Evaluation of an Object-Based Data Interoperability Solution for Air Force Systems

by

Jeffrey R. Doering

Submitted to the Department of Electrical
Engineering and Computer Science in
Partial Fulfillment of the Requirements for the
Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

Massachusetts Institute of Technology

May 2000
June 2000

Signature of Author.....
Department of Electrical Engineering and Computer Science
May 22, 2000

Certified by.....
Dr. Amar Gupta
Thesis Supervisor

Accepted by.....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

EVALUATION OF AN OBJECT-BASED DATA INTEROPERABILITY SOLUTION FOR AIR FORCE SYSTEMS

by

JEFFREY R. DOERING

Submitted to the Department of Electrical of Electrical Engineering
and Computer Science on May 22, 2000 in partial
fulfillment of the requirements for the
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Data interoperability between computer systems is critical for businesses. One design proposed for future Air Force systems, the C2STA data architecture, attempts to provide standardized object-oriented interfaces to data, independence from underlying data storage technologies, and implementation transparency. If successful, such an initiative would greatly simplify data interoperability issues. This thesis examines the details of the C2STA data architecture and presents the results of one prototype implementation. Further, research on other data architectures that complement this investigation is described. This thesis concludes with suggested modifications to the C2STA data architecture.

Thesis Supervisor: Dr. Amar Gupta

Title: Co-Director, Productivity From Information Technology (PROFIT) Initiative
Sloan School of Management

Acknowledgements

I would like to begin by thanking my colleagues from the MITRE Common Data Environment Office for their assistance in researching this topic. My many discussions with Ed Housman on data interoperability issues and data model standardization provided an essential introduction to this field. Dr. Scott Renner's excellent grasp of the C2STA's purpose, its relationship to other interoperability initiatives, and his introduction to data interoperability in general helped me organize my often jumbled thoughts. I must thank Jay Scarano for giving me the opportunity to study the C2STA data architecture and for his initial introduction to the material. Jeanne Fandozzi deserves credit for making my research in the Datalab possible. Finally, I cannot ignore the contributions of Ray Spinosa. Ray's ability to end the most technically abstract conversation on the beauty of object-oriented interfaces with questions like "Why are we doing this? Will this save the Air Force money?" was critical to keeping this project focused on the underlying reasons for studying data interoperability.

Having addressed the assistance I received at MITRE, I must now thank the individual who made this entire project possible. Dr. Amar Gupta provided me with the opportunity to work in the MITRE Datalab. Further, he provided important guidance on the development of this thesis and the search for related research. And of course Dr. Gupta provided the appropriate deadlines to make sure I actually got around to writing this thesis.

Table of Contents

List of Figures

Chapter 1

Introduction

Virtually everyone who has used a computer system is familiar with interoperability problems. The specific problem may come in the form of a computer user with one word processor trying to open a file created in a second word processor. In a different situation, a computer user on a private network might want to exchange messages with users on the Internet. Or perhaps the problem arises because a Web user wants to obtain contact information for a company but that company's Web site is written in French and the user only understands Russian. Each of these problems can be categorized as some kind of interoperability issue. The first two issues can be solved using computer systems specifically designed to provide interoperability between otherwise incompatible technologies. The third problem is more difficult to solve using automated systems as language translation algorithms are far from perfect. In such a situation an interoperability solution might include a human translator creating parallel web sites in multiple languages. The interoperability situations described above qualify as problems when users cannot achieve their goals because there is no solution in place. Interoperability problems can arise even when specific provisions have been made to facilitate interoperability. For example, an interoperability solution might rely on the existence of a reliable communications channel between systems; if such a channel fails interoperability is no longer possible.

The preceding examples provide a glimpse of the interoperability problems faced by computer system designers. Further, they demonstrate that interoperability problems can occur on many levels. The example of the web site in French highlights the case where all of the technology-related interoperability issues have been solved (e.g. a common network protocol such as TCP/IP, a common data exchange format such as HTML, and full communication reliability) yet interoperability is still not achieved. Thus, interoperability must be addressed at many levels in a system. The following research is focused on the data interoperability between systems. An introduction to data interoperability and some of the approaches used to achieve data interoperability between systems provides a useful context for researching this problem.

1.1 The Data Interoperability Problem

One of the goals of the Common Data Environment (CDE) office of the Air Force Electronic Systems Center (ESC) is to investigate data interoperability issues that affect Air Force systems. It is important to define data interoperability so that such issues can be distinguished from other kinds of interoperability issues (e.g. communication interoperability). Data interoperability is the ability to correctly interpret data that crosses system or organizational boundaries [Renner, 1999]. Thus, moving data between systems (communication interoperability) is not enough to qualify as data interoperability. On the other hand, because data interoperability specifically addresses data that "**crosses** system or organizational boundaries" it cannot occur without some kind of communication interoperability. The consequence of this dependency is that the definition of data interoperability does not eliminate the need to consider other

interoperability issues. Although CDE research focuses on the data interoperability problem, it must address other kinds of interoperability as well. It is preferable to adopt existing solutions to lower-level interoperability issues (e.g. using the already standardized TCP/IP protocol for network communications). However, this is not always possible. For example, communications between two different relational database management systems (RDBMSs) probably requires some solution to application programming interface (API) level incompatibilities. Although this is not technically a data interoperability problem, any proposed solution must at a minimum explain what existing technology could be incorporated to solve this difficulty. At a maximum, a data interoperability architecture might need to completely solve the underlying problem if no existing technology can be utilized.

Finally, some classes of interoperability problems occur at a higher level than the data problem. For example, if two systems interoperate but some information needed by one system is not available through the other full interoperation is impossible. This is a process interoperability issue and cannot be solved through any data interoperability approach [Renner, 1999]. As such, problems of this nature can be ignored. One must assume that process interoperability has already been addressed between the systems under consideration.

1.2 Various Data Interoperability Approaches

Based on the stated definition, no data interoperability issues can arise within a single organization using a single system. However, it is extremely likely that the introduction of even one additional system to such an environment will introduce data

interoperability needs. In an environment as large as the Air Force, the existence of many organizations each using many systems results in a large number of interoperability requirements. The Department of Defense (DOD) as a whole is an even more complicated example. Although several examples of data interoperability needs have been cited and more fundamental requirements such as communication interoperability discussed, the specifics of such systems have not been addressed.

Plain old telephone service (POTS) can often enable individuals in an organization to solve data interoperability issues. Suppose there is a need to combine information from two systems for some decision-making process. One solution might involve an individual with access to one system calling another individual who has access to the second system and asking the second individual to provide the appropriate information. The first individual can then associate the additional information with the information already available in the first system and supply the result to the decision-making process.

In this example, data interoperability has been achieved. However, it is quite likely that this is a time-consuming process and much more expensive than a solution where some automated mechanism exists to facilitate the data interoperability. It is still important to keep such possibilities in mind because such mechanisms do solve many data interoperability issues. If a data interoperability need is very infrequent or unique it might be more cost effective to use human intervention than to build an automated interoperability mechanism. Nonetheless, the following research focuses on automated data interoperability solutions. Even within this context interoperability solution can work at widely varying levels of granularity. On one extreme, a solution might try to

standardize all of the low-level details of several systems to allow interoperation. In essence, the multiple systems are combined into one super-system. At another extreme, a small number of very specific data interoperability needs might be defined and a very specific solution implemented that only supports interoperability of the defined data. The following examples of data interoperability initiatives illustrate various levels of interoperation granularity.

1.2.1 Data Model Standardization

Many current Air Force systems rely on RDBMSs for storage and retrieval of persistent information. RDBMSs store information according to relational models. These models provide an abstraction of the real world and a key for interpreting the data in a RDBMS [Renner, 1999]. The abstraction provided by a data model exists because the model specifies precisely what information a system will store and provides methods for accessing that data. Real-world details which are not present in the data model are assumed unimportant for the purposes of the given system. Data models allow interpretation of data because they define how data are structured, they describe how various structures relate to one another, and they usually provide a description of the real-world object being modeled.

Data model standardization offers one possibility for addressing data interoperability issues. Because a data model defines how an application "sees the world", systems with common data models can achieve data interoperability relatively easily. As already explained, a data model describes how to interpret the data in a

system. Systems with a shared model share a common interpretation mechanism. This allows them to easily guarantee the required correct interpretation of data across systems.

A very simple (although admittedly contrived) example makes the issue of data interoperability more concrete. Imagine two systems that share a common data model, have some data element named "sky color", and both report the value as "blue". They can quickly conclude that they agree on the color of the sky. However, it is very likely that two systems will not have a common data model. That one will have an element called "sky color" and the other will have an element called "color_sky". The first will report that value of "sky color" is "blue". The second will report that the value of "color_sky" is "0,0,255". Add a third system to the scenario with an element "sky_color" with a value of "14". The first system has stored "sky color" using the human understood (although not necessarily very precise) concept of the color name "blue". The second system has stored "color_sky" in a 24-bit red-green-blue (RGB) form. This is probably the same as the value of "blue" in the first system although an interpreter would have to be careful about the precise definition of "blue". Finally, the third system could be referring to all sorts of system-specific identifiers for color. An index into an internal color-palette is realistic possibility. The fact that all three systems use different names for the same concept further complicates the situation. While a human could easily guess that the three names refer to the same real-world characteristic, a computer could not make this determination with certainty (in fact even a human might only be guessing).

A data model standardization effort would take the three systems in the second scenario and force them to agree on a common name attribute storing the color of the sky.

Further, they would have to agree on a common format for storing the value as well. This is a non-trivial effort in real systems that might have existing models describing thousands of attributes. Further, the example only dealt with a single data attribute. Relational models actually define much more complicated entities and relationships between entities. While it might not require substantial effort to rename the attribute storing the color of the sky, redefining entities and the relationships between them can be very complicated. (On the other hand, if a system's applications are tightly coupled to its data model it might even require a fair amount of effort to rename the color of the sky.) Subtle differences in seemingly similar data models can greatly increase the effort required to achieve data model standardization. Further, a standard data model requires systems to model the world in the same way. This means that systems must agree on what aspects of the world they are interested in and to what level of detail. Although it is possible that no single application adopts all parts of a standard data model, data interoperability is only achieved for those parts that two interoperating systems have in common.

Data model standardization offers a very high level of data interoperation at a high implementation cost. Beyond the actual effort required to standardize data models across systems, such efforts are bound to encounter control issues particularly as they cross organizational boundaries. Both the DOD and the Air Force have attempted various data model standardization efforts. This approach will most likely succeed in tightly coupled systems crossing a minimal number of organizational boundaries. Because many systems that do not fit this profile require data interoperability, data model standardization cannot be viewed as a complete solution to the problem.

A final note on data model standardization is in order. The earlier discussion of data interoperability noted that it depended on other mechanisms such as those supporting communication interoperability. Data model standardization by itself does not provide a complete solution. Two systems might share a data model but not have a communication mechanism. However, it is relatively easy to build communication interoperability between two systems using existing network technologies. If the systems already share a data model, data interoperability is achieved. This is quite a different situation from the common case where two systems have communication interoperability but do not have a high degree of data interoperability.

1.2.2 Message Passing

Any data interoperability solution tackling the "color of the sky" problem will have to include some agreement on identification of the attribute of interest as well as the universe of possible values for the attribute. The high costs of data model standardization arise because this agreement is being implemented at the data storage layer in systems. This means that the agreement fundamentally affects the internal operations of the systems as well as the data interoperability between systems. Message passing architectures represent a very different solution space. Message passing focuses only on agreement for data interoperability between systems. The internal data representations are unconstrained by the architecture.[1]

Message passing architectures support data interoperability when two systems agree on common messages. The messages two systems define in common represent

[1] Of course internal data representations must actually represent the information needed for data interoperability, thus the only requirement is that some translation from the internal state to the message data format be possible.

their shared view of the world. By translating internal system data representations into messages the systems provide a means for both transferring data and guaranteeing a common interpretation of that data. Assuming that communication interoperability mechanisms are in place, the requirements for data interoperability are satisfied. A typical system might require an auxiliary message creation and parsing module to support this solution. Ideally no other changes to the original system would be required. Such an approach allows much more autonomy between interoperating systems. This can reduce implementation costs and system control issues.

Given that message passing provides an opportunity to address data interoperability with a reduced impact on existing systems, one might question the utility of data model standardization. One important factor to remember is that the two solutions provide dramatically different levels of interoperability. Data model standardization offers the potential for systems to share a large portion of their internal state. Message passing is better suited to situations where the data interoperability needs represent a smaller part of a system's state. If a large amount of interoperability is required, the cost of defining messages and building message-parsing systems could easily approach the cost of standardizing data models. Further, data model standardization allows for other system optimization such as shared data servers. Additionally, common data models allow for re-use of system modules whereas a pure message passing solution requires each system to build its own independent message handling module. Finally, message passing may not be appropriate for situations where data timeliness is critical. In this case shared data models and the potential for shared

database systems offer a better platform for guaranteeing one consistent view of global state.

As in the case of data model standardization, message passing is not a one-size-fits-all solution to data interoperability. However, it is a powerful tool in a data interoperability repertoire. Many current Air Force systems use message passing in a format called the United States Message Text Format (USMTF)[2] for data interoperability. Ongoing research initiatives are investigating the potential benefits of adopting Extensible Markup Language (XML)[3] and the commercial tools that support it. Such efforts will probably achieve the most success between relatively independent systems with modest interoperability requirements.

1.2.3 Object Oriented Data Wrappers

The two data interoperability approaches discussed thus far represent extreme positions on the solution spectrum. The first placed significant requirements on a system's internal data representations. The second placed almost no restrictions on the internal operation of a system. The third approach presented falls somewhere between these two positions. Data wrapping technologies attempt to standardize the interfaces to a system's internal state. This allows the system the freedom to adopt database level data models and even database technologies based on factors such as performance. Interoperability is achieved at the interface level. While such an approach is not constrained to object oriented interfaces (a functional interface could achieve the same

[2] For information on USMTF see: http://www.forscom.army.mil/interop/USMTF/DEFAULT.htm

[3] For information on XML see: http://www.w3.org/XML/

goals), objects are a natural choice for today's programming languages, tools, and environments.

Object oriented data wrappers require a lot of the same agreements as data model standardization. However, because such wrappers are implemented above the database level they can sometimes be introduced into existing systems without modification to the existing database or applications. Instead the wrappers provide a common access mechanism for new applications. This has the added benefit of allowing the underlying database to evolve without affecting the data interface used by applications.

Data interoperability in such a system is achieved when system modules are designed to communicate with the wrapper objects. When two interoperating systems support the same objects, they can access each other's objects in a seamless manner. Communication interoperability can be achieved through distributed object technologies that support network communications.

Unfortunately it is unlikely that a system's physical data model can achieve complete independence from its object interface. The semantics of the data objects will be somewhat constrained by the underlying data representations. The success of the object wrapper approach depends on the benefits of having an abstract object model outweighing the cost of any dependence between the object model and the underlying data model. Because of this constraint, object wrapper solutions will probably achieve the most success in systems that support similar data models. This condition improves the chance that common objects can be instantiated from the independent systems.

Current research projects both within the Air Force and in the computer industry as a whole examine various object wrapper architectures.[4] The following chapters describe a CDE experiment involving one such architecture. This architecture is then compared to other object wrapper approaches.

1.3 Literature Survey on the Data Interoperability Problem

The preceding explanation of data interoperability and the examples of data interoperability solutions provided an informal introduction to the subject. However, it is important to realize that this problem has been the focus of a significant amount of research. While the informal explanation is sufficient to grasp the general nature of the problem, a brief synthesis of formal research related to data interoperability will provide an academic context for examining the issue. Further, it will facilitate later comparisons between the data architecture studied in this thesis and existing research.

The data interoperability problem already cited is often called the need for *semantic interoperability* among *autonomous* and *heterogeneous* systems [Scheuermann et al., 1990, Sheth and Larson, 1990, Hurson et al., 1994]. The terms autonomous and heterogeneous refer to the fact that the systems under consideration are neither centrally managed nor based on the same technologies. For example, several Solaris servers running Oracle 8i to support different systems for independent branches of the DOD would be autonomous but not heterogeneous. While not all data interoperability problems within the Air Force meet both criteria, some do. Because solving the more general problem will also solve less-difficult problems, any approach to data interoperability should address both autonomy and heterogeneity of systems. Semantic

[4] For examples of object wrapper architectures see Chapter 4.

interoperability means the meaningful exchange of information [Goh, 1997]. Thus, it is clear that the definition of data interoperability used in this thesis closely parallels the concept of semantic interoperability among autonomous and heterogeneous systems cited in the literature.

Having established that the data interoperability issue examined in this thesis is in fact closely related to the problems addressed in existing research, it is useful to further consider how the problem has been classified. Both the concept of autonomy and that of heterogeneity have been broken into categories [Goh, 1997].

Autonomy is prefixed by one of the following: design, communication, or execution [Scheuermann et al., 1990]. Communication and execution autonomy refer to systems choosing who they communicate with and when, respectively. However, these distinctions are not of direct concern to the data interoperability problem as addressed in this thesis. On the other hand, design autonomy, or a system's ability to choose its own information, data model, and implementation, is a fundamental concern.

Two major classes of heterogeneity emerge. These are data heterogeneity and system heterogeneity [Goh, 1997]. The first issue addresses the organization and interpretation of data while the second addresses data models, data manipulation languages, concurrency controls, etc. As a whole, data heterogeneity leads to data conflicts. In his PhD dissertation, Goh identifies three categories of data conflicts from the relevant literature and further breaks these into nine individual cases. These cases will be very briefly described to illuminate the specific kinds of problems that must be solved by a data interoperability solution.

Schematic conflicts as documented in the literature account for four cases of data conflicts [Kim and Seo, 1991, Krishnamurthy et al., 1991]. These conflicts generally relate to differences in structure, or logical organization, of data [Goh, 1997]. The four cases are: data type conflicts, labeling conflicts, aggregation conflicts, and generalization conflicts. Data type conflicts arise when two systems use a different fundamental data type for the same piece of information. Perhaps a phone number is a number in one system and a string in another. Labeling conflicts refer to the naming of schema elements. The problem includes name collisions as well as name discrepancies. The earlier interoperability problem involving the color of the sky exhibited this problem. Aggregation conflicts involve differences in the choices for entities and attributes [Smith and Smith, 1977]. Attribute A of entity E in one system might correspond to attribute E of entity A in a second system. Finally, generalization conflicts concern differences in the relationships between entities.

Semantic conflicts have also been widely documented [Sheth and Kashvap, 1992, Naiman and Ouskel, 1995, Garcia-Solaco et al., 1996]. These kinds of conflicts can be categorized as naming conflicts, scaling conflicts, and confounding conflicts [Goh, 1997]. Semantic naming conflicts arise when different systems use different attribute values for the same concept or the same values for different concepts. For example one system might identify the United States by the abbreviation "US" while another might use "USA". Scaling conflicts involve the use of different units for the same concept. This was the problem in the NASA mission failure already cited. Confounding conflicts involve equating concepts that are actually different. For example, although two airlines might report the prices of their tickets as an attribute price, the two concepts of price

cannot be equated if one airline includes taxes in the attribute value and the other does not.

Goh identifies the final category of data conflicts as intentional conflicts [Goh, 1997]. Such conflicts relate to differences in information content present in data sources. These conflicts fall into two categories: domain conflicts and integrity constraint conflicts. Domain conflicts arise when different data sources have different (possibly implicit) domains. For example an Air Force database might claim to include tail numbers for all combat airplanes. However, comparison with a similar DOD database might reveal that the Air Force database excludes Navy planes. Someone trying to account for all combat aircraft owned by the United States could not use the Air Force database exclusively. Integrity constraint conflicts arise when different systems have different integrity constraints. For example, if two systems use different primary keys then one unique identifier for some real-world entity might not uniquely identify all of the information stored about it across several systems.

The preceding taxonomy of data conflicts provides a useful model for considering specific data interoperability problems. Some problems such as labeling conflicts can often be resolved through existing mechanisms such as RDBMS views. Other problems such as aggregation conflicts are more difficult to solve. The following investigation of a proposed data architecture for Air Force systems will not specifically address all of the types of data conflicts. However, any data interoperability must either address these conflicts or require that underlying systems eliminate certain classes of conflicts.

Having explained the academic background of the data interoperability problem it would be natural to now present the academic background of data interoperability

solutions. However, this discussion will be deferred until Chapter 4. Because the data architecture under consideration was not specifically presented as a solution to the data interoperability problem, nor was it justified in the context of existing literature on data interoperability, it is useful to first present the architecture and then try and classify it according to other literature in the field.

1.4 CDE Data Interoperability Investigation

The introduction thus far has provided the appropriate background for the following explanation of the CDE investigation of the potential impact of the Command and Control (C2) System Target Architecture (C2STA) data architecture on Air Force systems. Chapter 2 introduces the context of the data architecture as well as its specific details. The explanation is designed to outline the general goals of the architecture while providing important details for subsequent analysis. The third chapter describes in detail the CDE experiments designed to test the feasibility of using the C2STA data architecture to solve data interoperability problems involving existing systems. The chapter includes a description of the specific experiment scenario, details of the experiments, and a presentation of results. Chapter 4 examines other data architectures similar to that of the C2STA. The discussion draws on observations from the CDE experiments to highlight how other architectures might contribute to an improved C2STA data architecture. The fifth chapter summarizes recommended changes to the C2STA data architecture to improve its ability to support data interoperability between existing systems. Finally, the appendix lists the acronyms used in this thesis.

Chapter 2

C2STA Data Access Architecture

The previous chapter defined data interoperability and summarized various approaches to solving data interoperability problems. CDE research has focused on attempting to implement one particular data access architecture and analyzing that architecture's potential for solving data interoperability issues. The data access architecture examined was developed as part of a broader design initiative. The following is an introduction to that initiative followed by its data architecture. This provides a context for the CDE experiments and analysis.

2.1 The Command and Control System Target Architecture

The C2STA advocates an object oriented component-based design for future Air Force C2 systems. This design is aimed at achieving Air Force goals of an Integrated C2 System (IC2S) [C2STA, 1998]. Some of the potential benefits of such a design include reusability of systems and plug-and-play integration of systems.

The C2STA defines a multi-tiered architecture based on "capabilities". The C2STA uses the term component to define the lowest level structural element of software. Individual components do not have to comply with C2STA requirements. A capability is a software module designed to accomplish some C2 task. Capabilities are

composed of components and must comply with C2STA requirements. Figure 2.1[5] illustrates the C2STA multi-tiered capability architecture.

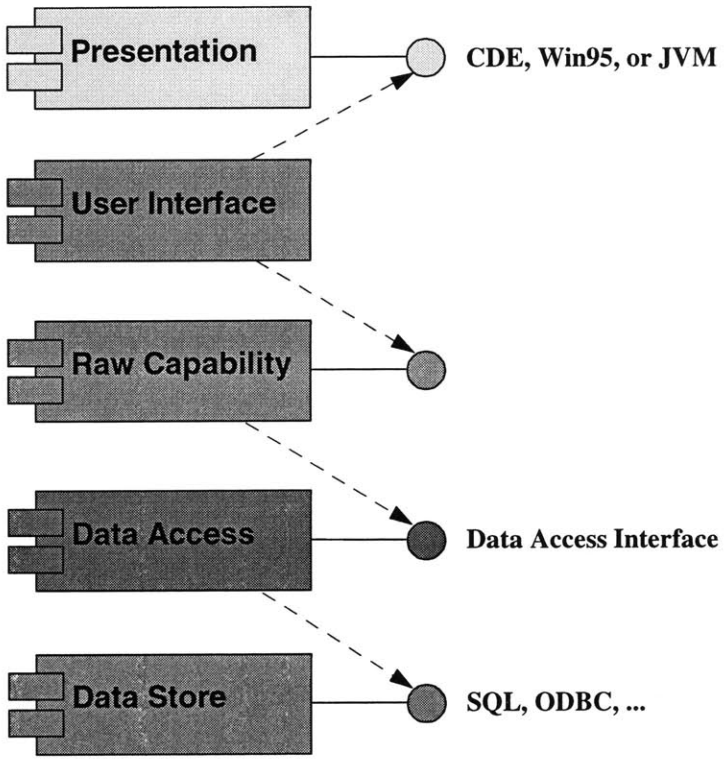

Figure 2.1 C2STA Capability Layering

The C2STA is presented as five distinct views: the capability view, the data view, the distribution view, the security view, and the production view. The following investigation of the C2STA data architecture focuses largely on the data view.

Having introduced the C2STA as a whole, the discussion now shifts to the details its Data View. This background is important because it provides the original definition of the C2STA data architecture studied by the CDE. Further, it provides insight into the design goals of the architecture. This allows for both a discussion of what exactly

[5] Appears in the C2STA as Figure 3-1.

constitutes the C2STA data architecture and evaluation of how successfully real implementations can achieve the original goals that motivated this design.

The C2STA specifies that its data view encompasses all data needed by C2 functions and capabilities. However, it goes on to identify five specific categories of data [C2STA, 1998]. These categories are: data already being shared, data from migrating C2 databases, data from existing commercial-off-the-shelf (COTS) desktop applications with internal databases, DOD and service reference data, and draft data. Each of these categories presents different constraints when building systems to support the C2STA data architecture. The experiments and resulting conclusions described in later chapters focus entirely on data from legacy systems. Although these systems offer a limited view for critiquing the C2STA data architecture, they represent the single most critical data category, as large parts of the envisioned IC2S will certainly evolve from legacy systems. Nonetheless, it is useful to keep all five of the categories in mind, as some details of the C2STA data architecture seem tailored to specific categories.

Before introducing the specifics of the C2STA data architecture, it is useful to review the design goals of the data view. While the general C2STA capability goals apply to the data architecture, many data-specific goals are stated as well. The C2STA explicitly lists key properties of C2 data solutions [C2STA, 1998]. These properties include:

- Providing access to C2 data anytime from anywhere.
- Accommodating data from migrating systems.
- Properly handling data with varying security constraints.
- Making data available to arbitrary C2 capabilities and applications.

- Allowing for the sharing of data with external systems.
- Providing data consistently over a range of communications conditions.
- Exhibiting robustness in the face of changing applications and C2 capabilities.

The second goal listed above will receive the primary focus in the following evaluation of the C2STA data architecture. However, information interoperability is much more complicated than merely facilitating data communication. Existing technologies such as the open database connectivity (ODBC)[6] standard for relational database programming interface access and the structured query language (SQL)[7] standard for formulating relational database queries provide some compatibility for moving data between systems. The C2STA data architecture attempts to provide integrated C2 data access with a much higher level of interoperability. Thus, it is important to remember that accommodating data from migrating systems requires a more sophisticated design than one needed only to "ship" data from such systems to C2 capabilities.

This introduction to the C2STA data architecture makes the provisions for separating data access by C2 capabilities from the underlying data storage mechanisms clear. This separation was designed to provide a high level of information interoperability while achieving the design goals previously listed.

The C2STA Data View includes both general descriptions of the intended data architecture and specific requirements for systems adopting the architecture. This information is mixed together in the data view and extended (with some duplication) in the C2STA appendices. The following summary of the information is presented in a

[6] For information on ODBC see: http://www.microsoft.com/data/odbc/

[7] For information on the SQL standard see: http://www.jcc.com/SQLPages/jccs_sql.htm

different format. First, the general architecture is introduced and then specific C2STA requirements are listed. Further, the specific requirements are organized into groups that summarize their purposes. While the C2STA does this to some extent, many related requirements are addressed in separate sections of the data view discussion. For example, this summary addresses change notification requirements in one unified section whereas the C2STA data view mentions change notification requirements in two different sections as well as in an appendix. Grouping related requirements helps clarify the specific details of the C2STA data architecture and eliminate redundant requirements.

2.2 Data Access Interface + Modules + Implementations

The C2STA data architecture is defined in terms of three separated specifications. The highest-level concept is the Data Access Interface (DAI). The DAI abstractly represents the complete set of data available across all C2 systems. The DAI view of C2 data is object-oriented and ideally independent from the underlying data store technologies. Further, the DAI is intended to shield C2 capabilities from data location issues. However, the abstract nature of the DAI means that capabilities cannot actually rely on the DAI to handle the details of attaining C2 data. Clearly some rendezvous mechanism is needed to allow C2 capabilities to locate C2 data without depending on a particular location. Although the C2STA does not address this requirement, some of the technologies it suggests (e.g. CORBA) do include support for such service discovery.

Because the C2STA requires that all C2 data be accessed via interface definitions that are a part of the DAI, the DAI is a virtual super-database of all C2 data. However, it is important to understand that the DAI is completely abstract. The C2STA does not

require any physical instantiation of the DAI. While it would certainly seem useful to have some central repository of the interface definitions that make up the DAI this is not a requirement of the C2STA. Because system builders never have to explicitly implement the DAI it is easy to overlook this unifying feature of the C2STA data architecture. The most important point to remember is "to C2 capabilities, the DAI **is** the database." [C2STA, 1998] (Emphasis added.)

The second element in the C2 data architecture is basically a mechanism for partitioning the DAI. Although this partitioning is designed to occur in a logical sense, it also facilitates physical partitioning of the data stores to some degree. Data Access Interface Modules (DAIMs) represent subsets of the C2 data. Like the DAI, a DAIM is not a physical system. Instead it is an interface definition that precisely models some particular C2 data. DAIMs are supposed to be defined to either manage a particular class of C2 data or to provide access to data from a migrating system. Because the DAIM architecture does not require one-to-one mappings between DAIMs and underlying data stores or visa versa, DAIMs can group data with limited concern for the underlying data locations. However, there is a limit to this independence, particularly in the case of legacy systems, as object relationships within a DAIM will often correspond to foreign-key mappings in an underlying RDBMS. Because these mappings often do not correlate across different RDBMSs a single DAIM might not be able to link data from completely different sources. In such a case it would be more natural to have different DAIMs define the data contained in the different sources.

While the concept of a single comprehensive database unburdened by the details of persistent data store technologies and locations is very appealing, it is obvious that C2

data must at some point pass through real systems. The final element in the C2STA data architecture is the DAIM Implementation (DAIMI). As the name suggests, DAIMIs are real systems. A DAIMI supports a particular DAIM interface. It facilitates the actual transfer of C2 data in response to a request conforming to the DAIM interface specification. A DAIMI supports only one DAIM, however many different DAIMIs can support the same DAIM according to the C2STA.

The relationship between the three elements of the C2STA data architecture is easily summarized. A DAIMI concretely supports the abstract interfaces of a DAIM that comprises some subset of the C2 data defined by the DAI. Figure 2.2[8] makes this relationship explicit while highlighting some possible configurations for defining DAIMs

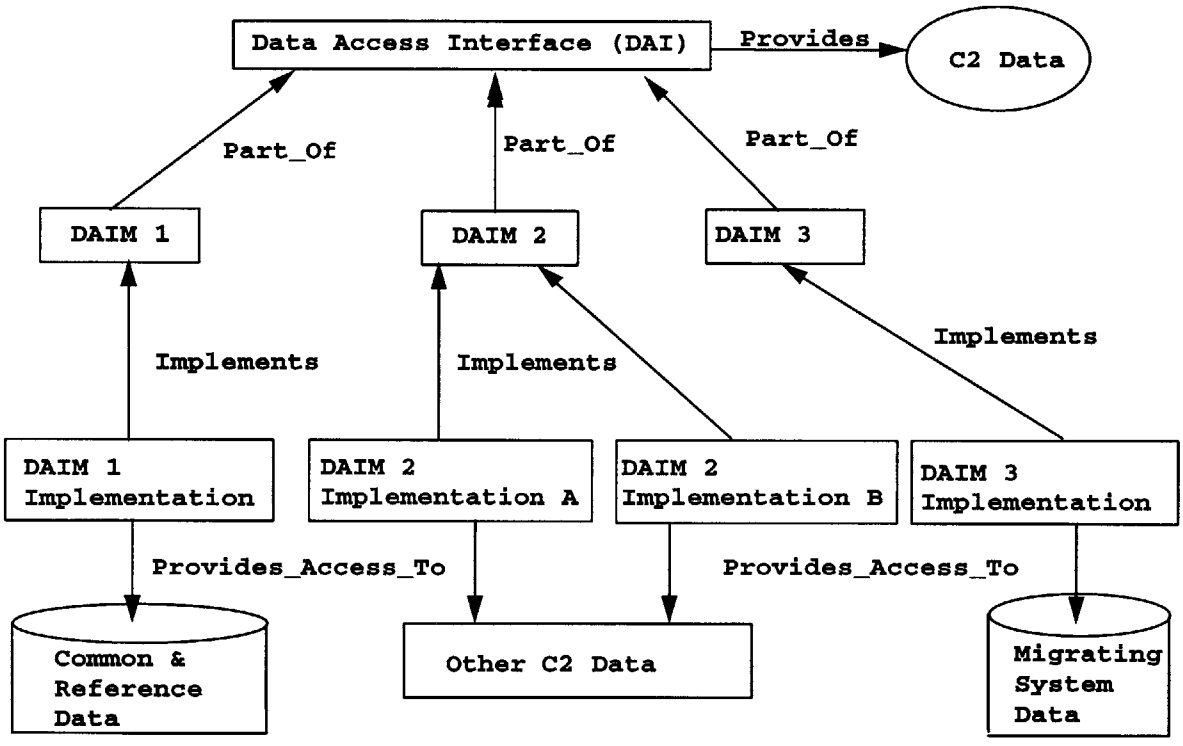

Figure 2.2 An Example C2STA data system highlighting the relationship between the DAI, DAIMs, and DAIMIs.

[8] Appears in the C2STA as Figure 4-3.

and building DAIMIs.

Having completed a general description of the C2STA data architecture, the following sections present the specific C2STA requirements for the various elements. These requirements will further detail the C2STA vision of a data solution. They will also further restrict the C2STA data architecture providing an opportunity for analysis of the architecture's feasibility (particularly for accessing data in legacy systems). Finally, by omission they provide an opportunity to compare the C2STA data architecture with other access models and suggest problems the C2STA overlooks.

2.3 C2STA DAI Requirements

The C2STA provides the following requirements regarding the DAI. Many of these requirements have been rephrased so their meaning remains clear absent their original context. It is also important to note that many of these requirements apply equally to DAIMs since the DAI is just the collection of all DAIMs. Often replacing "the DAI" with "a DAIM" will result in a requirement on DAIMs that preserves the original intent.

- C2 Systems will use the DAI for data access.
- The DAI will provide transparency of data location.
- The DAI view of C2 data shall be platform-independent and capability-independent.
- The DAI will allow integrated access to local and remote data.
- The DAI will isolate clients from the underlying storage details of data being accessed (e.g. schema).

- The DAI will be defined by the set of DAIMs defined for the Integrated C2 System (IC2S).
- The DAI will provide access to metadata about C2 data objects and their attributes.
- The DAI will be extensible.
- The DAI will maintain backward compatibility by defining new access methods and object rather than changing existing interfaces.
- Both the reading and writing of C2 data will occur through the DAI.

The three major goals of the DAI architecture are clear from the preceding requirements: (i) the DAI is the database (for both reading and writing) for C2 capabilities, (ii) the DAI is designed to shield clients from as many of the implementation details of actual data stores as possible (location, technology, etc.), and (iii) the DAI is extensible while maintaining backward compatibility. The second and fourth requirements listed above seem to overlap. Location transparency would seem to imply that local and remote data could be accessed in an integrated fashion. However, by listing both properties the C2STA makes it clear that DAI location independence is not simply a matter of masking some single location of data but rather a mechanism for allowing complex data replication and storage while hiding all such details from C2 capabilities.

2.4 C2STA DAIM Requirements

The majority of the C2STA data architecture's requirements apply directly to DAIMs. This is natural as the C2STA deliberately avoids detailing DAIMIs as much as

possible. The DAIM requirements cover a range of issues from basic DAIM design to specific functionalities such as change notification and finally formal interface requirements. General DAIM issues are addressed first and then separate sections focus on major design sub-categories.

- A DAIM shall present abstract data types (ADTs) to other components.
- The abstract data objects provided by a DAIM have well defined semantics agreed upon across C2 domains.
- A DAIM should provide "a complete model of the C2 data." This should include both persistent attributes as well as other "inherent attributes" of an object that can be calculated from persistent values.
- Structured data elements within an ADT shall be available via abstract references to the embedded ADT via the appropriate DAIMI, etc.
- All data access operations shall be atomic.
- DAIMs shall operate in either implicitly persistent mode or explicitly persistent mode.
- A DAIM shall be constructed such that capabilities can be completely ignorant regarding the details of the DAIMI they access.
- DAIMs should embody knowledge of data store replication used for data requiring high timeliness or static data.
- Migrating systems may require a "wrapper" DAIM.
- A DAIM may use the services of another DAIM or access data from multiple underlying data stores as necessary.

- Multiple DAIMs may access the same data store if the data stores hold various classes of data.

The second requirement that ADTs have semantics agreed upon across C2 domains helps maintain the interoperability of C2 capabilities. However, this same requirement means that constructing "wrapper" DAIMIs might require a lot more effort than simply layering an object model on top of an existing relational data model. It is likely that legacy systems will already include their own relational models of data. These legacy models could easily have overlapping data without any general agreement on semantics, structure, etc. DAIMs defined to access such systems would have to translate from the legacy model to the ADTs. The CDE experiments highlight some of the challenges in realizing such goals.

The requirement that DAIMs include access to both stored data and "inherent" calculated attributes based on persistent data is designed to provide flexible data access to C2 capabilities. However, it will later become clear that the C2STA does not reconcile some consequences of this requirement with its data access interfaces.

The statement that migrating systems might require wrapper DAIMs directly motivates the CDE experiments. However, observations from these experiments highlight incompatibilities between C2STA requirements and constraints of building wrapper DAIMIs.

The requirement that DAIMs support one of two modes of persistence requires some explanation. In implicitly persistent mode all instance creation operations and attribute modifications are immediately made persistent. In explicitly persistent mode creation operations and attribute modifications are temporary until the temporary instance

is explicitly made persistent at which time all temporary changes are simultaneously made persistent.

Several categories of additional requirements follow the previous examination of some of the general DAIM requirements. Some of these categories address high-level requirements such as what kind of information a DAIM should supply. Other categories are much more specific and require DAIMs to support specific interfaces.

2.4.1 DAIMs & Business Rules

The following rules help clarify the multi-tiered architecture envisioned by the C2STA. Because traditional RDBMSs often provide support for stored-procedures, triggers, and other programmable functions they act as more than passive persistent storage mechanisms. It is possible to include business rules and other application level functionality inside the database. However, this is not the design advocated by the C2STA. Instead, the C2STA separates business rules and application logic from the data store technology. The following specific requirements clarify this intent.

- A DAIM should not include operations that embody business rules that are not a fundamental part of the data object.
- DAIM operations can embody mission-independent business rules inherent to a data object. These rules may be instantiated in a DAIMI or in the underlying data store. (e.g. range constraints, etc.)
- DAIMs are responsible for maintaining the integrity of their persistent data.

Together these three requirements make it clear that DAIMs should not embody business rules but they are not passive storage locations. They are required to validate

data. The C2STA identifies a general guideline that data integrity rules should be instantiated as close to the persistent store as possible. This implies defining integrity rules at the data store level rather than the DAIMI level. However, this is only a guideline and not a rule. While separating business rules from DAIMs is a commonly accepted practice for building component systems with good encapsulation properties, legacy systems might have to violate these requirements. Their business rules are often instantiated inside the underlying DBMS (e.g. in the form of triggers, stored procedures, etc.) and any DAIMI wrapper would have to expose operations subject to the underlying business rules. The C2STA does not address this issue.

2.4.2 DAIMs & Metadata

The C2STA identifies metadata as an important aspect of its data architecture. Metadata is literally data about data. For example, a data element representing velocity might have metadata describing the units and precision. Metadata is a more important issue when supporting data interoperability between systems than it is when considering the operation of a single system. This is not because a single system does not care about metadata but because metadata within a system might be implicit. For example, a single system might always assume that velocity is measured in miles per hour. Data interoperability between systems is very dependent on metadata because explicit metadata helps guarantee the correct interpretation of data. A recent failure of a NASA Mars mission was attributed to a data interoperability failure. Two interoperating systems failed to account for differences in units of measurement causing the complete failure of a one hundred and twenty-five million dollar mission [Isbell et al., 1999].

Explicit metadata could have made the need for a data conversion apparent. The following requirements clarify the C2STA metadata requirements.

- DAIMs shall include attributes that describe other attributes (metadata) "when feasible."
- DAIMs shall allow metadata to be either ADT-wide or instance specific as appropriate.
- DAIMs shall handle metadata as additional attributes of an ADT.

2.4.3 DAIM Containers

Given the C2STA's object-oriented nature it is natural for sets of objects to be managed by some generic container object. The C2STA explicitly provides for this by requiring a Container ADT. The following requirements detail mandatory container behavior.

- DAIMs shall define a generic Container ADT designed to hold references to other ADTs.
- The Container shall operate in non-persistent mode and shall support a call to make itself persistent.
- A Container can be processed by the standard iterator mechanisms first, next, and last.
- A Container may be empty.
- A Container may be enhanced in arbitrary ways to support retrieval of ADT instances based on type-specific selection criteria.

The second and fifth requirements from above deserve special attention. The requirement that containers support persistence is somewhat vague. Does this mean that the container persistently stores all of its references as well? Or is the requirement aimed at storing any selection criteria associated with the container? These two points are not critical from a high-level design perspective, but a DAIMI would have to make explicit assumptions regarding container semantics. The allowance for arbitrary enhancements to support selection criteria for ADTs is more interesting. The C2STA makes no other reference to data search mechanisms in its data architecture. This is a major oversight. The C2STA advocate using DAIMs to hide underlying data store technologies, however a large part of the functionality of current data stores such as RDBMSs is to provide powerful search capabilities such as ad hoc querying via the SQL standard. If DAIMs hide this power without providing a reasonable search mechanism of their own they might force higher-level components to perform excessive data filtering. This also increases the amount of data being transported between systems and thus increases communication costs. The C2STA probably included the allowance for ADT selection criteria for precisely this reason. However, its failure to suggest even basic architectural support for such selection criteria (e.g. How would one specify the criteria? Hopefully not in a "SELECT * FROM ..." format!) could result in different DAIMs introducing radically different selection mechanisms. Such differences could increase a given capability's dependence on specific DAIMs and reduce the potential for future interoperability with different DAIMs.

2.4.4 DAIM Exceptions

The C2STA requires the DAIMs define exceptions for dealing with error conditions. The following requirements often refer to DAIMIs rather than DAIMs because DAIMs define things whereas DAIMIs do things. Nonetheless the following statements essentially constrain the definition of DAIMs.

- DAIMIs shall raise an exception if a caller supplies inappropriate selection criteria.
- DAIMIs shall raise an exception if a requested attribute is inappropriate for a specified ADT.
- DAIMIs shall raise an exception if the requested data are not available.
- DAIMIs shall not provided detailed exception explanations that violate security purposes.
- DAIMIs will supply a "no data available" exception when communication links are unavailable.
- DAIMIs shall raise an exception if the DAIMI is unable to complete an operation due to any error condition other than those already listed.
- DAIMs may define additional exceptions.

Most of these points are fairly straightforward. However, the purpose of the second requirement is somewhat unclear. Typically an object oriented programming environment tracks the types of object references and only allows calls relating to methods and attributes defined for that type. This is known as strong typing. However, some mechanisms such as casting can result in incorrect type classification. Perhaps the C2STA was trying to address this possibility.

2.4.5 Required Interfaces

Because the C2STA does not mandate a specific technology for DAIMs it cannot define interface requirements in terms of programming language specific specifications (e.g. C++ header files or Java class definitions). Instead it gives a very high-level interface definition. The following two statements require two different interfaces addressed in the next two subsections.

- A DAIM is a specific type of C2 capability and as such must include a general capability interface (GCI).
- Every DAIM will define a functional interface conforming to C2STA requirements.

2.4.5.1 General Capability Interface

The General Capability Interface (GCI) is a required feature of every C2STA-compliant capability [C2STA, 1998]. The interface is required to allow automatic run-time discovery of the IC2S capabilities available in a computing environment.

- The GCI shall be defined in IDL.
- The GCI shall be implemented via either COM[9] or CORBA[10] technologies.
- The GCI must be accessible to software programmed in various languages. (Ada and Java are given as specific examples.)

[9] For information on COM see: http://www.microsoft.com/com/

[10] For information on CORBA see: http://www.corba.org/

- The GCI must be usable by newly developed, migrating, and off-the-shelf software. This means it must support a variety of access methods including wrappers, proxies, and adapters.
- The GCI must include the following capability attributes: resources, interfaces, distribution mechanism, and execution environment.

The GCI is important for the overall C2STA to ensure that a generic mechanism exists for examining an unknown capability. It is important to DAIM designers because they must support it but it does not have a significant impact on the C2STA data architecture.

2.4.5.2 Abstract Data Type Access Interface

The C2STA defines three categories of data access interface requirements. The first category applies to all ADTs and specifies method for getting containers with ADT references and specific attribute values. The second and third categories define separate interfaces requirements for creating and modifying ADTs depending on the persistence mode of the DAIM.[11] A listing of all three categories will be followed by comments on the implications of these requirements.

All DAIMs shall include the following operations for each ADT <Typename> and ADT attribute <Attributename>:

- Get<Typename>
  - Purpose: Retrieve instances of a particular ADT.
  - Arguments: None.
  - Returns: Container[12] referencing all instances of that ADT.

[11] See Section 2.4.

[12] See Section 2.4.3.

- <Attributename>Of
  - Purpose: Retrieve a specific attribute of a specific ADT instance.
  - Arguments: ADT instance being examined.
  - Returns: The value (either a reference to another ADT or a primitive value) associated with this attribute for the specified ADT instance.

- Delete<Typename>
  - Purpose: Remove a specific ADT instance from persistent storage.
  - Arguments: ADT instance to delete.
  - Returns: Unspecified, probably a status flag representing the success or failure of the operation.

All DAIMS providing explicit persistence shall include the following operations for each ADT <Typename> and ADT attribute <Attributename>:

- New<Typename>
  - Purpose: Create new ADT instances.
  - Arguments: None.
  - Returns: Reference to a new non-persistent instance of the ADT with all attributes set to their default values.

- Define<Attributename>
  - Purpose: Set the value of an ADT instance's attribute.
  - Arguments: The instance being modified and the new value (either a reference to another ADT or a primitive value).
  - Returns: Unspecified, probably a status flag representing the success or failure of the operation.

- Set<Typename>
  - Purpose: Makes non-persistent New and Define calls persistent. Either updates the persistent state of the given instance if one exists or creates a persistent state for the instance. Either way, after the call the persistent state will completely match the state of the non-persistent instance referenced in the call.
  - Arguments: The non-persistent instance to make persistent.
  - Returns: Unspecified, probably a status flag representing the success or failure of the operation.

All DAIMS providing implicit persistence shall include the following operations for each ADT <Typename> and ADT attribute <Attributename>:

- New<Typename>

    - Purpose: Create new ADT instances.
    - Arguments: None.
    - Returns: Reference to a new persistent instance of the ADT with all attributes set to their default values.

- Define<Attributename>
    - Purpose: Set the value of an ADT instance's attribute.
    - Arguments: The instance being modified and the new value (either a reference to another ADT or a primitive value).
    - Returns: Unspecified, probably a status flag representing the success or failure of the operation.

Despite the object oriented nature of the C2STA design, its required interfaces are described in a functional manner. For example, image an Aircraft ADT with a TailNumber attribute. The C2STA requires that a DAIM support a GetAircraft operation to return a Container with references to all of the Aircraft managed by the DAIM. Further the DAIM must support a TailNumberOf operation and a DeleteAircraft operation both of which take a single Aircraft reference as an argument. A typical object oriented interface would make TailNumber an attribute of an Aircraft and Delete a method of an Aircraft. This eliminates the need for passing an Aircraft's reference as an explicit argument. Of course the underlying system still has to manage object references. However, it is surprising that the C2STA did not specify its requirements in such terms given that it specifically requires IDL definitions backed by COM or CORBA implementations.

Setting aside the specific format of C2STA required operations the general access paradigm is fairly restricted as well. The C2STA defines a single constructor for objects that takes no arguments and initializes the new object with some default values. It then dictates that all data reads and writes go through a basic get/set interface (the actual calls are AttributeOf and Define, however get and set capture the semantics of the operations

in simpler terms). The C2STA seems to imply that every attribute should support get and set methods. This seems natural given the requirement that objects have default values at creation-time. Without a set operation an attribute could never change from its default value. Unfortunately this does not account for some important cases. First, the C2STA requires that DAIMs expose both persistent data of an ADT as well as "inherent" attributes. An example from the C2STA helps clarify this requirement and the problem it introduces. Assume a Mission ADT persistently stores a StartTime and an EndTime attribute. The C2STA mentions that the Mission ADT should also provide a Duration attribute calculated from the StartTime and EndTime. However, the interface requirements presented here seem to require that Duration have both a get and a set. It is unclear that all calculated attributes should have a set operation. While providing a get operation in such cases might be convenient for some capabilities accessing a DAIM, a set does not have clear semantics. Should the StartTime or the EndTime be altered when a capabilities sets the duration? Clearly the DAIM must enforce the constraint that Duration = EndTime – StartTime. Requiring symmetric get and set operations in this case is a mistake.

The constructor, get, and set operations introduce problems with regard to legacy systems. A constructor will often require multiple arguments that are used to create a new object. This allows the constructor to perform data integrity operations considering several factors simultaneously. The object then need not provide set operations for all of its attributes. This is a reality of many object oriented designs. This reality extends to other designs as well. Consider relational database systems based on the SQL standard. According to the standard, new rows can be inserted into a table using an INSERT

INTO... statement. Subsequently any field in that row can be modified using an UPDATE ...WHERE statement (the row must have a unique identifier to guarantee a single update to the correct data element). SELECT statements can be used to read any field in the row. In this case INSERT corresponds to the object constructor, UPDATE to the set operation, and SELECT to the get operation. In theory using these statements on individual data elements is sufficient for performing all persistent storage tasks. However, a reality of many systems based on relational databases is that applications using the system never execute INSERT statements on base tables. Instead, a stored procedure (acting very much like an object constructor) performs some operations and inserts the row. With this kind of an interface applications do not always need UPDATE access to every field in a table. The C2STA provides no provisions for wrapping such systems with DAIMs. A DAIMI could not provide the data access semantics required by the C2STA without bypassing the existing data store restrictions and thus fundamentally impacting the existing system.

2.4.6 Change Notification Service

The basic C2STA data access paradigm is request and response. However, other existing initiatives in the Air Force have recognized the value of alternative data transport paradigms such as publish/subscribe or push. Such paradigms can substantially improve data interoperability as they can eliminate the need for specific pair wise configuration support. The C2STA recognizes the value of such data transport mechanisms. As such it requires DAIMs to define a basic change notification service. Such a service provides a basis for building more sophisticated systems such as those supporting publish/subscribe.

2.4.6.1 General Functional Requirements

The following requirements detail the functionality required of a DAIM change notification service. Detailed interface descriptions appear in the next section.

- A DAIM will provide a change notification services.
- The notification service shall alert "interested" capabilities that an element of data has changed.
- A DAIM shall include a registration service allowing capability to express their interest in notifications about a particular data element.
- A DAIM's registration service shall support interest to be specified in the following categories:
    - Any change to any instance
    - Any change to a specific instance
    - Any change to a specific attribute of a specific instance
    - Any change to a specific attribute of any instance

- A DAIM's registration service shall require a client to supply a callback interface when registering an interest.
- The notification service shall notify the callback of the type of change (create, update, delete), the attribute(s) that changed, and the old and new value(s).
- The DAIM shall define a Notification ADT that represents a client's notification registration.
- The DAIM shall define an Interest ADT that represents a client's interest in data changes as described before (e.g. interest in the change of a specific attribute of a specific instance).

The C2STA change notification requirements introduce a constraint on data access interfaces in the C2STA. Although DAIMI requirements specifically allow for multiple interfaces to C2 data[13], the change notification requirement dictates that all interfaces must use some common mechanism for change notification purposes. While this is not explicitly mentioned, it is necessary to guarantee that clients of one interface are notified of changes made through a different interface. The only alternative is frequent polling of data values to check for changes. This is impractical due to the enormous overhead it would place on the data store.

Requiring that all data access interfaces using a single data store share some change notification mechanism dictates that either the data store provide the notification service or that all data access interfaces are themselves clients of some layer between the DAIMI and the data store. These restrictions are particularly severe for migrating systems. Because existing applications use an existing data access interface, the C2STA

[13] See Section 2.5.

cannot require that they go through a DAIMI. Therefore consistent change notification is only possible if the change notification service is below both the DAIMI and the existing data access interface. This essentially means that the data store must support change notification. If this is not the case it is impossible to support the required C2STA change notification services using wrapper DAIMs and DAIMIs without changing the existing system.

2.4.6.2 Required Change Notification Interface

The C2STA requires the following change notification operations:

- RegisterInterest
  - Purpose: To associate a Notification instance, an Interest instance, and a client callback interface in the notification service's internal state. Implicitly calls TurnOnNotification for the supplied Notification instance as well.
  - Arguments: A reference to a new Notification ADT, A reference to the Interest instance being registered, and a reference to the client's callback interface.
  - Returns: Unspecified, probably a status flag representing the success or failure of the operation.

- UnregisterInterest
  - Purpose: Undoes a RegisterInterest call. The association is deleted from the notification service's state.
  - Arguments: A reference to a Notification instance.
  - Returns: Unspecified, probably a status flag representing the success or failure of the operation.

- TurnOffNotification
  - Purpose: Temporarily disables a Notification. The registration is maintained but the client callback is not invoked even if a change matching the Notification's Interest occurs.
  - Arguments: A reference to a Notification instance.
  - Returns: Unspecified, probably a status flag representing the success or failure of the operation.

- TurnOnNotification
  - Purpose: Re-enable a Notification after a call to TurnOffNotification.

- Arguments: A reference to a Notification instance.
- Returns: Unspecified, probably a status flag representing the success or failure of the operation.

- SetCallback
  - Purpose: Allow a client to change the callback interface associated with an already-registered Notification.
  - Arguments: A reference to a Notification interest and a reference to the new client callback interface.
  - Returns: Unspecified, probably a status flag representing the success or failure of the operation.

Further, the C2STA requires the following behavior with regard to notification:

- A DAIM shall invoke a client's callback interface by calling that interface with a reference to the particular Notification instance that resulted in this callback and a indication of the type of change as previously specified.

None of the requirements introduced in this section warrant additional comments. They merely make concrete the generic service already discussed.

2.5 C2STA DAIMI Requirements

The following requirements specifically address DAIMIs. There are very few requirements in this category precisely because the C2STA was not intended to deal with implementation issues.

- The C2STA places no restrictions on the technologies used for building DAIMIs.
- The GCI and functional interfaces of a DAIM will be accessible via a CORBA or COM implementation.
- A DAIMI may provide alternative access implementations as well.

The third point deserves discussion. The C2STA specifically allows DAIMIs to support interfaces other than COM/CORBA. It suggests that such a decision might be

based on performance considerations. However, Figure 2.3[14] makes it very clear that such access interfaces are still required to implement the DAIM data access operations. As such this flexibility does not account for the case of migrating systems where a DAIM is defined and a DAIMI implemented by legacy parts of the system still use an alternative access method completely independent of the DAIM. As already mentioned, this situation causes problems for change notification services as well as complicating the issue of how to expose the business rules of the original system.

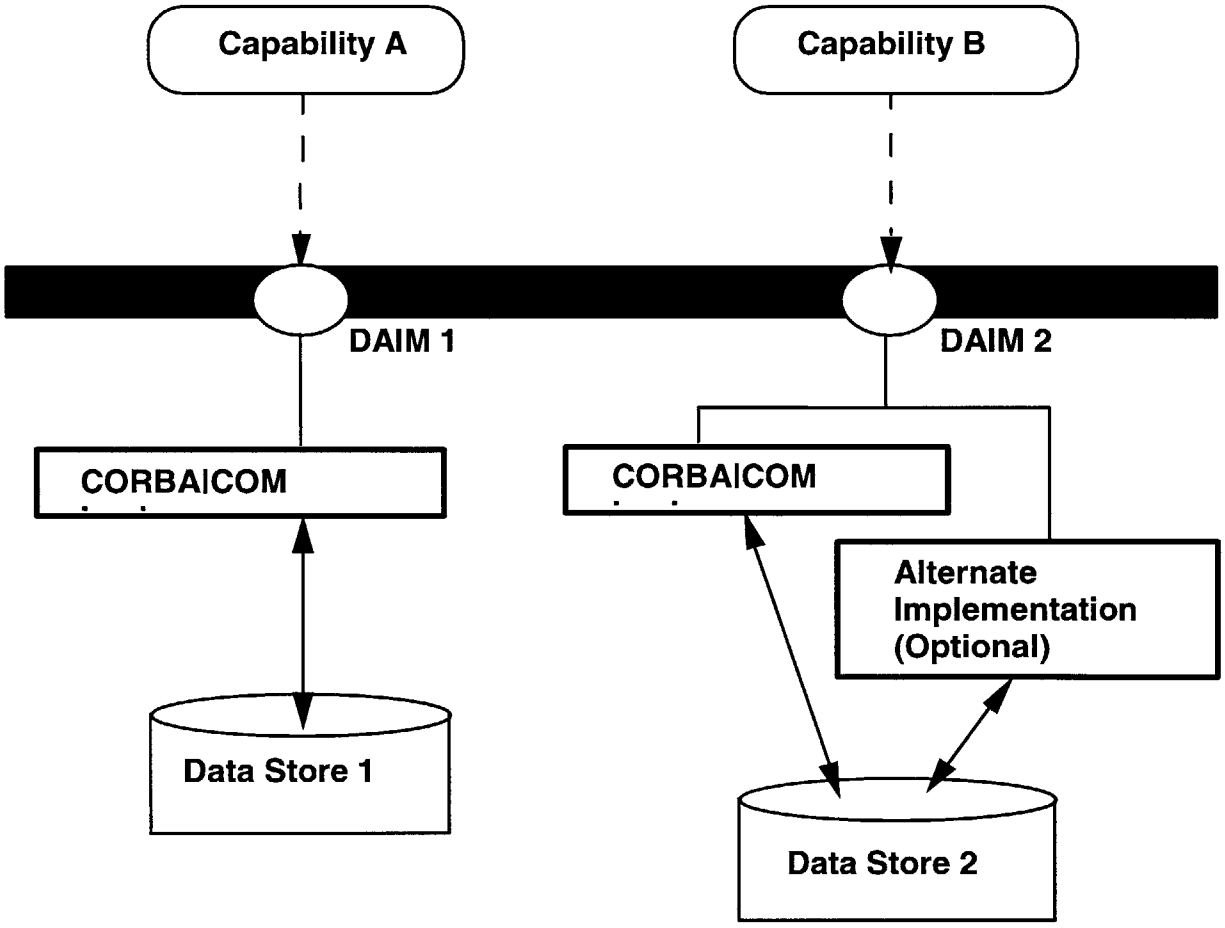

Figure 2.3 DAIMI Multiple Interface Support

[14] Appears in the C2STA as Figure 4-1.

Chapter 3

CDE Scheduler Experiments

The previous two chapters introduced the data interoperability problem and the C2STA data architecture. This discussion repeatedly referred to the data interoperability issues surrounding legacy systems and the possible application of the C2STA data architecture to those problems. This chapter provides a detailed explanation of CDE exploration of some of those issues.

The CDE Office of the Air Force Electronic Systems Center is testing the use of commercial middleware products to improve data interoperability among Air Force C2 Systems. In particular the CDE Office wants to understand how commercial products can improve inter-system data flow among systems using DAIMs, as described in Chapter 2. Basic DAIM functionality is summarized below. This description is consistent with the full DAIM architecture. However, it is important to note that the scheduler experiments never attempted to implement a fully C2STA-compliant DAIMI. In fact, many of the previous observations regarding the implications of C2STA requirements were motivated by a re-examination of the C2STA data architecture **after** the completion of these experiments. Therefore the following explanation of DAIMs and DAIMIs serves more as a clear description of the data architecture underlying the CDE experiments rather than a clear description of the full C2STA data architecture.

A DAIM defines an object interface to data in the form of abstract data types (ADTs). This provides a specific interface for an application to access the data it needs. ADTs are designed with application needs in mind rather than physical data storage constraints. Thus, a DAIM with its abstract object model stands architecturally between an application and one or more databases, which may be of different types and have different structures. The DAIM decouples the application from changes in the underlying data infrastructure. The C2STA dictates that DAIMs expose their ADTs via either CORBA or COM interfaces. A CORBA-based DAIM defines its ADTs using the CORBA Interface Definition Language (IDL). These definitions completely specify the DAIM. However, they do not indicate how data will actually be retrieved, processed, etc.

The task of retrieving data from a stable data store, manipulating them as necessary, and making them accessible through the ADT interfaces described by a DAIM falls on a DAIMI. A DAIMI is the actual system that a client application uses to make calls to a particular DAIM specification. The separation between a DAIM's specifications and a DAIMI's method of fulfilling those specifications allows client applications and underlying data sources to evolve independently. Changes to underlying data sources should not affect the DAIM definitions at all. Instead, changes would need to be made to the DAIMI layered above the data source so that it could continue to retrieve and process data properly in response to clients accessing ADTs.

The DAIM concept offers potential benefits to those solving interoperability problems between legacy systems as well as those implementing new data stores and client applications. Implementing DAIM technology to solve legacy data interoperability problems allows interoperability to be achieved by agreeing on some common set of

ADTs needed for communication between systems. Once these ADTs are defined, DAIMIs must be built to map the actual data stores of each system to the ADTs. While this does require that each system's data model be mapped to the ADTs, it alleviates the need for mapping a system's data model directly to that of another system. This is particularly beneficial when there is the possibility of more than two systems interoperating. If N systems need to interoperate, direct mapping between their various data models could requires $O(N^2)$ different mapping definitions. By agreeing on common ADTs for the interoperation, the number of mappings needed is reduced to O(N). Further, a change in one data model requires redefining only one mapping in this situation rather than redefining O(N) mappings in the previous situation.

The use of the DAIM architecture can also benefit interoperability in new systems. By writing new client programs to access their data through a DAIM, system designers can implement the underlying data store using whatever model and technology offers the greatest benefits as long as it can be mapped to the DAIM's ADTs. Additionally, a single client program can potentially access data from a variety of data sources with no dependence on their specific implementations. Finally, a DAIM layered above a new data store can potentially support other high level applications developed completely independently from that data store (e.g. a generic data browsing and indexing program).

Although the DAIM architecture offers many potential interoperability benefits, it must compete with other potential solutions such as those described in Chapter 1. In light of this competition, the DAIM architecture must prove some competitive advantage at least for some subset of interoperability problem if it is to achieve widespread use. One

critical measure of DAIM viability is the cost of development. While DAIMIs can obviously be custom programmed for specific data sources (and this might be required for some situations), this approach will likely prove expensive in both initial development and long-term maintenance costs. However, if commercially available middleware could be used for DAIMI development, the cost might be substantially reduced.

The CDE scheduler experiment was an initial investigation into the suitability of commercial middleware to DAIMI development. In particular, the project was interested in demonstrating the feasibility of providing read/write access to multiple legacy C2 databases using a commercial middleware based DAIMI. Such a capability could greatly advance the ability to provide linkage to a broad set of data sources to support future integrated Air Force systems.

This experiment provided insight into applying a commercial middleware product, Virtual DB from Enterworks[15], in a scenario involving the scheduling of air missions. This insight helps answer questions about building DAIMIs and motivates question regarding the C2STA data architecture as a whole.

3.1 Test Scenario

The test scenario for this evaluation was a continuation of earlier DAIM experiments performed by MITRE. The goal was to allow a generic scheduling application developed at MITRE to schedule air missions in two independent C2 databases. The following C2 databases were selected for the trial:

[15] It appears that Virtual DB has been renamed to Enterworks Content Integrator. The CDE experiments dealt exclusively with Virtual DB 3.0. All discussion regarding Virtual DB applies only to this version. For more information on the Enterworks Content Integrator see: http://www.enterworks.com/products/index.html

- Command and Control Information Processing System (C2IPS), which manages the flights of transportation aircraft worldwide.
- Theater Battle Management Control System (TBMCS), which controls air operations in a combat theater.

The C2IPS and TBMCS systems have many common functions, such as receiving flight requests; assigning aircraft and crews to missions; scheduling takeoffs and landings; tracking missions, and arranging for in-air refueling. Because there is already a need for flight information exchange between these systems, demonstrating that a single application can access flight data from both systems without knowing anything about their individual data models provides an excellent opportunity to evaluate the potential for solving data interoperability problems using the DAIM architecture.

A demonstration of this kind of interoperability is far from trivial. While the databases supporting these two systems contain some similar data, they have different table structures, inconsistent naming conventions, and incongruent data types. This is a challenging test of the ability of a DAIM to provide an abstract object model that can be mapped to both physical data models in a meaningful manner.

3.2 Preliminary MITRE Experiment

The previous iteration of this experiment also used the MITRE scheduler application to access C2IPS and TBMCS flight data. The scheduler application was written to access mission data through C++ classes that defined the ADTs for the DAIM. The DAIMI consisted of Windows dynamic-link libraries (DLLs) that actually retrieved

data from TBMCS and C2IPS databases and exposed it via the C++ ADTs. These DLLs were created by code-generators developed at MITRE. The generators took as input files specifying the mapping of database table and column names to ADT attributes. They output C++ code capable of performing most of the data translation and mapping required by the DAIMI. Specific translations not handled by the code generators had to be manually coded. Finally, the code was compiled to provide the DLL DAIMI.

The code generators demonstrated that it was possible to apply the DAIM architecture to the test scenario. However, the generators had to be developed in-house by MITRE and thus could not achieve the cost effectiveness of commercial-off-the-shelf (COTS) tools. Further, the DLLs were highly dependent on the underlying database technologies. For example, the preliminary tests used C2IPS and TBMCS data stored in Microsoft Access databases. Changing the scenario to include the databases on Oracle RDBMS servers (as they would likely be in actual operation environments) required changes to the code generators and recompiling the DLL DAIMIs. On the other hand, the custom code generated for each DAIMI allowed a great deal of flexibility in manually adding additional code for specialized data mapping problems.

The CDE experiment was designed to duplicate the benefits of the code-generated DAIMIs while demonstrating the suitability of commercial middleware to the DAIM architecture. Further, this experiment implemented the CORBA server based DAIM interface specified by the C2STA. This would allow the DAIMI to operate as an ADT server independent from individual client applications.[16]

[16] This contrasts with the DLL DAIMIs which essentially operate as a part of an individual client application.

3.3 Virtual DB DAIMI Architecture

The C2STA DAIM architecture specifies the operations a DAIM interface should provide. However, the particular design used for building DAIMIs is left unconstrained. The scheduling scenario for the Virtual DB experiment dictated that the DAIMI had to access C2IPS and TBMCS data stored on Oracle servers and had to provide clients with access to CORBA objects as specified by the ADTs defined in the earlier scheduler experiment and coded into the scheduler application.

The Virtual DB server is capable of accessing data from a wide-range of underlying sources. It can then perform data translations and renaming as well as combining data from multiple sources. This allows for the creation of an abstract data model that clients can access for both reading and writing.[17]

Because Virtual DB provides access to an abstract relational data model, it cannot stand as a complete DAIMI on its own. The C2STA DAIM architecture requires that a DAIM provide CORBA or COM objects for its ADTs. However, this does not undermine Virtual DB's utility as part of a DAIMI architecture. Building a DAIMI using Virtual DB requires an additional layer that translates Virtual DB's abstract data model into CORBA objects representing the DAIM ADTs. The DAIMI architecture designed for this experiment included a "VDBtoADT" server to provide this functionality. Ideally, a COTS based DAIMI would use commercial middleware for this layer as well.

[17] It is important to note that writes can only be performed on views that map to a single data server. While this constrains some types of write access, it does not affect the scheduling scenario where the client must to read data from two sources but always writes changes back to the data's original source. Although writes to views that provide data from multiple servers are theoretically possible, they require a complicated two-phase commit protocol that Virtual DB does not support.

However, no commercial products that fulfilled these needs were identified so a prototype VDBtoADT server was implemented in Java.

The Virtual DB + VDBtoADT architecture described is capable of supporting the C2STA DAIM architecture. Although a completely COTS based architecture would have been preferred, this solution offered significant reductions in development costs when compared to a completely custom programmed DAIMI. Further, examination of this architecture provided an opportunity to assess the applicability of COTS middleware to DAIMIs. Because Virtual DB stands as one type of interoperability solution on its own, it provided an opportunity to critique the C2STA DAIM architecture as well.

The remainder of this chapter describes details of the Virtual DB DAIMI implementation completed for this experiment, the CDE Datalab configuration, observations regarding the applicability of Virtual DB to DAIMIs, and observations regarding the C2STA DAIM architecture as a whole.

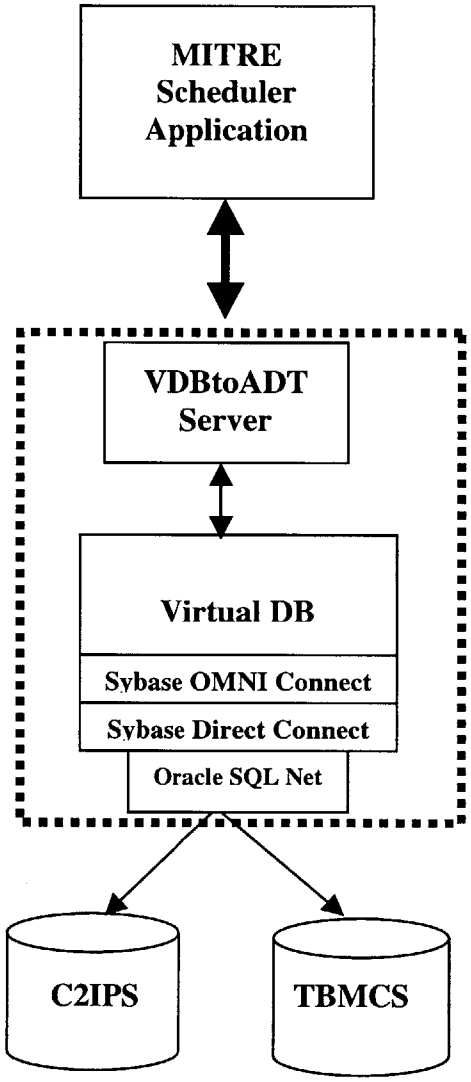

Figure 3.1 Virtual DB DAIMI Architecture

3.4 The Test Environment

The Virtual DB based DAIMI has four distinct software layers. These are the VDB to ADT translator, the Virtual DB server itself, the Sybase Adaptive Server (OMNI), and finally the Sybase Direct Connect for Oracle server. The complete test environment requires the Scheduler client program and two Oracle databases as well. The maximally distributed client server setup could actually include as many as eight separate machines. However, this degree of distribution did not seem realistic or particularly useful to test. Instead, the target test environment consisted of an Oracle server hosting both the C2IPS and TBMCS databases, a DAIM server hosting all of the VDB DAIMI services, and a client machine running the Scheduler application. However, experiments to date have not used this precise configuration. The next section describes the actual configurations tested as well as the configurations planned for further analysis.

The C2IPS and TBMCS databases were initially installed on two separate Sun SparcStations (maddog.mitre.org and krishna.mitre.org) physically located at the MITRE A building. Both of SparcStations ran Oracle 7.x database servers. This configuration provided an excellent model of the distributed data stores likely to be encountered when installing DAIMIs in real operating environments. However, this configuration complicated VDB DAIMI testing because the CDE Datalab resides at Hanscom Air Force Base in Building 1600. Network connections from Hanscom to the MITRE complex require special TCP/IP tunneling software that proved unreliable on the test machines. Further, the Oracle databases running on maddog.mitre.org and

krishna.mitre.org support other critical applications and were therefore less than ideal for experimentation purposes.

The initial Virtual DB server for DAIMI testing was configured on a Sun Sparc 10 (guinness.mitre.org) in Building 1624 at Hanscom Air Force Base. Unlike the current CDE Datalab, this location had a direct connection to the MITRE complex network. This avoided potential problems with the VTCP tunneling software. Guinness.mitre.org was configured with version 3.0 of the Virtual DB software as well as the Sybase OMNI server and the Sybase DirectConnect for Oracle server required by Virtual DB. These products were configured to access the C2IPS and TBMCS databases running on krishna.mitre.org and maddog.mitre.org. However, performance of the Virtual DB server running on guinness.mitre.org was unacceptably slow. Response times were so bad that basic data modeling tasks completed by the Virtual DB client software often failed due to server timeouts. For example, defining a single view attribute took over thirty seconds to complete. Worse yet, because of the delay, the Virtual DB GUI timeout and refused to display the new attribute even after the update occurred. This required a reloading of the GUI and re-logging on to the Virtual DB server (a several minute task in itself). Consultation with Enterworks contacts (who initially configured guinness.mitre.org as a Virtual DB server) suggested that guinness.mitre.org was underpowered for the services it was running and possibility configured incorrectly as well. Due to the slow response time of guinness.mitre.org as well as its physical separation from the Datalab, Virtual DB was installed on a Pentium III based Windows NT workstation in the Datalab configured with 128 MB of RAM. For the short-term another Datalab Windows NT workstation hosted Oracle8i instances of C2IPS and TBMCS to support development work in place of

krishna.mitre.org and maddog.mitre.org.[18] This configuration was more than adequate for the small datasets used in this experiment.

3.5 Building The Virtual DB DAIMI

Constructing the Virtual DB DAIMI involved several discrete tasks. The following subsections explain each task, the challenges encountered, and lessons learned.

3.5.1 Configuring Sybase

Virtual DB relies on the Sybase Adaptive Server Enterprise (also referred to as OMNI) product to access underlying data sources. Thus, Sybase Adaptive Server Enterprise in conjunction with Sybase DirectConnect for Oracle had to be configured to access the Oracle instances hosting C2IPS and TBMCS.

While the Sybase products provide the ability to access over twenty kinds of data stores, their installation and configuration is far from trivial. Experience installing and testing the Sybase products required by Virtual DB indicates a need for TCP/IP expertise, Oracle SQLNet experience, and ideally experience with Sybase Adaptive Server Enterprise.

3.5.2 Configuring VDB

Once the Sybase server products were configured to provide access to the Oracle instances, the next requirement was to configure Virtual DB to access the data through

[18] A Sun UltraSPARC hosting Oracle8i has been configured for future Datalab tests. Both the C2IPS and TBMCS databases will be hosted on this server, although they will use separate Oracle instances. This will provide a simple simulation of two separate databases without the excessive hardware and software requirements of actually hosting them independently.

Sybase. Fortunately, this step in the configuration was straightforward and closely followed the Virtual DB documentation. Again, TCP/IP experience is beneficial. Finally, because Virtual DB depends on the Visibroker CORBA ORB, care must be taken when using Visibroker for other services on the same machine. This situation arose because the VDBtoADT layer also needed to use the Visibroker ORB. Experience indicates that Visibroker should be installed after Virtual DB into the Virtual DB Visibroker directory.

3.5.3 Defining ADTs

Once the Virtual DB and Sybase middleware has been configured, the DAIM ADTs had to be defined before they could be modeled using Virtual DB. For the CDE experiment, the ADT modeling consisted of translating the C++ classes used in the code-generated DAIMI experiment into CORBA IDL. Defining ADTs would be one of the major tasks in creating a new DAIM. However, because the experiment involved an application designed with predefined ADTs they only had to be translated to a CORBA based architecture. While the Scheduler application can utilize a variety of ADTs, the initial experiments focused on Mission and Sortie objects. The DAIM IDL definition for the Mission ADT is listed next.[19]

[19] Notice that the following interface is similar to that required by the C2STA but not compliant with the C2STA. The CDE experiment focused on the feasibility of the major C2STA goals rather than the specifics of its requirements.

```
// Daim.idl
module Daim {

      interface DaimMission {
            string GetMissionID();
            string GetMissionState();
            string GetMissionSubtypeCode();
            string GetIFFSIF_Packed();
            string GetIFFSIF_Mode1();
            string GetIFFSIF_Mode2();
            string GetIFFSIF_Mode3();
            string GetStartTime();
            string GetEndTime();
            string GetUnionViewSources();
            long   SetMissionID(in string id);
            long   SetMissionState(in string state);
            long   SetMissionSubtypeCode(in string code);
            long   SetIFFSIF_Packed(in string
iffsif_packed);
            long   SetIFFSIF_Mode1(in string iffsif_mode1);
            long   SetIFFSIF_Mode2(in string iffsif_mode2);
            long   SetIFFSIF_Mode3(in string iffsif_mode3);
            long   SetStartTime(in string start);
            long   SetEndTime(in string end);
            long   Update();
            long   Delete();
      };

      typedef sequence<DaimMission> DaimMissions;

      interface Server {
            DaimMissions GetDaimMissions();
            DaimMission NewDaimMission();
      };
};
```

3.5.4 Modeling ADTs in Virtual DB

Once the ADTs were defined, the core DAIMI data mapping task had to be completed. For the Virtual DB based DAIMI, this meant building a relational model in Virtual DB representing the ADTs. A Virtual DB server maintains an object called the Metacatalog for storing data mapping and translation information. The Metacatalog stores a variety of data modeling objects including the base layer, foreign key mappings, processing blocks, and terms. The base layer stores the actual data models present on the data sources accessible to Virtual DB. This provides a starting point for mapping the

physical data models to the abstract data model. Foreign key mappings allow Virtual DB to automatically perform joins between abstract views in the abstract data model. These joins can link data from a single data source or even data from multiple data sources. Processing blocks allow the introduction of custom code into the Virtual DB data mapping services to operate on query parameters before passing them to the underlying data sources. Finally, the terms represent the actual mapping from the underlying data source models to the abstract data model. Terms allow for the definition of "virtual views" and "union views" that make up the abstract data model accessible through Virtual DB. Virtual views map abstract columns and their associated data types to actual columns and tables in the underlying data source. A single column in a virtual view can map to only one underlying data source. Union views allow data from multiple virtual views to be combined so it appears as one large dataset. This allows a single column in a union view to contain data from multiple data sources. The Metacatalog allows views to be stored in a hierarchical system very similar to folders on a hard disk.

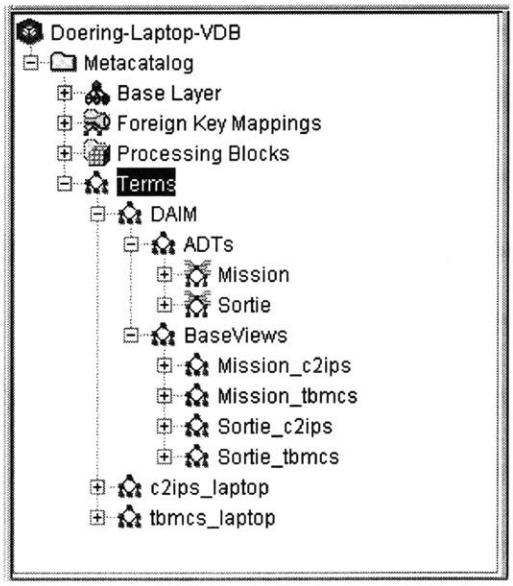

Figure 3.2 Metacatalog View Hierarchy

With an eye towards building a self-configuring VDBtoADT server, the view folders were structured as follows. A top-level folder called "DAIM" was created with two sub-folders called "ADTs" and "BaseViews". The ADTs folder was created to hold a single union view for each ADT in the DAIM. The BaseViews folder holds the virtual views mapping specific data sources to

specific ADTs. Thus, the ADTs folder contained a virtual view called "Mission" and one called "Sortie" while the BaseViews folder contained virtual views called "Mission_c2ips", "Mission_tbmcs", "Sortie_c2ips", and "Sortie_tbmcs". The naming convention dictated that the union virtual views in the ADTs folder had the name of the ADT being mapped while the virtual views in the BaseViews folder shared the name of the ADT being mapped followed by an underscore and the Virtual DB name of the underlying data source. Attributes of each view had the name of ADT's attributes.[20]

Virtual DB's view system allows for the renaming of data as well as the combining of data from multiple sources. The actual mapping of virtual views to the underlying data sources allows for basic SQL data manipulation. Data types can be changed, simple data manipulation performed, and SQL grouping operations used. For example, a column in a virtual view might be the concatenation of underlying data source columns, an algebraic manipulation of underlying columns, or some other basic SQL manipulation. It is important to realize that some manipulations will exclude write-back operations to the view. Virtual DB provides a second method for performing data manipulations. Processing blocks can be designed to operate on the parameters to a query or the values returned by a result set. A processing block is a piece of code that takes as input the parameter to a query or the value of a column in a result set and outputs a new value which is either passed on to the underlying data sources in the case of a query parameter or is substituted into the result set. Processing blocks can perform more complicated data manipulations than simple SQL operations. Further, processing blocks might help maintain write capabilities in some situations because they are asymmetric in

[20] Note that IDL previously listed had methods such as GetMissionID and SetMissionID. The virtual view attribute name for these methods would simply be MissionID.

that they can be independently defined for the input versus the output of a view. However, one important limitation of processing blocks is that they are only functions of their single input [Enterworks, 1999d]. For example, a processing block designed to operate on values for updating a view cannot depend on the current value of a column in the view. Processing blocks were not necessary for data modeling in the CDE experiment although they would have been useful had they been capable of functioning on both the input to an update query and a view's current data values.

Figure 3.3 provides some example mappings used for the Mission and Sortie views. The Mission ADT has three attributes called IFFSIF_Packed, IFFSIF_Mode1, IFFSIF_Mode2, and IFFSIF_Mode3. IFFSIF_Packed is a thirteen character string while IFFSIF_Mode1 is the first three characters of the string, IFFSIF_Mode2 the next five, and IFFSIF_Mode3 the final five. The IFFSIF was broken apart this way because C2IPS stores the packed version while TBMCS stores the three modes. Ideally, the ADT would have chosen a single representation and mapped both data sources to this representation. However, the Mission ADT for the original scheduler experiment instead included both representations and mapped each data source to both. This increased flexibility at the cost of complexity.

Virtual View Attribute	C2IPS Mapping	TBMCS Mapping
IFFSIF_Packed	C2ips.ATMSS_IDENT.IFF_SIF_CD	tbmcs.AIR_MSN_ACFT. AIR_MSN_ACFT_IFFSIF_MODE_1_CD || tbmcs.AIR_MSN_ACFT. AIR_MSN_ACFT_IFFSIF_MODE_2_CD || tbmcs.AIR_MSN_ACFT. AIR_MSN_ACFT_IFFSIF_MODE_3_CD
IFFSIF_Mode1	SUBSTRING(c2ips.ATMSS_IDENT.IFF_SIF_CD FROM 1 FOR 3)	tbmcs.AIR_MSN_ACFT. AIR_MSN_ACFT_IFFSIF_MODE_1
IFFSIF_Mode2	SUBSTRING(c2ips.ATMSS_IDENT.IFF_SIF_CD FROM 4 FOR 5)	tbmcs.AIR_MSN_ACFT. AIR_MSN_ACFT_IFFSIF_MODE_2
IFFSIF_Mode3	SUBSTRING(c2ips.ATMSS_IDENT.IFF_SIF_CD FROM 9 FOR 5)	tbmcs.AIR_MSN_ACFT. AIR_MSN_ACFT_IFFSIF_MODE_3

Figure 3.3 Example Data Model to Virtual View Mappings

3.5.5 Building the VDBtoADT Server

Once the ADTs were mapped in Virtual DB, one important component was still required for the Virtual DB DAIMI. The Virtual DB abstract data model had to be presented as CORBA objects as defined by the IDL definitions for the ADTs. Ideally commercial middleware would provide this functionality. Unfortunately, no such solution was discovered. This situation mandated the development of the VDBtoADT server.

Two major options were considered for the VDBtoADT server. The first design was to write specific code for exposing the scheduler ADTs. The second design was to build a generic VDBtoADT server capable of exposing arbitrary ADTs based on the abstract data model exposed by a Virtual DB server. The first design was pursued for the initial experiment. It was a simpler solution to program and offered increased potential for custom mappings (the VDBtoADT server could handle mappings beyond those supported in Virtual DB). However, a generic VDBtoADT server would be preferable for larger DAIMIs because it could be written once and then used universally. The investigation of such a solution will be a part of future DAIMI experiments.

The VDBtoADT server was implemented in Java due to the language's cross-platform potential as well as its high compatibility with the Visibroker ORB product used to expose the CORBA ADTs. The server connect to Virtual DB using the Virtual DB CORBA objects and exposed CORBA ADTs based on the IDL defined for the scheduler DAIM. The server then converted client "get" and "set" requests to Virtual DB queries as appropriate.

3.5.6 Linking the CORBA ADTs into the Scheduler

After completing the various steps in building the Virtual DB DAIMI, the final step was to modify the scheduler application to access data through the VDBtoADT server. As stated earlier, the experiment made use of the Visibroker ORB to provide distributed access to DAIM ADTs. The Visibroker environment includes tools for automatically generating C++ wrapper classes for accessing CORBA objects. Thus, the ADT IDL was converted directly into C++ source and header files for use by client applications. Simple test programs verified that a C++ program compiled for Win32 platforms could access the ADTs. However, significant difficulties arose when the ADT support was linked into the scheduler application.

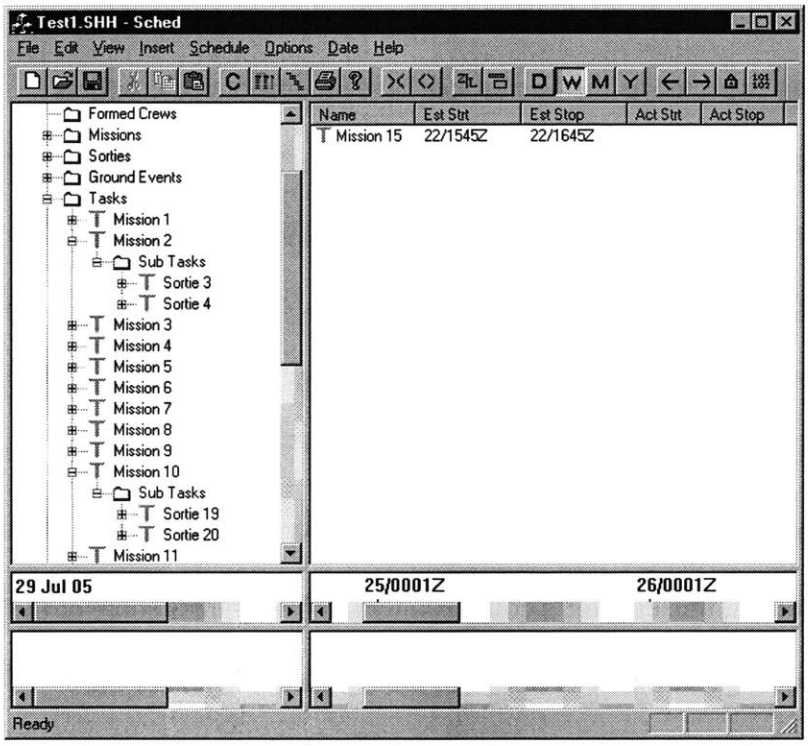

Figure 3.4 Scheduling Application User Interface

The scheduler application was developed under the Microsoft Visual C++ 5.0 environment using Microsoft Foundation Classes (MFCs). The application consists of several separate pieces that expose their capabilities through Windows DLLs. A graphical interface then integrates these components to provide a scheduling application.

The scheduling application is show in Figure 3.4. The linking of Visibroker headers and ADT support into the scheduling application data access DLLs allowed the scheduler application to access data through the Virtual DB DAIMI. This linking process was much more complicated than expected as the Visibroker headers introduced conflicts with the MFC header already present in the scheduler application. This incompatibility significantly increased the difficulty of the experiment. Future experiments might consider alternative CORBA implementations to reduce this cost.

3.6 Lessons Learned

Having described experiences from trying to build a DAIMI based on one commercial middleware product, this chapter's conclusion will now address some of the insights the experiment provided and tie the results into the broader analysis of the C2STA data architecture. A few disclaimers are in order. It should be clear that the DAIMI design presented in this chapter falls far short of the C2STA requirements described in Chapter 3. Change notification was completely ignored and the CORBA interfaces did not conform to the interface requirements of the C2STA. Further, metadata was not addressed. It might seem inappropriate to comment on the C2STA data architecture given that so much of it was ignored. However, these areas are details of the data architecture rather than its core. The C2STA data architecture provides three core benefits: (i) data are accessed through a common object oriented interface convenient for typical development tools, (ii) data are presented through abstract interfaces that hide the underlying storage representation, an finally (iii) data are presented as location independent. The DAIMI designed for the CDE experiment addressed all three of these

areas. Although it did not implement change notification and other details of the data architecture, experience working with its fundamental character provides insight into these details as well.

The Virtual DB DAIMI architecture provides important insight into the components needed for any DAIMI accessing legacy systems. A legacy DAIMI must perform three major functions:

- Solve communication and API interoperability issues.
- Map underlying data representations to an abstract model.
- Expose the abstract model via an object oriented interface.

The first function relates to actually accessing the legacy data. The data must be retrieved using some communication mechanism. In the VDB DAIMI design the Sybase server products provided this functionality. They handled network communications with database servers and translated various SQL dialects to one common SQL language.

The second function of a DAIMI is the particularly challenging one. Many commercial products deal with providing unified access to heterogeneous databases systems.[21] Likewise, many technologies layer object interfaces on top of traditional relational models.[22] However, none of these systems provides a mechanism for mapping a physical model to an abstract model. Virtual DB was particularly attractive because is offered this capability. However, it is important to understand that within a single RDBMS SQL defined views can also provide some basic translation from a physical data

[21] For example, both Sybase's OMNI middleware and Information Builders' EDA middleware address this problem. See http://www.sybase.com/products/middleware/ and http://www.ibi.com/products/eda/overview.html respectively.

[22] Examples include Microsoft's ADO technology as well as Sun's JDBC. See http://www.microsoft.com/data/ado/default.htm and http://java.sun.com/products/jdbc/ respectively.

model to abstract data models. Virtual DB provides this functionality across multiple databases and adds the concept of processing blocks. Despite this functionality significant restrictions are still placed on the possible mappings. Basic SQL commands can perform complicated data manipulation. However, this manipulation often precludes writing to a given view. Even Virtual DB's processing blocks cannot overcome some mapping problems. As noted before, the blocks can only operate on their input rather than on their input plus the current state of the database. While workarounds can probably be developed, these kinds of limitations significantly reduce the benefits of designing an abstract model of data.

The final task of a DAIMI, to provide an object-oriented interface, also deserves some discussion. While object oriented interfaces are convenient for building component-based software, the C2STA data architecture might hold more appeal for legacy systems if this requirement were modified. Data interoperability benefits greatly from standard technologies. The C2STA chose COM and CORBA as requirements to try and address this need. However, it now seems that other technologies such as XML hold more promise (and therefore receive more vendor support) for future inter-system communication. Does the rise of XML force the Air Force to scrap the C2STA data architecture? Of course not. The architecture could be redefined to preserve its core benefits while adopting XML technologies. This leads to the conclusion that the core contribution of the C2STA data architecture (one unified view of C2 data independent of the individual data servers) should be promoted without a strict connection to COM or CORBA technologies.

The next chapter will explore additional insights regarding the C2STA data architecture while considering existing literature and research projects related to data access. This background provides a comparison of competing data interoperability solutions, examples of solutions similar to the C2STA data architecture, and a peek at the technologies that might support an improved C2STA data architecture.

Chapter 4

Relevant Research

As mentioned in Chapter 1, data interoperability problems arise frequently in both intra-business and inter-business contexts. Because of the importance of this topic many papers address the issue. Further, many commercial middleware products and research projects address data interoperability issues. This chapter introduces the range of data interoperability solutions found in the literature in an attempt to characterize the state of the art technology in data architectures. This will provide additional insight into recommended improvements to the C2STA data architecture.

4.1 Literature Survey of Data Interoperability Solutions

As explained in Section 1.3 the review of literature regarding data interoperability solutions was deferred to allow for the C2STA data architecture's introduction without the burden of positioning it relative to existing research. Because the C2STA does not refer to any related research, it is natural to first explain the architecture and then position it in relation to other data interoperability solutions. Goh categorizes interoperability approaches according to two different axes: the choice of underlying data model used for conflict resolution and the use of either a loose-coupling or tight-coupling strategy [Goh, 1997]. While the data model chosen for conflict resolution is critical to the implementation of particular interoperability solutions, it can be largely ignored for basic

comparison purposes here. It is sufficient to mention that different solutions can be categorized as using a deductive data model, an object oriented data model, a functional data model, or a relational data model. The Virtual DB DAIMI used a relational data model for data conflict resolution.

The interesting distinction for data interoperability solutions is that between loosely-coupled strategies and tightly-coupled strategies. The critical differences between loosely-coupled and tightly-coupled systems involve who resolves conflicts and when [Goh, 1997]. Tightly-coupled strategies are characterized by the a priori development of shared schemas to globally represent the data available in heterogeneous systems. This requires the identification and resolution of data conflicts in advance. Once this task has been accomplished, systems can interoperate by issuing queries against the shared schema. Of course some mechanisms must exist to allow clients to query the multiple data sources. Numerous research projects have employed this approach. Some commonly cited in the literature include Multibase [Landers and Rosenberg, 1982], ADDS [Breitbart and Tieman, 1985], Mermaid [Templeton et al., 1987], Carnot [Collet et al., 1991], Pegasus [Ahmed et al., 1991], and SIMS [Arens and Knoblock, 1992]. The C2STA data architecture also qualifies as a tightly-coupled approach. The C2STA mandates the definition of a global schema (the DAI). This requires that underlying data sources map their schemas to their associated DAIM in advance. Dayal and Hwang propose a general strategy for solving data conflicts by creating supertypes and mapping the data from various sources to these supertypes [Dayal and Hwang, 1984]. Most of the projects listed above use this general strategy, as did the Virtual DB DAIMI.

Having categorized the C2STA data architecture as an instance of tight-coupling, it is useful to understand the major alternative. Loose-coupling approaches assume that maintaining a shared schema is too large a burden and does not scale well. Instead they attempt to detect and resolve data conflicts at request time by working between the data source and receiver. Loose-coupling research has typically focused on data manipulation languages designed to query multiple sources and transform the results [Goh, 1997]. The MRDSM [Litwin and Abdellatif, 1987] system is a well-known example of a loose-coupling approach. While tight-coupling carries the burden of developing a shared schema, loose-coupling carries the burden of conveying new conflict resolution procedures to all clients and receivers.

Goh argues that the COntext INterchange (COIN) strategy described in his thesis is a new kind of solution to the data interoperability problem [Goh, 1997]. The COIN strategy relies on both information sources and receivers having explicit context. Axioms that describe facts about the systems are defined. Given two contexts (the receiver's and the source's) and the previously mentioned axioms, the system can figure out how to properly satisfy a query from the receiver's context with information from the sender's context. This strategy is designed to eliminate the need for explicitly solving all data conflicts, as required by tight-coupling strategies. Further, it uses a Context Mediator to perform the context translations. This reduces the burden receivers face in loose-coupling approaches.

The Context Mediator used in COIN introduces another important concept from data interoperability literature. Dr. Gio Wiederhold of Stanford University has written extensively on the use of mediators to solve heterogeneous information system problems

[Wiederhold, 1991, Wiederhold, 1997, and Wiederhold, 1999]. While Goh claims that COIN uses a new approach within a mediator, the idea of mediators is not new. Wiederhold advocates a three-tiered approach to accessing heterogeneous data. The first and third tiers are the client application and the existing data source, respectively. The middle tier is composed of mediators. Mediators are tasked with providing useful access to information. This means that mediators must accept client queries, identify the appropriate data sources to satisfy a query, and return results processed for maximum information density. The processed results should summarize data or identify exceptional data to return the information a client really wants rather than just returning large quantities of data. Notice that the concept of mediators is largely architectural leaving the opportunity for a wide-range of implementations. The COIN system is one such implementation as is the TSIMMIS project introduced in Section 4.3.

How does the C2STA data architecture relate to mediators? The C2STA does use a middle tier (DAIMIs) to provide access to data sources. However, this is not a mediator as the middle tier is designed only to support a logical schema. Although the C2STA data architecture does not adopt the mediator design, the C2STA as a whole does allow for similar services. The raw capability layer in Figure 2.1 is designed to facilitate the kind of value-added services that characterize mediators in Wiederhold's architecture. The C2STA has decided that intelligent processing of data and the abstract representation of it should be separate tasks while the mediator approach assumes that the same agent that intelligently processes data should deal with its representation as well. The relative merits of these two approaches depend largely on assumptions regarding mediation technologies available. If highly automated mediation approaches such as COIN truly

provide mediation without manual intervention and without constraining the underlying data sources, it seems unnecessary to build the C2STA's DAI. Simply building raw capabilities as clients of COIN would allow the capabilities to access the data needed for their intelligent processing without worrying about the underlying data structures. In this case the C2STA data architecture and raw capability layers could be combined into one mediation layer using a COIN-like approach as well as mediators for post-processing of data. On the other hand, if one assumes that manual intervention will be required for building mediation systems (as assumed by the TSIMMIS project), the current C2STA data architecture is useful for shielding intelligent processing from the changes introduced by new resolutions to data conflicts. Because the C2STA generally avoids implementation issues, it can actually support a wide-variety of proposed data interoperability solutions. However, its mandate of a global abstract schema largely constrains it to the tightly-coupled strategies.

Having explained the spectrum of data interoperability solutions in the literature, the C2STA's data architecture can now benefit from the results of previous research. Proponents of loosely-coupled architectures will no doubt condemn the C2STA data architecture as destined to fail due to the difficultly in defining a global schema. On the other hand, many papers attempt to address building such a schema, so tightly-coupled strategies have proponents as well. Pitoura, et al. [1995] survey a variety of such approaches. In fact, Dogac, et al. [1995a, 1995b] undertook a CORBA based heterogeneous data project very similar to the CDE investigation. These efforts have not collapsed in complete failure. Instead they have identified a range of implementation technologies that might support future DAIMIs. The clear conclusion of research on data

interoperability is that the biggest challenge facing tight-coupling strategies is the design and maintenance of a global schema. The success or failure of the C2STA data architecture will likely hinge on whether or not the scope of C2 data is "too" large to be supported by a global schema. Of course it is difficult to measure "too" without actual implementation experience.

Having discussed the literature on the categories of data interoperability solutions, we will now focus on a few projects aimed at achieving data interoperability. These projects do not introduce new theory on data interoperability, but rather provide some practical examples of the important characteristics of data interoperability solutions. This provides an opportunity to examine elements missing from the C2STA data architecture.

4.2 Enterprise Business Objects

The jBusiness product of Novera Software, Inc. includes a feature called Enterprise Business Objects (EBO) [Orstein, 1999]. EBO technology provides a means to expose relational data as Java objects. Although the scheduler experiment dealt with CORBA objects, this mapping is very similar to that required for a DAIMI. Several features of EBOs are relevant to the discussion of the C2STA data architecture.

Like the Virtual DB DAIMI, EBOs provide basic schema transformation. However, they translate a relational schema directly into an object model instead of into an additional model. This allows EBOs to provide object oriented "get" and "set" methods for their attributes. Further, EBOs address issues such as the updating of aggregate properties. EBOs also provide convenient object collections for accessing related data (as a substitute for joins in the relational model). All of these features

suggest that EBOs might provide an excellent foundation for a DAIMI. Two other features of EBOs highlight areas in which the C2STA data architecture is lacking. EBOs provide an object oriented query interface and caching of data retrieved from a database. The C2STA's lack of a required query interface has already been highlighted as a major weakness. Support for caching is clearly the kind of implementation detail that the C2STA specifically avoided. However, it is important to note that much like change notification, caching services might be significantly complicated by the multiple data access interfaces allowed by the C2STA.[23] While EBOs closely mirror the problems encountered in the CDE experiments, they are fundamentally tied to relational source models and probably model objects in a manner relatively similar to the underlying relational model. Researchers are working on other data architectures that allow more flexibility in this area.

4.3 TSIMMIS Data Wrappers

The Stanford-IBM Manager of Multiple Information Sources (TSIMMIS)[24] project provides abstract object access to underlying data. However, unlike the Virtual DB DAIMI or EBOs, TSIMMIS wrappers are custom written for each underlying data source. The developers of TSIMMIS note that writing wrappers is a significant effort [Hammer et al., 1997]. The TSIMMIS system provides a toolkit for wrapper develop to reduce this burden. It is also important to realize that custom written wrappers provide the ultimate in data model translation flexibility. A custom wrapper can perform

[23] See Section 2.5.

[24] For more information on TSIMMIS see: http://www-db.stanford.edu/tsimmis/tsimmis.html

arbitrarily complex manipulations in response to a "get" or "set" call. This is similar to the custom coding done in the original MITRE DAIMI experiment with one noteworthy exception. The original DAIMI experiment included code for the accessing, mapping, and final presentation of data. On the other hand, a TSIMMIS wrapper only has to access and map data. The TSIMMIS system provides a framework for presenting and querying the data.

The support for querying in the TSIMMIS system is very important. The TSIMMIS system is capable of matching an application query to an underlying data source's exposed data, retrieving the underlying data from the data source, and returning the results to the application [Papakonstantinou et al., 1995]. Even more impressive is the fact that the TSIMMIS system supports post-retrieval querying that allows applications to pose more complicated queries than those supported by the underlying data source. In this case, the TSIMMIS system performs filtering on the results returned by the data source. This kind of support is very useful when designing data wrappers for data sources with limited query interfaces. Although this does not apply to most RDBMSs, the C2STA data architecture was designed to provide access to all data sources. A unified query architecture layered above existing data sources could provide both the desired standardization of query interfaces and provide the actual filtering implementation for those data sources that do not include their own. The potential benefits of adding such a layer to the C2STA data architecture is considered in the next chapter.

4.4 Garlic Middleware

Garlic middleware was developed to support integrated access to multimedia data stored in legacy systems. Like other middleware solutions, the Garlic project uses wrappers to access existing data sources and provides object-oriented interfaces to client applications. Garlic middleware then allows heterogeneous querying of the underlying data through a unified interface. So far this sounds very similar to the TSIMMIS system already described. Again there is an emphasis on query interfaces when dealing with heterogeneous data. Garlic also supports querying data sources that lack their own query interface [Roth and Schwarz, 1997].

However, Garlic's designers were particularly focused on integrating both schemas and data. They note that integrating legacy schemas often involves dealing with overlapping schemas [Haas et al., 1999]. This requires both composition and decomposition of underlying data structures to provide a unified object interface. The garlic system provides this functionality by supporting transformations applied to its object interface. This support addresses the functionality needed to provide the abstract data access advocated by the C2STA data model. The Garlic project includes an additional component that addresses the practical issues of building abstract data representations.

Building the Virtual DB DAIMI highlighted the difficulties involved in mapping an existing data model to some abstract representation. While the effort involved several development steps, some of these were required because the middleware employed did not support al of the functionality needed. Even with ideal middleware the task would involve two major steps. One step is defining abstract data representations. As

mentioned earlier, this step includes developing the agreements essential to data interoperability. Once the data representations are defined, actual data models must be mapped to the abstract model. The Garlic project includes work on a tool called Clio designed to facilitate the mapping task. Clio is designed to facilitate the mapping of data and schemas from sources to some target representation [Haas et al., 1999]. The tool reads in source schemas and translates them to an internal format using Schema Readers. It then uses a Correspondence Engine to identify matches between the source and target schemas. Finally, a Mapping Generator creates views that transform data from a source schema to data in the target schema. The Correspondence Engine is designed to support both a graphical interface for human-aided matching and a data-mining engine to automate the process. The Clio tool was in the early stages of development as of March 1999 [Haas et al., 1999].

4.5 YAT Model Translations

The final presentation of research related to the CDE data interoperability investigation focuses on the YAT/Tran-Scm[25] system. YAT is a data model for describing object using a labeled tree design. This model includes a language, YATL, for specifying rule-based translations from a source model to a target model. The Tran-SCM part of the system is a mechanism for automatically generating some translations [Abiteboul et al., 1999]. Because source and target schemas often have a lot in common, a rule-based algorithm is used to automatically generate translations where possible. A system designer then specifies the remaining translations using YATL. This design attempts to maximize translation flexibility while minimizing the cost of developing

[25] For more information on Yat/Tran-Scm see: http://www-rocq.inria.fr/~simeon/YAT/

translations. Experience with the Virtual DB DAIMI indicates that this flexibility is essential. Virtual DB provided a relatively simple (although not automated) method of generating simple translations. However, its ability to support custom-coded complex translations was too limited. An ideal DAIMI platform would support automatically generated simple translations as well as arbitrarily complex custom translations. Having discussed various research projects related to DAIMI needs, enough background is in place to propose changes to the C2STA data architecture based on the pervious discussions about the architecture and experiences from the CDE experiments.

Chapter 5

Conclusions

Chapters 1 through 4 introduced data interoperability, the C2STA data architecture as an interoperability solution, the CDE experiments with this architecture, and finally additional architectures that address data interoperability. All of this background supports recommendations on the C2STA data architecture and a discussion possible implementation issues surrounding the architecture.

The C2STA data architecture should be separated into two parts. The first layer should be responsible for communicating with underlying data sources, mapping source schemas to a target schema (in some internal format), providing change notification services, and providing a common query mechanism. The second layer should be responsible for presenting the abstract data to client applications. However, the second layer should not be required to implement a COM or CORBA interface. Instead, this new architecture could support any of a variety of interfaces including COM, CORBA, XML, Enterprise Java Beans (EJBs), etc. Many systems within the Air Force are already considering adding XML and other interfaces to their data systems. However, it is not clear that many systems intend to support COM or CORBA. By separating the C2STA data architecture from the particular interface, the architecture can succeed regardless of which interface becomes widely adopted. Further, by allowing multiple interfaces above the common translation layer, the problems regarding separate change notification

services are solved. This will not solve change notification issues involving legacy access methods to data servers. Such problems are difficult and the C2STA data architecture should be modified to better address legacy systems.

Legacy systems present a wide range of problems for the C2STA data architecture. Many such problems were discussed in Chapter 2. The C2STA should provide separate requirements for legacy systems and new systems. This might be accomplished by defining two different standards with different criteria for compliance. For example, it was already noted that legacy systems might be unable to separate their data interfaces from their business rules. While such a separation is useful, it is better to have access to the data through and interface that encompasses the business rules than to have no access at all. Other exceptions for legacy systems should deal with change notification requirements that might be very difficult to implement below legacy interfaces. Finally, the C2STA data architecture should include suggestions for a migration path from the less restrictive requirements for legacy systems to the more stringent requirements for new systems.

A common feature of the research projects from Chapter 4 motivates another recommendation for the C2STA data architecture. The architecture must mandate a flexible query interface. Such an interface is very useful to client applications and essential to promoting the C2STA as a primary data interface for systems. System designers are unlikely to embrace a data architecture that replaces traditional RDBMSs with a new interface that lacks flexible query features. RDBMSs provide extensive support for ad hoc querying. System designers are accustomed to this support so any replacement data architecture must provide similar features.

A final recommendation for changes to the C2STA data architecture involves its assumption of symmetric "get" and "set" operations. Reading data is often much simpler than writing data when accessing data stores. This situation arises due to security measures as well as read-only view issues. Therefore, it is generally easier to create read mappings from a source schema to a target schema than to create the equivalent write mappings. While supporting both read and write access is critical if the C2STA data architecture is to provide the only interface to data sources. However, in the case of legacy systems other interfaces are likely to exist. In this case, it is useful to have a read only representation of data for interoperability even when write access is unachievable. The C2STA data architecture should recognize this asymmetry and provide guidance on providing read-only access while preferring read/write access.

Having discussed possible changes to the C2STA data architecture, a few comments regarding technologies that might support such an architecture will conclude this discussion. The review of other data architectures in Chapter 4 as well as experiences implementing the Virtual DB DAIMI suggest that flexibility of schema translations is critical to any DAIMI platform. The ideal solution would provide automatic mappings for simple translations while allowing arbitrarily complex custom components for sophisticated mappings. Further, the platform would support the separation of mapping functionality and interface presentation already recommended. The system would allow a single set of mappings to be represented through several interfaces simultaneously. The research projects examined suggest that such a tool might be available in the not too distance future. While automatic mapping functionality will

probably take quite awhile to refine, the other goals could be realized with current technology.

The C2STA data architecture addresses important techniques for achieving data interoperability. It represents a middle ground in the solution space for data interoperability as defined in Chapter 1. The architecture is particularly appealing because a message passing system could be implemented above a system's abstract data interface while a schema standardization effort could take place below the abstract data interface. Therefore this solution is completely compatible with other data interoperability solutions. A new revision of the C2STA data architecture incorporating the suggestions outlined in this chapter could support significant data interoperability solutions both inside and outside the Air Force.

Appendix

Acronyms Appearing in this Thesis

Both the military world and the computer science world are littered with acronyms. Some are broadly used and recognized (e.g. HTML) while others are very specific and obscure (e.g. GCI). In fact, it is not uncommon for acronym collisions to occur. Therefore, the following table lists in alphabetical order all of the acronyms in this thesis for the reader's convenience.

Acronym	Expansion
ADT	Abstract Data Type
AF	Air Force
API	Application Programming Interface
C2	Command and Control
C2IPS	Command and Control Information Processing System
C2STA	C2 System Target Architecture
CDE	Common Data Environment
COIN	Context INterchange
COM	Component Object Model
CORBA	Common Object Request Broker Architecture
COTS	Commercial Off-the-shelf
DAI	Data Access Interface
DAIM	Data Access Interface Module
DAIMI	Data Access Interface Module Implementation
DBMS	Database Management System
DLL	Dynamic Link Library
DOD	Department of Defense
EBO	Enterprise Business Objects
EJB	Enterprise Java Bean
ESC	Electronic Systems Center
GCI	General Capability Interface
HTML	HyperText Markup Language
IC2S	Integrated C2 System
IDL	Interface Definition Language
MFC	Microsoft Foundation Class
ODBC	Open Database Connectivity
POTS	Plain Old Telephone Service

PROFIT	Productivity From Information Technology
RDBMS	Relational Database Management System
RGB	Red-Green-Blue
SQL	Structured Query Language
TBMCS	Theater Battle Management Control System
TCP/IP	Transmission Control Protocol / Internet Protocol
USMTF	United States Message Text Format
XML	Extensible Markup Language
YAT	(Unknown to author)
YATL	YAT Language

Bibliography

Abiteboul, S., Cluet, S., Milo, T., Mogilevsky, P., Siméon, J., and Zohar, S. (1999). Tools for data translation and integration. *IEEE Data Engineering Bulletin*, 22(1):3-8, March 1999.

Agarwal, S., Keller, A. M., Wiederhold, G., and Saraswat, K. (1995). Flexible relation: An approach for integrating data from multiple, possibly inconsistent databases. In *Proc. IEEE Intl Conf on Data Engineering*, Taipei, Taiwan.

Ahmed, R., Smedt, P. D., Du, W., Kent, W., Ketabchi, M. A., Litwin, W. A., Raffi, A., and Shan, M. C. (1991). The Pegasus heterogeneous multidatabase system. *IEEE Computer*, 24(12):19-27.

Arens, Y. and Knoblock, C. A. (1992). Planning and reformulating queries for semantically-modeled multidatabase systems. In *Proceedings of the 1st International Conference on Information and Knowledge Management*, pages 92-101.

Batini, C., Lenzerini, M., and Navathe, S. (1986). A Comparative analysis of methodologies of database schema integration. *ACM Computing Surveys*, 18(4):323-364.

Breitbart, Y. J. and Tieman, L. R. (1985). ADDS: Heterogeneous distributed database system. In Schreiber, F. and Litwin, W., editors, *Distributed Data Sharing Systems*, pages 7-24. North Holland Publishing Co.

Brodie, M. L., and Stonebraker, M. (1995). *Migrating Legacy Systems: Gateways, Interfaces, and the Incremental Approach.* Morgan Kaufmann Publishers, Inc., San Francisco, CA.

Carey, M. J., Haas, L. M., Schwarz, P. M., Arya, M., Cody, W. F., Fagin, R., Flickner, M., Luniewski, A. W., Niblack, W., Petkovic, D., Thomas, J., Williams, J.H., and Wimmers, E. L. (1995). Towards heterogeneous multimedia information systems: the garlic approach. In *Proceedings of the Fifth International Workshop on Research Issues in Data Engineering (RIDE): Distributed Object Management*, Taipei, Taiwan, Mar 6-7, 1995.

Cohen, W. (1998). Integration of heterogeneous databases without common domains using queries based on textual similarity. In *Proceedings ACD SIGMOD*, 27(2):201-212, Seattle, WA, June 1998.

Collet, C., Huhns, M. N., and Shen, W. M. (1991). Resource integration using a large knowledge base in Carnot. *IEEE Computer*, 24(12):55-63.

Dayal, U. and Hwang, H. Y. (1984). View definition and generalization for database integration in a multidatabase system. *IEEE Software Engineering*, 10(6):628-645.

DII-AF Chief Architects' Office. (1998). Command and Control System Target Architecture (C2STA) Specification. Release 2.0 Draft I5. Electronics Systems Center (ESC/DIE), Bedford, Massachusetts, August 31, 1998.

Dogac, A., Dengi, C., Kilic, E., Ozhan, G., Ozcan, F., Nural, S., Evrendilek, C., Halici, U., Arpinar, B., Koksal, P., Kesim, N., and Mancuhan, S. (1995a). METU Interoperable database system. In *ACM SIGMOD Record*, 24(3), Sept 1995.

Dogac, A., Kilic, E., Ozhan, G., Ozcan, F., Nural, S., Evrendilek, C., Halici, U., Arpinar, B., Koksal, P., Kesim, N., and Mancuhan, S. (1995b). Experiences in Using CORBA for a multidatabase implementation. 6th International Conference on Database and Expert Systems Applications Workshop presentation, London, September 1995.

Enterworks, Inc. (1999a). *Virtual DB Administrator's Guide Version 3.01.* Enterworks, Inc., Ashburn, Virginia, May 1999

Enterworks, Inc. (1999b). *Virtual Application Developer's Guide Version 3.01.* Enterworks, Inc., Ashburn, Virginia, May 1999.

Enterworks, Inc. (1999c). *Virtual DB Data Modeler's Guide Version 3.01.* Enterworks, Inc., Ashburn, Virginia, May 1999.

Enterworks, Inc. (1999d). *Virtual DB Processing Block Reference Version 3.01.* Enterworks, Inc., Ashburn, Virginia, May 1999.

Garcia-Solaco, M., Saltor, F., and Castellanos, M. (1996). Semantic heterogeneity in multidatabase systems. In Bukhres, O. A. and Elmagarmind, A. K., editors, *Object-Oriented Multidatabase Systems: A Solution for Advacned Applications*, chapter 5, pages 129-202. Prentice-Hall.

Goh, C. H. (1997). Representing and reasoning about semantic conflicts in heterogeneous information systems. PhD dissertation. Massachusetts Institute of Technology, Sloan School of Management, December 1996.

Haas, L. M., Miller, R. J., Niswonger, B., Roth, M. T., Schwarz, P. M., and Wimmers, E. L. (1999). Transforming heterogenous data with database middleware: beyond integration. *IEEE Data Engineering Bulletin*, 22(1):31-36, March 1999.

Hammer, J., Breunig, M., Garcia-Molina, H., Nestorov, S., Vassalos, V., and Yerneni, R. (1997). Template-based wrappers in the TSIMMIS system. In *Proceedings of the Twenty-Sixth SIGMOD International Conference on Management of Data*, Tucson, Arizona, May 12-15, 1997.

Hammer, J. and McLeod, D. (1993). An approach to resolving semantic heterogeneity in a federation of autonomous, heterogeneous database systems. In International Journal of Intelligent & Cooperative Information Systems, World Scientific, Volume 2, Number 1, pp. 51-83, 1993, M. Papazoglou and T. Sellis, editors-in-chief. Available online at: http://dbpubs.stanford.edu:8090/pub/1993-14

Hurson, A. R., Bright, M. W., and Pakzad, S. H. (1994). *Multidatabase Systems: an advanced solution for global information sharing*. IEEE Computer Society Press.

Inprise Corp. (1998a). *Visibroker for C++ Version 3.3 Programmer's Guide*. Inprise Corp., Scotts Valley, CA.

Inprise Corp. (1998b). *Visibroker for C++ Version 3.3 Reference*. Inprise Corp., Scotts Valley, CA.

Inprise Corp. (1998c). *Visibroker for Java Version 3.3 Programmer's Guide*. Inprise Corp., Scotts Valley, CA.

Inprise Corp. (1998d). *Visibroker for Java Version 3.3 Reference*. Inprise Corp., Scotts Valley, CA.

Isbell, D., Hardin, M., and Underwood, J. (1999). Mars climate orbiter team finds likely cause of loss. NASA news release, September 30, 1999.

Kim, W. and Seo, J. (1991). Classifying schematic and data heterogeneity in multi-database systems. *IEEE Computer*, 24(12):12-18.

Krishnamurthy, R., Litwin, W., and Kent, W. (1991). Language features for interoperability of databases with schematic discrepancies. In *Proceedings of the ACM SIGMOD Conference*, pages 40-49.

Lakshmanam, L., Sadri, F., and Subramanian, I. N. (1996). SchemaSQL – a language for interoperability in multiple relational databases. In *Proceedings of the Conference on Very Large Databases (VLDB)*, Bombay, India.

Landers, T. and Rosenberg, R. (1982). An overview of Multibase. In *Proceedings 2nd International Symposium for Distributed Databases, pages 153-183.*

Litwin, W. and Abdellatif, A. (1987). An overview of the multi-database manipulation language MDSL. *Proceedings of the IEEE*, 75(5):621-632.

Miller, R. J. (1998). Using schematically heterogeneous structures. In *Proceedings of ACM SIGMOD*, 27(2):189-200, Seattle, WA, June 1-4, 1998.

Naiman, C. F. and Ouskel, A. M. (1995). A classification of semantic conflicts in heterogeneous database systems. *J. of Organizational Computing*, 5(2):167-193.

Orenstein, J. A. (1999). Supporting retrievals and updates in an object/relational mapping system. *IEEE Data Engineering Bulletin*, 22(1):50-54, March 1999.

Papakonstantinou, Y., Garcia-Molina, H., and Widom, J. (1995a). Object exchange across heterogeneous information sources. In *Proceedings of IEEE Conference on Data Engineering*, pp. 251-260, Taipeh, Taiwan.

Papakonstantinou, Y., Gupta, A., Garcia-Molina, H., and Ullman, J. (1995b): A query translation scheme for rapid implementation of wrappers. In *Deductive and Object-Oriented Databases.*

Pitoura, E., Bukhres, O., and Elmagarmid, A. (1995). Object orientation in multidatabase systems. *ACM Computer Surveys*, 27(2):141-195.

Quass, D., Rajaraman, A., Sagiv, Y., Ullman, J., and Widom, J. (1994). Querying semistructured heterogeneous information. In *Proc International Conference on Deductive and Object-Oriented Databases.*

Renner, S. (1999). Data issues. Presented at the Air Force Architecture Workshop, November 18, 1999.

Roth, M. T. and Schwarz, P. (1997). Don't scrap it, wrap it! A wrapper architecture for legacy data sources. Very Large Databases '97, Athens, Greece, August 26-29, 1997.

Scheuermann, P., Yu, C., Elmagarmind, A., Garcia-Molina, H., Manola, F., McLeod, D., Rosenthal, A., and Templeton, M. (1990). Report on the workshop on heterogeneous database systems. *ACM SIGMOD RECORD*, 19(4):23-31. Held at Northwestern University, Evanston, Illinois, Dec 11-13, 1989. Sponsored by NSF.

Sciore, E., Siegel, M., and Rosenthal, A. (1994). Using semantic values to facilitate interoperability among heterogeneous information systems. *ACM Transactions on Database Systems*, June 1994.

Sheth, A. and Kashyap, V. (1992). So far (schematically) yet so near (semantically). In Hsiao, D. K., Neuhold, E. J., and Sacks-Davis, R., editors, *Proceedings of the IFIP WG2.6 Database Semantics Conference on Interoperable Database Systems (DS-5)*, pages 283-312, Lorne, Victoria, Australis. North-Holland.

Sheth, A. P. and Larson, J. A. (1990). Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Computing Surveys*, 22(3):183-236.

Smith, J. and Smith, D. (1977). Database abstractions: aggregation and generalization. *ACM Transactions on Database Systems*, 2(2):105-133.

Templeton, M., Brill, D., Dao, S. K., Lund, E., Ward, P., Chen, A. L. P., and MacGregor, R. (1987). Mermaid – a front end to distributed heterogeneous databases. *Proceedings of the IEEE*, 75(5):695-708.

Wiederhold, G. and Genesereth, M. (1997). The conceptual basis for mediation services. *IEEE Expert, Intelligent Systems and their Applications*, 12(5), Sept-Oct 1997.

Wiederhold, G. (1999). Mediation to deal with heterogeneous data sources. Vckovski, Brassel, and Schek, editors, Interoperability Geographic Information Systems, Springer LNCS 1580 (Proc. Interop'99, Zurich, March 1999), pp. 1-16.

Wiederhold, G. (1991). Obtaining information from heterogeneous systems. In *Proceedings of the First Workshop on Information Technologies and Systems (WITS'91)*, Dec 14-15, 1991, Cambridge, MA, MIT Sloan School of Management, pp. 1-8.

Information Resources Cited

- CORBA website: http://www.corba.org/
- Enterworks Virtual DB information: http://www.enterworks.com/products/index.html
- Information Builders EDA information: http://www.ibi.com/products/eda/overview.html
- Microsoft ADO technology website: http://www.microsoft.com/data/ado/default.htm
- Microsoft COM technology website: http://www.microsoft.com/com/
- Microsoft ODBC technology website: http://www.microsoft.com/data/odbc/
- Sun Microsystems Java JDBC website: http://java.sun.com/products/jdbc/
- SQL standard information: http://www.jcc.com/SQLPages/jccs_sql.htm
- Sybase OMNI product information: http://www.sybase.com/products/middleware/
- TSIMMIS Project homepage: http://www-db.stanford.edu/tsimmis/tsimmis.html
- USMTF website: http://www.forscom.army.mil/interop/USMTF/DEFAULT.htm
- W3C XML information website: http://www.w3.org/XML/
- YAT Project homepage: http://www-rocq.inria.fr/~simeon/YAT/