure enterprise-level server side components. MTS provides a surrogate for in-process server-side components. MTS can also be defined as a component-based programming model. An MTS component is a type of COM component that executes in the MTS run-time environment. MTS supports building enterprise applications using ready-made MTS components and allows you to "plug and work" with off-the shelf MTS components developed by component developers. Just as a COM component can be modeled on the basis of interfaces and their implementation, MTS enforces modeling based on the component's state and behavior. MTS, through the COM infrastructure, handles

communication between components using DCOM. This allows MTS to expose its components to Windows applications from anywhere on the net or the web. As long as you are only running on Windows NT, MTS components can be deployed on top of existing transaction processing systems including traditional transaction processing monitors, web servers, database servers, application servers, etc. Legacy system integration with non-Windows systems and MTS is achieved using the COM Transaction Integrator (COM-TI) technology. Also, it is important to realize that MTS is a stateless component model, whose components are always packaged as an in-proc DLL. Since they are composed of COM objects, MTS components can be implemented in a variety of different languages including C++, Java, Object Pascal (Delphi), Visual Basic and even COBOL!

EJB is a *middleware component model* for Java and CORBA and is a specification for creating server-side, scalable, transactional, multi-user and secure enterprise-level applications. It defines a set of specifications and a consistent component architecture framework for creating distributed n-tier middleware. It would be fair to call a bean written to EJB spec a Server Enterprise Bean. Most importantly, EJBs can be deployed on top of existing transaction processing systems including traditional transaction processing monitors, web servers, database servers, application servers, etc. Since these components are written using Java, EJBs containing the business logic are platform-independent and can be moved to a different, more scalable platform should the need arise. If you are hosting a mission-critical application and need to move your EJBs from one platform to the other, you can do it without any change in the business-logic code. This allows you to "plug and work" with off-the-shelf EJBs without having to develop them or have any knowledge of their inner workings. EJB provides both a stateless and a

stateful model and are packaged as jar files. Since EJB is built on top of Java technology, EJB components can only be implemented using the Java Language. The architectures of CORBA, DCOM and Java/RMI provide mechanisms for transparent invocation and accessing of remote distributed objects. Though the mechanisms that they employ to achieve remoting may be different, the approach each of them take is more or less similar.

CHAPTER 3. MODEL-BASED DISTRIBUTED OBJECT COMPUTING

3.1 Introduction

MDOC is the solution approached that provides a versatile object oriented frameworks to solve the mention problems, the use of old methods and integrate the isolated legacy application needed to finalize a design. As mentioned in the introduction, since MDOC is based on Java/RMI, MDOC takes advantage of RMI to create heterogeneous distributed application and the java programming language to create object oriented environment.

MDOC also benefits from RMI and the java programming language to create a whole new environment that enables engineers to share the design of complex modeling applications. MDOC gives the user the robustness to share the design with any engineer on the net with high security architecture manner and a very intelligent environment. MDOC use is also to improve the design and decrease the time frame of that design.

3.2 The Benefits From MDOC

The advantages of MDOC are:

- Object Oriented: MDOC robustness makes its growth exponential. MDOC architecture was designed in a way to make its library of objects expendable. It is very easy to add an object to the library.
- Mobile Behavior: MDOC can move behavior client to server and server to client. For example, you can define an interface for examining rigid beams stress analysis at a specific connection to see whether they conform to

current defined constraint. When a constraint is created, the client can fetch an object that implements that interface from the server. When the constraints change, the server will start returning a different implementation of that interface that uses the new constraints. The constraints will therefore be checked on the client side-providing faster feedback to the user and less load on the server-without installing any new software on user's system. This gives you maximal flexibility, since changing constraints requires you to add a new object on the server host.

- Design Patterns: Passing objects lets you use the full power of object-oriented technology in distributed computing, such as two and three tier systems. When you can pass behavior, you can use object oriented design patterns in your solutions. All object oriented design patterns rely upon different behaviors for their power; without passing complete objects-both implementations and type the benefits provided by the design patterns movement are lost.
- Safe and Secure: MDOC uses built-in security mechanisms that allow your system to be safe when users download implementations. MDOC uses the security manager defined to protect systems from hostile objects thus protecting your systems and network from potentially hostile downloaded code. In severe cases, a server can refuse to download any implementations at all.

3.3 Technical Rationale

MDOC provides a Knowledge Based Engineering (KBE) framework that captures knowledge from the modeled domain and creates parametric models with that knowledge. MDOC is "adaptive" in that it can be used to model a wide range of

domains that have interacting components and constrained behavior between them. MDOC can be adapted to diverse engineering applications.

3.4 Object-Oriented Framework

MDOC framework supports a single underlying object-oriented architecture. MDOC's object-oriented design paradigm provides two different types of relations. A class-subclass relation allows data abstraction, encapsulation, sharing of data structures and behavior, and polymorphism. Subclasses can be derived from any of the MDOC classes, or from user-defined classes. Multiple inheritance is available for classes. A class can be derived from an existing class, and new *properties* can be added, or *formulas* and *values* redefined for existing *properties*, as shown below:

(define-class class-name

:inherit-from (class-list)

:properties (property-list)

:sub-objects (subobject-list)

)

The *class-list* is a list of classes to inherit from; the *property-list* is a list of object properties that are defined similar to the objects. The *subobject-list* is a list of objects directly located under the object being defined in the part-subpart hierarchical assembly. A part-subpart relation enables the creation of a tree structured unified part model where the children of any node of the tree represent sub-objects. Various aspects of the problem can be structured hierarchically according to the domain being modeled (Design, Analysis, Manufacturing, Costing & Inspection).

MDOC provides an expanded set of classes that support a wide spectrum of applications. Such classes are the basis of various modules supported within MDOC.

3.5 System Architecture

The MDOC modeling framework consists of several modules (sets of classes and methods), relating to the different knowledge domains, each focusing on a different functionality. All the modules are written within the MDOC object-oriented architecture although they do communicate with external programs through the Virtual Layer Architecture. Additional modules can be defined and loaded into MDOC to adapt the language for a specific purpose. Since MDOC is modular, only the necessary systems need to be loaded into MDOC. If a problem requires a modeling framework without graphics or geometry, only the kernel needs to be loaded for that application. Applications invoking different aspects of the system built in MDOC will utilize a common user interface to the system. User familiarization is required only with one interface irrespective of the applications. The Design, Analysis, Manufacturing, and Inspection modules all utilize a common MDOC interface.

3.5.1 The MDOC Kernel

The lowest level of the MDOC modeling paradigm provides the language constructs for defining classes, methods, and the constraint mechanism. All subsequent objects simply augment the language. The MDOC core system provides the ability to dynamically instantiate classes and methods and to add, edit, and delete objects and properties at runtime. The constraint mechanism, the part hierarchy, and other basic language constructs are also provided by MDOC.

3.5.2 Dynamic Models

Since design is inherently iterative and dynamic, MDOC is also dynamic in nature. Values and formulas of properties can be changed after the model is instantiated. MDOC also permits the addition and deletion of objects and properties after the model is instantiated. Methods can be defined against any MDOC class or one derived from an MDOC class. This lets the user modify behavior of classes according to the needs of the application. Polymorphism as well as `virtual functions' can be utilized.

3.5.3 Property Objects

MDOC permits the definition of new property classes inheriting from existing ones such as property-object. Properties can be added to and methods written against a property class like any other class. In fact, a property-object is also derived from the MDOC object class. The property objects are the basis for accessing the virtual interfaces to communicate with external databases, foreign applications, geometric modelers, etc.

3.5.4 Methods

Methods can be defined against any MDOC class or one derived from an MDOC class. This allows the user to modify behavior of classes according to the needs of the application. Polymorphism as well as `virtual functions' can be utilized.

3.5.5 Unified Part Model

Various aspects of a problem can be detailed through a single unified model in MDOC. The design strategy and related engineering and production processes are captured within a single part model, represented by a hierarchy of objects. An example of such an application is the design automation and analysis of a

combustion engine. A geometric design is created followed by the association of various physical attributes with the geometry. Then the attributes for a finite element model, and the strategy required for generating the mesh model along with various input files for the analysis solver, are maintained. MDOC enables the various aspects of the engineering processes to be stored in a single model in a structured fashion. Furthermore, knowledge for manufacturing, inspection, cost, and tooling can be incorporated into the same model for the automation of the manufacturing and inspection process plans. Feedback could be provided at various stages to different entities in the model. A complete user interface for the problem including input and output forms, menus, etc. can also be associated with the same part model that encompasses the various aspects of the application.

3.5.6 Modular, Virtual Layers Architecture

MDOC supports a modular underlying architecture consisting of a number of virtual layers that make it easy to interface to foreign applications or modules. Different modules within MDOC can be easily replaced or extended.

A number of solid modelers are supported. An MDOC application can interface among different solid modelers without changes in the application code. Through the MDOC virtual layering interface, additional applications such as engineering analysis solvers, other solid/surface modelers, or any other engineering applications can be seamlessly integrated. The MDOC syntax is independent of the underlying foreign application since all communication is handled though the virtual layers. MDOC objects are fully portable and can seamlessly interface with other modelers or applications without changes to the user defined objects in the application source code.

Since MDOC supports a modular underlying architecture, additional modules required by the user can be easily integrated. Existing third party applications can be integrated with MDOC independent of the language or the operating system that they are developed in. Existing modules are written in C, C++, FORTRAN, JAVA, and LISP. The wide range of languages is proof that MDOC allows easy, open integration of additional modules.

The MDOC paradigm provides a common interface to a number of solid modelers in addition to different mesh generators and analysis solvers. This interface is implemented through a Modular and Virtual Layer Architecture, providing a common consistent interface to underlying foreign applications. Another advantage of the Virtual Layer Architecture is that a design can be created using different modeling engines. For example, the same geometric design could be created using the SHAPES, ACIS or Parasolid solid modelers to compare accuracy and performance, or imported via STEP or IGES from various CAD systems. Similarly an analysis model could be exported to various solvers such as FLUENT or SPECTRUM without requiring changes in the analysis model.

3.5.7 Constraint Mechanism

MDOC's underlying constraint mechanism supports demand driven and dependency backtracking behaviors. Demand driven refers to the fact that the value of a property is not calculated until it is demanded. Until a value is demanded, an internal flag refers to the property value as being unbound or the property being smashed. Several properties that affect a certain property can be modified, but the affected property does not need to be recalculated every time, only when it is finally needed. Dependency backtracking is the mechanism that actually propagates constraint changes throughout the part model. When a

property is modified, all the properties in its effect list are smashed. When a property is smashed, it further smashes all properties in its effect list, propagating the change by notifying entities that they need to be recalculated when demanded next.

3.5.8 Model Tree Search

MDOC employs a unique mechanism for associating *properties* and *objects* within *formulas* and *values* referred to as *the* referencing. The *the* mechanism provides the means for querying the model for properties and objects as well as establishing constraint relationships. Additional functionality and *querying* mechanisms are provided to enable extended definition of *constraints* and *dependencies*. The *select-object* function provides a means of selecting objects over the entire model, or for a particular branch of the tree.

3.5.9 Parametric Design

The constraint mechanism coupled with the tree-search provides the parametric modeling environment. Several properties of the model can be changed and then the results can be computed, which may result in new geometry or different outputs. A "what-if" scenario can be achieved without the user having to manually notify entities of change. The change-value and change-formula methods assist in modification of properties.

3.5.10 Event Triggers

Although a demand driven paradigm suffices for several applications, situations arise that require notification of entities that may be outside the domain of dependencies and certain actions need to occur at the moment an event takes place. The system defines classes that can be inherited into objects to define

actions on creation, deletion, change and smashing of properties or objects. An example of the use of an event trigger is in the way MDOC handles the freeing of external pointers on notification from MDOC events. When a property that holds a pointer external to MDOC is smashed, a trigger is activated that invokes the system call to destroy the pointer and unbound the property. This cannot be done using dependencies alone since the pointer is external to the constraint network. This mechanism is unique to the MDOC functionality of the Virtual Interface that independently manages automatic allocation of memory to external applications.

3.6 Geometry

MDOC supports advanced parametric solid and surfacing capabilities including "web" geometry with complete topology access and mixed Boolean operations. The advanced geometry module supports seamless virtual interfaces to Shapes, Parasolid, and ACIS. Additional solid/surface modeling kernels could be easily integrated via the MDOC Virtual Layer Interface (VGL). MDOCs VGL enables the user to preserve the MDOC application code while extending its compatibility to other applications and modelers.

The various applications of MDOC including design automation, layout and configuration, manufacturing planning, and finite element modeling and analysis, have different geometric requirements. These individual requirements are satisfied through the capability of augmenting the part model for different representations to satisfy various demands of the different applications. These different representations are manipulated through a unified part model. MDOC presents a number of objects/classes for modeling simple primitives in addition to complex geometrical operations. Complex Boolean operations for mixed-dimensional solid, surface, and/or wireframe representations, incorporating non-manifold

topology, are also supported. Additional objects for advanced modeling of free-form surfaces (NURBS, Beziers, etc.) also exist. MDOC supports a STEP and IGES interface to external CAD systems.

3.7 Geometric Reasoning

MDOC supports geometric reasoning for process planning automation to integrate the part design with the manufacturing, inspection, or analysis plans for simultaneous engineering. Various queries and methods to enhance and modify the part geometry as required by the manufacturing, analysis, and inspection processes, are also supported by MDOC. For example, queries about a distributed set of points on a free form surface, along with their normals and connected path could be dynamically queried and presented in a separate object. This object could also be used for an inspection plan.

3.8 Process Planning

MDOC provides a suite of objects, which are fully integrated, supporting the general requirements of manufacturing and inspection process planning automation. Present manufacturing capabilities includes machining, supporting milling and hole making process planning as well as cost estimation. Also a unique tool path planner for NC is fully integrated within MDOC. This dynamic tool path generation supports up to five axis NC tool path planning. The automated path planner supports geometrical reasoning capabilities which are suitable for applications in spot welding, arc welding, spray painting, water jet cutting, Eddy Current inspection, and other applications that require the position and motion control of a tool/probe moving on or about complex shapes and surfaces. Capabilities to control multiple path offsets, distance from, and

orientation between the tool and the surface are also automated. The uniqueness of the integrated and dynamic link between the geometrical modeler and the path planner comes from MDOCs single underlying object-oriented architecture.

3.9 Graphics

Through its virtual layer capabilities, MDOC supports advanced visualization techniques for manipulation and display of the geometrical objects. These capabilities include limited functionality for post processing and visualizations, such as color mapping using various grouping techniques. MDOC supports various objects for creating dynamic charts such as bar charts, curve fitting charts, etc.

3.10 Parametric Feature Based Design

MDOC provides a unique interactive design environment. It is parametric, constraint driven, free form feature based, and has solid and surface modeling capabilities. Geometric as well as non-geometric features can be modeled. The system enables easy referencing and parametric association for feature properties that could be linked to external processes as a part of the Virtual Layer interface.

3.11 User Interface

MDOC provides a complete set of interface classes including forms, buttons, radio boxes, check boxes, input forms, and pop up menus supported on Unix/Motif and Windows NT/95. Since the user interface model is represented using the same knowledge representation system as the applications, it too is dynamic in nature, and the attributes of the user interface entities, (color/size of buttons), can be altered dynamically.

3.12 Attribute Tagging And Propagation

MDOC provides a unique mechanism for facilitating association of information with entities in a geometric model. Attribute Tagging and Propagation is being utilized for facilitating association of engineering processes information with entities in a geometric model. This information typically needs to be conveyed to downstream processes such as manufacturing, inspection, meshing or analysis. Typically, when a geometric model is constructed, several stages of construction geometry are required. These construction entities are then booleaned, transformed, swept, revolved, etc., to create the final model. In a parametric modeling environment, reconfiguring the model involves the modification of parameters at the construction level and the regeneration of the geometric model. All supplementary information would need to be reconveyed to the final geometry as well as downstream processes every time the model is reconfigured. This could be a very tedious task requiring major interaction with the user and delays in the engineering cycle. MDOCs unique parametric modeling paradigm provides the capability to propagate attributes through geometric operations thereby automating the procedure of geometrical enhancement of the final geometry to extract the required data for complete automation of the finite element modeling and analysis processes as well as any other engineering processes.

First, using Attribute Tagging, the supplementary information is associated with construction configurable geometry. Next, every downstream operation, including final design, analysis, and manufacturing has the information passed on through Attribute Propagation. Attribute Propagation ensures that every operation on the geometry, including reads and writes, Booleans, sweeps, etc., propagate the attributes through the operation. As a result, when the model is reconfigured, i.e.,

upstream design entities are modified in geometry or other properties, the Attribute Propagation mechanism ensures that supplementary information is passed downstream automatically. The Attribute Tagging and Propagation mechanism is integrated with the demand driven and backward propagation mechanism using Event Property objects.

3.13 Mesh Generation

The Automatic Mesh Generation system is an MDOC module that allows tight integration of various mesh generation and analysis applications. MDOC provides a Virtual Interface to support various third party mesh generators. The system permits the selection of geometry to mesh, tagging the vertices, edges and faces of geometry for selective refinement of the mesh, and meshing the geometry by calling the external mesh generator. It provides objects for meshing as well as a user interface. It also provides various objects and user interfaces for visualizing the mesh and querying the mesh database created by the mesh generator.

3.14 Analysis Modeling

The Finite Element Analysis (FEA) Modeling module will enable the definition of an analysis problem by defining regions of interest, material models, boundary conditions, solution strategies, and other requirements for analyzing various problems utilizing a Mesh Generator and a Finite Element Solver. The various entities of interest are modeled as MDOC classes that can be utilized to instance a complete FEA problem model. The problem can be associated with the geometric objects as well as the mesh. The system generates several files that the solver can read and execute to generate results. This system can be extended to provide a Virtual Solver Layer that can talk to various FEA solvers.

3.15 Compact And Portable Architecture

The complete MDOC paradigm, including the various modules, requires less than 30MB of disk space. MDOC requires 100MB of swap space with 64MB of RAM recommended (32MB RAM could meet the requirements of various applications). The relatively small requirements of swap and memory size reflect MDOC's object oriented unique architecture. MDOC is supported on UNIX platforms and Intel based PC's running WINDOWS NT, 98, or 95. The compact size of the MDOC system is proof of its innovative advanced architecture when compared to the requirements for storage, swap size, and electronic memory of other competitive products.

3.16 Summary

The MDOC paradigm provides a versatile, parametric modeling environment supporting a unified part model for integrated design and process automation. The MDOC object-oriented modeling framework allows the user to develop applications using dynamic objects for composing adaptive models that can be tailored to various engineering requirements at runtime. The dynamic environment is suited to simulating "what-if" scenarios and iterative modeling environments. MDOC offers a flexible modeling environment that can be used for a wide spectrum of engineering problems requiring the integration of various engineering disciplines all supported within an adaptive object oriented part model. MDOC enables the abstraction of the modeled domain into a set of interacting entities that can be applied to problems requiring a high degree of visualization and complex geometric operations for integrated design and process automation.

MDOC is a revolutionary modeling framework that shortens the design-to-manufacturing cycle, resulting in rapid part production with lower cost. Complex parts and detailed process plans for manufacturing, analysis, and inspection are concurrently designed and developed in a fraction of time compared to current methods. Improved quality and efficiency is realized by producing intelligent, error-free designs.

Finally MDOC is now mostly used to solve electrical and mechanical problem like car design, boiler design, etc … MDOC enabled VOLVO to save millions of dollars (G. Björkman, Personal Communication, January 21, 2000).

CHAPTER 4. CASE STUDY

4.1 Introduction

The following case study will detail how VOLVO used the MDOC technology to analyze a truck Cab. The study will go over the steps that VOLVO went through to achieve this analysis. The result VOLVO reached is demonstrated with a few snapshots.

VOLVO started the analysis by importing the Cab geometries from Parasolid/IGES and using the power of MDOC to automatically identify these geoms and map them into objects.

The middle surface service was the next things to do. Because of the lack of powerful *Hex mesh* engines nowadays, the middle surface of each objects is automatically generated by MDOC with an interface that empowers the users to validate these mid-surfaces manually. After the creation of the middle surfaces the mesh service is then triggered and generates a mesh element for each object. Engineers then check the quality of the FEA and with a graphical interface filter the object with bad mesh in order to repair the finite element mesh then Fix the filtered objects mesh and re-mesh. Iterate till a satisfied solution is reached. Regenerating geometry out of the repaired finite element is a very powerful service that is useful if engineers were unable to reach a satisfied solution. This service provides the ability to generate the surfaces of the objects (shapes) from a mesh.

Weld service is responsible for identifying the weld seed points and weld these objects together with a graphical representation of the weld elements. Now the

next step is to mesh these objects again and check the quality of the finite element produced by the mesh engine after welding them together.

Iterate on repairing the mesh and re-mesh until the approach of an acceptable quad finite element. Note that getting rid of 100% of tri mesh was something impossible but optimizing them to quad mesh was the responsibility of quality control service and the iteration effort of engineers.

Generation of LsDyna files takes place and gets delivered to LsDyna server which produces an output and this output gets loaded to GL view to simulate a crash and view the distortion on the cab from this crash (E. Persson, Personal Communication, March 24, 2000).

MDOC and this case study were implemented at VOLVO Car in Sweden. A team of 7 engineers including me worked for almost one year and used this case study to test MDOC. Research and development initially took place followed by the architectural design and implementation of the MDOC code. After several iterations, the project was facing a critical dead end and VOLVO was too close to stop funding it. Thanks to the team effort, perseverance and had work were able to resolve this project and turn it into a success. More than 350 objects, and approximately 14364 line of code were written.

4.2 *Import*

The right side of the image below shows a graphical representation of the truck cab objects and the finite element mesh of a couple of them. The right side shows the tree service, the list of objects imported from Parasolid/IGES, and the palette that provides the user the capabilities to rotate, shade, unshed, hide, unhide, and draw/undraw the surfaces of each object and its mesh.

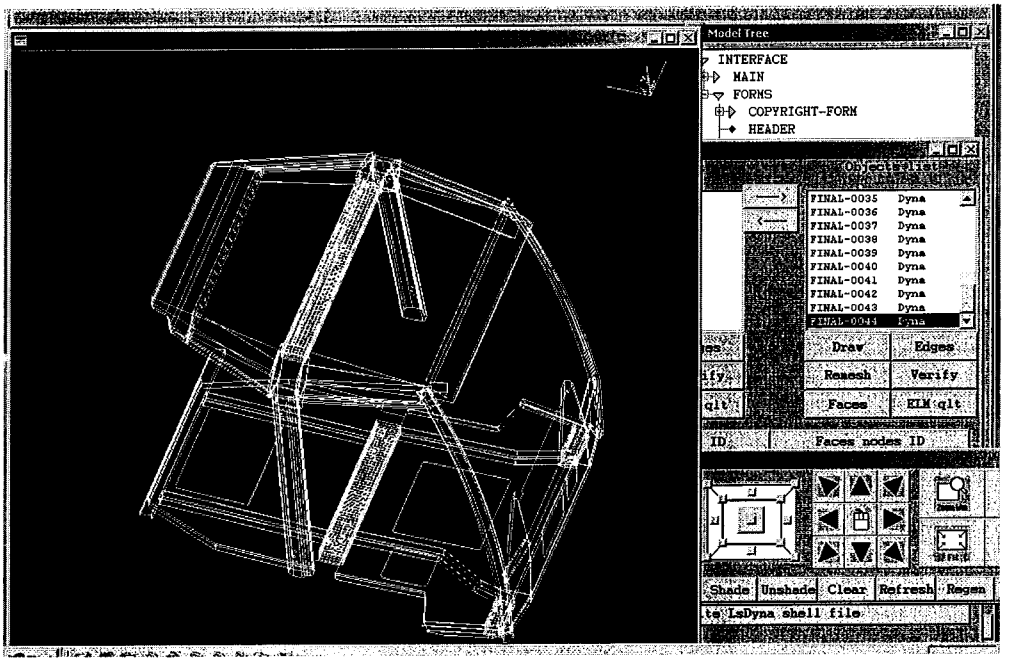


Figure 4-1 Imported CAB

4.3 Filtering

The left side of the image below shows a graphical representation of the finite element. The right side shows a form that provides the ability to draw the mesh of each object separately, and in turn verify this mesh and control the quality of its elements. It also has the ability to repair this mesh, remesh and generate new surfaces out if the repaired mesh.

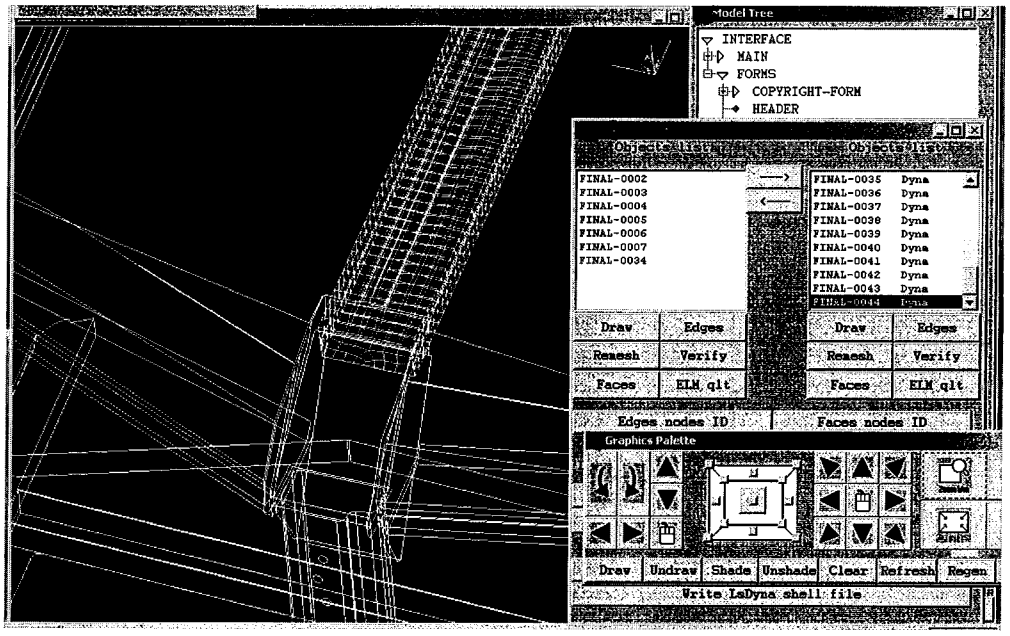


Figure 4-2 Filtered CAB

4.4 Weld

The right side of the image below shows weld elements generated automatically by the weld service. The right side shows a form that provides the ability to control the automation of the MDOC process. As you can see this form gives you access to trigger each service separately and the power to stop and start each service as needed to optimize time.

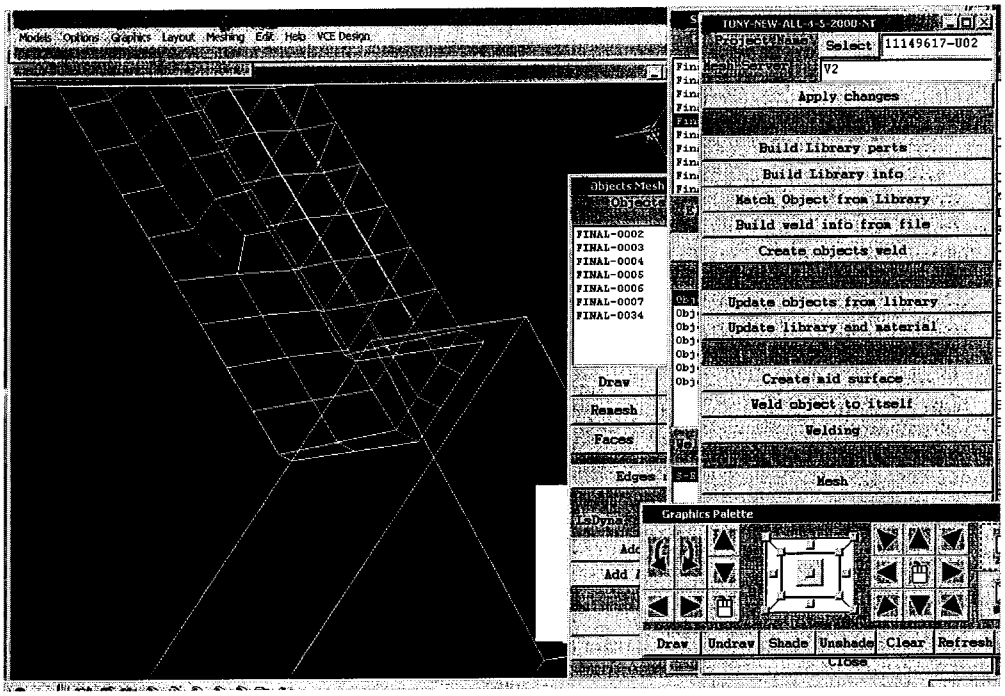


Figure 4-3 Welded CAB

4.5 Final Result

The image below shows the final result of a mesh of an object after being welded to other objects. Notice the presence of the tri-mesh and how the quality of the mesh is no longer as finite as before the weld. The left side also shows a form that provides the ability to draw the shell, the mesh and the seeds.

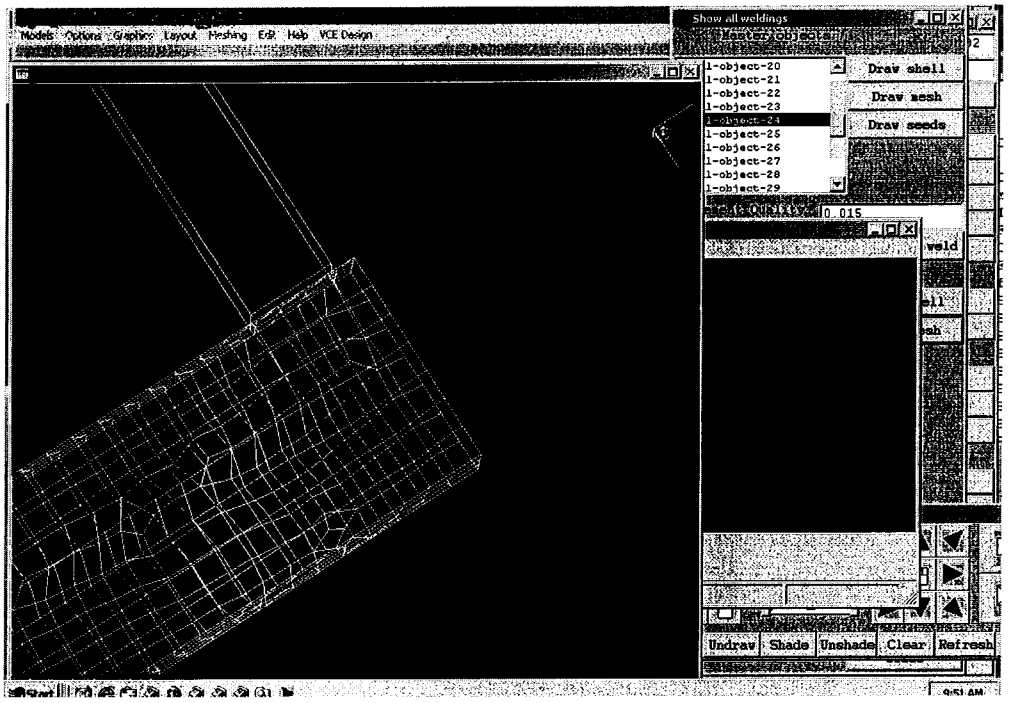


Figure 4-4 Final Result

4.6 Steering Knuckle Design

The following pictures show the geometry for an environment for the design and analysis of automobile steering knuckles. The environment enables engineers to rapidly design and analyze a steering knuckle with a few mouse clicks. This environment takes a minimal amount of input design parameters and dynamically creates a complete 3-D solid model, engineering drawings, and a Finite Element mesh/analysis. Any changes made to the geometry dynamically update in the mesh and in the corresponding Finite Element model (loads and constraints).

Figure 4-5 Steering Knuckle Design

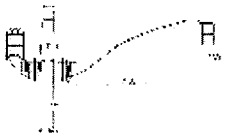


Figure 4-6 Steering Knuckle Design Geometry

Figure 4-7 Steering Knuckle Design Meshed

4.7 Cabin Climate Control

The following picture shows a sample of the cabin integrated control volume detailed design with the various air-handling components. The cabin features that were selected for detailing include the A, B, and C pillars, front and rear windshields, instrument panel, side mirror reflecting areas, consoles, front and back seats of the car, demister inlet and exit surfaces etc. The system architecture of the analysis model is designed and implemented to enable integration with the defroster system shown below. The result provides the engineer with a powerful tool to configure and simulate the flow of air in and out of the cabin model.

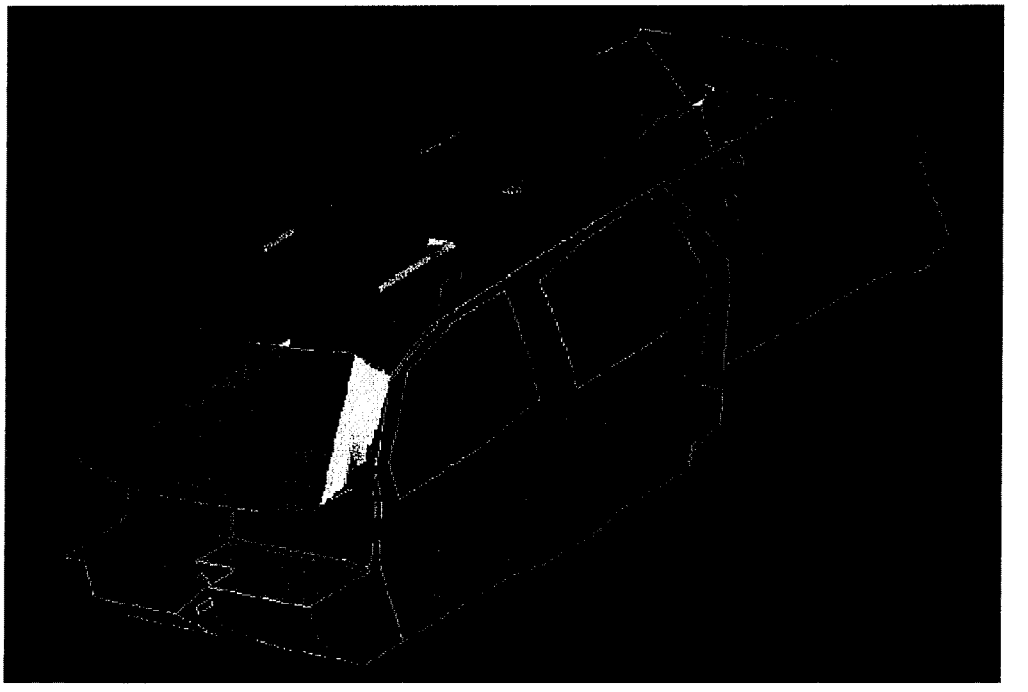


Figure 4-8 Cabin Climate Control

CHAPTER 5. CONCLUSIONS AND FUTURE WORK

With the help of VOLVO, MDOC technology is now used by engineers to produce cars.

MDOC saved engineering efforts, minimized trial and error, and reduced the creation of physical prototypes crash (C. Vikstedt, Personal Communication, May 4, 2000).

With MDOC all the models are now centralized, secured, and accessible from everywhere via the web. Therefore, the versioning issue is no longer a problem.

The MDOC set of services, that enable the regeneration of the geometry after improving their mesh element quality and the ready access to visual feedbacks, was a great success in saving lots of human efforts and money.

MDOC provided an Object oriented framework highly effective for modeling. Helped in minimizing development effort, and in integrating legacy applications.

An object in the MDOC library that other services automatically adapt to, represent the integrated application.

Engineers at VOLVO are nowadays asking for additional features to optimize time and money. The most important one is to support object thickness to be able to handle hex mesh. The integration of GL view is VOLVO's next step to be able to simulate the distortion of a CAB.

Creation of CABs from library of parts is in progress nowadays.

Web services-based architecture is the next challenge and the creation of a framework for wrapping legacy applications into web services is MDOC's next step.

GLOSSARY

DOC: Distributed object computing.

MDOC: Mode-based Distributed object computing

CORBA: Common Object Request Broker Architecture

OMG: Object management group

ORBs: Object request brokers

ORPC: Object remote procedure

DDCF: Distributed Document Component Facility

DII: Distributed Document Component Facility

DSI: Dynamic skeleton interface

DCOM: Distributed component object model

COM: Component object model

RMI: Remote method invocation

JVM: Java virtual machine

RPC: Remote procedure calls

JNI: Java's native method interface

IDL: Interface definition language

BOA: Basic object adapter

POA: Portable object adapter

ORPC: Object remote procedure call

JRMP: Java remote method protocol

MTS: Microsoft transaction server

EJB: Enterprise javabeans

BIBLIOGRAPHY

Box, D. Essential COM. MA: Addison-Wesley, 1997.

Christopher, B. Professional windows DNA. Building Distributed Web Applications With VB, COM+, MSMQ, SOAP, and ASP. Canada: Wrox press, 2000.

Gilbert, H. Introduction to TCP/IP. NY: PCLT, 1995.

Johnson, R. Frameworks = patterns + components. Communication of the ACM, 1997 Oct. 40.

Morgan, B. CORBA meets Java. [On-line]. Available: http://www.javaworld.com/jw-10-1997/jw-10-corbajava.html (1998, October).

Reilly, D. Introduction to remote method invocation. [On-line]. Available: http://www.davidreilly.com/jcb/articles/javarmi/javarmi.html (1998, October).

Reilly, D. Java & CORBA - a smooth blend. [On-line]. Available: http://www.davidreilly.com/jcb/articles/javaidl/javaidl.html (1998, October).

Reilly, D. Mobile Agents - Process migration & its implications. [On-line]. Available: http://www.davidreilly.com/topics/software_agents/mobile_agents/ (1998, November).

Vinoski, S. Corba: integration diverse applications within distributed heterogeneous environments. IEEE Communication Magazine, 1997 Feb. issue14.

Wollrath, A. Riggs, R. & Waldo, J. A distributed object model for the java system. USENIX Computing Systems, 1996 Nov. issue 9.

Zinky, J. A., Bakken, D. E., & Schantz, R. Architectural support for quality of service for CORBA Objects. Theory and Practice of Object Systems, 1997 Feb. issue 3.