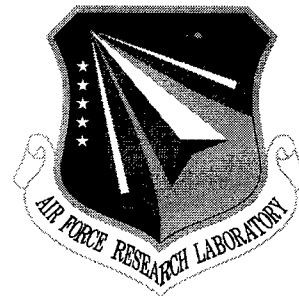AFRL-IF-RS-TR-1998-197
Final Technical Report
October 1998

ADVANCED SYSTEM ENGINEERING AUTOMATION (ASEA)

Modus Operandi, Inc.

John Faure

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

**AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
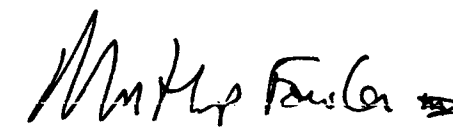ROME, NEW YORK**

DTIC QUALITY INSPECTED 3

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-1998-197 has been reviewed and is approved for publication.

APPROVED: *[signature]*

FRANK S. LAMONICA
Project Engineer

FOR THE DIRECTOR: *[signature]*

NORTHRUP FOWLER III
Technical Advisor
Information Technology Division
Information Directorate

REPORT DOCUMENTATION PAGE			*Form Approved* *OMB No. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE October 1998	3. REPORT TYPE AND DATES COVERED Final Apr 93 - Nov 98	

4. TITLE AND SUBTITLE

ADVANCED SYSTEM ENGINEERING AUTOMATION (ASEA)

5. FUNDING NUMBERS

C - F30602-93-C-0123
PE - 63728F
PR - 2527
TA - 02
WU - 28

6. AUTHOR(S)

John Faure

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Modus Operandi Inc.
122 Fourth Ave
Indialantic FL 32903

8. PERFORMING ORGANIZATION REPORT NUMBER

N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

AFRL/IFTD
525 Brooks Rd
Rome NY 13441-4505

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

AFRL-IF-RS-TR-1998-197

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION AVAILABILITY STATEMENT

Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Information is one of the most important products of the systems and software development life cycle, and today's computer-based tools do not fully exploit it. In general, currently available tools operate in isolation from each other, exist on incompatible platforms, are difficult to learn to use, may have excessive per user costs, and have limited supported lifetimes.

This report describes the development of an open integration framework, known as Catalyst, which was designed to increase the effectiveness of system and specialty engineers. Through a software "building block" approach, the Catalyst concept supports the engineering, communication, and management activities of a system software development. Catalyst, based on a Common Object Request Broker Architecture (CORBA), provides for the creation of a unified web of interrelated system data that can be browsed, analyzed, corrected, manipulated, and moved from tool to tool. These capabilities help to capture, preserve, access, and fully utilize the information created during the system software life cycle, as well as support the evolution and modernization of a project's tool suite.

14. SUBJECT TERMS

system engineering, CORBA, tool integration, framework, object oriented

15. NUMBER OF PAGES
108

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Table of Contents

List of Figures

List of Tables

1. Introduction

This Final Technical Report documents the objectives, history, and results of the Advanced System Engineering Automation (ASEA) effort under contract F30602-93-C-0123, sponsored by the US Air Force Research Laboratory/ Information Directorate. The ASEA prime contractor is Modus Operandi, Inc., formerly known as Software Productivity Solutions. The ASEA subcontractors were Northrop Grumman Surveillance and Battle Management Systems (SBMS), MTM Software Engineering, Virginia Polytechnic Institute, Odyssey Research Associates, and Ontos.

The primary product of the ASEA effort is the Catalyst system. An important goal of this report is to preserve and communicate the rationale for Catalyst's unique design and implementation and what we learned as Catalyst was designed. Many engineering environments have been created – with varying degrees of success. The designers of Catalyst participated in some of these efforts firsthand and followed the progression of others through the literature. These experiences drove Catalyst to follow some different paths, which resulted in a much more successful system at a fraction of the cost that was spent developing other environments.

This Final Technical Report (FTR) presents an overview of the Catalyst system, and then describes its parts and their development history. The items that were delivered are listed, followed by a series of descriptions of generic Catalyst applications. The applications of Catalyst on pilot projects are then described, followed by a list of enhancements that we believe are worthy of consideration for future implementation.

This FTR does not duplicate information contained in the other ASEA documentation, but provides a summary of the effort and gives insight and rationale into the resulting Catalyst system. The FTR is supported by the following Interim Technical Report documents:

- Volume 1 - Catalyst Interim Technical Report
- Volume 2 - System Engineering Key Processes
- Volume 3 - System Engineering Best Practices
- Volume 4 - Catalyst Security Study
- Volume 5 - ASEA ECP - Integrated Security Analysis Tools
- Volume 6 - Micro-Process Enactment Study

A large number of other documents were produced under the ASEA effort including an entire suite of 2167A documents. More detailed information about Catalyst tools and design can be found in these documents. The software user manuals and tutorials produced for Catalyst have been converted to HyperText Markup Language (HTML) and are available at the Catalyst web site under http://www.modusoperandi.com/mo_labs/catalyst/catalyst.html.

1.1 Background

The ASEA contract was a follow-on to the Systems Engineering Concept Demonstration (SECD), Contract F30602-90-C-0021, also performed for the US Air Force Research Laboratory. SECD successfully laid the groundwork for the ASEA effort by researching system engineering automation requirements and defining the automation concept, known as Catalyst.

1.2 Objectives of the Effort

The objective of ASEA was to develop and demonstrate automated system engineering technologies that effectively support system engineering and specialty engineering-related activities throughout the computer-based system life cycle.

1.3 Scope of the Effort

The scope of the ASEA effort was to create a complete and robust initial version of Catalyst that was suitable for external dissemination, evaluation, and trial use. This goal was achieved and Catalyst has been delivered to numerous outside users.

1.4 Executive Summary

Catalyst was developed to help engineers who are developing and maintaining complex systems. Catalyst uses advanced software technologies such as the Common Object Request Broker Architecture (CORBA) to integrate the tools and databases used by an engineering project team. The integration and other services provided by Catalyst add value by helping to realize the full benefit of all information created during the system development.

Information is one of the most important products of engineering and today's tools do not fully exploit it. Current tools operate in isolation from each other, exist on different platforms, are difficult to learn to use, may have high per user costs, and have limited supported lifetimes. Catalyst provides transparent access to data in integrated tools and allows data to be linked together across tool boundaries. Catalyst allows data in any tool to be annotated using text, graphics and audio. Catalyst supports the creation of a unified web of interrelated system data that can be browsed, analyzed, corrected, manipulated, and moved from tool to tool. These capabilities help to capture, preserve, access and fully utilize the information created during system development and maintenance.

Catalyst is a generic distributed information integration system and has been applied outside of the engineering domain. Catalyst has been successfully applied and demonstrated in several real world applications including Joint STARS and STRICOM. Additional applications are underway at Warner Robins ALC, MCC, NSWC, DARPA and several universities.

2. Catalyst Overview

Catalyst is an open framework designed to increase the effectiveness of system and specialty engineers so they can produce more successful, higher quality systems. By integrating the tools and databases used during system engineering, Catalyst enhances the communications, management, and engineering activities performed by systems and specialty engineers. Catalyst is a high performance, scaleable, distributed environment based on the open industry CORBA specification. CORBA is a specification produced by the Object Management Group (OMG), the world's largest software consortium. The CORBA specification defines a system for managing and communicating with distributed objects. Transparent distribution and scalability are several benefits of this rapidly growing, vendor-independent specification.

Modus Operandi, Inc. (formerly Software Productivity Solutions, Inc.) is the Prime Contractor responsible for designing and implementing Catalyst. Catalyst has benefited from Modus Operandi's ten years of practical experience in designing and building engineering environments and in participating in the engineering environment community. Catalyst has been designed to address the practical problems that hinder the widespread use of engineering environments. The keystone of our design philosophy is to make Catalyst adaptable to a user organization rather than vice versa.

In a perfect world, Catalyst would only include tools specifically written to conform to its own interfaces and conventions. However, since we recognized that this is not feasible, we tried to accommodate actual practice (i.e., the user's existing tools) in the design of Catalyst. An organization adopting Catalyst must determine what tools they wish to use with it, how they will be integrated, and how they will be applied to the engineering effort. An issue that must be dealt with is the possibility that a user will access data through the tool in a way that is not consistent with Catalyst. A tool integration must address the fact that data may still be created, deleted, locked, or modified through the tool's own interface. A high quality integration should detect and handle these events either as they occur or as the affected data is accessed.

Catalyst is designed for easy and incremental adoption, for providing capabilities and benefits unavailable through Commercial-Off-The-Shelf (COTS) products, and for performing and scaling well enough to support large engineering efforts. Each engineering organization uses a unique combination of tools, methods, and processes. An overview of Catalyst is illustrated in Figure 2.1-1. Supporting organizations by adapting to their choices was a primary motivation in the design of Catalyst. Using the portable and open CORBA specification as the basis for Catalyst will prevent technological obsolescence and reliance on a specific hardware, software or operating system vendor. Catalyst is well positioned for supporting new levels of system engineering integration and productivity into the 21st century.

2.1 Design Considerations

One of the most important considerations in making a tool integration environment acceptable to potential user organizations is supporting the tools *these organizations already own and use. Most organizations have a major investment in cost, time, training and experience in their tools, which typically* include a mixture of COTS, Government-Off-The-Shelf (GOTS) and custom tools. Considering this fact, it was recognized early in the effort that few organizations would abandon their existing tool suite (and its accompanying legacy data) in favor of another tool suite that happened to be integrated into an environment. In addition, trying to choose the most "popular" set of tools and integrating them is difficult because there are many popular tools in use in many combinations. To address these issues, Catalyst was designed to make it as easy as possible to integrate tools, which enables an organization to integrate its own choice of tools.
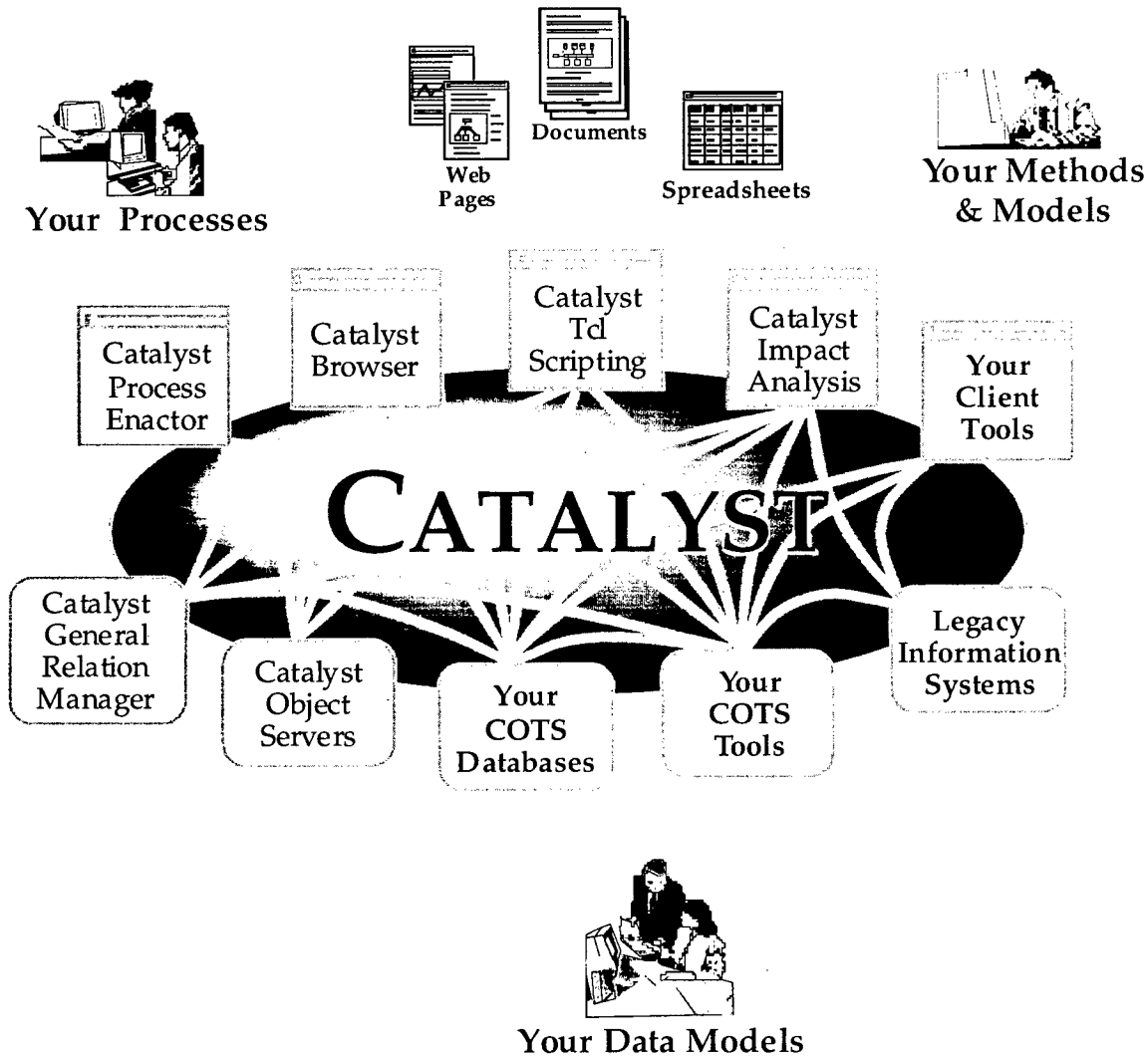
Figure 2.1-1. Catalyst Overview

4

Another important consideration was to allow Catalyst to be introduced into an organization incrementally, with incremental benefits realized. Most user organizations are very reluctant to embrace any new technology that requires a very high initial investment before any benefits are realized. Any new technology that disrupts on-going projects or radically changes the development process is not likely to be accepted. As a result, Catalyst was designed as a number of "building blocks" that can be used as desired with associated benefits. In addition, using distribution based on CORBA enables Catalyst to integrate a tool's data "in-place." This design allows a tool to continue to store its own data and to be used as designed while Catalyst gains access to its data. This non-intrusive integration paradigm is transparent to existing users of the tool. Such design choices allow Catalyst to be adopted incrementally without disrupting the normal operations and culture of an organization, preserving vital corporate information.

To make incremental adoption viable and to improve overall acceptance of Catalyst, the Catalyst team has carefully avoided relying on any COTS packages with large up-front or per-user license fees. Catalyst has a lightweight infrastructure based on CORBA, which supports most Catalyst operations. Currently, SunSoft's NEO implementation of CORBA on Solaris is being used, which has very low runtime fees per user and no high initial fees. Future plans include using VisiBroker from Borland to support other hardware platforms.

While it is important for a user organization to select its own choice of tools, it is equally important that they design their own data models and their own processes. Catalyst does not dictate the data model to be used, but rather allows an organization to define the data model that is appropriate for the methods and standards being applied. Various design methods and Government standards model similar information (such as requirements, architectures, designs, tests, and documents) differently; hence, it was recognized that any environment that dictated the data models based on one method or standard would greatly limit its applicability. Catalyst provides pieces of the data model that support its own operations, examples of how various engineering activities can be modeled, and the ability for an organization to easily and quickly create and tailor the data models to their own needs.

Catalyst supports process improvement through its process definition and enactment capabilities. In keeping with the overall philosophy of adaptability, the Catalyst process capabilities are intended to provide a great deal of flexibility in defining and enacting processes and in defining how rigidly a process must be followed. Processes such as change management can be defined with very tight controls to ensure proper adherence, while more creative processes such as requirements development can be defined very loosely to allow for iteration and evolution. Catalyst's process capabilities were designed to help users correctly complete their tasks, and not to obstruct or control their activities. An essential goal for these capabilities is that it be easier to perform work *using* them than *without* using them.

Performance was another critical consideration when we designed Catalyst. Engineering environments are complex data-intensive systems that tend to require many system resources. An environment must perform at least well enough that it does not become the limiting factor in how fast work can be performed by the user. An environment must also be capable of scaling up to large teams, because environment support is most needed for large efforts. Using CORBA as the basis for Catalyst addresses a number of significant performance problems by distributing the disk and CPU load across *all* the workstations being used in the project. Using CORBA also enhances flexibility in load balancing and management. Because CORBA provides location-transparent distribution, servers and objects can be moved without affecting tools or being visible to the user. A user can store objects of interest on his own workstation, reducing the impact on other team members' workstations.

Other design issues related to performance are object granularity and host network speed. Catalyst has been optimized for dealing with "fine-grained" objects, which are better suited for use in an engineering environment than "coarse-grained" objects. An example of a coarse-grained object might be *an entire document.* This type of object is much easier to implement, but it does not provide any insight into the structure or content of the document. An example of a Catalyst fine-grained object might be a *single requirement,* a single *test case, a single device in a circuit design,* or a *document paragraph.* This level of granularity gives automated tools and analyses enough visibility to perform useful work. Catalyst has also been optimized for high-speed network (LAN or WAN) use as opposed to Internet-wide use. This choice was made because (co-located) project teams usually interact across high-speed networks. Future initiatives are being pursued to study and support geographically separated project teams.

2.2 Architecture Overview

Catalyst has two basic "sides." The server "side" integrates the tools and databases that store data. This enables the data stored in these tools and databases to be viewed inside Catalyst as CORBA objects. This is done by wrapping the data sources with the Catalyst Core IDL, which puts a standard interface on all the integrated data. The server side presents an interconnected sea of engineering data, which appears as Catalyst objects and relationships. Relationships between the data stored in various integrated tools are stored by the Catalyst General Relation Manager (GRM). The GRM is a special server for managing relationships.

The client "side" integrates the user tools that use the sea of integrated data to perform interesting and useful processing for the user. The client tools take advantage of the standard interface to data. All they need to understand is how to talk to objects and relationships. There are no special cases. They do not need to know where the data is stored, or by what tool.

An example of a client is the Catalyst Browser Tool. This tool allows a user to graphically browse and manipulate all the data integrated into Catalyst. The

user does not need to understand how any of the integrated tools work or how to use them. The user may browse data from multiple tools and may follow links from one tool's data to another, all without knowing or caring that more than one tool is involved.

Catalyst derives much of its utility from being able to access data using *one* interface and by being able to link data that is stored in different tools. One of the major shortcomings of current system and software engineering tools is that they do not integrate well with each other. Some tools provide special bridges between their products and other popular products. These bridges are often available only for certain versions of certain tools. Worse, most of the time it is better for the data to remain in the tool that created it. Transferring it into another tool only creates redundancy. Catalyst overcomes these problems by supporting links between data in different tools and by requiring that a tool only be integrated once into Catalyst before it can interact with all other integrated tools.

The two "sides" of Catalyst are tied together using CORBA. In CORBA, interfaces are defined using the Interface Definition Language (IDL). IDL is used to define object classes and the operations that each class can perform. The IDL is compiled and converted to "stub" classes that clients can call, and "skeletons" that are filled in by the programmer to create server programs. When a client calls an object operation using a stub, the stub calls the CORBA Object Request Broker (ORB). The ORB then locates the server where the object resides and invokes the same operation on the server skeleton. The code the server programmer has written is then executed and the result is passed back to the client.

The ORB may need to send messages across the network if the client and server are not running on the same machine. It does this automatically, providing a capability known as location transparency. The client does not have to know where the server for the object is. It just invokes the operation it wants and the ORB does the rest.

The most important feature of Catalyst is the IDL interfaces, which are defined to specify what operations can be performed on Catalyst objects and relationships. These operations are defined by the Catalyst Core and GRM IDLs. These IDLs define the interfaces between the Catalyst clients and the Catalyst servers. The clients make calls to objects using operations defined in the Catalyst Core IDL. These operations are implemented by the servers, which integrate data sources into Catalyst. Example operations are get class name, get attribute value, set attribute value, etc. Clients make calls to the Catalyst GRM IDL to create, retrieve, and delete relationships. These operations are implemented in the Catalyst GRM (which is a server).
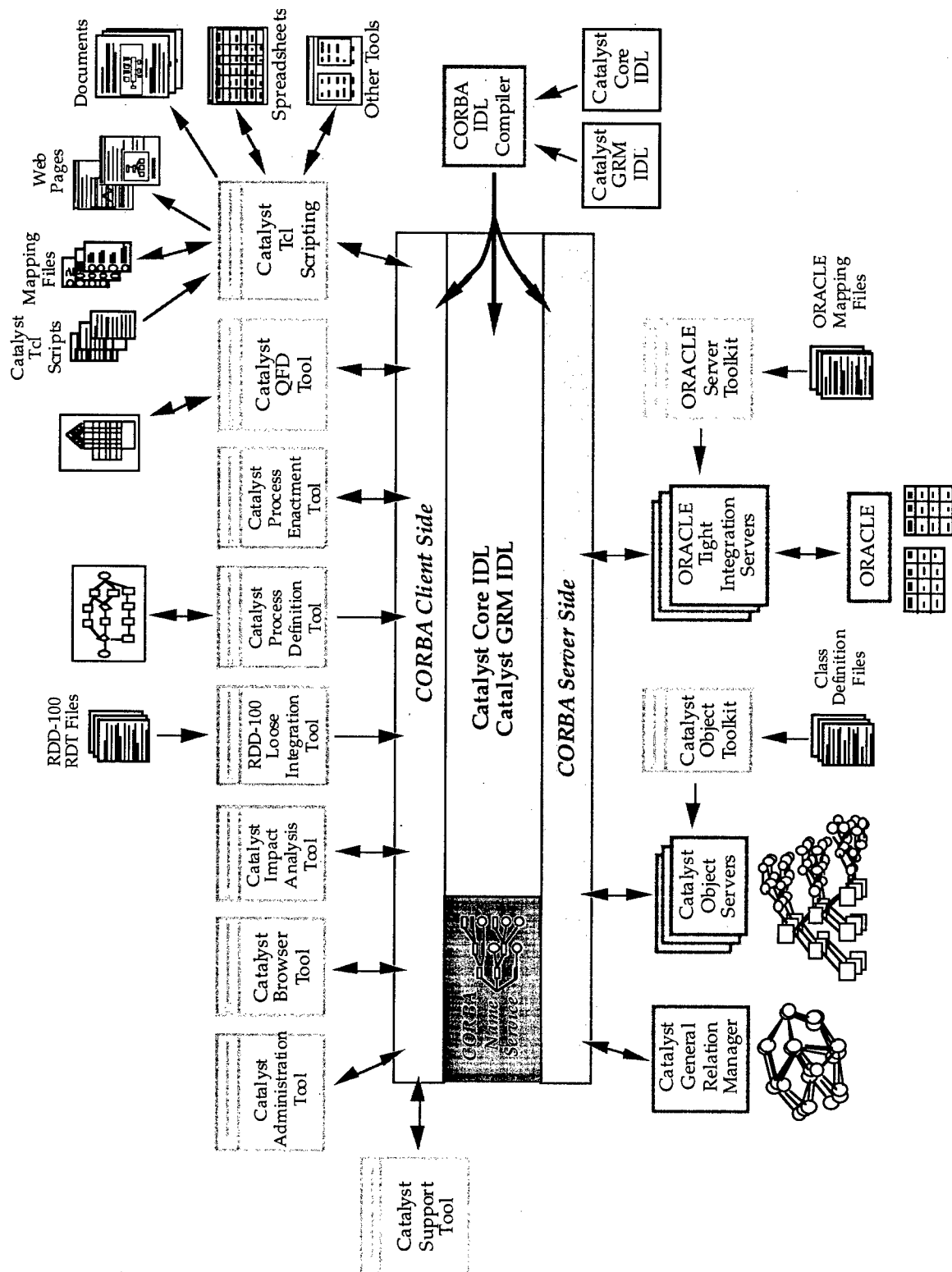
Figure 2.2-1. Catalyst Architecture

The Catalyst architecture is illustrated in Figure 2.2-1. The figure attempts to show the role of the Catalyst IDLs and the presence of clients and servers. The Catalyst IDLs are not very long or complicated, and reflect the Catalyst philosophy of providing simple but powerful and generic operations.

2.3 General Benefits

This section contains a high level description of the types of benefits to be realized using Catalyst. More specific scenarios for applying Catalyst are described in Section 4.0.

An organization using Catalyst will realize many significant, integration-related benefits. Many of these benefits are built around exploiting the uniform access to information provided by Catalyst's CORBA interfaces. Information is one of the most important assets at each stage in the lifecycle of a complex system, and the integrations made possible by Catalyst help to improve the capture, utilization, and quality of all types of system information, as well as reducing the costs associated with information collection and management. The following paragraphs describe the integration benefits as they relate to information transfer; information analysis; information capture, linking, and communications; and state-based process capabilities.

Integration benefits related to information transfer:

- Information Transfer from Tool-to-Tool

- Information Transfer from Site-to-Site

- Information Transfer from Tools-to-Products

One of the most obvious benefits of integration is the ability to transfer data from one tool to another. This has many applications, from subjecting a particular design to analysis by different tools to migrating project data from an older tool to a newer one in order to stay abreast of advances in technology. This is particularly beneficial to programs with 10-30 year life spans. A less obvious benefit is the ability to share data with geographically separated sites by exporting it from one site, transmitting or physically transporting it to another site, and then importing it. Catalyst has sophisticated facilities for merging imported data and for maintaining the data consistency of the individual sites. Information transfer from integrated tools can also be used to create unified reports, graphs, spreadsheets, and other products from data that is stored in multiple tools within Catalyst.

Integration benefits related to information analysis:

- Information Analysis for Correctness, Completeness, and Consistency

- Information Analysis for Impact Analysis and Facilitation

Catalyst contains a powerful scripted capability for analyzing data stored in multiple tools. A script can traverse any of the data visible to Catalyst and identify problems that may span several tools. The traceability of information between tools and the consistency of data stored in separate tools can be analyzed. Missing information such as unallocated or untested requirements can be found quickly and easily. Custom analysis can be performed easily because the Catalyst capability is scripted rather than hard-coded.

Catalyst contains complete support for identifying and assessing the impacts of proposed changes to products and processes. This capability was designed to assist with analyzing and documenting complex scenarios in which the effects of changing multiple items must be assessed at once, as is usually the case in engineering changes. Catalyst performs (perhaps several different) user-defined searches, starting from each changed item. The potentially impacted items are collected and organized by the changed item causing the impact. The user may add or remove impacted items as deemed necessary. An assessment of each impacted item related to each changed item is collected, as is the overall impact. Organizing impacted items according to changed items allows the assessment to be easily modified if the set of changed items is revised, as often occurs when engineering changes are proposed or developed. This capability will considerably facilitate the important but often informal process of impact assessment.

Integration benefits related to information capture, linking, and communications:

- Information Capture and Linking

- Transparent Information Browsing

- Improved Team Communications

As a program progresses, the rationale and informal background information that explains why program decisions were made and how a system was intended to operate is often lost or forgotten. Paper-based documentation standards and practices are not set up to accommodate the informal information that is often crucial for understanding a system. Personnel turnover also usually results in an irreplaceable loss of knowledge.

Catalyst addresses these problems through very flexible information annotation and linking capabilities. Text, graphic, and audio annotations can be created and linked to any object in Catalyst, regardless of where the object is stored. For example, a requirement stored in an ORACLE database may be annotated with a graphic diagram and verbal discussion, which are linked through Catalyst to the requirement and available for review at any time. Furthermore, objects that are conceptually related, such as a trade study and the subsystem it pertains to, may be physically linked so that the trade study can be located from the subsystem and vice versa. Annotations on objects may themselves be linked together into arbitrary networks. Catalyst supports the creation of an interrelated network of

many formats of information that can be stored and browsed at a later date. With Catalyst, there is no reason for formal or informal information of any type to be lost or separated from the project components to which it applies.

All the information stored in Catalyst can be graphically browsed using a point-and-click interface. The Catalyst Browser transparently accesses information through Catalyst, regardless of the integrated tool in which it is stored or the format in which it is stored. This allows novice users to view data that may be stored in ORACLE, RDD-100, or any other integrated tool without knowing anything about how to operate these tools. Furthermore, it is possible to follow interconnected data from tool to tool and document to document without any special tool skills. For example, it would be possible using a few clicks to follow a system requirement to the software and hardware requirements to which it traces, and then to its test plans, descriptions, cases, and results. Traditionally, this traversal would require leafing through many separate documents and would require far more time and effort. In Catalyst, it is possible to graphically follow the traceability of the requirement through many documents to its test results using just a few mouse clicks.

The easy graphical browsing made possible by Catalyst will help the project team understand the project and will reduce the learning curve of new team members. It will also foster communications between different specialties and different teams within the project by allowing people to easily browse and understand complex designs and data created by team members.

Integration benefits related to state-based process capabilities:

- Integrated, State-Based Process Capabilities

The Catalyst process capabilities provide unique benefits in defining, enacting, and improving engineering processes. The process capabilities exploit integration by driving the execution (enactment) of processes off the actual states of work products being created and manipulated by integrated tools. Catalyst allows processes to be seamlessly changed and improved in mid-stream. The state-based approach developed for Catalyst allows each work product in the process to "find" its place in a new process automatically. This important feature is crucial for supporting smooth process improvement in on-going projects.

2.4 Lessons Learned

This section contains brief descriptions of key lessons we learned or principles we followed that we feel were important in successfully creating the Catalyst system.

2.4.1 Catalyst Principles

Know exactly what problem you are solving and why. We realized early on that duplicating COTS functionality was not a good idea. We focused on providing

integration and other services not found anywhere for sale. Knowing these two things helped tremendously to scope and control the effort. They stopped us many times from wandering down what would have been dead ends. They also helped to divert a lot of flack from the project, because there is no need to take criticism for problems you are not trying to solve. You can't solve every problem, so pick your problem carefully and stick to it.

Learn the financial and political realities your project will have to face and design your architecture to accomodate them. The most convincing technical arguments are political and financial ones. If your system conflicts with the way users do business, they won't use it. If your system carries too much risk for your users, they won't use it. If your system is too expensive for your users, they won't use it.

Use standards where possible for many reasons. Using standards helps to avoid becoming involved in "religious" wars because many decisions people would argue over have been made for you. Using standards leverages existing efforts.

2.4.2 Building Catalyst

Use a small team of people to design and build a complex system so that communication overhead and problems don't complicate the effort. A small team also helps to ensure that the design and principles of the system stay consistent and do not get compromised.

Give each team member complete responsibility for a well defined part of the system and communication overhead will be very low and the sense of ownership will be very high. Each of the tools and servers in Catalyst was built and owned by one person. This proved to be a very effective way to bring new people up to speed and to ensure that a coherent and quality design resulted.

Use automated testing to reduce testing costs and to improve quality. Packaging the GRM and Object Server tests so the end user can run them really paid off. Test code which would normally be just overhead to write is now actually a useful Catalyst feature.

Technology will constantly evolve and change. If you remain flexible this can be very benficial. Porting all the clients to Java will take a lot of work but will allow universal access to them. Porting all the servers to VisiBroker will allow them to run on many platforms.

Don't be afraid to throw out or rewrite parts of the system if a better design is discovered. The original design for object community interchange used a custom scripting language and a parser and interpreter written in C++. When Mark Wallace joined the team he suggested using Tcl, a standard, extensible scripting language. We gladly threw out 2500 lines of working code and in two weeks of effort had Catalyst integrated with an industry standard scripting language which provides far more functionality.

12

Stress test the core technologies being used to build your system with operational size test cases. We tested and timed NEO using programs which created and deleted thousands of objects and sent tens of thousands of messages before we were convinced it was suitable for the task.

Stick to the core capabilities of a new and emerging technology and you will reduce the risk that it will evolve in a way which leaves your project stranded. We used only core CORBA features and were able to port Catalyst across six different versions of NEO (SunSoft's CORBA product) without significant impacts. During the ASEA effort, the CORBA specification was changed to drop several services, revise most of them, add several new ones, and in four years never completed the specifications of several others.

Message traffic between components is a critical performance cost in distributed systems. Systems must be designed to optimize message throughput and avoid message bottlenecks.

Multithreading and concurrency management are the the hardest parts of CORBA programming. We discovered that Neuron Data's graphics and data structure libraries were not multithread safe. This forced us to reimplement several of its key data structures in a multithread safe way.

Keep the system simple yet powerful with no special cases anywhere. The Catalyst Core and GRM IDLs have required only minor changes in the four years since they were written.

Validate the system architecture by building a vertical slice running from the user interface through the lowest level server, then add to the system by building outwards. This was done very successfully in Catalyst, reducing the technical risks very early in the program and allowing us to have a working system early in the development.

Spend enough time to completely design and understand complex parts of the system before starting to build them. Almost half of the time required to build the Process Enactment Tool was spent on designing and thinking about it. No significant design changes were required during its implementation or use.

3. Catalyst Development

This sections describes the development history of Catalyst, with emphasis on those activities that contributed to the success of the effort. Development in general is discussed in Section 3.1. The development of each piece of Catalyst software is discussed in Section 3.2.

3.1 Development History

3.1.1 Risk Reduction

Many risk reduction activities were undertaken on the ASEA project. Many other organizations had tried to build software engineering environments with limited or no success. Catalyst is a system engineering environment, but in many ways the engineering and technical challenges are the same. Catalyst benefited from Modus Operandi's years of experience in building and marketing engineering environments and tools. This experience helped us identify areas of technical and marketing risk that needed to be addressed. The following sections describe the technical risk abatement activities that helped us succeed.

3.1.1.1 Technology Evaluations

A considerable amount of effort was invested in evaluating COTS technologies that could be used to help develop or be integrated with the environment. We followed a systematic approach to performing the evaluations. Detailed information about the evaluations is presented in the ASEA Interim Technical Report, Volume 1. Highlights of this work and their relevance are presented in this report.

Object Oriented Databases

Modus Operandi evaluated five object-oriented databases:

- Objectivity/DB 2.0, an object-oriented database product developed and marketed by Objectivity, Inc.

- VERSANT Release 2, an object-oriented database product developed and marketed by Versant Object Technology Corp.

- ONTOS DB 2.2, an object-oriented database product developed and marketed by ONTOS, Inc.

- Object Store, an object-oriented database product developed and marketed by Object Design, Inc.

- GemStone, an object-oriented database product developed and marketed by Servio Corporation.

Each product's technical documentation was reviewed, discussions were held with vendor representatives, and some hands-on benchmarking and test program development was performed. The functionality, usability, and platform

support of each Object Data Management System (ODMS) was evaluated. Functional capabilities included basic database functionality such as concurrency control and recovery, as well as object-oriented database features such as inheritance and versioning. Usability measured the application development and maintenance process.

The net result of the evaluation was there was no ODMS that was sufficiently mature to support its intended role within Catalyst. Therefore, no ODMS was selected, and the selection decision was deferred until the second Catalyst build. Problems with the ODMSs that we evaluated include: they were difficult to use, they lacked any industry standards, they did not support schema migration, and they were in a rapid state of evolution.

In retrospect, it is interesting to note that over the last four years, ODMSs have still not gained much market share or widespread use. Several of the vendors, such as ONTOS, no longer sell ODMS. There are two major reasons for this trend. First, creating an Object Data Management System that maintains the classic database Atomicity, Consistency, Isolation, and Durability (ACID) properties, supports distributed clients to access it, and still performs adequately is very difficult to achieve using a generalized solution. This difficulty has prevented any one ODMS vendor from creating an easy to use ODMS that will "do it all" and still perform well.

The second reason for the lack of widespread ODMS usage is that CORBA has solved many of the distribution and management aspects of the problem. If CORBA is used to provide location transparency and distribution, there is no need for an ODMS to perform the same functions. The ObjectStore database eventually was used in Catalyst, but only the single-user Persistence version. This was sufficient because each Catalyst Object Server maintained its own Persistence database and did not share it with any other process. The distribution and multi-user features of an ODMS were not needed.

Interleaf

Interleaf was evaluated as a possible programmable document interface to Catalyst. The idea was to use Interleaf's programming interfaces to get data from Catalyst, format it as a document, and present it to the user. The user would be able to add to, modify, and delete the document data. These changes would be propagated to the Catalyst objects. This would provide WYSIWYG editing of data in tools integrated into Catalyst.

Unfortunately, it was found that Interleaf programming was very difficult, not reliable, and that Interleaf was not very supportive in helping people program their product. This risk reduction short-circuited what would have been a time consuming dead end.

Multiplatform GUI Development Tools

Modus Operandi evaluated five multiplatform GUI development tools for use in creating the user interfaces for Catalyst:

Galaxy by Visix

XVT by XVT

C++ Views by Liant

Open Interface by Neuron Data

Bedrock

C++ Views was an early front runner in the selection process, however, when Neuron Data dropped their run time fees, Open Interface was selected for use. Open Interface was used very successfully on Catalyst for four years. It has a few bugs and quirks, particularly with printing, but overall it works reliably and is fairly easy to program. It was selected partly because it has been around since 1984 and is therefore a mature product.

Its maturity probably lead to one of the more serious shortcomings of Open Interface. In the late 1980s multithreaded programming became more and more prevalent. We discovered that Open Interface was not multi-thread safe, even with regard to its most basic data structures. This lead Modus Operandi to create new implementations of the Open Interface list and hash table classes, using the same interfaces as the old ones. The new implementations are thread safe.

It is interesting to note that Java has pretty much destroyed the market for other multiplatform GUI development tools.

ToolTalk

The ToolTalk service is a network-spanning interprocess message system that promotes cooperation among independently developed applications. Cooperating application developers agree on a message protocol and use the ToolTalk service to deliver the messages. ToolTalk was completely superseded by CORBA, which is much more general purpose.

Orbix

At the time of the evaluations, Orbix did not provide enough of the CORBA architecture to support Catalyst requirements. IONA was also a very small company, perhaps lacking the resources for completing a production quality implementation of CORBA. For these reasons, it was decided that Orbix would be inappropriate to use in Catalyst. In the four years since the evaluation, Orbix and IONA have grown considerably. However, they still are not the market leader in either performance or market share.

The new Internet Inter-ORB Protocol (IIOP) specification – approved by OMG and supported by most ORB vendors – will allow Catalyst to talk to both clients and servers implemented in Orbix and other ORBs in the near future. The lack of a standard API still prevents source code compatibility across vendor products and is a serious shortcoming with CORBA as of early 1998.

DOE (later renamed NEO)

A lengthy evaluation that included lots of performance testing was carried out using Distributed Objects Everywhere (DOE) from SunSoft. The Early Developers Release EDR 1.1 was found to be complete, reliable and fast. At the time of the evaluations, it was judged to be the best quality product from a well established vendor and was selected for use in Catalyst. Since that decision was made, Catalyst has been ported across many versions with little disruption: DOE EDR 1.1, EDR 2.0, DOE pre-Beta, DOE Beta, NEO 1.0 and NEO 2.0. Now, ironically, six versions of the product later SunSoft, has decided that it is not strategic for platform vendors to sell middleware products and has dropped the product in favor of reselling VisiBroker from Visigenic.

Rogue Wave Tools.h++

The C++ library from Rogue Wave was evaluated and selected to provide standard data structures such as strings, lists, hash tables etc. Once development got underway we realized that Open Interface already provided lists and hash tables, and their types were used as parameters to many user interface operation calls. If Rogue Wave lists and hash tables were used, code and run time would be required to convert from Rogue Wave lists to Open Interface lists before the calls could be made, etc. In the end, only Rogue Wave's string management, regular expression, and string tokenizer classes were used. The string class provides advanced reference counted and late copying management of string data, and proved very useful. The regular expression and string tokenizer classes were very useful in the Catalyst Process Enactor Tool's parsing of expressions specified by entry and exit conditions. Open Interface data structures were used for everything else. This is probably a common reality when using a user interface class hierarchy.

3.1.1.2 CORBA Performance Testing

The performance characteristics of CORBA were used to influence and guide the design of Catalyst. Modus Operandi's experience in building several other distributed object-oriented systems also demonstrated the importance of a few key performance characteristics in determining the responsiveness and feasibility of the resulting system. These characteristics stem partly from the nature of object-oriented systems in general and partly from the engineering domain in which the systems operated.

Because Catalyst was to be an object-oriented system, it would represent and manipulate all data as *objects* connected together by *relationships*. This is a connection-oriented model where data is located and used by navigating from

one object to another via the relationships between the objects. Catalyst was to use a fine-grained model, meaning that an entity such as a requirements document would be represented by many objects connected together. An object in a fine-grained model may represent a single document paragraph, a requirement, a test case, an interface, etc. In a coarse-grained model, an object may represent an entire document. The advantage of a fine-grained model is that the structure and content of the data is explicitly represented, making it possible to perform detailed analyses and transformations.

Fine-grained object-oriented systems that represent engineering data tend to create, manage, and operate on fairly large numbers of objects and relationships. For example, a complex weapons system can easily have 10,000 requirements that are linked to functions, system components, and test plans and procedures. A complex board-level design can contain hundreds of devices interconnected with thousands of circuit paths. Handling this magnitude of data requires that objects and relationships be light-weight in time and space. Because each object is usually linked to 5–10 other objects, there tends to be one order of magnitude more relationships than objects in a system.

Experience with other object-oriented systems dictated that a useful and responsive system would have to support on the order of 100 object operations and 1000 relationship operations per second. Empirical observations suggested that because data access times tend to be the most important, it was acceptable for object and relationship creation and deletion operations to be one order of magnitude slower than their corresponding access operations.

Using these estimates, we built and measured several representative prototypes to determine the actual performance characteristics of the COTS CORBA implementation. We found that NEO supported up to 330 object operations per second on a SPARCstation20, well within our desired goal for Catalyst object performance. Our measurements indicated that the performance bottleneck was the time required to traverse the network to reach the desired CORBA object, and not the time spent actually performing the operation. This measurement demonstrated to us that any implementation in which one Catalyst relationship was implemented using one CORBA object would be too slow to meet our goals by close to an order of magnitude.

To achieve the desired relationship performance, we designed a General Relation Manager object (GRM) that would store very large numbers of relationships using only one CORBA object. This improved performance by allowing one CORBA object operation to include a group of Catalyst relationship operations. This was a winning design since the typical pattern of access in object-oriented systems includes many relationship group operations, such as *retrieve all the relationships for a particular object*. The Catalyst GRM performance ranges from 330 relationship operations per second for single relationship operations to 6000 relationships per second for large groups of relationships. This design also reduced the number of bytes required to store a relationship from around 3000 (required for a CORBA object) to only 48 bytes.

Using past measurements and estimates helped us to set the necessary performance goals for Catalyst. By creating and measuring prototypes early in the development cycle, we were able to create a design architecture that achieves the system goals. Having hard numbers to work with, both for estimates and actual results, was a key factor in creating a usable and practical Catalyst system.

3.1.1.3 Using Only Basic CORBA

CORBA was a relatively new and evolving technology when Catalyst was being designed. To minimize the risk to the project, only the most basic CORBA features were used to build the core Catalyst system. The motivation for this strategy was to avoid having CORBA features (that Catalyst relied on) changed or dropped in new versions of CORBA. Sticking to the features that were central to CORBA reduced this risk. Catalyst primarily relies on being able to define IDLs, create and manage objects and send them messages. This is the most powerful capability of CORBA. Of the many CORBA Services that were being proposed, only the Name Service was used.

This proved to be a winning strategy. Catalyst has been ported across six different versions of NEO and two versions of CORBA itself. Each of these ports was accomplished in less than a week with no major impacts to the system. Many of the early CORBA service specifications have been heavily modified or dropped, and most have never been implemented by any of the COTS vendors. The Name Service – one of the most basic services – has been implemented by NEO since the first release, and has been implemented by other COTS vendors since. Many of the really useful CORBA Services such as the Query Service do not have well designed specifications and are not implemented by any vendors.

As CORBA matures, Catalyst can take advantage of any new and useful features that become available. The Object Transaction Service has recently been implemented by several vendors, and may be useful for supporting distributed transactions.

3.1.1.4 Vertical Slice Development

A vertical slice of the Catalyst system was built as soon as it was possible to do so. Building a vertical slice involved creating a Neuron Data user interface that talked to the Object Cache, a demand based object and relationship buffering system which was created to simplify client development, reduce network traffic, and improve client performance. The Object Cache in turn talked to the Catalyst Object Servers and GRM. These components represented a slice of the system from the user interface down to the lowest level servers. Building this slice provided early confirmation that the technologies selected and the architecture created using them would perform well and would work reliably. This is a critical risk reduction function when new technologies are being used. In Catalyst, almost every technology from Neuron Data, C++, CORBA, Object Caching, to General Relation Management were all new to the design team.

Building distributed object-oriented systems was familiar territory but all the tools were new.

The vertical slice confirmed that the Catalyst architecture was sound and that performance of the system was very good. Stress tests were performed in which 10,000 dummy requirements were loaded and linked together. Catalyst handled this load without any problem and without any performance degradation. Later tests with 100,000 dummy requirements yielded similar results. Eventually the ObjectStore and GRM databases that store the objects and relationships would fill up the disk drives where they are located but there is no inherent limitation in Catalyst or CORBA.

Once the vertical slice was operational the system was built outward horizontally. New client tools such as the Process Definition, Process Enactor, Quality Function Deployment (QFD), Impact Analysis, Support, and Administration were added. More servers were added, and more tools such as ORACLE and RDD-100 were integrated. Very little risk was associated with any of these tools, since the overall architecture had been validated very early in the development.

None of the code used in the vertical slice was prototype code that was thrown away. No code was ever written in Catalyst with the intention of throwing it away one day and replacing it with the "real" code. The "real" code was always written when it was needed, removing the need to go back later and rewrite it. Code has been rewritten or thrown away when better ideas and better designs have been discovered. Writing the real code from the start, but not being afraid to throw it out or rewrite it later when it made sense, has contributed to the overall high quality of the Catalyst software.

3.1.1.5 Comprehensive Automated Testing

The key parts of Catalyst that do not have user interfaces have always been tested using automated and comprehensive test programs. These programs exercise all the features of the code and automatically determine if the code passes or fails the test. The results are summarized at the end of the test. This allows tests to be rerun effortlessly any time during the project that the developers need to check the software. When changes were made or new features were added, all the tests were rerun to make sure new bugs were not introduced. Many times the automated tests identified problems that were overlooked by the developers.

The Object Cache, Object Servers, and General Relation Manager all have comprehensive test suites. The Object Server and GRM tests can even be run by the end user to make sure Catalyst was installed correctly and that it is still working correctly. When new tools are integrated, the Object Server tests can be run to make sure the integration was done properly. The Object Server and GRM test suites also include timing tests to help evaluate the performance of changes to the code and of different hardware platforms.

Creating automated test suites took perhaps 25% more time than not doing so, but it has paid off immeasurably. The end user's ability to run the full test suites is a very valuable capability.

3.1.2 Catalyst Traceability

Catalyst was used to develop and track the traceability of its requirements and tests. As soon as enough of it was operational, the Microsoft Word documents for Catalyst were parsed using a custom (2-page) C++ program and the contents stored in Catalyst. This technique was used to capture the 610 system level requirements for Catalyst. When the 1100 software and 25 interface requirements were created, they were created and linked in Catalyst. The traceability of all the requirements to the tests was also created and linked in Catalyst.

The complete Catalyst requirements traceability tables are created using a two page Catalyst Tcl script. These tables are 225 pages long and give the tracing of system requirements to software and interface requirements and then to tests. A second set of tables gives the inverse tracing from tests back to requirements. Using Catalyst to maintain its own traceability saved many man months of effort. New and completely up to date tables can be generated any time they are needed in a few minutes. Using the Catalyst Browser to review and follow the traceability is actually much faster and more convenient than looking through the paper traceability document.

3.1.3 Catalyst Change Management

The Process Definition Tool was used to define a process for performing change management. This process allows users to suggest changes to a system being developed. The changes must be reviewed by management and then assigned to an engineer for completion or rejected.

The change management process was followed on Catalyst using the Process Enactment Tool. Changes to Catalyst and items of work that had to be completed were entered into Catalyst by the engineers working on the project. Catalyst was used to keep track of who was working on what, what its priority was, when it was due, and how far along each piece of work was. This was extremely helpful on a complex project such as ASEA. At times there were more than 150 different items to be completed. Keeping track of them all manually would have been a very time consuming and error prone process. Catalyst continues to be used to track work items and changes in the follow-on programs.

3.1.4 Virginia Polytechnic Institute Efforts

Virginia Polytechnic Institute was a subcontractor. The Virginia Tech (VT) team was lead by Dr. Wolter Fabrycky, who is a very well known systems engineer and author. Dr. Fabrycky and his team provided valuable insight into the systems engineering principles that were used to design Catalyst. Dr. Fabrycky continues to pursue applications of Catalyst to system engineering in areas

including nuclear site decontamination and disposal. Dr. Fabrycky has created the System Engineering Design Laboratory (SEDL) at Virginia Tech to foster academic and industry cooperation in system engineering research and application.

Virginia Tech performed several important tasks under the ASEA effort. They were responsible for identifying processes and practices that were part of good system engineering practice for possible automation by Catalyst. Virginia Tech identified many candidate engineering practices, some of which were presented in detail to the ASEA reviewers and end users. Of these, the Quality Function Deployment (QFD) technique was chosen for implementation. QFD was of interest to both Northrop Grumman and the Air Force as an efficient way to prioritize and summarize the many interrelated factors affecting any engineering effort.

Virginia Tech identified many potential engineering processes. An actual process used by Northrop Grumman for performing Functional Configuration Audit (FCA) was defined by VT using the Process Definition tool. This effort provided very valuable feedback on process definition and enactment to the Modus Operandi team. VT also modeled the System Software Engineering Process using Catalyst. This is a large and comprehensive process that also provided much insight into process modeling.

The complete results of the Virginia Tech efforts are documented in ASEA Interim Technical Report Volume 2, *System Engineering Key Processes,* and Interim Technical Report Volume 3, *System Engineering Best Practices.*

3.1.5 Northrop Grumman Efforts

Northrop Grumman Surveillance and Battle Management Systems (SBMS) was a subcontractor. The Joint Surveillance Target Attack Radar System (Joint STARS) program being performed by Northrop Grumman was the original target end user for Catalyst. Joint STARS is a very complex airborne side looking RADAR system designed for detecting and track ground based moving targets. Joint STARS is a very complex systems and software engineering effort which made it a good candidate for Catalyst application. Northrop Grumman was responsible for providing insight into the system engineering tasks and problems confronted by a real world, very complex system development.

The selection of ORACLE and RDD-100 for integration into Catalyst was made to support Joint STARS. The original system requirements for Joint STARS were captured using a custom ORACLE database designed by Northrop Grumman known as the Requirements Allocation Database (RADB). No capable COTS requirements tools existed when the system requirements were developed forcing the use of a custom solution. Some of the later Joint STARS subsystems were developed using the COTS RDD-100 system engineering tool. RDD-100 provides many features not implemented in the RADB. Some of the system requirements in the ORACLE database either partially overlap or logically trace

to requirements stored in RDD-100. Capturing and using the connections from ORACLE data to RDD-100 data was a perfect application of Catalyst.

Northrop Grumman provided the requirements management part of the schema for the ORACLE Requirements Allocation Database (RADB) used to manage and trace Joint STARS requirements. Providing this schema allowed Catalyst designers to work with the same database structure that was used by Northrop Grumman. The schema helped provide insight into what needed to be supported by the Catalyst ORACLE integration.

Northrop Grumman provided the schema for RDD-100 as it is used on Joint STARS. This schema was useful in designing the Catalyst RDD-100 integration.

Northrop Grumman provided example data sets for both ORACLE and RDD-100. These data sets had been sanitized so heavily that little content remained. Even so, the example data sets were very useful, since the structure of the data remained the same. The interconnections between data objects is very important in Catalyst, and probably contains more information than the contents of the objects. The example data sets were used to test the integrations of ORACLE and RDD-100.

Northrop Grumman helped Modus Operandi create two sets of public domain data. These data sets have the same structure as real Joint STARS data, but do not contain any real Joint STARS information and do not accurately describe Joint STARS in any way. The first public domain data set was known as FONEY RADAR, indicating that it was the design of a non-existent radar system, known as the FONRAD. The content of this data set was created by Northrop Grumman. This was a small but useful data set that was used in many demonstrations.

The second public domain data set described a possible but different design for Joint STARS. The content of this data set was created entirely by Modus Operandi from public domain data sources. Northrop Grumman reviewed this data set and made suggestions for making it appear to be plausible while verifying that it did not describe the real Joint STARS design. The second data set is much larger, containing over 400 requirements.

More information on the Joint STARS application of Catalyst is contained in Section 5.1.

3.1.6 Integrated Security Analysis Tools ECP

The ASEA contract was modified by an Engineering Change Proposal (ECP) to integrate a set of security analysis tools developed by Odyssey Research Associates, Inc. (ORA). ORA enhanced and integrated together several security analysis tools and performed a design study on how to integrate them into Catalyst. Unfortunately, due to a funding cut the tools were never integrated into Catalyst. More information on this ECP can be found in ASEA Interim Technical Report, Volume 5, Integrated Security Analysis Tools.

3.1.7 Trusted Ontos Prototype ECP

The ASEA contract was modified by an ECP to develop a multilevel secure object-oriented database prototype. ONTOS performed this work using workflow-based security analysis modeling. The prototype database system was delivered and demonstrated. More information on this ECP can be found in the documents that were produced by ONTOS, including the *Philosophy of Protection Document* and the *Demonstration Users Manual*.

3.1.8 Micro Process ECP

The ASEA contract was modified by a two-part ECP. The first part was to study and design an approach for supporting micro processes. The second part, which is discussed in the next section, was to perform a security analysis of Catalyst and make recommendations for improving security.

Modus Operandi performed a study and completed a design for supporting micro processes in Catalyst. Current process enactment mechanisms, including the one developed for Catalyst, focus on enacting standard, pre-planned processes, such as the System Engineering Management Plan (SEMP) (e.g., establish configuration management), Project Planning (e.g., conduct the System Design Review), and System Analysis (e.g., allocate system requirements). While these "macro-process" enactment approaches are very useful, they are not appropriate for smaller, more spontaneous processes common within engineering teams (e.g., to accomplish and coordinate a multi-disciplinary tradeoff analysis). Studies and interviews of system engineers' needs accomplished under SECD identified that significant portions of time are spent assigning, coordinating, and monitoring these "micro-processes."

Modus Operandi documented how a micro-process definition and enactment capability could be created in Catalyst. The document also discussed the differences between micro- and macro- processes, how micro-processes could be supported using the current Catalyst framework, how micro- and macro-processes could be integrated together, and some risk abatement tasks that could be performed when attempting to implement the proposed micro-process capability. More information on this study is available in the ASEA Interim Technical Report, Volume 6, *Catalyst Micro-Process Study*.

3.1.9 Catalyst Security Study ECP

As part of the Security Study/Micro Process ECP, Odyssey Research Associates performed a study of CORBA security and how security could be implemented in Catalyst. They found that the CORBA Security Service specification provided a great deal of flexibility in implementing security policies for CORBA systems.

The CORBA Security Service is essentially a specification that defines where hooks must be placed in CORBA systems. These hooks are calls to a security manager that must decide whether or not to allow an operation. Hooks are placed for sending messages to objects, creating objects, deleting objects,

manipulating servers, etc. The specification defines when these hooks must be called, but it does not specify what processing should be done or how to make the decision to allow/not allow the operation.

A security policy is actually implemented in CORBA by a) writing code to plug into the hooks, and b) making the decision whether or not to allow operations. The idea behind the specification is that no one security policy is going to satisfy all the potential uses for CORBA. It is better to allow security policies to be customized by the application writers rather than dictating how it should work. We determined that COTS solutions for various security policies would be supplied by vendors. Application developers for CORBA would then select a COTS security package that met their needs and plug it in.

In retrospect, this is actually what happened. There are now COTS CORBA security packages available, the first from Concept Five technologies. For some Catalyst users, it may make sense to use one of these packages to control access to Catalyst objects. In practice, most Catalyst users with security requirements rely heavily on physical security to protect their projects. ORA ultimately stated that implementing multilevel security in Catalyst would cost many times more than the rest of the project and would take many years to gain NSA approval. Even with a large investment of time and money, there would still be considerable risk that acheiving this level of security would not be successful.

Modus Operandi recommends using physical security to protect classified information stored in Catalyst. This is not only much less expensive, but it is also much more secure than relying on a computer-based security solution. Using a COTS CORBA security product may add value in terms of helping groups of people work together without accidentally deleting or changing each other's work. Catalyst already provides the locking mechanism to address the majority of groupware concerns, but a COTS package may provide different levels of access or more flexible control.

3.1.10 Northrop Grumman Demonstration ECP

The ASEA contract was modified to include the integration of ORACLE and RDD-100 and the creation of a Catalyst demonstration based on Joint STARS. The ORACLE and RDD-100 integrations are described in Section 3.3.4 and 3.3.5, respectively. The demonstration was created and presented at the ASEA final delivery in November of 1997.

3.2 Catalyst Software

This section describes each software component of Catalyst. The architecture, design motivations, history, purpose and construction time are described as appropriate.

Catalyst software required about four calendar years to develop. During development, the team ranged in size from one person to five full time

programmers. Several Catalyst team members left and several new members joined the team. John Faure, the team leader and chief architect was the only member of the team present from the start of the development through the final delivery. The exact number of man months spent strictly on the development of the software is very difficult to estimate because the ASEA project also required the development of over one hundred documents and reports, several thousand PowerPoint slides, three software builds – each with four review meetings, four ECPs, and many subcontractors to meet with and manage. The entire ASEA effort consisted of about 30 man-years of effort for Modus Operandi.

3.2.1 Catalyst Core IDL

The Catalyst Core IDL is the heart of the Catalyst integration framework. This section explains the purpose of each of its parts.

Catalyst integrates data sources such as databases and tools by "wrapping" them with a standard interface. This standard interface allows the implementation of many beneficial generic operations, such as browsing, reading, writing, moving, and analyzing data. The Catalyst Core IDL defines the standard interface used by Catalyst for accessing objects, attributes, and schema information. It defines the interface for standard factories used to create object instances. Most of the types used by the Catalyst GRM IDL are defined in the Catalyst Core IDL, since both interfaces use them. One of the most basic properties of Catalyst as an integration framework is the presence of standard protocols for framework operations. Requiring even one special case for accessing objects or relationships would double the size of the object access code and seriously compromise the principles of Catalyst.

3.2.2 Catalyst Objects

The Catalyst Core IDL is organized as a series of type and interface declarations that collect related functionality. The interfaces specify the standard Catalyst protocols. They are not meant to be implemented on their own; they are essentially virtual interfaces that should be inherited by the CORBA servers that integrate the data sourced into Catalyst. Standard Catalyst code sections are available to reduce the effort needed to implement the standard interfaces and thus to integrate a new data source.

To tightly integrate a data source into Catalyst requires creating a CORBA server for each class in the data source. The IDL for this CORBA server must inherit at least the ObjectAccess interface. The CORBA server should also inherit the Notification interface if the data source or its integration can benefit from notification of external events.

If the data source being integrated stores relationships between the objects it manages, then the RelationAccess interface should also be inherited. Relations have a query object flag that indicates to the GRM whether or not it should ask the object servers for relationships. If a data source stores relationships internally

and implements the RelationAccess interface, then the query object flag on those relationships should be set to true. This will cause the GRM to query the object server for relationships and then add those relationships the GRM itself stores, and return the combined list to the requestor.

One principle of the CORBA architecture is that for each class defined by IDL interfaces, there must be at least one server program in existence before any objects of that class can be created and manipulated. Server programs exist to handle the method calls defined in IDL interfaces. Servers are programs written in C++, Java, Smalltalk, etc. which usually must be compiled and linked before they can be used. This makes creating a new class in Catalyst a fairly significant operation. To create a new class requires creating, compiling, linking, and registering the server with CORBA.

This principle makes CORBA data models much more static than systems like relational databases, where new tables can be created using a single SQL statement. This makes it even more important to carefully design the object schema for the data to be represented. Work has been done in the CORBA community for dynamically creating server implementations. This may make it possible to someday automate the process of creating server classes for certain types of servers.

In Catalyst, the process of creating a class that is stored by Catalyst has already been automated. The source code for the class is generated by a program. The program reads a definition file created by the user that describes the class name, and its attributes and their data types. The generated code is compiled and linked using a makefile that is supplied. The whole process takes about 5 minutes for the compilation and linking. The data for the class is transparently stored in ObjectStore. The ObjectStore implementation has proven to be very reliable over the four years in which Catalyst has been operational.

The following is an outline of the Catalyst Core IDL module:

```
module Catalyst {
    object types...
    interface ObjectAccess...

    notification types...
    interface Notification...

    relation types...
    interface RelationAccess...

    interface Factory...
};
```

The ObjectAccess interface provides a standard protocol for unique object identification, a capability that was not included in the CORBA specification, on

the grounds that it seriously limits the implementation choices for various ORB operations. CORBA ObjRefs are many-to-one with Objects, which means many different ObjRef values can denote the same object. This characteristic prevents ObjRefs from being used as the basis for general relationships. In Catalyst, the unique identifiers provided by the ObjectAccess interface are used to create and use relationships. Catalyst identifiers are also used to build object structures such as lists, search trees, etc., where object identity is important.

The ObjectAccess interface uses an object identification strategy that is low cost and reliably unique. A standard *include* file provides operations for creating and comparing unique identifiers, reducing server implementation cost and increasing Catalyst uniformity.

The ObjectAccess interface also provides a method for retrieving the name of an object's class, which is crucial information for performing generic browsing, analysis, etc. A method for returning a display "image" that concisely represents the object is included to support generic browsing and navigation. The display "image" is essentially a standard human readable identification protocol, although the display string does not have to be unique. This "image" capability is very important for creating efficient object browsers and navigators.

The ObjectAccess interface contains a method that returns the standard attributes of an object in one operation. These items, in concert with the object's ObjRef, are the most important and commonly used attributes in Catalyst. Because distributed message calls are on the order of milliseconds, providing *one* method to retrieve them *all at once* substantially improves the performance of the framework for a minimal implementation effort.

The ObjectAccess interface also supports more detailed levels of access to an object. A method is provided for retrieving the attribute definitions for an object. The attribute definitions retrieved from an object of a certain class apply equally to all objects of the same class. If CORBA had the equivalent of class objects, this operation would be supported by the class object and not by all the instance objects. The interface also defines a standard protocol for getting and setting attribute values. These capabilities are necessary to support generic browsing, editing, forms display and editing, import/export, etc.

The Notification interface provides a standard protocol for informing objects of the occurrence of events. Notifying objects of events is a powerful and important technique for implementing semantics in the object model, maintaining the consistency of the data, and reducing the amount of custom code needed to implement framework and application operations. Normally, the object being notified is the range of some relationship, and the notifier is the domain of the same relationship, as will be explained below in the discussion of relations.

The RelationAccess interface provides a way to retrieve the relations defined for a class. It also provides ways to retrieve, create and delete relationships between objects. This interface should be inherited by servers for data sources that

internally store relationships between the objects they manage. Inheriting and implementing this interface will give Catalyst access to these relationships.

The General Relation Manager architecture supports relationship operations for objects whose servers do not implement the RelationAccess interface. It also provides the capability for establishing relations and relationships between objects stored by different servers. All relation operations are performed by calling the General Relation Manager, which takes care of calling the RelationAccess interfaces of the object servers as necessary.

The Factory interface is used to create instances of the class. The Factory also contains operations for returning schema information. All object servers must inherit the Factory interface. The *create* operation returns the instance as an ObjectAccess object. This supports generic code that can create instances of any Catalyst class, which is a very important capability for tools such as generic browsers and editors, and for importing, exporting, versioning, baselining, etc.

The Catalyst ObjectAccess interface defines the simplest level of integration with the Catalyst system. The ObjectAccess module requires the existence of three standard attributes that all Catalyst objects must possess. These three standard attributes are class (class name string), image, and unid (unique identifier). The standard attributes support the minimum capabilities thought necessary for a useful integration: to identify an object and be able to relate it to other objects (unid), to classify an object (class), and to display or print a descriptive string for an object (image).

All objects in Catalyst must support a set of standard attributes. These attributes were defined to provide a standard protocol for accessing important characteristics of objects. A few were defined as specified by Catalyst requirements. The standard attributes must be defined in the order and with the type, mode and size definitions specified in Table 3.2.2-1.

Defining standard attributes improves the consistency and uniformity of Catalyst client tools and object servers. They reduce the development effort and run time required by client tools by making the standard attributes consistently and easily accessible. The standard attributes are important to any client tool or object server because they define crucial object information; making them readily accessible is very important. Because the standard attributes already require special attention in an object server implementation, the effort required to adhere to this convention in object servers will be small.

Table 3.2.2-1 Catalyst Standard Attributes

name	type	mode	size in bytes
class	adt_string	read_only	32
unid	adt_unid	read_only	20

image	adt_string	read_write	0
baseline	adt_string	read_only	8
locked_by	adt_string	read_only	32
created	adt_time	read_only	8 (from gettimeofday)
modified	adt_time	read_only	8 (from gettimeofday)
mod_count	adt_long	read_only	4

3.2.3 Catalyst Relations

In Catalyst, the term *relation* refers to the existence of the *definition* of a connection between objects (i.e., the meta-information), while the term *relationship* refers to the existence of a *connection* between existing objects (i.e., an instance of a relation). Relations in Catalyst define a connection between two objects of different classes, or connect objects of one class to each other. New relation types may be defined at run-time. This capability is supported by the Catalyst General Relation Manager.

Relations are named using the form <domain class> <relator> <range class>. The primary direction is from the domain class to the range class. The inverse direction is from the range class to the domain class. The inverse is named using the form <range class> <inverse> <domain class>. An example relation name is "Document contains Section". The inverse name for this relation might be "Section contained_by Document".

Catalyst relations are typed. To successfully create a relationship using a particular relation, the domain object must be an instance of the domain class *and* the range object must be an instance of the range class. Catalyst will reject relationships where these conditions are not true. Catalyst will also reject trivial relationships, which are those in which the domain and range are the same object.

Relations are bi-directional, and have a primary direction and an inverse direction. The direction of the relation is very important to the meaning of the relation. Catalyst maintains the bi-directionality of relationships automatically. If one direction is created, the other direction is automatically created. If one direction is deleted, the other direction is automatically deleted. Relationships may be followed in either the forward or inverse direction by specifying the relation name using the forward or inverse name.

Catalyst enforces *set semantics* on all relations. This means that there can only be one instance of a particular relation between two objects. If the relationship is created more than once, the creation operations after the first one have no effect. There may be many relationships between the same two objects as long as they are of different relations. This is an important point because it means two objects are either related by a relation or they are not. There is no degree to being

30

related. If such a capability was required it can be implemented by using an object between the related ones that stored the degree. N-ary relations can be implemented in a similar fashion.

Recursive relations are those in which the domain class and the range class are the same. For example, Component decomposes_into Component. The inverse of the this relation might be Component decomposes_from Component. This relation forms a recursive hierarchy that can be any number of levels deep. To go down the relation, specify Component decomposes_into Component (the forward relation name). To go back up the hierarchy, specify Component decomposes_from Component (the inverse relation name). Using the forward and inverse names allows one to travel up or down the recursive hierarchy at will.

Using this relation, Component1 could be related to Component2 (this is the forward direction), and Component2 could be related to Component1 (this is also the forward direction). This forms a two object cycle that is legal and could be traversed from object to object in both the forward and inverse directions. This set of relationships do not violate the one relationship between objects rule because the relationship direction is different. This makes the two relationships completely different as far as Catalyst is concerned.

Relations have one of four cardinalities: one-to-one, one-to-many, many-to-one, and many-to-many. The cardinality determines what combinations of relationships may be established between a group of domain and range objects.

• A relation whose cardinality is *one-to-one* specifies that *each* object may be the domain of no more than one instance of the relation, and that each object may be the range of no more than one instance.

• A relation whose cardinality is *one-to-many* specifies that each object may be the domain of zero or more instances of the relation. However, each object may be the range of no more than one instance.

• A relation whose cardinality is *many-to-one* specifies that each object may be the domain of no more than one instance of the relation. However, each object may be the range of zero or more instances.

• A relation whose cardinality is *many-to-many* specifies that there is no restriction on an object's participation in the relation.

Cardinality in Catalyst has never been enforced due to performance penalties with the current implementation. Their enforcement could be implemented in a more efficient manner in the future. Currently, cardinality information provides information about the meaning and intent of the object schema but is not guaranteed to be followed.

3.2.4 Relation Semantics

Relations in Catalyst may possess one or more of a set of well-defined semantics that are part of the basic Catalyst data model. In addition, a relation may possess

application-defined semantics, which may be added at any time. It is important to keep in mind that relations are meta-information so that their characteristics (or modifications of those characteristics) apply immediately to all instances of the relation.

A "hook" has been provided to allow Catalyst user applications to define their own semantics information. This has been implemented as a long word (appl_semantics) that is maintained by Catalyst for each relation. This provides up to 32 separate semantics flags for other user-defined purposes. All semantics are meta-level information, so user-defined semantics are defined on the relation and apply to all instances (relationships) of the relation.

Catalyst relation semantics were defined to support implementation of event-driven operations across networks of objects. Using traditional algorithms to support these types of operations is usually very difficult, so event-driven or access-oriented algorithms based on notifications have become common in object-oriented systems.

An example operation is the deletion of an object and all the objects that are logically "part" of the object. This forms a hierarchical network of objects, in which the top level object may be a car, the next level may be the engine, tires, seats, body, and the next level engine parts, etc. To delete the car using a traditional algorithm would require writing a series of nested loops that traverse the various relationships forming the hierarchy. If a new relation was added from any part to any new part, such as "Engine contains CatalyticConverter" a new loop would have to be added to traverse this relationship and delete any CatalyticConverters that were part of the car.

Using semantics, each part of the car would be related to the car using a relationship with *parts* semantics. Deleting the car would cause each part related directly to the car to be deleted. As these parts were deleted it would trigger notification messages to the subparts. This process would continue recursively until all the car's parts and subparts and subsubparts, etc. were deleted. If a CatalyticConverter was added in the future, it would automatically get deleted, as long as it was connected to the car with a relationship that had *parts* semantics. All the user has to do to delete the car is send it a delete message, Catalyst semantics take care of the rest.

This simple example demonstrates how semantics can be used to create very resilient and flexible capabilities with minimal effort. The work required to carry out the applications is distributed throughout the servers and the GRM.

Semantics are implemented in Catalyst partly by the GRM and partly by the object servers. When something happens to an object that could affect semantics, the object server is responsible for telling the GRM that it happened. The GRM then determines what objects are related to the affected object, and what notifications should be sent to them. The notified objects may take some further

action that they would inform the GRM about. The process continues until all the affects have been propagated to all the objects that need to be informed.

The GRM and object servers must be implemented carefully to avoid any deadlocks or erroneous actions that can be caused by the propagation of semantics notifications. This turns out to be fairly easy, as long as the following principle is followed: no server (including the GRM) should make any distributed calls while holding any resource locks. If this principle is not followed, it is inevitable that one of the distributed calls will eventually call the original object and deadlock with itself on the locked resources. This is particularly true of the GRM. If the GRM makes any distributed calls while holding locks, it is almost guaranteed that those calls will deadlock in the GRM before long.

To conform to the principle requires using the following processing strategy. First, acquire the locks needed, and while the locks are in place do any processing required, such as changing the object or relationship state. Second, while still holding the locks, collect the list of other objects that might be affected by this action. Third, release the locks. At this point, no changes can be made that affect the object or relationships state. Make the distributed calls to the list that was collected. This simple strategy has proven itself to be effective and deadlock-free for several years of operation.

The Catalyst built-in semantics *value, existence,* and *parts* are implemented partly within the GRM. When the state of an object is modified, the object server is required to notify the GRM using the *object_modified* method. This method finds all objects related to the modified object by relations with value semantics. The GRM then sends each of these objects an *object modified* notification, which includes the object reference and unique identifier of the object that was modified.

When an object is about to be deleted, but before the deletion takes place, the object server is required to notify the GRM using the *object_deleted* method. The GRM then finds the list of all objects related to the object to be deleted by relations with existence semantics. The GRM sends each of these objects an *object deleted* notification, which includes the object reference and unique identifier of the object to be deleted. These notifications are sent before the object is deleted, and with the object in an unlocked state, so that other objects that are informed that it is about to be deleted can communicate with the object before its demise. This is important for maintaining the consistency of complex object networks. Before an object leaves the network, other objects can talk to it and then update themselves appropriately. For example, an object maintaining a sum might subtract the value of the deleted object from the sum, rather than requerying all the objects and computing a new sum. Once the *object deleted* call to the GRM returns, the object can delete itself.

A second action taken by the GRM when it receives an *object deleted* message is to handle parts semantics. Existence semantics are always handled before parts semantics. To handle parts semantics, the GRM finds the list of all objects related

to the deleted object by relations with parts semantics. The GRM then deletes each of the objects related by parts semantics. This can trigger a wave of existence notifications and further parts semantics processing as subparts and subsubparts are deleted. The GRM has been constructed to handle these conditions correctly and without deadlocking. Parts semantics are used in the Process Enactor and Administration tools to delete old processes and their parts from projects.

The built-in semantics in Catalyst propagate through object networks in a synchronous fashion. This means that if an object is modified and it sends an *object modified* message to the GRM, the message will not return until *all* the notifications to *all* the objects have been sent out. Since this may trigger many levels of notifications through a hierarchy of objects, this can take some time to occur. The rationale for using synchronous propagation instead of asynchronous propagation is that the built-in semantics were designed to help support maintaining the consistency of complex networks. This is much easier to design and implement if notifications spread out synchronously. The object that was modified knows that all the required changes to the object network have been made before it continues processing after the *object modified* notification message to the GRM returns. Many times, this is a valuable simplifying assumption. User-defined semantics can be implemented to propagate either synchronously or asycnhronously.

User-defined semantics can be implemented completely within the object servers that are required to support them. This can be done as follows. When an object receives a notification message that could trigger the user-defined semantics, the object server should query the GRM for the list of objects related with relationships that possess the user-defined semantic. The object server should know or look up which relations possess the user-defined semantic, and use this information to get the correct set of related objects from the GRM. The object server can then send the appropriate notification message to each of the related objects, or take the appropriate action on each of the related objects, such as using one of the operations defined by the object. Implementors should follow the deadlock avoidance algorithm outlined above to ensure the correct processing of their semantics. Performance of user-defined semantics should be comparable to the built-in semantics.

Catalyst defines a set of relation semantics that are not currently enforced or checked. These semantics may be enforced or checked by later implementations of Catalyst. These semantics serve several important purposes. They provide a standard hook or interface for the existence of these semantics that can be used to facilitate cooperation among different tools and servers that are interested in them. This cooperation is enhanced by the existence of a standard mechanism for storing and accessing them. They also provide information about the meaning and intent of the object schema to be used by people wishing to understand the schema, and by automated tools processing the schema, such as the Model Explainer from CoGenTex.

Catalyst defines *acyclic relation semantics* to mean that there is no path using only relationships of the acyclic relation from any object back to that object.

Catalyst defines *hierarchical relation semantics* to mean that there are no objects in a network that have more than one parent within a single network of objects connected with the hierarchical relation.

Catalyst defines *transitive relation semantics* to mean that if a is related to b, and b is related to c, then a is related to c, assuming all relationships are of the transitive relation.

Catalyst defines *commutative relation semantics* to mean that if a is related to b, then b is related to a.

3.2.5 Catalyst GRM IDL

The Catalyst General Relation Manager (GRM) IDL defines the interface to a general purpose server for storing and managing relations and relationships in Catalyst. This server stores relationships between objects in different servers, and between objects in the same server when that server does not support relationships. The GRM provides a unified source for all the relationships of an object by adding together the relationships stored by the servers to the relationships it stores itself. This allows client programs to see all the relationships for an object with one call to the GRM and without having to know where the various relationships are stored. Thus, the GRM provides important integration and transparency services in the Catalyst integration framework.

The GRM IDL provides operations for finding, creating, modifying and deleting relations. These operations support definition of the object schema.

The GRM IDL provides operations for retrieving, creating, deleting, locking, and unlocking relationships between objects. Many of these operations work on groups of relationships rather than a single relationship at a time. This improves Catalyst efficiency because invoking operations on CORBA objects is fairly expensive. Cutting down on object message traffic through grouping is a significant savings. In the future, we may add an even more powerful grouping operation for relationship retrieval that retrieves all objects related by a list of relations instead of a single relation.

The GRM IDL provides the *object created* operation to inform the GRM when a new object has been created. This is useful for making sure all objects are registered in the GRM so that they can be looked up later by their unique identifier. It is also used during *autopublishing* to enter an object so that its tool key may be stored. Autopublishing is used by tight integrations to make data visible in Catalyst and is discussed in Section 3.3.4.

The GRM IDL provides the *object modified* and *object deleted* operations to inform the GRM when objects are modified or deleted. These operations are critical for implementing built-in semantics.

The GRM IDL provides operations for storing, removing and looking up keys that identify an object within an integrated tool. These operations were added to the GRM to make it easier to write tight integrations. The GRM already stores and indexes the information related to the object that was needed, with the exception of the tool key, which was added.

The GRM IDL provides interfaces for exporting, importing, and debugging GRM information. These interfaces are currently not used. Importing and exporting the GRM contents to text files could be implemented to make upgrading the GRM server easier.

3.2.6 General Relation Manager

Catalyst was initially designed about the same time the CORBA 1.2 specification was approved by the Object Management Group (OMG). At that time, the Relationship Service specification was approved. After careful analysis, it was concluded that the current specification was not appropriate for implementing an engineering integration framework. The Relationship Service Specification states that up to three objects can be used to implement one relationship. There are several advantages to using objects to implement relationships. It means that any support, such as backup/restore, searching, etc. that applies to objects immediately can be applied to relationships as well. Unfortunately, there is a considerable performance penalty with this design, since objects are relatively expensive in CORBA.

At the time Catalyst was being designed, Modus Operandi had a great deal of experience in building object-oriented systems that made heavy use of relationships. It was found that there were generally an order of magnitude more relationships in such systems than there were objects. Relationships were generally created and deleted much more frequently than objects, as well. This placed time and size constraints on relationships that had to be met in order to create a useable integrated system.

A much more lightweight implementation had to be created in order to meet the expected performance constraints on relationships. For this reason, relationships in Catalyst are not stored as objects. A special lightweight relationship service was developed, based on the Catalyst Core and GRM IDLs. To store a single object in CORBA requires somewhere around 3000 bytes and requires approximately 100 ms to create or delete. By not storing relationships as objects, the Catalyst GRM is able to use only 48 bytes per relationship and take about 1 ms to create or delete one. This architectural decision probably made the difference between the current, very useable Catalyst system, and one that would have been too slow and memory-intensive to be useful. Catalyst has been optimized to handle many lightweight relationships in a LAN environment, because this meets the performance requirements *and* the operational requirements for most engineering development.

The current Catalyst General Relation Manager was written in C++. It is essentially a custom database for storing relations and relationships. It uses Solaris memory mapped files to provide very fast access to the disk files that store its information. The current implementation can handle approximately 300 individual relationship operations per second. When relationship group operations are performed, it can handle as many as 5000 relationship operations per second.

The current implementation is not very portable because it relies on Solaris memory mapped files. Other operating systems – such as other UNIX flavors and Windows NT – provide similar features. The GRM could be implemented using an object base such as ObjectStore or using a relational database such as ORACLE. The performance of such an implementation is hard to forecast, but it would probably be lower than that of the current implementation because it is hard to manage disk interaction any more efficiently than it does.

3.2.7 Catalyst Object Servers

The Catalyst Object Servers are CORBA servers written in C++ that inherit and implement the ObjectAccess, Factory, and Notification interfaces from the Catalyst Core IDL. These servers were created to store objects of classes that were not stored by any integrated tool. Any time an object class was needed and there was no tool to store it, a Catalyst Object Server was created for it. This includes all the classes needed to support process enactment, impact analysis, and project modeling.

The Catalyst Object Servers use the Object Development Framework (ODF) facilities provided by NEO to store their object data persistently in ObjectStore databases. This persistence capability is bundled with NEO, which saved a lot of development time and provided a high-quality persistence capability. The ObjectStore databases provide transactions and database recovery after failures.

The Catalyst Object Servers are actually generated by a C++ program. This program takes a definition file that describes the attribute names, modes, types, and sizes and generates the IDL, DDL, and C++ files needed for the server. The server is then compiled and registered with NEO and a new Catalyst object class is available.

Generating the server code instead of writing it by hand saved an enormous amount of repetitive coding. There were over 50 Catalyst Object Servers in Catalyst at the time it was delivered.

Now that NEO is no longer supported, these servers will have to be ported to VisiBroker, which does not come with any bundled persistence solution. ObjectStore Persistence Pro will probably be used to provide object persistence, since this technology has already been proven to work well in the current NEO servers.

3.2.8 Catalyst Object Cache

The Catalyst Object Cache is a key part of the Catalyst architecture. It caches CORBA objects and relationships in memory in a client application. Since the objects and relationships are probably stored by a server in a remote machine, this caching makes the client applications far more responsive to the user. The object cache works by storing all objects, relationships and meta-data in memory as they are accessed by the client application. The object cache retrieves data from the servers when it is demanded by the client application. The caching is totally transparent to the client. The object cache tries hard to minimize both the amount of information sent across the network and the amount cached in memory. Using the object cache, scrolling, screen redraws, data analysis, data editing and any other operation which works on CORBA data can proceed at memory speeds rather than the approximately four orders of magnitude slower CORBA access speeds. Caching objects and relationships in memory creates redundancy which must be carefully managed to avoid race conditions and other data corruption problems. The architecture for the object cache is shown in Figure 3.2.8-1.

3.2.8.1 Cache-to-Server Locking Policies

Catalyst uses object and relationship locking in the servers to prevent data race conditions between multiple users. The locking model is based on a check-in, check-out paradigm similar to configuration management tools familiar to most engineers. Rather than enforce a single model of locking, Catalyst allows the user to select the model which fits the activities they are performing.

Catalyst servers support a simple model of data locking, on top of which many interesting client level locking models can be implemented. Catalyst server locks associate a user identifier with each piece of locked data. Only the user holding the lock may modify, delete, or unlock the data. Catalyst server locks last without expiration until they are unlocked. This supports the long-term, multi-session locking of work products necessary for modeling real-world work and development practices.
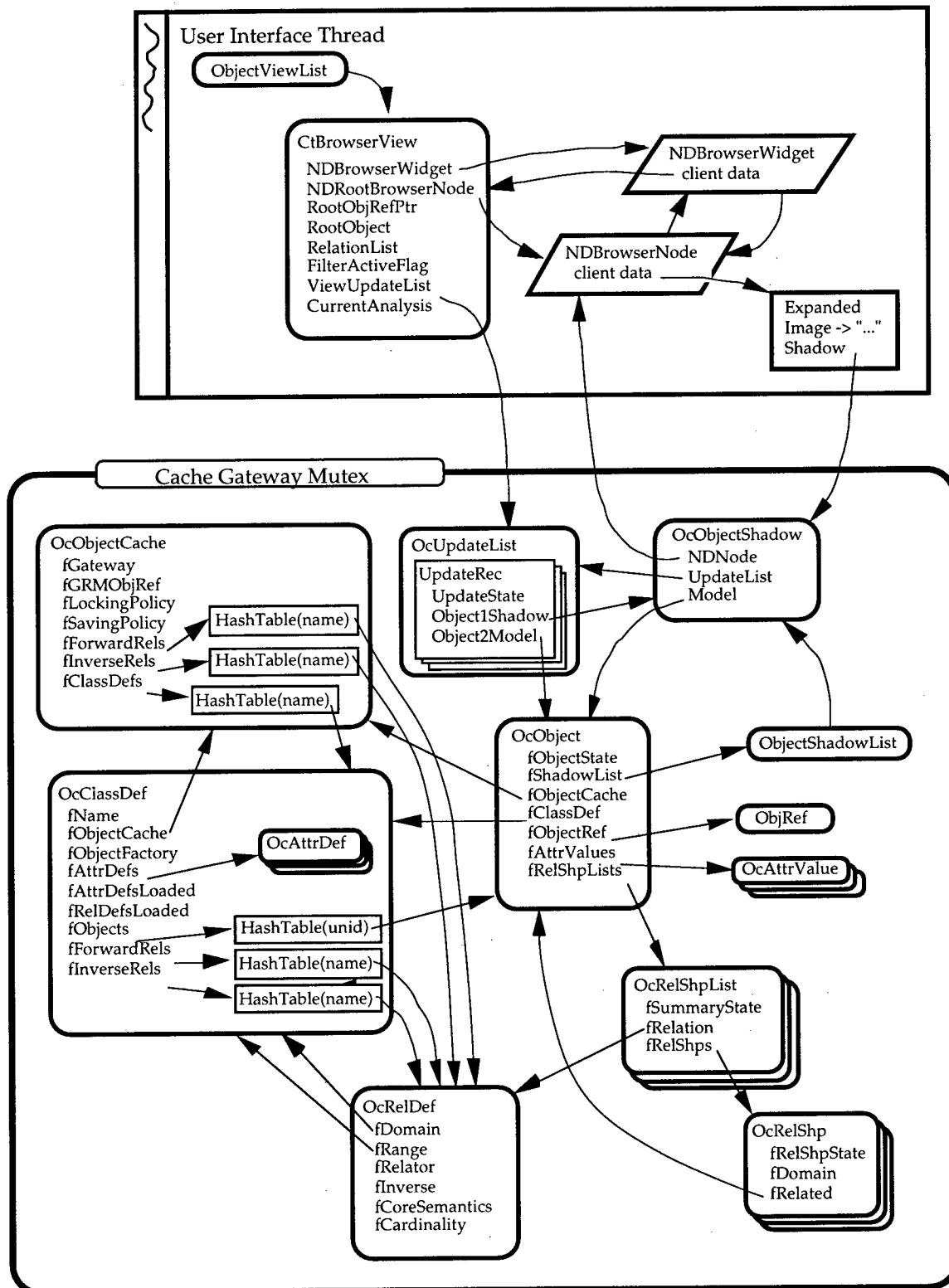
Figure 3.2.8-1 Catalyst Object Cache Architecture

The object cache supports three locking policies at the client level. The current policy in effect may be selected by the user or the application. Each policy has advantages and disadvantages which make it appropriate for certain types of usage patterns or situations. The locking policies are:

(a) No Locking

(b) Just In Time Locking

(c) User Directed Locking

Policy (a) no locking, is the weakest type of locking, and runs the greatest risk of race conditions, and failures to write. A failure to write occurs when the user has modified data in the cache, but the modifications can not be written to the server because another user has the data locked. This policy is the easiest to use and also the most efficient to execute because nothing is ever locked. This locking mode may be appropriate when little or no chance of conflicts exists (i.e. no other clients have either visibility or write access to the data), or when a small number of minor changes needs to be made. When shared server data must be accessed, this is a poor choice.

Policy (b) just in time locking, is safer than no locking and is usually faster than user directed locking. Under this policy, the object cache will attempt to lock data just before modifying it. Once locked, the data will remain locked until unlocked by the user or application. Using this policy, only data which is actually modified is locked; data which is only read is never locked. The drawbacks of this approach are that there is a chance that race-conditions or failures to write may occur. This policy is not implemented in the current object cache. It could be implemented in about two weeks of effort.

Policy (c) user directed locking, is the strongest locking policy, providing the greatest protection but also the greatest costs. Under this policy, a subset of the object network is identified by the user or application and each data item in it is locked, prior to any modifications being made. Once the entire subset has been locked, the user or application is free to make any modifications they want with confidence that there will be no race-conditions or failures to write when the modifications are saved. This is probably the most useful locking policy when a large number of changes will be made to shared server data.

The user directed locking policy can be used very naturally to mimic the way in which groups of people currently collaborate to create shared work products. Each person is assigned a subset of the overall work product, which may be a document, requirements network, design hierarchy, or any other subsettable object network. Each person then uses their applications or the object browser to lock their assigned subset. The work group can then proceed to cooperate on building the total work product without the possibility of race-conditions or failures to write.

Acquiring the locks under the user directed locking policy should be assisted by the application or object browser. It should be both easy and logical to specify what parts of the object network to lock. The user should not have to enumerate what to lock or unlock manually. For example, in a document editor the user would request that a document section be locked, and the document editor would automatically lock all the (possibly) many objects making up the section. Another example would be a requirements editor locking all requirements in hierarchy starting at a user selected object. The requirements editor would automate the necessary tree traversal and locking. The most general way to acquire locks under this policy would be to use the object browser. This tool will provide several types of general traversals and searches which can be used to identify and lock or unlock any selection of the object network.

Because Catalyst server locks last until they are unlocked, the enforced division of the object network under this policy can remain in place for days or weeks or until the overall work product is completed. Management and modification of who has what parts locked may be necessary at times during the work process creation. This reflects the natural changing of the logical ownership of parts of the work product as its creation proceeds, or as the work products flow through their lifecycles.

3.2.8.2 Cache-to-Server Saving Policies

Orthogonal to the issue of how to lock and unlock data is the issue of when to write modified data from the cache back to the object servers. The Catalyst client architecture defines two data saving policies. Like the locking policy, the current policy in effect may be selected by the user or application. The saving policies are named:

(a) Immediate Saving (Write Through Cache)

(b) User Directed Saving (Write Back Cache)

Policy (a), immediate saving, causes all data modifications to be written to Catalyst servers as soon as the modifications are made. This policy is similar to the write through cache policy used in some computer architectures. This policy makes client changes visible to the rest of the network as soon as possible. It also allows active object networks to propagate changes in the order in which changes were made. For locking policies in which failures to write are possible, this saving policy alerts the user to the failure as soon as possible. The drawbacks of this policy are that the user must wait for each save operation to complete, and that if multiple changes are made to the same object, the cost of saving it to the server must be paid each time.

Because of its immediate feedback on failures, this saving policy is a good choice when the no locking or just in time locking policies are used.

Policy (b), user directed saving, keeps track of cache modifications and saves them only when directed by the user. This policy is somewhat like the write back

cache policy used in some computer architectures, except that the user has the option to discard changes without ever saving them. One major advantage of this policy is that the cache data may be modified freely at memory speeds without ever having to wait for the Catalyst servers. The main purpose of the object cache is to allow applications to run at memory speeds most of the time, only having to pause infrequently to load or save data. Another advantage is that the user can save data when it is in a consistent state, rather than incrementally. This may be very important when many other clients are reading and using the data. A final advantage is that multiple changes to the same data only have to be saved once, improving both client and server performance.

Using the user directed saving policy allows the user to collect and work on a network of data, and save it only when they feel it is consistent or complete. Should they decide that their modifications are incorrect, the entire object cache can be unlocked and discarded, essentially a powerful undo capability at the object network or working set level.

Switching saving policies from (a) to (b) causes the cache to begin accumulating changes rather than immediately saving them. Switching saving policies from (b) to (a) causes all pending changes to be written back, and then all subsequent changes to be written immediately.

3.2.8.3 Cache Loading and Swapping

The object cache is loaded with objects, relationships and meta-data incrementally, as needed by the activity of the client application. Objects and relationships are brought into memory transparently, when accessed by the client application. The goal of the object cache is for it to appear to the client application as if all objects and relationships are available in memory at all times. In actuality, the object cache retains in memory all objects and relationships accessed by the client application until the upper limit on total cached objects is reached. At this time, objects will be swapped out on a least recently used basis, to make room for incoming objects. When an object is swapped, all its relationships are swapped out as well.

Initially the object cache contains no meta or instance data. It is "primed" with one or more calls to functions which supply the CORBA stringified object reference or naming context path to an object. The object cache obtains a connection to the actual CORBA object in its object server and uses the Catalyst Core IDL to obtain meta and instance information about it. The object's unid (Catalyst unique identifier), display image, and class name are retrieved. The names and definitions of the class's attributes are retrieved and stored in a class definition object (OcClassDef). The relations defined for the class are retrieved and stored in relation definition objects (OcRelDef). All this meta information is indexed by the object cache (OcObjectCache). The Catalyst Core IDL contains methods for retrieving meta-information from object instances, because CORBA is a one-level object oriented system, and therefore has no meta-objects to ask for meta-information.

42

Next, the client application will request additional objects by retrieving relationships belonging to the "priming" objects. These requests cause the object cache to retrieve and store the relationships for the priming objects. At the other end of each relationship is a new object, which is cached and interrogated the same way the priming objects were. Thus new meta-data in the form of class and relation definitions can be discovered and stored. When objects of existing classes are found, then no meta-data operations are performed; this work is always done when the first object of a new class is found.

Over time, a working set of meta and instance data has been loaded into the cache, decreasing the server traffic to an occasional access when a new object is encountered. The typical Catalyst client tool session will be slightly slower at first as the working set is built up, then should proceed faster as all the working set objects are cached in memory. The client tool will slow again when new areas of the Catalyst object network are loaded or when cache saves are performed. This adaptive behavior will optimize client tool performance by automatically tuning cache contents to the user's actions.

3.2.8.4 Cached Object Structure and Swapping

When the number of objects in the cache passes a limit, then objects are swapped out to make room for new ones. An object in the cache consists of four parts. Only pointers to OcObject objects are visible outside the object cache. These are essentially handles to the cached object information. The OcObject object contains pointers to the other three parts of a cached object. The fData pointer is a link to the object's attribute and relationship data. The fObjRef pointer is a link to the CORBA object reference used to talk to the actual server object. The fShadowList pointer is a link to the list of OcObjectShadow objects which maintain connectivity with the views currently displaying the object.

The capability for swapping out data has been built into the current object cache implementation, however the code for checking for "object overflow" and for actually swapping objects out was never implemented. Even with thousands of objects in the cache there was never a problem with running out of memory. This may change when Java or PC based clients are used. The swapping out capability could be implemented in two weeks of engineering effort.

Because OcObjects are pointed to directly by many data structures and threads, destroying them would be very complex, if not impossible to perform reliably. In fact, destroying an OcObject would require synchronizing all threads with pointers to it on the object's imminent destruction, which would be very costly in programming and execution time. OcObjects are therefore never destroyed. When an object is server deleted by the user, all three pointers in OcObject are set to null. This indicates to all data structures and threads pointing to the object that it has been server deleted. The OcObject has become a "tombstone" indicating that the object is deleted, preventing access to deallocated memory and other memory management problems. Because OcObjects are only 12 bytes in size, having even 10,000 tombstones in existence would not be a serious

43

problem, and it is unlikely that any one application session will result in server deleting this many objects. Using tombstones to safely delete thread shared memory was a key design technique which helped make Catalyst clients very reliable.

When an object is swapped out, its CORBA object reference (fObjRef) is retained, because without it there is no way to ever swap in the object again. The object's shadow list (fShadowList) is also retained, because this information can not be reconstructed from the object server. As would be expected in a distributed system, the object servers are not aware of shadows or views.

When an object is swapped out, its attribute values and relationships are freed so this memory may be used to store other objects. (The memory is returned to the operating system rather than being pooled by the object cache). If the swapped object is accessed again in the future, the object cache detects this fact by noticing that its data pointer (fData) is null. It then uses the object reference it retained (fObjRef) and rereads the object's data from its server using the Core IDL. The process of rereading an object which was swapped out is called an object fault, and is analagous to a page fault. Object swapping and faulting are done transparently to the object cache caller. To the user of the object cache it should appear as if all meta and instance objects reachable from the priming objects are in memory all the time.

3.2.8.5 Class and Relation Definition Swapping

Class and relation definition objects are never swapped out. This is because they are small in size, few in number, and very interconnected to the rest of the cache. Swapping out a class definition would require swapping out all instances of the class and their relationships, and all relation definitions that class participates in. The effort required to swap a class definition out would therefore be very high and would only yield a small number of free bytes (in the class definition object itself) for reuse. Swapping a relation definition has similar complications and yields a similarly small payoff in free bytes. Because of these considerations, once a class or relation definition has been cached, it remains in memory as long as the client tool runs.

Understanding relationship swapping requires knowing a few facts about Catalyst relationships and how they are cached. Relationships in Catalyst are bi-directional. A domain object is connected to a range object by the forward direction of the relationship. A range object is connected to a domain object by the inverse direction of the relationship. Each object stores one direction of all the relationships it has with other objects. The related objects store the other direction. In essence, the object stores the links from it to the other objects its related to, and the other objects store the link from themselves back to the original object. This forms a complete bi-directional link.

The implementation of this is as follows. Each object keeps a list of all the range objects for those relationships in which the object is the domain. This is called

the forward list. Each object also keeps a list of all the domain objects for those relationships in which the object is the range. This is called the inverse list. Note that the relationships stored by the forward and inverse lists of one object are completely different. Each forward and inverse list stores half of a relationship. The other half is stored by the related objects.

3.2.8.6 Data ownership and sharing

Data ownership and sharing are very important concepts in a multi-threaded system where data flows between threads. In the object cache and Catalyst client architecture, the transfer of ownership of dynamic data is carefully documented whenever data may flow from one thread to another. This is complicated by the passing of certain lists of data because the ownership and status of the list structure may be different than that of the list contents.

All dynamically allocated data is classified as either thread shared or thread private. Each piece of thread private data is owned by one and only one thread. Only the thread which owns a piece of thread private data may access that data, or have a pointer to it. No other threads may access that data in any way. The owner of thread private data is responsible for deallocating it.

Thread shared data is jointly owned by all the threads which have a pointer to it. In the Catalyst client architecture, all thread shared data has a mutex lock associated with it which protects it from simultaneous access. The data may have its own mutex lock, or it may protected by the object cache Gateway mutex. Thread shared data may be accessed by a thread only when the thread is holding that data's mutex lock. If a thread is not holding the mutex lock, then that thread may not access the thread shared data in any way, other than by storing, passing or copying the pointer to it.

All thread shared data in the Catalyst client architecture has been carefully identified and designed, and is therefore always a well known part of the design. This is significant because it means that local variables and temporary data structures are never thread shared.

Thread shared data may be deallocated only when just one thread has a pointer to it. In the Catalyst client architecture, this can happen only when the user interface thread knows that all the other threads accessing a piece of thread shared data have terminated. At this time, it is safe for the user interface thread to deallocate the thread shared data, which may be the object cache, message queues, or result lists. During normal client application operation, thread shared data is never deallocated, because many threads are accessing it. Because of its role in coordinating the operation of the client tool, only the user interface thread will ever deallocate any of the thread shared data, because only it will know when it is safe to do so.

Extreme care must be taken in all coding not to accidently deallocate thread shared data, or to access it without locking the mutex. These programming errors are very difficult to track down, and cause failures which are not always

repeatable. Class interfaces where data is transfered between threads carefully document the status and ownership of each parameter after each operation.

Data flow between threads accessing the object cache is often accomplished by the transfer of ownership of thread private data in the course of an object cache call. For example, the user interface thread may receive ownership of a thread private list of data which was created by the object cache. Usually this kind of ownership transfer involves creating new lists or copies of data by one thread or the other so that its ownership may be transfered and therefore be considered thread private. Thread private data is desirable because the owner of the data may access it freely without locking or holding any mutexes, increasing concurrency. To continue the example, the user interface thread can unlock the Gateway mutex after receiving the list of data and continue to safely use that data. The down side of this type of transfer is that it requires time and memory to make the copies.

An additional facet to the problem is that often the status of a list of data is different than the status of the list contents. For example, calling OcClassDef::GetRelDefs returns a list of relation definition objects (OcRelDef) which belong to a class definition. The list returned is a new one created by the OcClassDef object, and it is thread private and owned by the caller after the call. The list contains pointers to OcRelDef objects which are thread shared and owned by the object cache. The caller may do anything they wish with the list object, however, they can only access the OcRelDef objects when they have Gateway mutex locked.

An effort has been made in the object cache design to balance the efficiency of thread shared data with the concurrency of thread private data. In addition, the status of all parameters of interfaces which may cross thread boundaries has been carefully documented to ensure that there is no question about who owns what data, who has to deallocate it, and whether mutexes need to be locked in order to access each part of it.

Care must be taken to deallocate lists correctly. When the list and its contents are thread private, then the list structure and its contents must be deallocated. When the list is thread private but the contents are thread shared, then the list structure must be deallocated without deallocating the contents.

3.2.9 Catalyst Browser

The Catalyst Browser is a client tool implemented in C++. The Browser was the first graphical Catalyst tool created, and it has undergone many changes and enhancements over its lifetime. It is difficult to estimate, but somewhere between 2 and 3 man-years have been spent on its construction and maintenance.
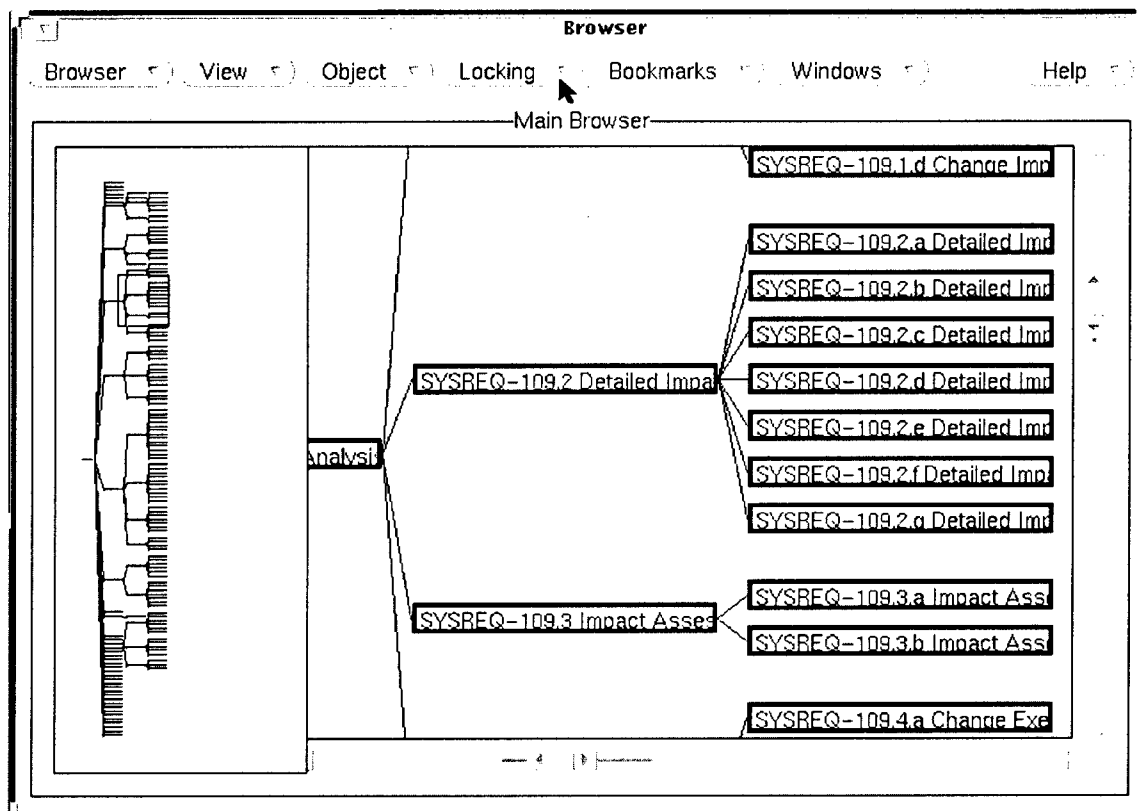
Figure 3.2.9-1. Catalyst Browser Tool

The Browser allows the user to graphically navigate and view any object, attribute, or relationship in Catalyst. The Browser also allows objects and relationships to be created, deleted, modified, locked and unlocked. The Browser is very useful for researching parts of a system being developed. It allows the user to look at data in many integrated tools without knowing how to operate any of the tools. The Browser is often used in conjunction with other Catalyst tools.

3.2.10 Catalyst Support Tool

The Catalyst Support Tool was implemented in C++. The purpose of this client tool is to perform setup, testing, backup/restore, and management functions for Catalyst. The Support Tool does not have a GUI-based user interface, which makes it easier for Modus Operandi to support remote Catalyst sites. Not having a GUI makes it easier to run the Support Tool across slow Internet and dial-up connections.

The Support Tool provides operations for creating, deleting, and testing the General Relation Manager. These operations allow Catalyst Administrators to set up Catalyst and to determine whether the GRM is operating correctly. Quick tests, timing tests and a full operational test suite are provided. The Support Tool determines automatically if the tests passed or failed and summarizes the results at the end of the test.

47

The Support Tool provides operations for testing object servers. Quick, timing and full operational tests are provided. The Support Tool determines automatically if the object server tests passed or failed, and summarizes the results at the end of the test. The object server tests are useful for determining if a data source has been tightly integrated correctly.

The Support Tool provides operations for importing and exporting relation definitions to and from the GRM. This capability is used to load new relation definitions into the GRM.

The Support Tool provides operations for importing and exporting object communities. Object communities are networks of related objects. They are specified by giving a starting object and a list of relations to be followed. Exporting an object community is performed by writing all the objects, attributes, and relationships in the community to an ASCII text file in the format defined by Catalyst. The Support Tool exports the starting object and then follows the instances of the specified relations recursively and exports the objects it encounters. The export process stops when all objects connected together have been exported.

Importing an object community requires reading an export file and creating all the objects, attributes and relationships it contains in Catalyst. An existing object community can be restored by recreating the objects using the same unique identifiers they had when they were exported. This technique can also be used to fix any holes where objects were accidentally deleted from an object community. Conversely, a totally new object community can be created by giving each object a new unique identifier as it is imported. This can be used to create a copy of an object community.

Using the Support Tool requires a little more expertise and low-level knowledge than it should. Object community operations such as defining, importing, exporting, copying, deleting, locking, unlocking should be added to the Catalyst Browser Tool and made more convenient for the user. The concept of object communities was discovered and evolved after Catalyst had been sufficiently implemented to support complex networks of objects.

3.2.11 Catalyst Impact Analysis Tool

The Catalyst Impact Analysis Tool was implemented in C++. The Impact Analysis Tool required about nine man months to design, code and test.

The purpose of the Impact Analysis Tool is to provide the user with the capability to determine how changes to an existing Catalyst object will propagate and cause impacts throughout a system's existing object network. Impacts are found by traversing the object network based upon an Impact Definition, which defines a starting object, a set of predefined relations, and conditions that "connect" it to other objects in the network. Consider the example object network shown in Figure 3.2.11-1.
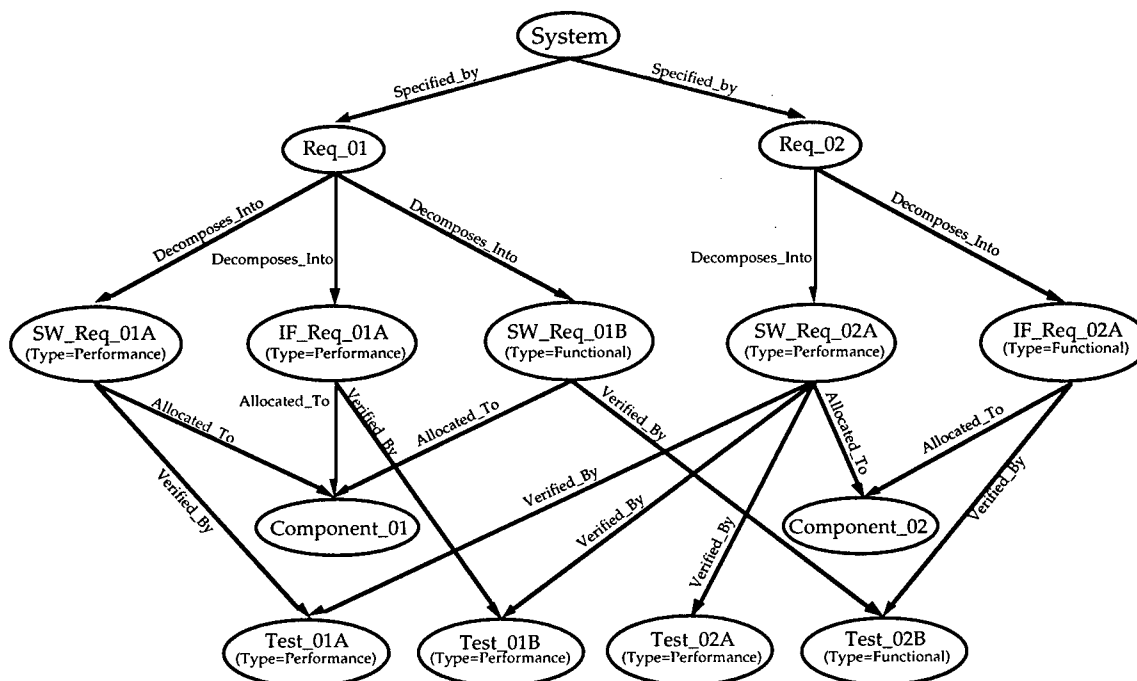
Figure 3.2.11-1. Example of an Object Network

Based on this object network, suppose we wanted to determine all the impacts if we change the object Req_01. The first step is to start from the object Req_01 and find the related objects. Once these related objects are found (SW_Req_01A, IF_Req_01A, SW_Req_01B), we check the objects related to them. For example, from the SW_Req_01A, we check related objects and find the Component_01, and Test_01A. Following a similar procedure for objects IF_Req_01A and SW_Req_01B, we find that Test_01B and Test_02B are also impacted.

Figure 3.2.11-2. Catalyst Impact Analysis Tool

The previous example accounted for all relationships (i.e., there were no conditions imposed on the relationships). Suppose we are only interested in the impacts to SW_Requirements that specify Performance characteristics. In this case, we are looking for relations among objects of the type Requirement that decompose into SW_Requirement, with the condition that the SW_Requirement.Type attribute equals Performance. This would lead directly from Req_01 to SW_Req_01A. To gain more information about the impacts, we

can also check relations from the SW_Requirement. For example, we may also be concerned with the relations SW_Requirement allocated to Component, and SW_Requirement verified by Test with the Test.Type attribute equal to Performance. Using these new conditions, and starting at Req_01, we find the only impacted objects to be Req_01, SW_Req_01A, Component_01, and Test_01A. Figure 3.2.11-3 shows the basic sequence of operations to perform an impact analysis.
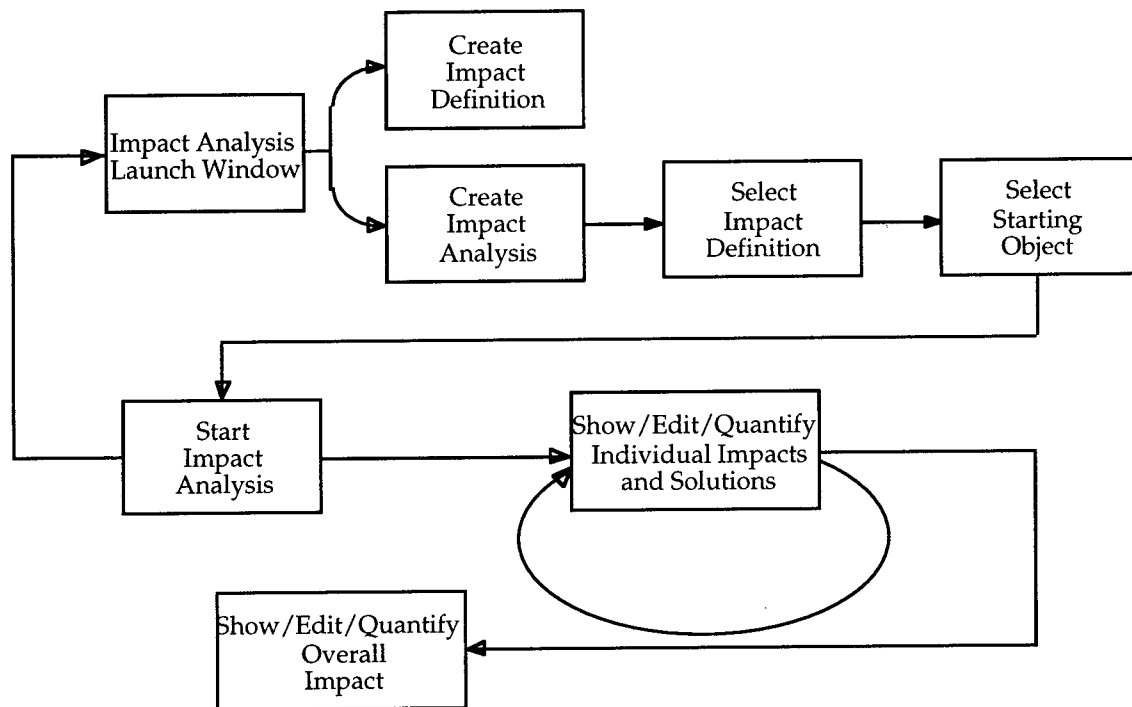
Figure 3.2.11-3. Sequence of Operations During Impact Analysis

It is important to note that an impact analysis may be run multiple times, each time with different starting objects and even different impact definitions, with the results either being truncated or concatenated based on the user's preferences. Figure 3.2.11-2 shows the Impact Analysis main window. Methods for Creating, Editing, and Deleting impact analyses and viewing the results are provided by the set of impact analysis user interfaces. An important feature of the impact analysis is that when an analysis is being run, the user may still interact with the Impact Analysis user interfaces. This is possible because the analyzer component of the Impact Analysis tool is designed to be multithreaded (i.e., it has its own thread of control). This also allows multiple impact analyses to be run simultaneously.

3.2.12 Catalyst Process Definition Tool

The Process Definition Tool was created using InSight, an object-oriented meta-modeling tool developed by Modus Operandi. InSight can be used to create tools which model complex object interactions using declarative constraints. Objects in InSight can be displayed in presentations which are also governed by

constraints. An engineer who is familiar with InSight can use it to create single user CASE tools in a matter of weeks without any C++ or other programming. The Process Definition Tool uses objects to model and display the various parts of a process being defined by the user.

The Process Definition Tool required only about 4 weeks to develop, and has required about 4 additional weeks of enhancements since then. Using InSight to create the Process Definition Tool saved an estimated 24 weeks of programming effort.
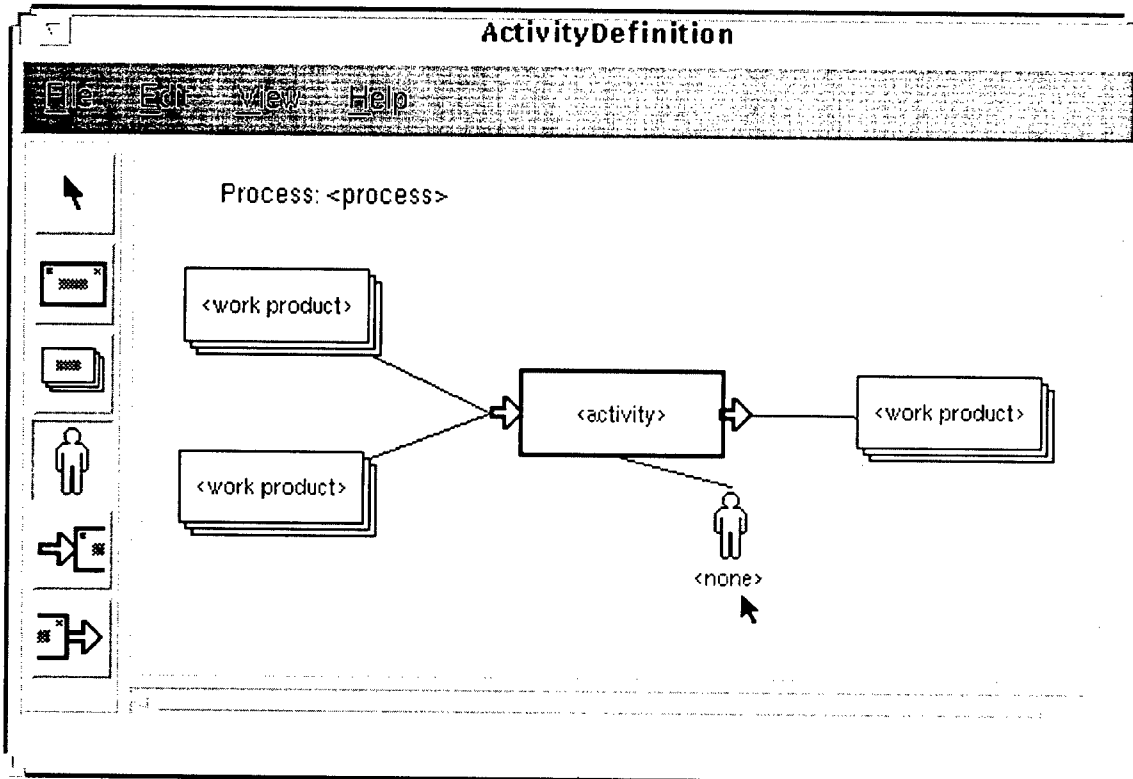
Figure 3.2.12-1 Catalyst Process Definition Tool

The Process Definition Tool allows a user to graphically create and document a process. Catalyst uses a process model that is similar to the one created by the Software Engineering Institute (SEI). An additional level of detail, known as steps, had to be added to the model to define a process in enough detail that it could be enacted by a tool. Once a process has been defined, it is exported from the tool to a file. This file is then imported into Catalyst using the Administration tool, where it is ready to be enacted.

3.2.13 Catalyst Process Enactment Tool

The Process Enactment Tool was created in C++. The purpose of this tool is to enact processes defined by Catalyst users. Process enactment is intended to reduce the effort needed to set up, manage, and track the performance of work on engineering developments using Catalyst. Catalyst enactment supports the

creation of work assignments to people, tracking the inputs and outputs from each activity, and the delegation of work to multiple users.

The Process Enactment Tool is fairly complex. Operational in 1996, it was the first CORBA-based workflow or process execution tool in the world as far as our research can discover. It required about 12 man months to design, implement, and test. At least half of this time was spent on designing the tool, before any code was written. The complete design, in terms of object schemas and algorithms was worked out for all parts of the process enactment before the implementation was started. This turned out to be a winning implementation strategy, because no design changes were required in the tool once it was completed.

Once the tool was operational and we gained some experience enacting processes, we made a few minor changes to the process model to help make the assignment of work products easier to control. A few new features have been added to the tool since its initial creation, but no flaws have been found in the original code or design.

Figure 3.2.13-1 depicts the Process Enactor Screen design. The design is centered around three main "views" – the Desktop View, the Assignment View, and the Activity View.

The user can display one of these views at a time, and can switch views at will. Through the Desktop and Activity views, the user can get to the Perform Activity window. From there, the user enacts defined processes by selecting a task/method, and executing method steps from the Perform Method window. The following paragraphs provide more details on each window and view.

Initially, the user must select a Project Assignment to work within. A Project Assignment associates a person with particular roles on a particular project. If the user has only one Project Assignment, then that assignment will be used. If the user has more than one Project Assignment, a selection window is displayed to allow the user to select the Project Assignment to work under.

The Desktop View displays all of the work products assigned to the user's active Project Assignment. The work product name, as well as its current state are shown. Double-clicking on a work product selects it for use and initiates an activity to use it in. If there is more than one possible activity in which the work product can be used in its current state, a selection window is displayed to allow the user to select the activity to perform. Once an activity is selected, if there are other work products that are also needed for the activity, a selection window is displayed for each needed work product type, allowing the user to select the other work products to use within the activity.

► If > 1 Project Assignment   Main Window

Select Project Assignment

xxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxx

menus:
View ── Desktop (1)
├─ Assignment (2)
└─ Activity (3)

(Active View)

Select Process

(1) Desktop View

Project,User,Roles

WP   WP

WP   (WP)

Select Activity

Select Other WPs

(2) Assignment View

Project,User,Roles

Joe
   WP
Bill
   WP
   WP

(3) Activity View

Project,User,Roles,Process

Select WPs

Perform Activity

Task 1
   Method 1
   Method 2
Task 2
   Method

Perform Method

Method: M

Step 1*

Step 2

Step 3

* modal dialogs for each step;
1 window per step type

Figure 3.2.13-1 Process Enactor Screens

The Assignment View displays all work products assigned to each user for all of the users on the current project (the current project is the project associated with the active Project Assignment). The work product type, its name, and its current state are shown. No actions can be initiated from this view, but it provides a good place for project managers and others to get an overview of the project by seeing where the work products are and what states they are in.

The Activity View displays all the activities within a process selected by the user. This view serves two main purposes. First, a user can use this view to get

refamiliarized with a process definition. All activities, work products, and roles are shown. Second, the Activity View provides a way to begin process enactment by specifying an activity, rather than selecting a work product. This is especially important for activities in which there is no initial work product, like in an activity in which a work product is initially created.
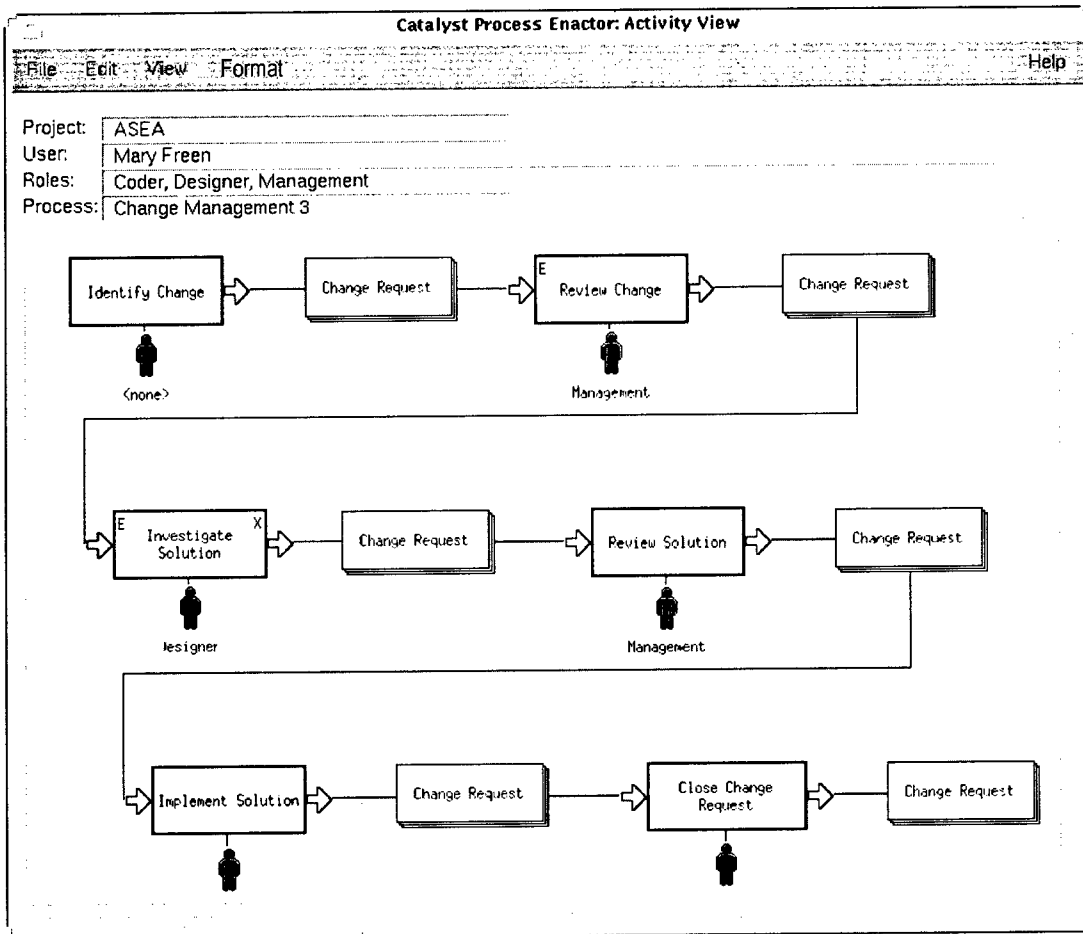
Figure 3.2.13-2 Catalyst Process Enactment Tool

When the user initially brings up the Activity view, if his project assignment involves more than one process, a selection window is displayed that allows the user to select the process to be displayed in the activity window.

The user can initiate an activity by double-clicking on the activity icon in the activity view. If there are work products required by the activity, a selection window is displayed for each work product type, which allows the user to select the work products to use with that activity.
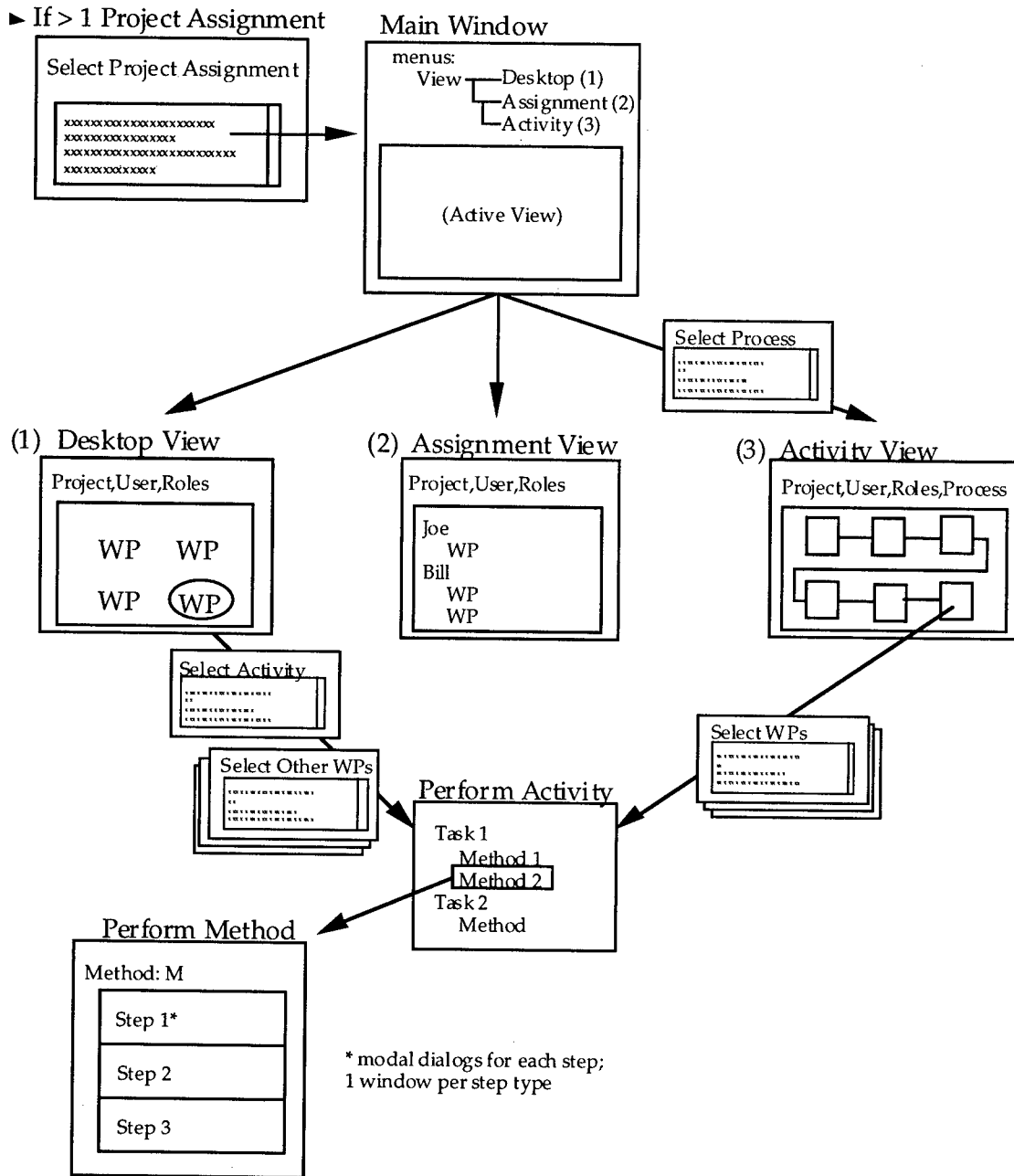
The Perform Activity window displays the tasks and methods within a selected activity. From this window, the user selects a method to perform. Only one method can be performed at a time. Selecting a method to perform brings up the Perform Method window.

The Perform Method window displays the steps within a method. From this window, the user performs each step, thereby enacting the process. Steps should be performed in the sequence displayed. However, the enactor does not enforce order.

There is a step window for each step type defined in the Process Definition Tool. Most of the step windows are simple dialogs or confirmation boxes. All step windows are modal, so that no two steps can be performed at the same time.

The step windows are implemented using functions rather than classes. Since the step windows are modal dialogs, defining them as functions rather than classes results in a simpler programming interface. Rather than creating an object, executing methods to get an answer, and then destroying the object, a single function call will display the window, get the result, perform any necessary processing, and return to the caller when done.

3.2.14 Catalyst Administration Tool

The Catalyst Administration Tool was created in C++. It required about five months of work to design, code and test. The Catalyst Administration Tool provides a graphical interface that makes it easier to set up and maintain enterprises and projects in Catalyst. Catalyst uses a simple enterprise model to keep track of what people work for the enterprise, what projects they work on, and what roles they are allowed to play on each project. This infrastructure supports Catalyst process enactment.
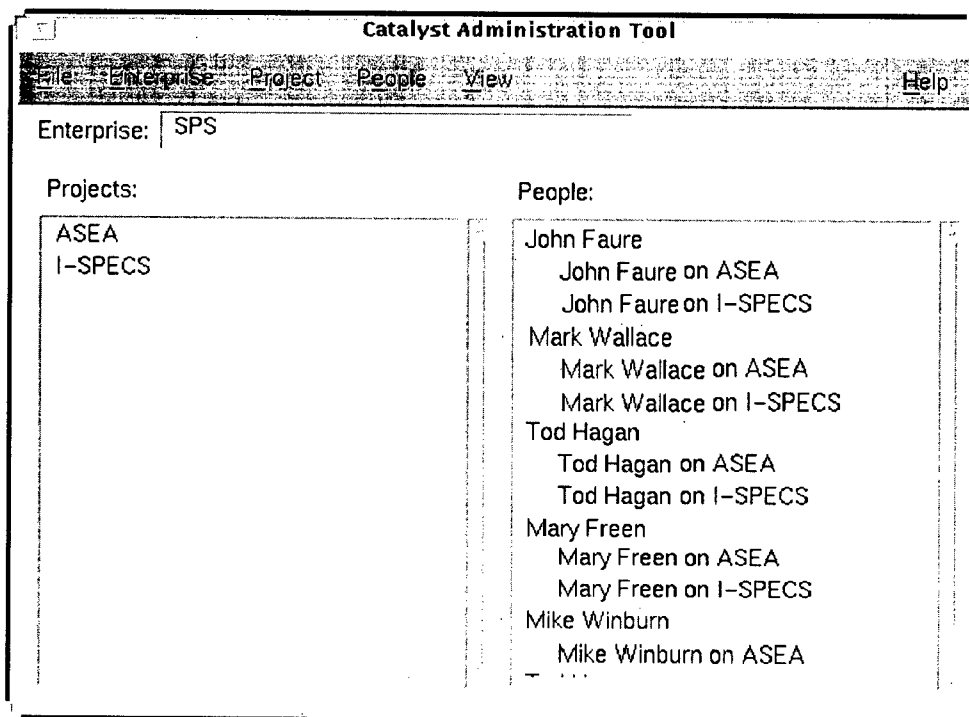

Figure 3.2.14-1 Catalyst Administration Tool

Strictly speaking, the work performed by the Catalyst Administration Tool could also be done using the Browser. Enterprises, projects, and users can be created and manipulated using the Browser. However, it requires a great deal of expertise to create and link together the correct objects and relationships to build a working enterprise and project. The Administration Tool was created to drastically lower the expertise required and speed up the management of projects and users. Creating and using this tool has demonstrated the value of having special purpose tools that hide the lower level Catalyst machinery from the user, and have suggested the future creation of even more sophisticated tools for assisting the user. The Administration Tool itself may be enhanced in the future to make managing other aspects of Catalyst easier.

3.2.15 Catalyst Tcl Scripting

The Catalyst Tcl Scripting capability was created using C++, and required about one month of effort to create. Later in the project, we spent an additional two weeks making several valuable enhancements. The short time required to create this capability is due in large part to its author's (Mark Wallace) extreme familiarity with Tcl technology.

This capability was originally named the Object Community Interchange (OCI) Tool. The OCI Tool was envisioned as custom scripting language for importing and exporting data to and from ASCII files used by engineering tools. Our objective was to provide a low-cost way to rapidly create loose integrations with tools that used simple file formats. These tools included word processors, spreadsheets, graphing programs, HTML, etc. The original OCI Tool was a parser and interpreter for a custom scripting language that was optimized for reading and writing ASCII files. The original OCI Tool was written in C++.

When Mark Wallace joined the Catalyst team, he educated the rest of the team about Tcl. Tcl is a standard scripting environment developed by Dr. John Ousterhout, a U.C. Berkeley professor who later worked for SunSoft. Tcl is a widely used public domain technology that is well documented in several mass market computer books. Tcl has been extended to support windowing, rule-based expert systems, and many other useful capabilities.

It did not take long to decide to abandon the original OCI approach for a Tcl-based solution. This decision has proven to be very beneficial, since many new capabilities have been made available to Catalyst for almost no cost. Tcl is a much more powerful scripting language and provides access to operating system functions. Tcl has recently been ported to Java in the form of Jacl. Tcl is widely supported and easy to learn due to the excellent books available on the subject.

The Tcl scripting environment is used to provide low cost loose integration with simple tools, as we originally envisioned. Several additional Tcl "commands" have been created to simplify writing loose integration scripts. Tcl is also used to perform data creation, manipulation, analysis, annotation, and searching.

Adding a rule-based expert system extension is being considered for future work. This would be useful for creating intelligent Catalyst agents.

3.2.16 Catalyst QFD Tool

The Catalyst Quality Function Deployment (QFD) Tool was created using InSight. About 4 man weeks were required to create the QFD tool. An additional 2 man weeks of effort have gone into enhancing it to provide more integration with Catalyst. Using InSight instead of C++ to create the QFD tool saved approximately five man months of programming effort. The QFD tool was implemented for two reasons. First, the QFD technique is useful for engineering prioritization and information capture, and was of interest to the Air Force and Joint STARS users. Second, it serves as an example of the kind of tool that can be implemented very easily and quickly using InSight.

Figure 3.2.16-1 Catalyst QFD Tool

The Catalyst QFD tool employs a systematic technique for implementing and obtaining quality – Quality Function Deployment. The Catalyst QFD tool was specifically developed for the Catalyst program. The tool adheres to an open architecture to provide a number of different facilities for integrating QFD with other Catalyst tools. This open architecture is necessary to allow QFD requirements to be populated with the actual requirements managed by Catalyst, to export information as Catalyst objects, and to support navigation from objects displayed in the QFD tool to related objects managed by Catalyst. The Catalyst QFD tool's open architecture allows demonstrations of these integrations.

3.3 Catalyst Tool Integration

Considerable benefit can be realized with each new tool integrated into Catalyst. Because Catalyst is a tool integration framework rather than a tool-to-tool integration system, integrating a tool with Catalyst immediately allows it to interact with all other tools integrated into Catalyst.

Before getting to the specific strategies, a few technical issues must be presented to make the discussion more meaningful. A central problem in integrating a tool into Catalyst is handling the mapping of data in the tool to objects and relationships in Catalyst. All data in Catalyst appears as CORBA objects with attributes, which are interconnected by bi-directional relationships. The "appearance" of objects and relationships in Catalyst is defined by the Catalyst Interface Definition Language (IDL) modules. IDL is used in CORBA to define the interfaces to objects. It is these IDL modules that define the uniform interface to all Catalyst data, and the task of the server side tool integrator is to support this client-server interface. The data in a tool being integrated must be mapped into either objects and their attributes or into relationships.

In Catalyst, all objects are represented as CORBA objects. CORBA objects are managed by object server programs that (usually) run in the background and handle requests to their objects made by clients using CORBA "object references." An object reference can be thought of as a pointer to a CORBA object. An unfortunate property of object references is that they are many-to-one with objects. In other words, many different object references can refer to the same CORBA object. This property makes object references impossible to use to uniquely identify an object.

To solve this many-to-one problem, Catalyst assigns each object an identifier that is guaranteed to be unique across all machines, processes, threads, and invocations of Catalyst. This uniqueness is critical to correctly identifying objects. To avoid the performance and reliability problems associated with a centrally managed agent for creating unique identifiers, Catalyst uses the following strategy for generating unique distributed identifiers. Assume a new object is being created by some thread of some process on some machine. Take the machine's Internet Protocol (IP) address, the process's identifier, and the thread's identifier. In addition take the second since January 1, 1970, and the microsecond within that second at which the object was created. This tuple of five integers is

fast and easy to compute (UNIX supplies each piece of information) and guaranteed to be unique. A utility in Catalyst is provided for generating these unique identifiers.

An object's identity within Catalyst is based on the value of this unique identifier. This property, which is sometimes referred to in the object-oriented community as immutable object identity, has many subtle benefits. The CORBA object representing the Catalyst object may be exported to a file, deleted, and then re-imported at a later date. As long as the old unique identifier is assigned to the new re-imported object, the old object has been restored as far as Catalyst is concerned. This allows objects to be moved between machines, and to be deleted and reconstructed on demand. It also allows the same object to be located at another site, so that exact duplicate networks of objects can be maintained and updated at both sites.

It is the responsibility of the tool integration software to assign and maintain the mapping between these Catalyst identifiers and the information needed to find the corresponding data within the integrated tool. How this is done depends on the type and quality of information made available by the tool. Tools that support some form of unique and immutable data identification lead to the most straightforward and high quality integrations. Various integration strategies based on the level of support provided by the tool are discussed below.

3.3.1 Loose Integration Strategy

The loose integration strategy is used when a tool only supports single user access to its data, when the tool has an inadequate or no application programmer interface (API), or when it is important to integrate a tool with minimal effort. This strategy centers around processing a data file generated by the tool and creating objects and relationships in Catalyst that represent the data in the file. Loose integration *is not* the optimal strategy, since it doesn't provide "live" access to the tool's data. However, loose integration can be very useful in certain situations.

Many tools that lack APIs have facilities for writing data to a file according to a format that is documented or can be reverse engineered. In a loose integration, "wrapper" software is written to process this file and then the desired information is extracted and replicated within Catalyst by creating objects and relationships. The information in Catalyst can be kept up to date if the integration wrapper writes a file that maps from the tool data to the Catalyst unique identifiers of the objects created. When changes are made to data in the tool, the integration wrapper can be run again, and it can use the mapping file to find the original Catalyst objects and update them. Bi-directional data exchange can be performed using a loose integration, if required, by making the wrapper extract information from Catalyst and write it in a format that is readable by the integrated tool.

Using a loose integration effectively requires adhering to a policy that reflects how an organization wants to apply the integrated tool. Due to the inherent redundancy of this approach, problems can arise if modifications are made to the data within Catalyst and the tool simultaneously. One option for handling this problem is to consider the tool as the primary method of creating and modifying the data and to consider Catalyst as a way of moving the data into other tools, and as a way of making the data more visible to other team members. In other words, the data should be treated as read-only within Catalyst. One way of dealing with the lack of live access in a loose integration is to transfer data between the integrated tool and Catalyst at certain checkpoints, such as when a consistent version of the data has been created.
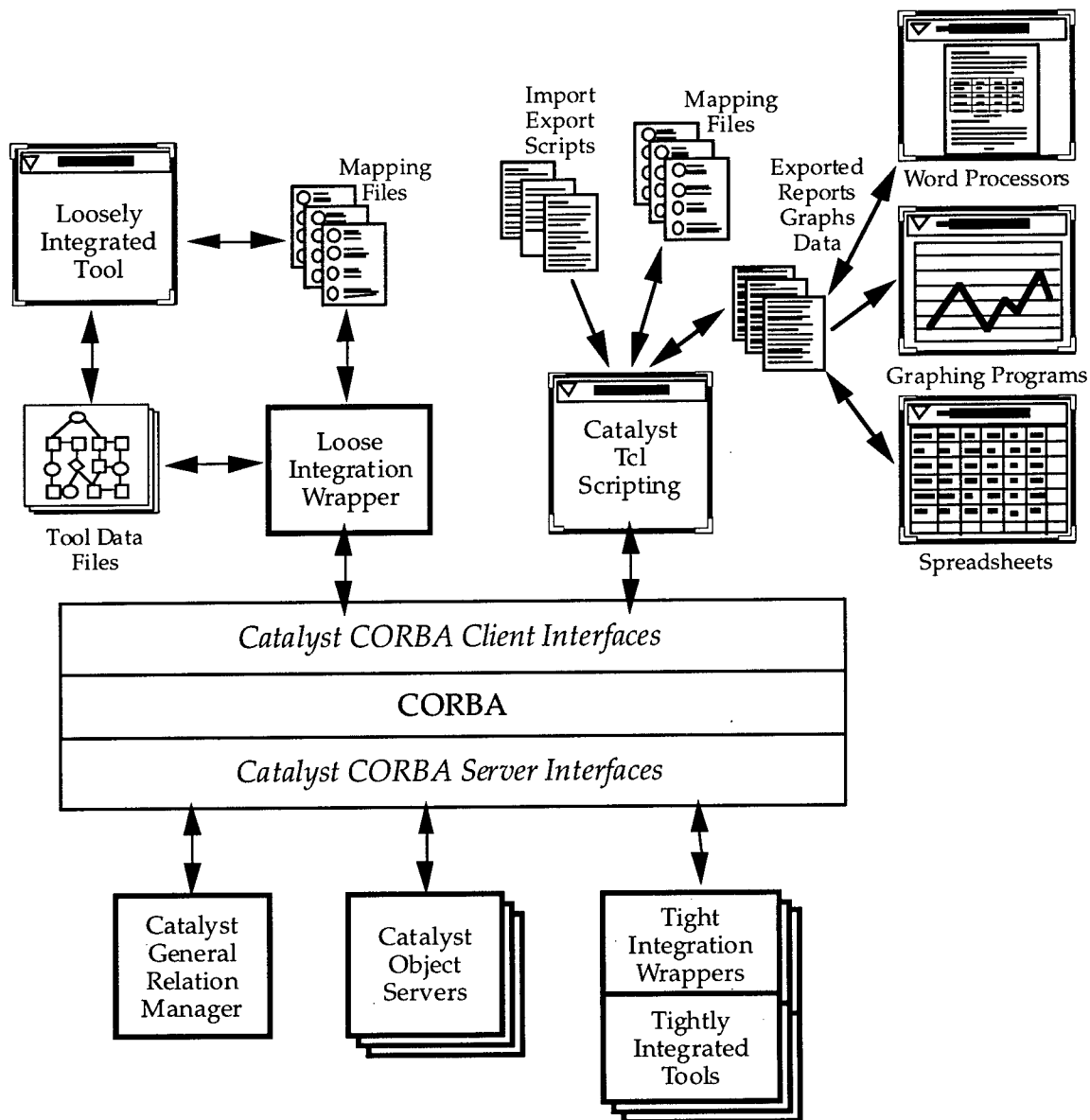

Figure 3.3.1-1. Loose Integration Strategy

A loose integration is especially useful for recovering legacy information (from past efforts) that is valuable reference material but will no longer change. A simple loose integration can be written to extract data from obsolete or unsupported tools and store it in Catalyst in new tools or databases. At this point, the old tool no longer needs to be supported.

A Catalyst Tcl scripting environment was developed to reduce the effort required to perform loose integrations with many tools. Catalyst Tcl scripts can be used to implement bi-directional loose integrations. A Catalyst Tcl import script can be written to read a tool's data file and create Catalyst objects and relationships for the data the file contains. A Catalyst Tcl export script can be written to navigate among Catalyst objects and relationships, outputting a data file and a mapping file as the script directs. Special Tcl procedures were developed and provided with Catalyst to make import, export, and map file management easier when implementing loose integrations.

The mapping file created by an export script can be used by an import script to update Catalyst objects with any changes made to data using the integrated tool. Catalyst Tcl scripting was created to handle bi-directional integration with file-oriented tools, such as word processors, spreadsheets, graphing programs, scheduling programs, etc. Tools that have simple or regular data formats can be integrated into Catalyst by writing the appropriate scripts.

3.3.2 Tight Integration Strategy

The tight integration strategy is used when a multi-user API is supported by the tool. This is a more complex integration strategy, but it provides "live" access to the tool's data and avoids data redundancy. The essence of this strategy is to leave the data within the tool and store information in a location that maps from the tool data to Catalyst objects. A CORBA object server is written, which uses this mapping information to make data in the tool appear as CORBA objects and relationships within Catalyst. The process of creating these "shadow" CORBA objects representing data in the integrated tool is known in Catalyst as "publishing. " The mapping information for the shadow objects can be stored within the object server, or within the tool itself.

To be a good choice for a tight integration, a tool must provide multi-user access to its data. If it does not, the object server integrating the tool can "tie-up" access to the data for long periods of time, making it inaccessible to team members. CORBA object servers are most efficient if they are configured to stay in memory for 30 to 60 minutes after their last access. This prevents constant swapping in and out of memory. For tools that support multi-user access, this requirement is not a problem.

One approach to storing the mappings needed for a tight integration is to put them in the CORBA "ref data." Every object in CORBA is provided with up to 1024 bytes of ref data. This ref data is intended to be used by an object server writer to map from CORBA Object References to the object's actual data. When

an operation on an object comes into the server, the ref data is provided to the server by CORBA as part of the operation. The ref data can be used to store the Catalyst unique identifier and other mapping information needed to identify the object within the tool. This is the most efficient way to construct a tight integration, but it takes the most programming effort and requires detailed knowledge of the workings of CORBA object servers.
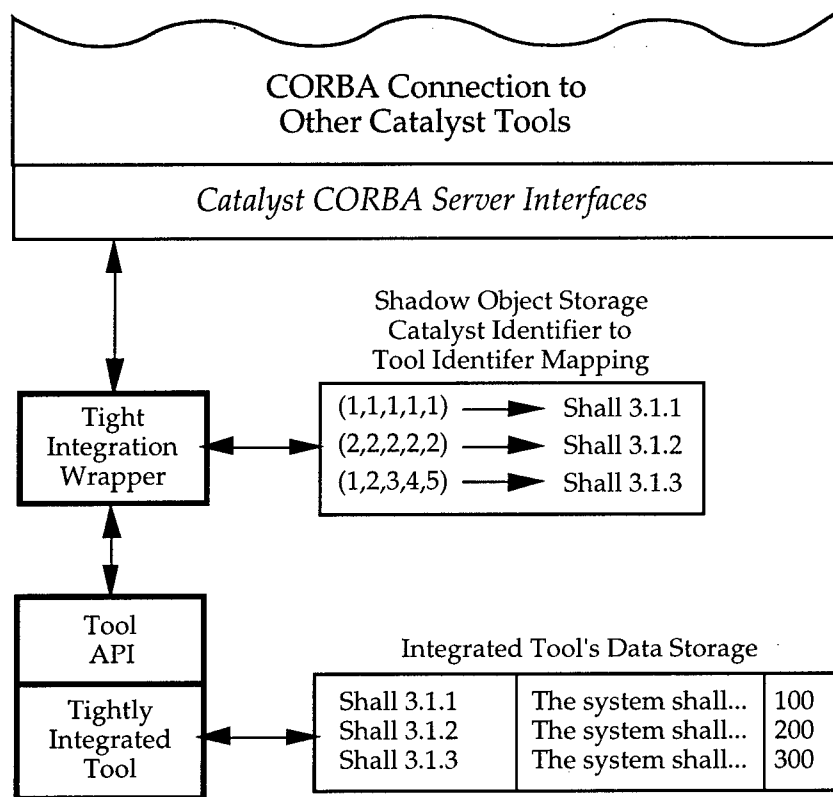
Figure 3.3.2-1. Tight Integration Strategy

A simpler approach is to use a lightweight server such as the one provided by SunSoft's NEO, which includes a "toolkit" for building servers. This toolkit hides much of the complex machinery involved. Using the toolkit makes server writing far simpler and has little impact on performance, since most of the actions performed by the toolkit server would have to be hand coded in the ref data server approach. Using this approach, the mapping information would be stored by the lightweight server, and used to locate objects within the tool.

A final approach is to store most of the mapping information in the tool being integrated. This approach may not be as advantageous in some cases because it is more intrusive to the tool. The other approaches are more transparent as far as the tool itself is concerned. All three approaches must handle translation of the links between integrated tool data into Catalyst relationships.

3.3.3 Client Integration Strategy

The essence of the Client Integration strategy is to use Catalyst as a distributed object base or "virtual repository." This strategy is appropriate for tools that are heavily oriented toward interacting with the user through a user interface. This strategy is also appropriate when a tool only provides *single user* access to its data, but *multi-user* access is desired. Accomplishing a client integration requires replacing the data storage and manipulation facilities of the tool with Catalyst's client side interfaces, in effect using Catalyst to store the tool's data. Because Catalyst is a distributed system, this type of integration allows the tool's data to be shared live with all the other Catalyst tools and users. This is a very powerful type of integration, particularly for tools where collaborative engineering efforts are required.
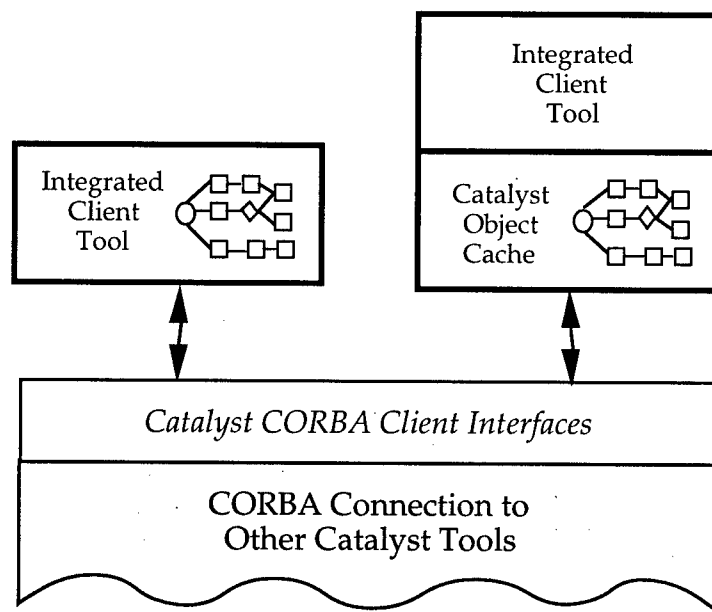
Figure 3.3.3-1. Client Integration Strategy

Catalyst provides two levels of client-side interfaces. The lower level requires sending Catalyst CORBA messages - talking directly to tightly integrated tools and Catalyst object servers. This level is straightforward to program, but requires the client to store objects and relationships. A higher level interface is available in the form of a special purpose object cache created for Catalyst. The object cache buffers object class and relation (model) information, object and relationship instances, and provides high level policies for handling object locking and saving. It provides considerable performance advantages by buffering a working set of objects and relationships in memory. Which level to use depends on whether the object cache capabilities will benefit the tool being integrated. The effort required for this integration strategy depends on the degree of difference between the integrated tool's current API and Catalyst's client-side interfaces.

Both CORBA and Catalyst use navigation-oriented searching, meaning that relationships are used to travel from object to object through the data. Tools that

are based on performing broad searches of all data (rather than on navigating by following relationships) may require more effort to integrate. A new specification for performing broad searches is being drafted for inclusion in CORBA. Search-oriented tools will become easier to integrate when this specification is supported by CORBA vendors.

The client tool's data may be stored in other tightly integrated tools or in Catalyst object servers. For example, if a relational or other database has been tightly integrated, the tool's data could be stored within it. New Catalyst object servers can also be created to store the tool's data. Catalyst contains a toolkit that can create a new object server from a description of the object in a matter of minutes. This toolkit is available to create servers for objects that have no existing server in Catalyst.

3.3.4 Catalyst ORACLE Integration

The Catalyst ORACLE integration was created in C++. It required about one man month to create. An additional two man-weeks have been spent on enhancing it to make it more reliable. The ORACLE integration currently provides read-only access to the database. Write access could be added in an additional man month. The ORACLE integration has proven to be very reliable and effective.

Making ORACLE data visible in Catalyst is a two-part process. Making data from an integrated tool visible in Catalyst in called publishing. The first part requires using the Browser to create an object of a class that maps to ORACLE. This process is easier if the object is from a "top level" class in the ORACLE schema, such as a system or document. The new object is opened in the Browser, so that its attribute may be modified. The ORACLE key of the data in ORACLE to be made visible is entered into the *key* field of the Catalyst object that was created. The Catalyst object is then saved. If the object is refreshed, the key value will be used by the ORACLE integration to retrieve the object's data from ORACLE.

The second part of the process requires following relationships from the new object. The ORACLE integration will determine what objects are related to the first one. If these objects have not been published in Catalyst before, it will automatically publish them. This is known as autopublishing. The related objects will appear in the Browser. The user can then follow relationships from these objects, which will result in more objects being published.

Autopublishing makes data in ORACLE visible in Catalyst on demand. This may occur when a user is using the Browser, or when a Tcl script is running and tries to access them, or when another tool such as the Impact Analysis Tool is running and searching for impacts. Autopublishing is a very transparent technique once the user has manually published the first object.

3.3.5 Catalyst RDD-100 Integration

The Catalyst RDD-100 Integration was created in C++. It required about one week to create. RDD-100 was loosely integrated using a custom RDD-100 RDT file parser written in C++. This approach was chosen due to the complex syntax used in RDT files. It might have been written in Tcl if much of the parser code had not been available for reuse from other projects.

RDD-100 is an object-oriented tool that represents data using objects, attributes, and relationships. The RDD-100 Integration reads an RDD-100 RDT file and replicates the objects and relationships it contains in Catalyst. A mapping file that describes how to map RDD-100 object, attribute, and relationship names to Catalyst names. The mapping process allows multiple RDD-100 classes to be mapped to a single Catalyst class, and multiple RDD-100 relations to be mapped to a single Catalyst relation. This help simplify the Catalyst representation of the RDD data and helps make RDD data fit better within an existing schema.

The Catalyst Object Cache was enhanced to support mapping integrated tool schemas to the Catalyst schema. These enhancements can be used to make it easier to loosely integrate future tools.

The RDD-100 Integration has been successfully applied to import real world RDD-100 files into Catalyst. Future enhancements may be required to process more of the specialized RDD-100 features, such as F-nets and T-nets, which represent function interactions and time-based events.

3.4 ASEA Deliverables

This section lists the items developed under the original ASEA contract. Many of these items contain multiple volumes and many items were delivered in multiple versions. A CD-ROM was delivered that contained electronic versions of all documents, graphics and presentations developed under ASEA. A second CD-ROM was delivered that contained all the source code and example data sets.

A001 R&D Status Reports (52 created)

A002 Contract Funds Status Reports (52 created)

A003 Conference Agenda (12 created)

A004 Conference Minutes (12 created)

A005 Presentation Material (12 created)

A006 Catalyst Software Development Plan (SDP)

A007 CSCI-01 Process Manager SRS

A007 CSCI-02 Browsers and Editors SRS

A007 CSCI-03 Meta-Modeling Toolkit SRS

A007 CSCI-04 Modeling and Analysis Toolset SRS

A007 CSCI-05 Documentation Toolset SRS

A007 CSCI-06 Catalyst Administrator SRS

A007 CSCI-07 Object Infrastructure SRS

A007 CSCI-08 Common Object Services SRS

A007 CSCI-09 Framework IDL SRS

A007 CSCI-10 Client Toolkit SRS

A008 Catalyst Interface Requirements Specification (IRS)

A009 CSCI-01 Process Manager SDD

A009 CSCI-02 Browsers and Editors SDD

A009 CSCI-03 Meta-Modeling Toolkit SDD

A009 CSCI-04 Modeling and Analysis Toolset SDD

A009 CSCI-05 Documentation Toolset SDD

A009 CSCI-06 Catalyst Administrator SDD

A009 CSCI-07 Object Infrastructure SDD

A009 CSCI-08 Common Object Services SDD

A009 CSCI-09 Framework IDL SDD

A009 CSCI-10 Client Toolkit SDD

A010 Catalyst Software Test Plan (STP)

A011 Catalyst Interface Design Document (IDD)

A012 Catalyst Computer Resources Integrated Support Document (CRISD)

A013 Catalyst Software Test Description (STD)

A014 Catalyst Software Test Report (STR)

A015 Catalyst Version Description Document (VDD)

A016 Catalyst Software Product Specification (SPS)

A017 Document Changes

A018 COTS Manuals

A019 Catalyst Overview Software Users Manual (SUM)

A019 Catalyst Support Tool Software Users Manual (SUM)

A019 Catalyst Administration Tool Software Users Manual (SUM)

A019 Catalyst Browser Tool Software Users Manual (SUM)

A019 Catalyst Process Definition Tool Software Users Manual (SUM)

A019 Catalyst Process Enactor Tool Software Users Manual (SUM)

A019 Catalyst Impact Analysis Tool Software Users Manual (SUM)

A019 Catalyst QFD Tool Software Users Manual (SUM)

A019 Catalyst OCI Tool Software Users Manual (SUM)

A019 Catalyst Tool Integration Guide

A019 Catalyst Administration Tool Users Tutorial

A019 Catalyst Browser Tool Users Tutorial

A019 Catalyst Process Definition Tool Users Tutorial

A019 Catalyst Process Enactor Tool Users Tutorial

A019 Catalyst Impact Analysis Tool Users Tutorial

A019 Catalyst QFD Tool Users Tutorial

A020 Training Course/Control Document

A021 Final Technical Report

3.4.1 Trusted Ontos Prototype ECP Deliverables

This section lists the items that were developed under the Trusted Ontos Prototype ECP:

A022+ SSDD

A023+ R&D Test & Acceptance Plan

Philosophy of Protection Document

Demonstration Users Manual

Install Instructions

Demonstration Software

3.4.2 Integrated Security Analysis Tools ECP Deliverables

This section lists the items that were developed under the Integrated Security Analysis Tools ECP:

A021R Final Report

A024+ Interim Summary - Integrated Toolset Detailed Design

A025+ Interim Summary - Extended Security Coverage

A026+ Interim Summary - Penelope & Romulus Integration

A027+ Interim Summary - Security Analysis Methods & Models, Security Toolset & Catalyst Integration Design

A028+ Interim Summary - Security Requirements Analysis & Multi-policy Tradeoffs (not delivered due to NSA funding shortfall)

A029+ Interim Summary - Testing

A030+ Interim Summary - Prototype Implementation

Computer Software

3.4.3 Security Study/Micro Process ECP Deliverables

This section lists the items that were developed under the Security Study/Micro Process ECP:

A031+ Technical Report - Catalyst Security Study

A032+ Technical Report - Catalyst Micro-Process Study

A033+ SRS - Micro-Process Enactment

A034+ Update to existing SSS

A035+ Update to existing SSDD

A036+ Update to existing OCD

3.4.4 Joint STARS Demonstration ECP Deliverables

This section lists the items that were delivered under the Joint STARS Demonstration ECP:

Catalyst System Traceability Document

Joint STARS Demonstration

Joint STARS Demonstration Data Sets

4 Catalyst Usage Scenarios

This section describes some of the many possible application scenarios for Catalyst. Many other uses exist because Catalyst provides general purpose integration and data processing capabilities, which may be combined in many ways for many purposes. Some of the following scenarios were suggested by outside Catalyst users who were applying Catalyst to the problems they needed to solve.

4.1 Browsing Heterogeneous Databases and Tools

One of the most basic uses of Catalyst is to conveniently browse data which is stored in heterogeneous databases and tools which are integrated with Catalyst. Catalyst solves the problems of accessing distributed data which may reside on heterogeneous platforms and which the data may be stored in many different formats. Using the Catalyst Browser, a user can view objects and follow relationships to other objects by simply clicking with the mouse. The user does not need to know how to use any of the integrated tools which store the data. The user does not even need to know where the data is physically stored. The user may follow relationships which connect data stored in different tools. This allows users to follow the traceability of data across many tools and engineering phases.

The Catalyst Browser is very simple and can generally be learned in a few hours. Browsing gives the user access to the entire set of integrated data without any training in using the integrated tools. When the Java version of the Browser is available, it will give users on Windows95, WindowsNT, Macintosh, and other platforms access to all Catalyst data. Browsing data is an important part of many of the more complex scenarios.

4.2 Storing Links Between Data in Multiple Tools

Catalyst may be used to create an integrated engineering information base which spans many tools and engineering activities. This allows the traceabilities and dependencies between data stored in different tools to be stored and used. The background information supporting design and prioritization decisions can be captured and linked directly to the relevant data objects. Because Catalyst is a multi-user distributed tool this information becomes available to all personnel working on the project.

The ability to link information among many tools is illustrated using a scenario involving the Catalyst QFD Tool. The information supporting the various values captured in the QFD matrix are linked directly to the values using Catalyst. This allows other users to understand the rationale for why each of the particular values were chosen.

The QFD matrix illustrated in Figure 4.2-1 captures the applicability of Global Command and Control System (GCCS) Core Operating Environment (COE)

functions (the Hows) to Air Force mission needs (the Whats). Each interaction element in the main QFD matrix is linked to a Catalyst Annotation object which gives the rationale for the interaction value in the matrix. In the scenario, the rationale for the interaction between Intelligence (mission need) and Security (function) is described.

Mission Area Plans (MAPs) are used by the DoD to define short and long term plans for some mission area. Mission needs identified by the mission area are listed in QFD matrix under the "WHATs" column (e.g., crisis planning, force deployment, intelligence, logistics, etc). These can be cross-referenced with the GCCS COE functions. These correlations can be used, for example, to determine which functions are most critical to implement based on the needs of the mission. The computed relative and absolute importance weights found in the bottom two rows of the QFD diagram can be used to help make this determination. Catalyst can be used to link a mission need (a.k.a., requirement) to a Catalyst Requirement object within the COE, a link which may be used later during impact analysis and requirements traceability reports.

The "roof" or "hat" part of the matrix allows you to draw correlations between the GCCS COE functions, and to link in Catalyst Annotation objects that describe the rationale for selecting either high-medium-low correlations within the roof. Intuitively, there would be a stronger dependency between Messaging and Network Management than there would be between Messaging and Office Automation.

The QFD matrix is linked to a Catalyst System object. Like other Catalyst objects linked to the matrix, it is possible to open the object using the Catalyst Browser. The user can traverse the hierarchy of components which comprise the system, this is illustrated by "System Architecture", in the top right of the figure.

The "Business Case Analyses" document (for improving organizational efficiency, quality of service, competitiveness, and customer satisfaction) is linked to the Customer Rating portion of the QFD Matrix. The Business Cases Analyses document can be improved by analyzing the data provided in the "Customer Rating" column of the diagram.
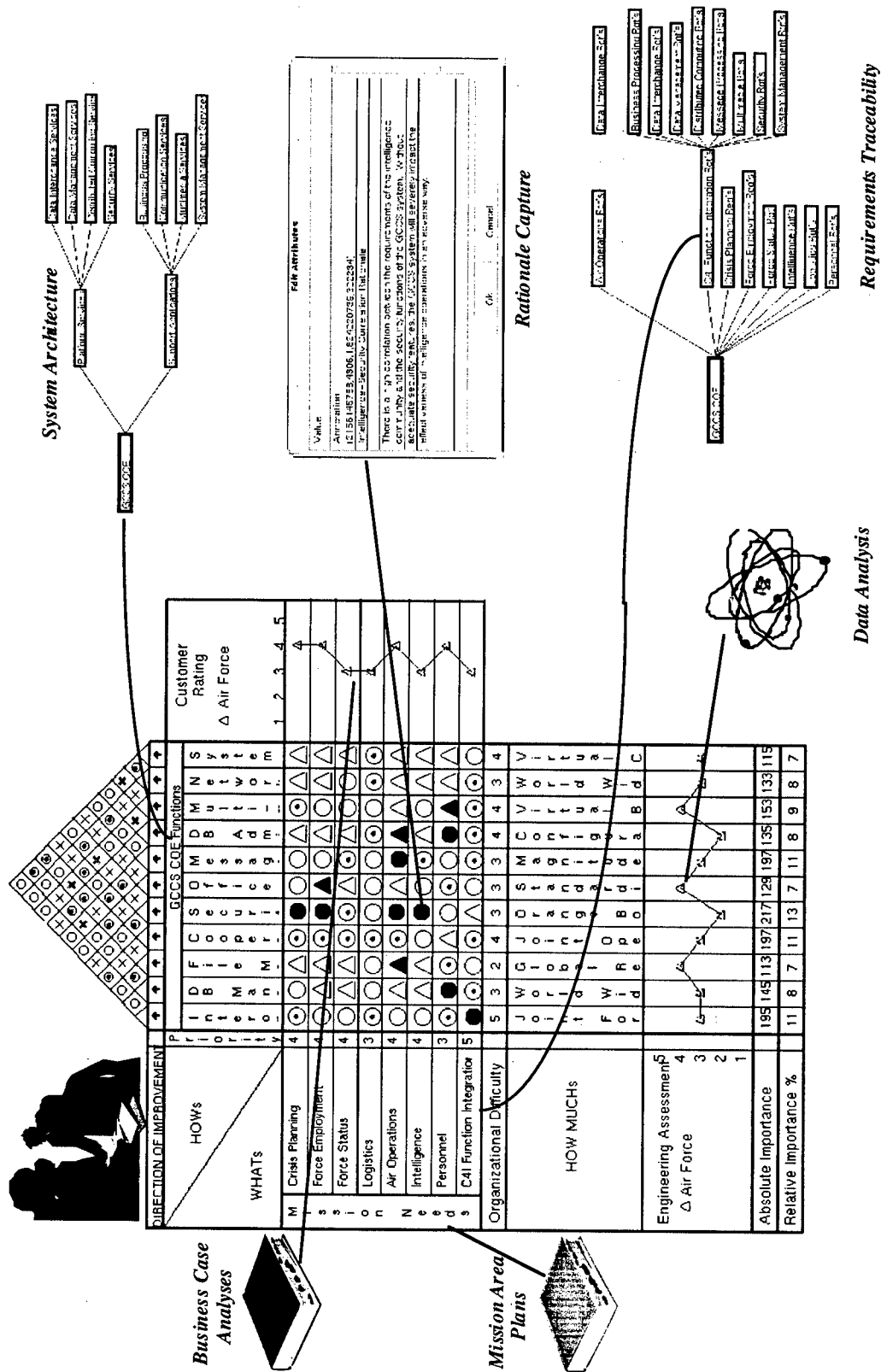
Figure 4.2-1   Storing Links Between Data in Multiple Tools

This scenario illustrates how rationale and other information can be captured and linked together so that it may be retrieved and used later.

4.3 Supporting Collaboration Using Locking

Catalyst supports groups of users in collaborating on creating complex designs, models, simulations, documents or other information. For example, if a group of users are cooperating in creating a document they could assign responsibility for one or more of the sections of the document to each of the group members. Each group member would lock their sections by locking the objects and relationships which make up the section. This would allow the other group members to view all the document sections, see which group member was responsible for which section, but will prevent any other Catalyst user from altering any group member's sections.

Catalyst locks remain in place until explicitly unlocked, allowing the group members to work on their sections for as long as it takes to complete them. Catalyst locks contain the name of the user owning the lock. This allows other users to see who has an object or relationship locked. The Catalyst Browser and other Catalyst tools provide assistance in automatically locking and unlocking object networks.

Catalyst allows locked objects to participate in new relationships. This policy allows a group member to review and comment on document sections created by the others. The reviewer can create Annotation objects containing their comments and attach them to the relevant parts of the document. This allows feedback and comments to be collected without allowing the original data to be modified. The Annotations can be linked together and to other objects, forming an integrated network of rationale and comments.

The document sections can be unlocked when they are completed or when the entire document is completed. The entire document could then be locked by the manager or engineer responsible for it. This would prevent further changes to it.

Although this scenario was described in terms of creating a document, it could be applied to a group of users collaborating on creating any object network in any domain.

4.4 Capturing and Using Rationale

One of the most powerful applications of the integration made possible by Catalyst is the ability to capture, link together and use information which formerly had no convenient storage and retrieval mechanism. Any Catalyst object, regardless of where and how it is stored, can be linked to other Catalyst objects using relationships. Catalyst provides the Annotation object class to capture notes, graphics, audio and other types of data. Relations are provided which allow Annotation objects to be linked to any other Catalyst object, as well as to each other. A user can also define their own object classes that model more

specific types of rationale, the Annotation class is provided simply as an example of what is possible.

Rationale information such as design studies, designer opinions, performance tests, customer direction, literature searches, etc. can be linked directly to the design components, requirements, tests, etc. to which they apply. This rationale can be discovered later by other users looking at the design components, etc.

A very powerful type of rationale is the existence of relationships between two items. For example, a design component may rely on an assumption about the implementation of a second one. The system will cease to function correctly if the second design component is changed without also changing the first one. A relationship could be created between the two design components using Catalyst. The presence of this relationship would alert engineers to the dependency between them. Using Catalyst, this type of relationship rationale can be captured and stored. Without Catalyst, if two items are stored in different tools then most likely this relationship rationale will be lost because there is no where to store it.

4.5 Impact Analysis

A scenario for performing an impact analysis using Catalyst is described in Section 3.2.11.

4.6 Process Enactment

The Catalyst process definition and enactment capabilities can be applied to many useful scenarios. Process enactment was once viewed in the engineering community as being beneficial by preventing the user from doing things they should not. Trying to control what the user did has not proven to be very easy or very useful. The real benefits of process enactment are realized by assisting the user in correctly following a process. A lot of manual tasks can be eliminated through process enactment. Similarly, a lot of useful information can be presented at the users fingertips at the time when they need it. The real benefit of process enactment may be in having knowledgeable users capture their expertise in a usable way by defining processes. Their expertise is then made available to less experienced users through enactment of the process.

Capturing expert knowledge is illustrated by the following scenario, which was implemented in the HLA Integrated Development Environment (HIDE) being developed by SAIC, Lockheed Martin and Modus Operandi for STRICOM. A Federation Development Process (FEDEP) which covers the complete HLA simulation development lifecycle was developed by Dr. Kent Bimson of SAIC. This process encapsulates knowledge about what activities to perform, what order to perform them in, what tools to use at each step, and why each step is being performed. During certain steps, the process automatically launches an internet browser and opens it to the appropriate government web site containing policy recommendations and standards for HLA development.

The benefit of this process enactment scenario is that it leads a novice user through the creation of an HLA simulation, it presents them with the information they need at each step, and launches the tools needed. The process reduces the expertise needed and automates some of the manual tasks required to develop the simulation.

4.7 Migrating Data Between Tools

Catalyst can be used to migrate data from one integrated tool to another. This capability has many useful application scenarios. For example, Catalyst can be used to migrate data in a legacy tool into a newer tool which provides more features or a lower cost of ownership. To migrate the data, both the tools must be integrated with Catalyst. These can be either tight or loose integrations, as long as the data writing capabilities were implemented in the integration of the new tool. A Tcl script is written to traverse the data in the Legacy tool and copy it to the new tool. This copying process can also perform any data transformations needed to make the data compatible with the new tool. A simple migration is illustrated in Figure 4.7-1.
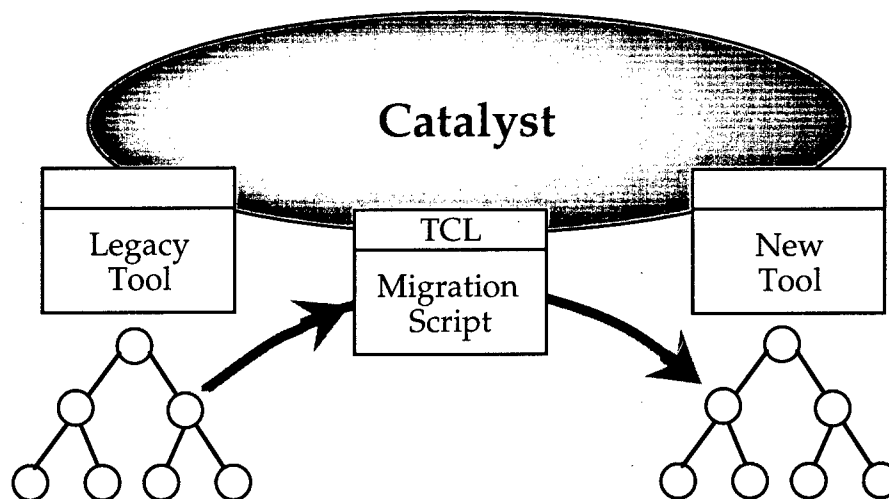
Figure 4.7-1 Simple Tool to Tool Data Migration

The time required to perform such migrations is dependent on the time required to integrate the tools and the complexity of the data transformations which need to be made in the Tcl migration script. Simple migration scripts for tools which are already integrated can be written in a matter of hours. More complex migration scripts may require days or weeks to be created. Because the full power of a programming language is available in TCL, almost any transformation can be made as long as the necessary information exists in the Legacy tool.

A more complex migration scenario illustrated in Figure 4.7-2 leverages Catalyst to reduce the effort required to set up simulation runs during a development

effort. The basis for the scenario is that an engineering design tool is used to store the baseline design for an aircraft which is under development. The baseline design is only modified using the design tool. When changes are made which require simulation, Catalyst scripts are used to migrate the design information to several integrated simulation and analysis tools. These scripts pull information about the aircraft from the baseline design and transform it as needed for each simulation tool. The simulations are then run and analyzed.
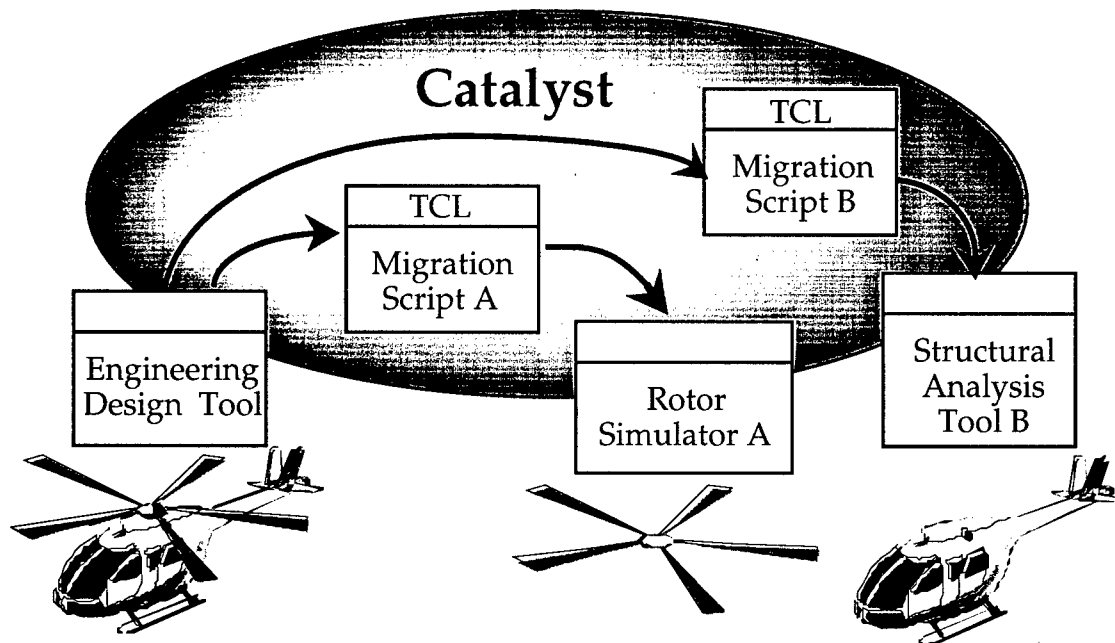
Figure 4.7-2   Migrating Design Data into Simulators

This scenario provides two immediate benefits. The first is that the effort required to set up simulations is greatly reduced since the process is now automated. This allows more simulations to be performed in the same amount of engineering time, allowing the designers to consider more alternatives and create a more refined and effective aricraft. The second benefit is that all the simulations are driven from the same design reducing the chance for errors. If several simulation tools are used, as is common, there is always a danger that there are descrepancies in the input data used to feed the simulators causing the production aircraft to not perform as expected.

Although not shown in the figure, the simulation results can be captured in Catalyst and stored to create a design history of the aircraft. The simulation results for various alternatives can be referred to in the future to recover the rationale for design choices. This scenario can be applied any time one or more tools needs to be driven from data which is maintained in another tool.

4.8   Backing Up Multiple Tool Baselines

The Catalyst Support Tool supports writing the contents of an object community to an ASCII Catalyst export file. This file contains the complete contents of the

object community. The export file can be imported back into Catalyst later to reload the object community. This provides a backup and restore capability for Catalyst. The object community can contain objects and relationships which are stored across many integrated tools, thus allowing baselines of information spanning many tools to be backed up and restored as a single unit.

The export file can be sent to other organizations and imported into their Catalyst installations. This allows organizations to deliver information to each other electronically, and to collaborate electronically on the development of complex information networks.

4.9 Contractor/Subcontractor Collaboration and System of Systems Development

Catalyst can be used to enable the development of an integrated engineering information base when multiple contractors are collaborating on building one system or on integrating a system of systems. This scenario applies equally to both activities.
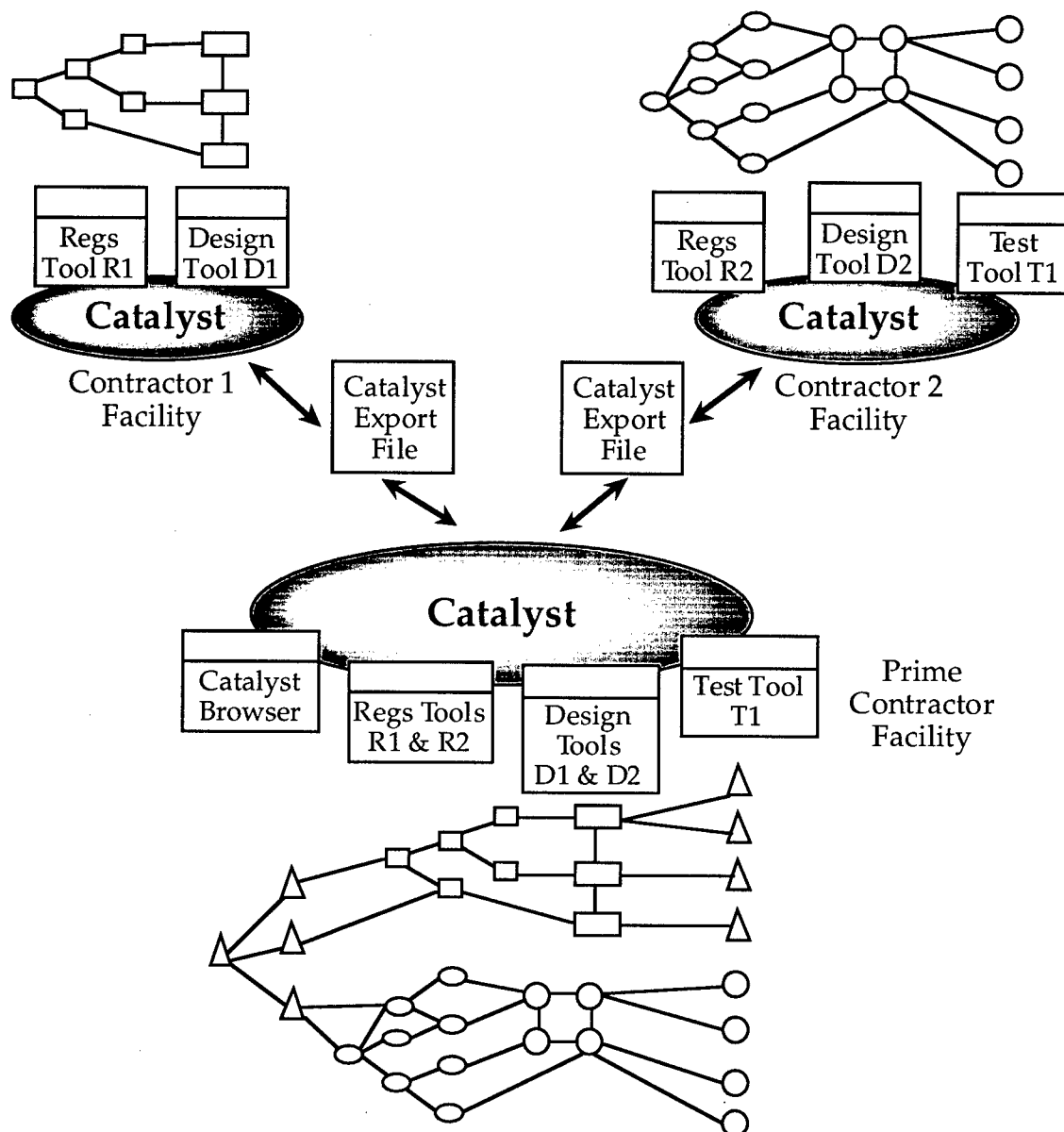
Figure 4.9-1 Contractor/Subcontrator Collaboration

A scenario involving a prime contractor and two subcontractors is illustrated in Figure 4.9-1. Each of the subcontractors in the scenario would be responsible for developing the requirements and design for one subsystem of the overall system. Further, one subcontractor would be responsible for developing the tests for their subsystem. The prime contractor would be responsible for integrating the subsystems and developing the tests for the other subsystem.

Each subcontractor would develop their subsystems using tools integrated with Catalyst. At the appropriate points in the development cycle, the subcontractors would export their data to Catalyst export files and ship the files to the prime contractor. The prime contractor would import the subcontractor data and link it

together into an overall information network which encompased the entire system.

The prime contractor could then browse and perform design reviews, traceability completeness, test coverage, and other analyses on the overall information network. The overall information network could be exported by the prime contractor and delivered to the subcontractors, if appropriate, and to the procurement, oversight, logistics and end user agencies for the program as required. Having an integrated information base spanning the entire effort would allow inconsistencies, errors, and missing information to be identified earlier in the effort, saving downstream costs. It would facilitate communication between the system stakeholders resulting in a higher quality system at a lower overall cost.

The subcontractors would periodically deliver updates to their information. This could be done at review points in the lifecycle such as PDR and CDR. Catalyst allows an existing object community to be updated using a newer version of the export file. This allows any annotations and links created with the earlier version to be preserved when the newer version is loaded and merged. Sophisticiated facilities for merging changes were designed for implementation under the GEODE SBIR effort sponsored by the Naval Air Warfare Center. Unfortunately, the implementation phase of the GEODE SBIR was not funded. Catalyst's current merging facilities are adequate for many uses.

4.10 Storing Tool Data When the Tool is not Available

Often organizations which have different toolsets must collaborate with each other. Catalyst provides a general method for addressing this problem, which is illustrated in Figure 4.10-1. A contractor is using Catalyst and a number of (potentially expensive or proprietary) integrated tools to develop a system. The procurement, oversight, logistics, and/or end users do not own these tools and can not afford to purchase them. Catalyst can still be used to allow electronic delivery and exploitation of the information developed by the contractor using these tools.

The data stored in the integrated tools used by the contractor is modeled at the contractor site in Catalyst using a set of object classes and relations. The object classes and relations are mapped to the integrated tools by integration "wrappers" as described in Section 3.3. Catalyst Object Servers can be created at the other organization's sites for these classes. The Catalyst Object Servers will store the tool data without using the integrated tools. Catalyst Object Servers rely on the ObjectStore database to store their data as described in Section 3.2.7. The same object and relation model can be created at the other organizations using Catalyst Object Servers, allowing them to store the same data.
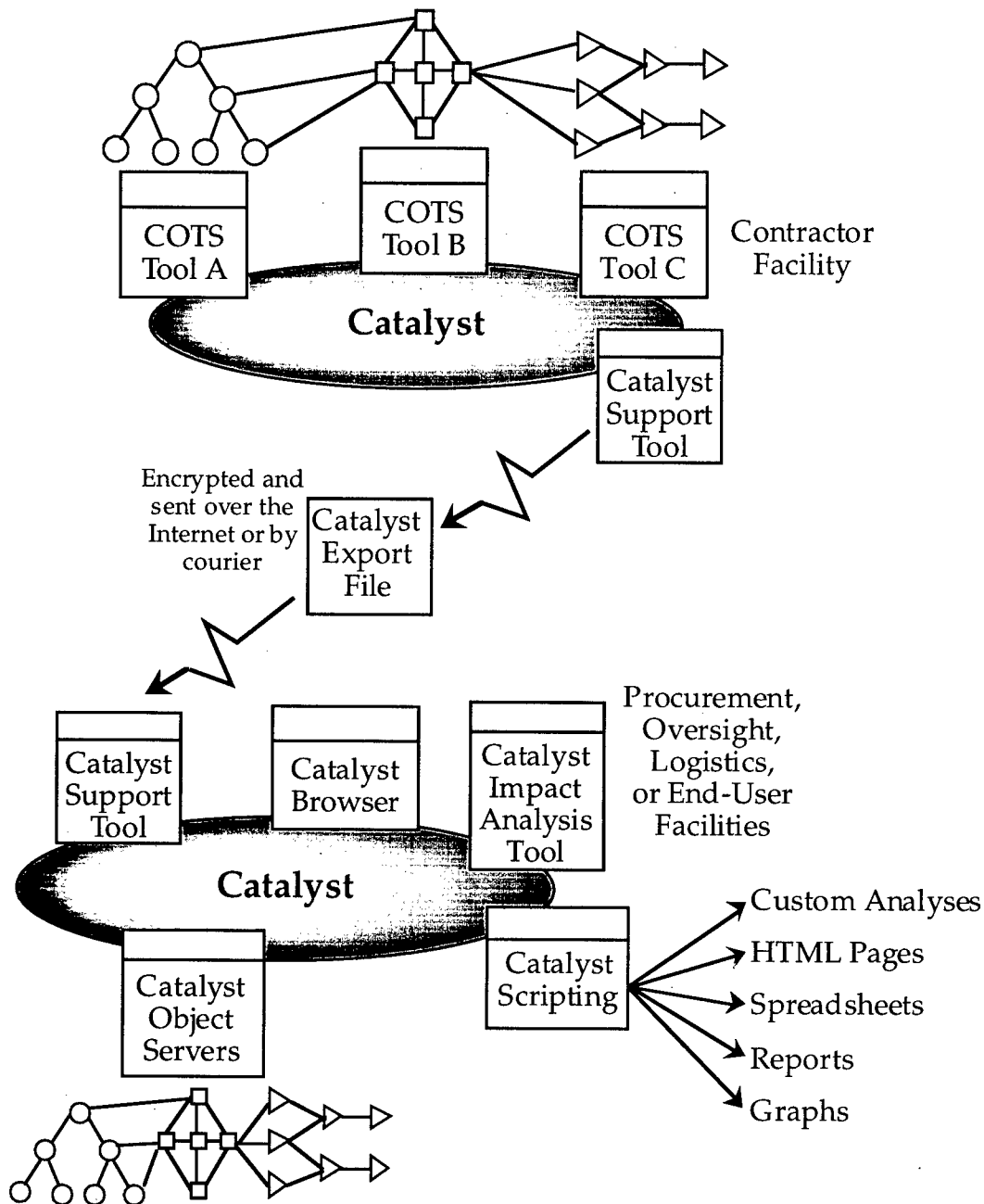
Figure 4.10-1 Storing Tool Data when the Tool is not Available

The data developed by the contractor is written to an Catalyst export file using the Catalyst Support Tool. This is an ASCII file which contains the full contents of an object and relationship network. The data file is moved to the other organizations in a manner consistent with its sensitivity. This may be through encrypted Internet transfer or by physical courier. The data set is imported into Catalyst at the other organization's site and the data is automatically stored in the Catalyst Object Servers. Catalyst will automatically use the correct Catalyst

Object Server as long as it has the same class name as the original tool integration wrapper server.

If the data set is very large it can be exported in pieces in mulitple files. The data set will be reassembled automatically by Catalyst as each file is imported.

This scenario gives the other organizations the ability to browse the data, perform impact analyses on it, and generate reports and graphs from it. Functionality which is specific to the tools used by the contractor will not be available, only access to the data created by the tools is provided.

4.11 Linking to WWW Pages

Catalyst supports a mechanism for linking objects to WWW pages and to other externally stored data items such as graphics, audio and video files. WWW pages may also be linked to Catalyst objects.

A scenario for using these capabilities is linking requirements objects stored in a database to the requirements document in which they were defined. The objects in the database are traversed using a Catalyst Tcl script. An Annotation object is created for each requirement object that links it to an HTML tag on the section of the document where the requirement was defined. The HTML tags are based on the number of the document section where the requirement was defined. This was chosen for the tag because it was easy to find in the HTML and was available in the requirements database. A second Tcl script was used to parse the HTML version of the requirements document and add the section number tags.

The Annotation objects allow the user to automatically launch an internet browser and bring up the requirements document at the section where the requirement was defined. Getting access to the full text of the requirements document allows a user interested in a requirement to read any background information and view any diagrams pertaining to the requirement. It also allows the user to look at other requirements defined before and after the one of interest. Since the requirements are defined in the document in a logical sequence, these other requirements provide valuable context information. This scenario was implemented on the Joint STARS program.

This scenario uses a feature of the Catalyst Browser. The Browser looks for objects which contain both "file_path" and "file_type" attributes. When the user views an object that contains both of these attributes the Browser adds a *display* button to the view. If the user presses this button the Browser will launch the tool specified by the "file_type" using the "file_path" as an argument. In this case, "netscape"" was used for the "file_type" and the section number tag was used as the "file_path." This feature allows links to be created to many types of external entities such as WWW pages, graphics, audio, word processors, etc.

4.12 Generating WWW Pages

Catalyst can also generate WWW pages using Tcl scripts. Any of the information stored in Catalyst may be used in generating these WWW pages. A scenario for generating WWW pages is the creation of an impact analysis report. A script has been implemented which takes an impact analysis and generates an HTML file which summarizes the results of the analysis. This script can be customized to generate as much or as little detail about the analysis as desired.

Generating WWW pages from Catalyst information can be beneficial for providing very convenient access to selected information for both internal and external use. Using scripts to generate the pages automates the process allowing them to be kept up to date with a minimal amount of effort. Arbitrarily complex pages can be created which may contain graphs, tables, icons, html links and Java applets. If necessary, the generation script can be run on demand so the information is current each time a user views it.

4.13 Providing a Common Interchange Platform

Catalyst can be used as a common interchange platform for collections of domain tools which operate on the same or related data. In this scenario, a collection of related tools would be integrated with Catalyst. Each tool could be integrated either loosely or tightly. Tools which operate on the same or very similar data could share one object model. Tools which operate on different but related data could use Tcl scripts to translate their data to and from other models.

This scenario would allow data to be exchanged among multiple tools. It would allow users with different tools to share data with each other. It would also allow one user to use multiple tools to operate on one set of data. A user could load the data into the tool most appropriate for the task they were performing. This would allow the user to take advantage of the best features of each tool without being restricted to any one tool.

This scenario is being implemented in the HLA Development Environment (HIDE) being developed by SAIC, Lockheed Martin and Modus Operandi for STRICOM. A collection of HLA model development tools are being integrated into Catalyst to permit exchange of data among them. A custom tool for developing and manipulating HLA data is also being developed to help hide the Catalyst implementation details from the end user, who is only concerned with HLA model development.

4.14 Building Object Networks from Documents

In this scenario, a document is parsed and the information it contains is captured in Catalyst as a structured network of objects and relationships. This scenario was applied to help automate the development of Catalyst's own requirements. Once enough of Catalyst was operational, Catalyst was used to create, store, and analyze its own requirements, tests, and their traceabilities.

The inputs used were the System Segment Specification document and the Software Requirements Specification documents. A custom C++ program was written to parse the requirements text sections and the requirements traceability tables. The program was approximately 500 lines of C++ code. The parsing was made easier by manual editing of the documents before running the parser. Only the relevant sections of the documents were parsed. All introductory and closing boilerplate sections were removed. The format of the sections was well defined because the documents follow the 2167A format. Parsing the requirements traceability tables was simplified because the column entries of the tables were separated by tab characters. A C++ program was written because at this time the Tcl scripting environment had not been created. A Tcl script would have been easier to create and could have performed the same task.

The parsing program was patched together quickly because it only needed to be run successfully one time in order to capture the requirements information in Catalyst. From that point on, the requirements traceability was maintained in Catalyst. Requirements traceability tables were generated from the Catalyst requirements network.

A set of Tcl scripts was written to assist in the development of the Catalyst requirements. These scripts looked for system requirements which did not trace down to software or interface requirements. The scripts found over a dozen of these. The scripts looked for software and interface requirements which did not trace from any system requirements. The scripts found over twenty of these. The scripts looked for system, software, and interface requirements which were not tested. Since the test traceability was still in development at this time, the scripts found over one hundred requirements which were not linked to tests. All these problems which had existed in the requirements documents were eventually discovered and corrected using the automated analysis scripts.

If it was ever needed, the Catalyst requirements network could be migrated into a COTS requirements development tool. The COTS tool, such as DOORS, would be integrated into Catalyst. A Tcl script would be written to traverse the Catalyst requirements network and replicate it in DOORS. This would allow DOORS capabilities to be applied to the requirements, while still permitting Catalyst to access it.

This scenario can be generalized to capture in Catalyst useful information that is stored in any electronic format that is readable and parsable. This includes documents, spreadsheets, intelligence messages, help files, ASCII files, and custom application data files. As long as there is some structure which can be used to interpret the data, it can be modeled and stored in Catalyst. This can be used to recover valuable information and rehost it in Catalyst where it can be made easily accessible to users and where it can be analyzed using scripts or migrated into newer tools.

Data stored in Catalyst can be analyzed much more easily than data stored in formats such as word processor documents. Catalyst explicitly represents ths

structure of information because it is by nature an object-oriented system. All data in Catalyst is stored in objects with well defined classes and attributes linked together by well defined relationships. Storing the structure explicitly makes the stucture of the information easy to use. If a system requirement traces to a software requirement this fact is explicitly represented using a relationship. Automated analysis tools such as scripts can easily discover this connection by requesting the set of software requirements related to the system requirement. Documents containing paragraphs of free text are much harder to parse and analyze.

Building object networks from documents and other information can be used to help with training and maintenance of older systems which are still in operational use. Parsing and capturing the information contained in the document set from an older system could be used to create an electronically browsable web of information about the system. Dependencies among parts of the system could be captured using relationships. Users performing maintenance on the system would be alerted to potential ripple effects of changes they were making. Users wishing to understand the system could research it electronically by following links from object to object rather than flipping through paper documents or searching the documents using a word processor. When the Catalyst Java Browser becomes available this scenario could be used to provide Internet access to the information base for existing systems.

5 Catalyst Applications

This section describes some of the applications of Catalyst to date.

5.1 Joint STARS

The Joint STARS program has been the first end user of Catalyst since the start of the ASEA project. Several Joint STARS personnel from Northrop Grumman have been Catalyst reviewers since the start of the effort and have provided many valuable comments on how an integration system could help real world engineering projects. Targetting Joint STARS drove the selection of ORACLE and RDD-100 as the first engineering tools to be integrated. This section describes the Joint STARS application at Northrop Grumman.

The Joint Surveillance Target Attack Radar System (Joint STARS) is the most advanced airborne ground surveillance radar system in the world. It provides key strategic surveillance capabilities to the United States in both war and peace time missions. Joint STARS provided US commanders with critical information in the successful execution of both Desert Storm and the Joint Endeavor I & II missions. The diverse threats facing the United States in the wake of the collapse of the Soviet Union make the continued and effective evolution and enhancement of Joint STARS an important need.

Joint STARS is a very complex hardware and software system, with a long development history and a long anticipated service lifetime. Developing and evolving Joint STARS has been complicated by a number of factors, including the sheer size and complexity of the system, the large number of people required to build and maintain it, the scarcity of system experts, limitations of the engineering tools used, and the evolution of engineering technology.

During the middle 1980s when Joint STARS development started, relational databases such as ORACLE were the only tools available for electronically capturing and maintaining the traceability of the tens of thousands of requirements needed to specify the system. Since that time, new engineering and requirements tools such as RDD-100, and desktop computer databases such as Microsoft Access have emerged to provide additional useful capabilities. Though these tools provide valuable functionality, they operate in isolation from one another, making it difficult to fully utilize the information they contain. These database tools provide one way modeling and viewing data which is useful, but do not capture the graphical and background information which is contained in the Word and Frame documents for the system. To perform engineering activities using the current set of tools requires access to multiple tools, expertise in using each of the tools, and duplication of effort to bring up and interact with each of the tools. Furthermore, there may be nowhere to store valuable information such as traceability between data items stored in different tools.

The Catalyst system was applied to provide valuable benefits to Joint STARS evolution by integrating the tools and information required to support it. Catalyst allows the user to browse and analyze all the integrated data using one simple and graphical interface. This reduces the expertise required to access key information and helps get the most benefit from the large base of information which exists about Joint STARS. Catalyst can also capture and use information such as linkages between data in different tools which will allow users to more accurately assess the potential impacts and "ripple" effects of changes and enhancements to the system. Catalyst technologies can be applied to link information in databases with the corresponding information stored in traditional documents. This linkage allows a user to transparently navigate between databases and documents, allowing effortless access to the information they need to perform their jobs. These capabilities enhance the accuracy and reduce the effort needed for evolving the system.

5.2 I-SPECS

I-SPECS is the *Integrated System for the Predictable Evolution of Complex Software Systems..* The scope of I-SPECS is to adapt and integrate critical software components and technologies into Catalyst to provide an integrated system for performing requirements development and negotiation. The resulting integrated system will function in support of predictable evolution of complex software systems. The initially identified components include two key technologies— *WinWin* and *I-Doc*—and their supporting tools, which will be adapted and integrated into *Catalyst*. An event monitoring and reasoning capability known as FLEA (which was developed by CS3, Inc.) and a database integration technology known as Sanctuary ((which was developed by the University of Colorado at Boulder) are also being integrated.

I-SPECS will be evaluated and demonstrated in the context of the Joint STARS program, which is a large, complex, software intensive, evolutionary system development effort.

5.3 Warner Robins Air Logistics Center

Modus Operandi is proposing (under AFRL PRDA 98-07-IFKPA) to support the Joint STARS annual release program at Warner Robins ALC using Catalyst. A new annual release program is started every January and runs for about nineteen months. The annual release program is a good fit for Catalyst because it will benefit from the technology and because it can generate both near term validation of the approach plus continuing and measurable benefits. Two annual release programs are already in progress. These programs will provide a baseline of existing measures for comparison and quantitative evaluation of the improvements realized using Catalyst technology.

5.4 Cape Canveral Launch Operations & Support Contract

Sverdrup, the Launch Operations & Support Contract prime contractor at Cape Canveral Air Station (CCAS), selected Catalyst as the key technology for integrating the diverse tools and data used in support Titan, Delta, and Atlas launches. Sverdrup and SAIC are partnering with Modus Operandi on the $25M effort to modernize and integrate the information systems used in support of CCAS launches. The user community for the integrated system consists of over 300 users working at a dozen installations around CCAS. Tools and data used by various facilities will be integrated using Catalyst and made available over the Space Port Launch Operations Intranet. The integration made possible by Catalyst will help to distribute information automatically to the people who need it, will allow data to flow from one tool to another as needed, and will improve mission readiness and efficiency by permitting global analysis of operations which are not possible in a traditional stove piped operation. The integrated system supported by Catalyst will provide new capabilities for supporting more efficient launch operations at CCAS.

5.5 STRICOM

Catalyst is gaining acceptance in the Modeling and Simulation (M&S) community. It is the integration framework for the U.S. Army's Simulation Training and Instrumentation Command's (STRICOM) High-Level Architecture Integrated Development Environment (HIDE). The High-Level Architecture (HLA) is DoD's standard for enabling interoperability among all types of models and simulations including C4I systems, and facilitating the reuse of simulation assets.

The formal definition of the HLA consist of three components: the HLA Rules, the HLA Interface Specification, and the HLA Object Model Template (OMT). The rules describe the responsibilities of simulation components within the HLA. The HLA Interface Specification defines the interface to the HLA Run-Time Infrastructure (RTI), which provides the mechanisms for simulations to communicate. The OMT provides a template for documenting HLA relevant information about classes of simulations and their attributes and interactions. In HLA nomenclature, the attributes and interactions of a class of simulations is referred to as a Simulation Object Model (SOM) and the object model that defines how a set SOMs interact is referred to as a Federation Object Model (FOM). A simulation that implements the attributes and interactions of a SOM is referred to as a Federate and the cumulative interaction of the Federates in a large scale simulation is referred to as a Federation Execution.

The entire development process, from concept formulation, to SOM and FOM development, to the Federation Execution is defined by the Federation Development Process (FEDEP). HLA tools are currently being developed to address each phase of the FEDEP. However, the tools are not interoperable and most are being developed in a stovepipe fashion. In HIDE, Catalyst provides the

common integration framework that supports a common object representation for HLA-based information, allowing data from various tools to be shared and semantically linked, thus giving an integrated view of related HLA FEDEP data. This is critical in providing a robust environment that supports the FEDEP process, where different tools support each phase. This linkage supports traceability among design artifacts from different phases and allows the Catalyst client tools to perform comprehensive analysis and operations on the entire set of data, regardless of which tool the data originated from. For example, a simulation Requirements Document defined during Concept Formulation may be linked to a particular Federate that fulfills a role in the Federation. If there are changes to the Requirements Document, a Catalyst Impact Analysis may be performed and it will identify the linked Federate as a potentially impacted object. Another benefit that Catalyst provides to the HIDE environment is the ability to enact process models. As mentioned previously, the development of a Federation Execution proceeds as specified by the FEDEP model. In the HIDE program, the FEDEP model has been defined using the Catalyst Process Definition Tool and may be executed using the Catalyst Process Enactment Tool. In the Process Enactment Tool, at each phase of the FEDEP, appropriate supporting tools may be launched for that phase. The design artifacts created from the tools from each phase may be semantically linked across phases, providing the benefits discussed previously.

In HIDE, the Catalyst CORBA infrastructure and the existing client tools will greatly support the development and maintainability of evolving large scale distributed simulations, in the same manner it supports large scale systems engineering efforts

The Browser provides the ability to browse integrate data from heterogeneous data sources; the Impact Analysis Tool provides the ability to determine how changes to an object may propagate and affect other objects; the Process Definition and Enactment tools provide for definition and execution of process models. In general, these tools perform the same functions and provide the same support as they do in any other domain. Catalyst is also allowing the Modeling and Simulation community to benefit from other tools developed for the System and Software Engineering communities. Under the Evolutionary Design of Complex Systems (EDSC) program, numerous engineering tools are being integrated with Catalyst. HIDE has already made use of one such integrated tool, the WinWin requirements negotiation and rational capture tool developed by the University of Southern California. This will help support the large scale collaborative requirements negotiation efforts that go into designing and developing a Federation Execution.

Since HIDE is currently a "work-in-progress", additional client tools are envisioned that are specific to the HLA domain. These include specialized SOM/FOM development tools that support Semantic Consistency and Completeness, Scenario Generation, and Federation Execution. Consistency and Completeness extends the basic impact analysis principles to determine how and

when "fixes" to a change resulting to an impact are consistent and complete. For example, if a FOM requirement is changed, semantic consistency can help determine when the FOM is consistent with the change. Scenario Generation allows a Federation scenario developer to use the integrated assets, which includes objects and their possible interactions, to define a simulation scenario for execution with those assets. Federation Execution support is the ability to initialize and control an executing federation. These ideas, as well as generic enhancements to Catalyst, will be evaluated in HIDE follow-on efforts.

5.6 MCC Software and Systems Engineering Productivity

MCC is conducting consortial research activities under the Software and Systems Engineering Productivity (SSEP) project. The goal of SSEP is to reduce systems and software development costs and time cycles by supporting the adoption of product-line strategies focused on the application of architecture-based generation and testing approaches.

SSEP has been engaged in evaluating Catalyst as a candidate for the infrastructure to achieve information integration and tool interoperability. Evalution activities include prototyping a tool integration framework using the loose integration strategy and managing relationships among objects which are originated from different information resources using the Catalyst General Relationship Manager (GRM).

5.7 NAWC GEODE SBIR

Modus Operandi conducted a Phase I Small Business Innovation Research (SBIR) project for the Naval Air Warfare Center (NAWC). The GEOgraphically Distributed System Engineering Environment (GEODE) project extended Catalyst to allow geographically separated personnel to effectively collaborate on complex engineering data. Synchronous collaborative technologies such as remote data browsing, audio, video, and whiteboard conferencing were evaluated and found to be inexpensive and effective. These technologies provide the immediate sharing and discussion of engineering data required for brainstorming, review, design communication and team building. An Internet based remote data browsing capability was prototyped and found to be effective for remote viewing of engineering data such as requirements traceability.

An asynchronous collaborative capability known as the Object Community Synchronization Tool was prototyped and evaluated. This tool supports correctly and consistently exchanging complex engineering models between remote sites. The prototype tool builds on existing Catalyst capabilities and allows remote sites to jointly develop and maintain baselines of complex and interrelated data. The ability to maintain remote baselines of information is critical for ensuring that distributed teams are all working from the same plans and goals. It also enables remote design reviews, remote program monitoring,

and other activities which allow an organization to make use of remotely located personnel expertise.

The Phase I efforts concluded that creating the GEODE collaborative engineering environment based on Catalyst is both feasible and effective for improving distributed engineering practices. The Phase II proposal recommended integrating widely available COTS video and whiteboard capabilities for supporting remote meetings and design reviews. It also recommended developing a production version of the Object Community Synchronization Tool to complete the initial collaborative engineering environment. Unfortunately, Phase II funding has not been secured as yet to continue this valuable effort.

6 Future Directions

6.1 Platform Support

One of the most frequently requested enhancements to Catalyst is the support of additional operating system and hardware platforms. Catalyst currently is supported only on Solaris 2.4 or higher. Many organizations have requested support for accessing the client tools from Windows95. This is the most requested client platform. Some organizations have asked for support for accessing client tools and running servers under WindowsNT. Following these requests have been a smaller number of requests for other UNIX environments, particularly Silicon Graphics, which is widely used by military simulation developers.

6.1.1 Java Clients

The most reasonable strategy for meeting the requests for wider client tool support is to port them to Java. This will allow them to be used from any platform that supports the Java virtual machine, which is almost every platform commonly in use. The Catalyst client tools will most likely be constructed as Java applications, given their size and the need for accessing files on the local machine. Smaller versions of them, particularly the Browser, may be created as applets to allow them to be downloaded and used more conveniently.

Some prototyping is underway under other related Catalyst R&D efforts to determine the feasibility of porting Catalyst clients to Java. The Object Cache has been ported and seems to be operating correctly. It is still too early to tell if this strategy will work, but with most of the world betting on Java for their user interfaces, it appears likely that Java will eventually mature enough to be usable.

6.1.2 VisiBroker Servers

Since SunSoft has discontinued the NEO CORBA implementation that Catalyst is built on, it is important to choose another ORB and port Catalyst servers to it. Modus Operandi has looked at various products, talked to their vendors, and received feedback from groups like MCC, who have tested most of the ORBs available today. The VisiBroker ORB from Visigenic Software looks like the best choice for replacing NEO in Catalyst. It is one of the fastest ORBs, and has the largest market share as of early 1998. It is also portable to many platforms requested by Catalyst users.

Porting Catalyst to VisiBroker will require porting the Object Cache, the Catalyst Object Servers, the General Relation Manager, and all tightly integrated tools such as ORACLE.

6.2 Browser Improvements

The Browser is a key Catalyst Tool because of its ability to view and manipulate all data in Catalyst. A number of important improvements have been requested by users or identified by Modus Operandi.

6.2.1 Network Browser

A Browser that showed objects and relationships graphically using a network instead of a hierarchy like the current Browser would be very useful. The hierarchy view is very useful for a lot of engineering data and would be preserved. The new network browser would be useful for understanding data that has cycles. Each object would appear on the network browser window only once. The network browser would use some heuristic to try to place expanded objects to minimize line crossing, but the user would be able to rearrange the object locations as desired. Objects could be excluded from the view by adding them to an exclude list.

6.2.2 Schema Browser

The Browser could be extended by adding a schema browser window. This window would allow the user to graphically browse the schema. Classes and relations would be graphically shown using a network of boxes representing classes and labeled arrows representing relations. The user could add classes to the view by selecting them from a list. The user could show the relations from a class by selecting the class and hitting an expand button. Classes could be excluded from the view by adding them to an exclude list. This would be useful for removing classes that are related to almost every class, such as Annotation. Various options for showing and hiding forward and inverse relation names could be supported. An option for showing the attributes defined by a class could be supported. A good schema browser would help users understand the data model used by their Catalyst installation.

A really ambitious schema browser would allow the user to edit the schema. New classes and relations could be added. Existing classes and relations could be modified or deleted. These features would be very difficult to implement, particularly if automated migration of the effected data was required.

A prototype schema browser was implemented using Tcl scripting. A special version of the Catalyst Tcl shell was created which links in GUI commands from a Tcl extension known as "Tk". The Tcl/Tk scripting shell allows window, menu, and drawing operations to be performed. A display only schema browser was prototyped using this shell in about 7 days of effort. Using the prototype has confirmed the usefulness of schema browsers in understanding the structure of Catalyst data.

6.2.3 Better Object Community Support

The Browser could be extended to make it easier to work with an entire object community as a single unit. An object community view or window could be added, which would show the user what object communities were available to them. Each object community would be represented as a single user-defined icon. The object community could be duplicated, locked, unlocked, imported, exported, or deleted using one menu operation. This would make it much easier to manipulate object communities.

The definition of the object community would be integrated with each actual object community instance. Object communities are defined as a set of starting objects, a set of member relations, and a set of reference relations. This set of items is the object community definition. An instance of an object community is specified by the set of starting objects and the object community definition. This is somewhat analogous to a class (which would be the object community definition) and an object (which would be the object community instance).

The enhanced Browser would contain a graphical schema viewing window that could be used to graphically create object community definitions. The objects and relations that were part of the object community definition would be selected by the user and added to the definition using a menu item. Member classes and relations would be shown in green. Reference relations and classes would be shown in blue. Non-member items would be shown in black. Object community definitions would be managed as icons in the enhanced Browser. They could be shared among multiple users (in which case the user responsible for them should lock them) and potentially used to define many object community instances. Each object community instance would keep a pointer to its object community definition, for use by operations on the community.

These enhancements would make it much easier to create and work with complex sets of data in Catalyst by raising the abstraction level above the level of individual objects and relationships.

6.3 Distributed GRM Support

A distributed implementation of the GRM would be useful for geographically distributed teams working in one Catalyst environment and for supporting teams with more than 100 engineers. The current GRM design is centralized and uses Solaris memory mapped files. Catalyst was designed for distributed relation management and several designs have been worked out for supporting distributed relation management. Probably the best engineering solution involves using one GRM for each platform that supports Catalyst servers. The relationships for the objects stored in the servers on each platform would be managed by the GRM that belonged to that platform. The GRM dedicated to a platform would not necessarily have to run on that platform, but could run on a companion machine nearby on the network.

Each relationship would be stored in either one or two GRMs. If both sides of the relationship were stored in servers on the same machine, the relationship would be stored only in the platform GRM. If the domain and range objects were stored in servers on different machines, the relationship would be stored in the GRMs for each of the two machines.

Clients would manipulate the relationships for an object by calling the RelationAccess interface of the object itself, rather than by calling the GRM as is done now. Since the Object Cache hides the details of relationship manipulation from client programs, this change would have minimal impact on the existing clients. In any case, relationship calls would be changed to replace the GRM with the domain or range object as the target of the call. This would require moving one variable from the parameter list to be the target of the call.

A significant study and prototyping effort needs to be conducted to ensure the technical soundness of any distributed relation management approach. A distributed relation manager would allow Catalyst data networks to grow without any limit that we are aware of. To add more capacity would require simply adding more machines, each of which would support a number of object servers and one GRM server.

6.4 Content Based Queries

Catalyst was designed to efficiently store and process fine-grained structured information, such as that produced and used during engineering efforts. Catalyst uses explicitly stored relationships between objects to indicate that the objects are somehow related or interact. This is a very efficient model for dealing with engineering information. It may be thought of as a navigational access model in which explicit relationships are followed from object to object to access the information which is desired.

Another, orthogonal, access model is to use content based queries to find desired information. This is the method traditionally used by relational and other types of databases which process ad-hoc queries by searching the content of information for the desired results. This is a completely different model which is much less efficient but has the advantage of allowing the user to retrieve information without having any explicitly stored path to it.

It was originally envisioned that Catalyst would emphasize explicit navigation but would also support content based queries using the CORBA Query Service. This would allow Catalyst to use a COTS solution for content based queries. Unfortunately, no COTS implementation of the Query Service has been produced which could be used in Catalyst. This is due in large part to the lack of a true SQL standard and to the existence of a separate standard for object base queries. It will be very difficult to create a unified Query Service specification for CORBA which can be implemented efficiently.

It would be possible and very worthwhile to investigate implementing some type of simple but powerful query capability for Catalyst. By making certain restrictions, it should be possible to create a very useful capability in a reasonable amount of effort. It would also be useful to create a more complex capability which would permit the so called heterogeneous join operation. This would allow information stored in one tool to be "joined" in the SQL sense with data stored in another tool. This would provide a very powerful data analysis capability. It could be used to "mine" integrated data sets for correlations, to perform data fusion, and to create relationships between tools automatically.

A query capability would greatly enhance Catalyst's ability to exploit the integration of information.

MISSION
OF
AFRL/INFORMATION DIRECTORATE (IF)

The advancement and application of information systems science and technology for aerospace command and control and its transition to air, space, and ground systems to meet customer needs in the areas of Global Awareness, Dynamic Planning and Execution, and Global Information Exchange is the focus of this AFRL organization. The directorate's areas of investigation include a broad spectrum of information and fusion, communication, collaborative environment and modeling and simulation, defensive information warfare, and intelligent information systems technologies.