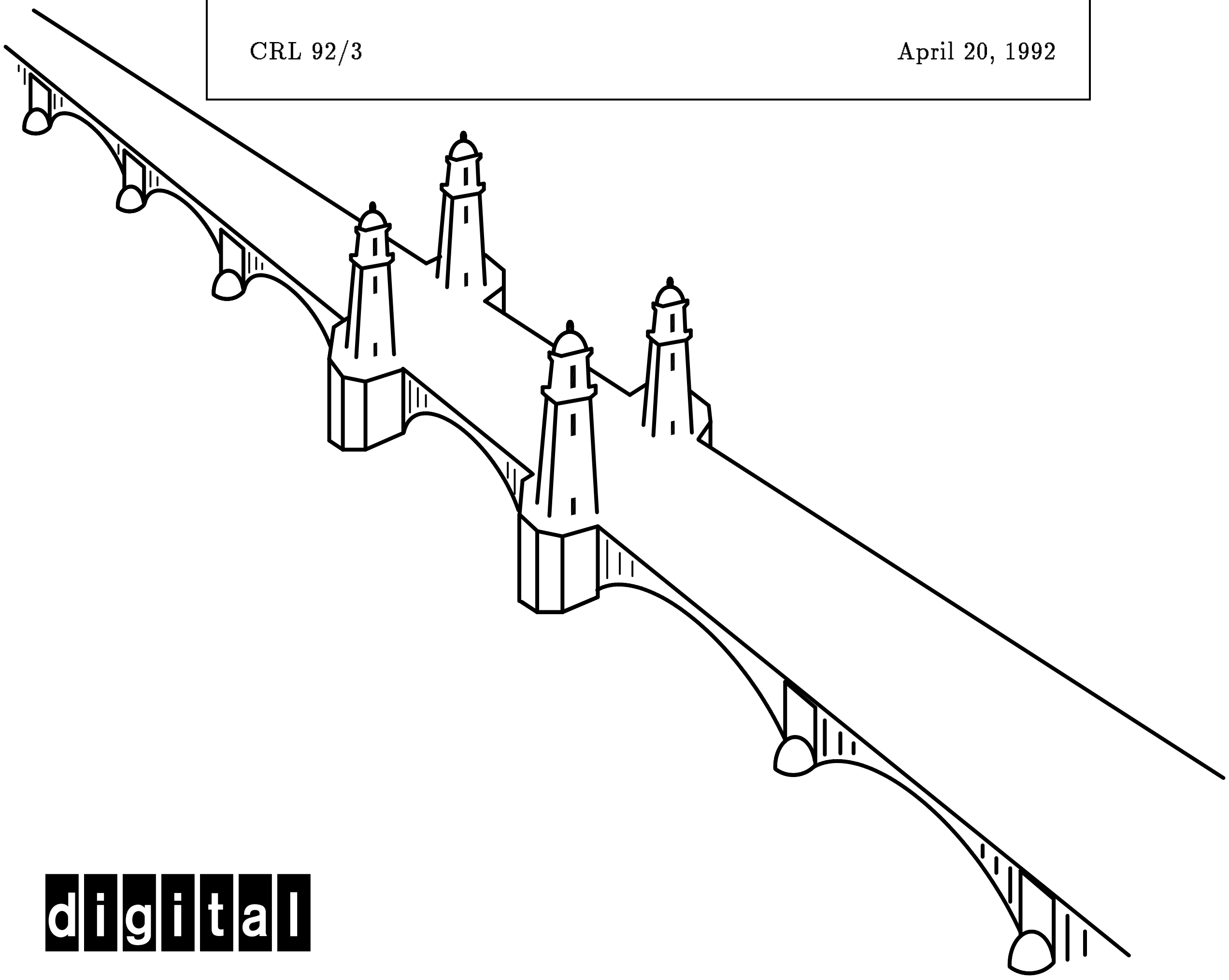
The Organization Engine: Virtual Data Integration

James S. Miller (DEC/CRL)
Carl Niedner (DEC/HCABU)
Jack London (Fox Chase Cancer Center)

Digital Equipment Corporation
Cambridge Research Lab

CRL 92/3 April 20, 1992



digital

CAMBRIDGE RESEARCH LABORATORY
Technical Report Series

Cambridge Research Laboratory
One Kendall Square
Cambridge, Massachusetts 02139

The Organization Engine: Virtual Data Integration

James S. Miller (DEC/CRL)
Carl Niedner (DEC/HCABU)
Jack London (Fox Chase Cancer Center)

Digital Equipment Corporation
Cambridge Research Lab

CRL 92/3 April 20, 1992

Abstract

The Organization Engine is an early example of Virtual Data Integration — providing the appearance of integration at the desktop without modifying existing infrastructure. Starting with the Organization Engine, eight programming days were needed to provide uniform desktop access to a CODASYL-compliant hospital information system and to a MUMPS-based radiology information system (the technique is equally effective for relational and other data bases). The resulting tool provides a seamless integration of these two systems, image storage, pre-recorded audio, and document storage. In addition to providing uniform access, the tool also allows healthcare providers to organize the data to suit their individual needs.

The key to this easy integration lies in two simple techniques: the transformation of data from all sources into a single, homogeneous representation; and the use of simple customization files to describe new object types and formats. The approach is sufficiently general to allow the integration of applications which present external interfaces of radically different forms. Two such forms are discussed here: data map publication and transactions.

1 Introduction: The Challenge of Integration

Most healthcare institutions today must struggle with a profusion of disparate systems, each carefully designed for a single specific purpose. Unfortunately, the majority of healthcare providers require information dispersed among these various systems. Worse yet, at times of greatest need data is likely to be required from the widest variety of systems. Historically, several approaches to solving this problem have been tried: specific pair-wise solutions (e.g., one vendor's lab system to another's billing system); agglomeration (constructing a centralized replica of portions of distributed departmental datasets); interfacing standards (HL7, MEDIX). None of these approaches allows the solution to be on the individual's desk and hence tailored for (and possibly by) the individual. VIRTUAL DATA INTEGRATION is the apparent integration of information at the desktop, without modification to the existing infrastructure of niche applications, custom and standard interface protocols, and organizational boundaries.

All of the current approaches to solving the integration problem alleviate some of the problem, but they still suffer from several drawbacks:

1. None of them provides UNIFORM ACCESS to the data. The user must still be aware of the origin of the data, and must deal with data from different sources in different ways.

2. None of them provides CUSTOMIZED ORGANIZATION of the data. Each individual application program still specifies the way in which its data is accessed and viewed. In particular, it is not possible to intersperse data from one source with data from another.

3. None of them provides USER EXTENSIBILITY to accommodate user preferences or new data types.

4. They require extensive (and expensive) custom programming and continual program maintenance.

Any realistic solution to the healthcare integration problem must address a specific set of requirements. First and foremost, it must solve these four problems. Second, it must capitalize on existing infrastructure — very few institutions can afford to discard their existing information technology investment. Third, it must be evolutionary and inexpensive. The Organization Engine is a prototype designed to demonstrate that virtual data integration satisfies all of these requirements.

The key to virtual data integration is providing each user with a single view of all data — regardless of such details as the data's location, storage format, metadata model, and so on — without disturbing the implementation of the various systems that manage the data. The choice of this single view is at the root of a successful virtual data integration effort. The Organization Engine explores one particular user view: all data is composed of a set of fields; each field has a value and an icon; the icons are used to indicate a user-tailorable combination of the operations that can be performed on the field's value and the origin of the data. For example, the Organization Engine provides icons for "folders" of user-configured data, as well as "folders containing medical reports as filed in the hospital's medical computing system."

The current Organization Engine prototype is connected to two pre-existing healthcare applications as well as a variety of desktop utilities that deal with multimedia datatypes (files of sound, clinical images, document images, flat text, and structured text). Extending the set of datatypes is straightforward (see Section 2.1). Most of this paper is devoted to the considerably harder task of integrating existing healthcare applications, which is demonstrated by examining the techniques used to integrate the DECrad radiology information system and the Fox Chase Cancer Center's medical computing service (MCS).

DECrad

DECrad provides hospital radiology departments with patient registration functions, exam tracking and scheduling, diagnostic report and film library management operations, accounting functions, and management reporting. DECrad is implemented in Digital Standard MUMPS (DSM), running on Digital's VAX/VMS platform.

DECrad's history is illustrative of the integration trend described in the introduction. Initially it operated in isolation, providing departmental management functions but no ability to present its information outside the radiology department. An increasing awareness of the need to share data led to the implementation of a proprietary protocol used by DECrad to communicate with other hospital units about the essential events in the life cycle of a radiology exam. Lacking standards, a number of other healthcare vendors adopted the protocol despite its proprietary nature. Recently, a drive for *de jure* standardization of similar interface protocols has produced (among others) the HL7 standard, and DECrad now supports this standard.

MCS

The Medical Computer System developed at the Fox Chase Cancer Center (FCCC) provides standard hospital information system functionality, as well as capabilities that are of specific value in the treatment of cancer patients. The MCS provides the staff at FCCC with the ability to register patients, perform patient admissions, discharges, and transfers, transcribe radiology reports, histories, physicals, and discharge summaries, schedule diagnostic procedures and outpatient visits, and track the location of patient medical record charts and radiology films. Interfaces were written to purchased systems for pharmacy and clinical laboratory, and data on patient services provided are passed over a local area network to a purchased hospital billing package. Oncology-specific features include cancer staging, available treatment display, drug side-effect information, and pre- and post-diagnostic "workup" guidelines.

Development of the MCS began in 1984 on a Digital VAX cluster platform with character cell terminals. Digital's DBMS network model CODASYL database product was used. The MCS was built as a central clearinghouse and repository for all clinical information. It is a typical example of the "agglomeration" approach. In 1989 a migration was initiated to a distributed network of Reduced Instruction Set Computers (RISC) with X-terminals as clinical workstations. Radiology images from CT and MRI are available at the X-terminals, along with text and graphics. The databases still remain on the VAX cluster, which is now effectively a database server. Database transactions pass between the RISC machines and the VAX cluster using an in-house developed protocol.

The remainder of this paper describes the Organization Engine prototype system. Section 2 describes the basic toolkit provided by the Organization Engine. It is this basic set of software that allowed the prototype to be constructed in eight programming days. The Organization Engine assumes that it has on-line access to its data sources, and Section 3 describes some of the issues surrounding the choice of network protocols. The majority of the integration work is spent designing and building the small pieces of code that connects the Organization Engine with a particular data source — the server application programming interface ("server API"). Section 4 describes two generally applicable models for server APIs and offers comparisons which may be useful to system designers facing virtual data integration problems. The final two sections contain observations, measurements, and conclusions.

2 The Organization Engine

The Organization Engine began as a research project in Digital's Cambridge Research Laboratory. The goal was to provide a software base for use in exploring the issues that arise from dealing with vast quantities of data, primarily located in "legacy repositories" — pre-existing systems that merge raw data with structuring information in individual, idiosyncratic ways. The software is divided into three distinct pieces, two of which were stable over a variety of information sources, and the third carefully tailored to each specific repository. The two generic pieces were a user interface (described in Section 2.1) and an integration toolkit (Section 2.2) which provides the connection between the user view of data and a transmission protocol. The repository-specific component implements this transmission protocol for a particular data repository. The design and implementation of the repository-specific component is the subject of Section 4.

2.1 User Interface

In building the Organization Engine prototype, care was taken to define an abstraction barrier (modularity boundary) between the user interface and the underlying integration system. This modularity allows the researchers to modify the user interface without affecting the implementation of the integration system, and vice versa. The choice of the initial user interface was thus seen as a "trial balloon" rather than an integral aspect of the design of the system.

The user interface of the prototype Organization Engine is built to encourage users to think about navigation (browsing) rather than search (querying) as a means of locating information. The goal is to make it natural for users to think of placing information in multiple places so that locating it later will be easy. "Copying" an object actually means copying a reference to the object, so it is very fast and efficient. In fact, the implementation of the system allows some simple queries to be constructed and executed through this browsing interface, but the user is unaware of these queries. From the user's perspective, the interface looks very much like a set of "blotters," each of which is similar to the Macintosh^{tm}FINDER or MS/DOS Windows^{tm}DESKTOP. The blotter contains various named icons corresponding to data objects, with the shape of the icon reflecting the operations that can be performed on the data and/or the origin of the data.

Using the mouse to double-click on an icon ACTIVATES the icon. The meaning of "activate" is quite flexible. A simple text file is used to customize the user interface; it maps the name of an icon to the operation that should be performed when it is activated. There is a standard operation that can be used to activate an icon to reveal another blotter (i.e. to make the icon behave like a folder). Another pair of operations allows a specific application to be run (the application name is in the customization file) and given either the name of the icon of the contents of the icon as a command argument. This mechanism makes it very simple to add new data types to the system: a new icon is designed and installed, and the customization file is edited to map that icon to the application that can handle the data type.

The Organization Engine's user interface is the focus of ongoing research. In particular, the ability to browse through data is rather limited in the current interface, and the lack of good query facilities limits its applicability. Finding a new metaphor that encompasses both is under active investigation.

2.2 The Integration Toolkit

At the core of the Organization Engine is a toolkit that is used to connect the user interface with a wide variety of different data sources. See Figure 1. The user interface communicates with this toolkit through an API that allows the user interface to ask for the field names within a record, for each of those ask for the contents and type of the field. It also allows the creation of new records, fields within records, modification of field contents and so forth. The user interface specifies a record by providing a record identifier which it has previously received from the central toolkit (as the value of a field of some record) or one of a set of "magic" identifiers that each data source specifies as the roots from which other records can be located.

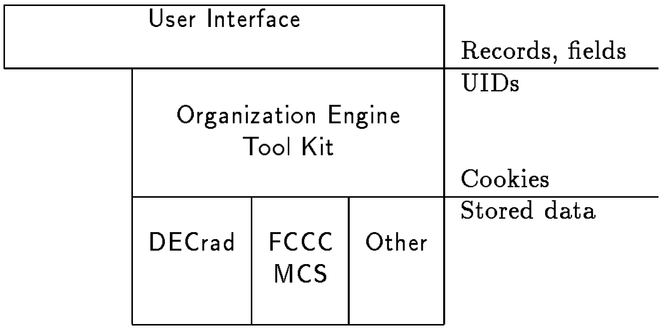


Figure 1: Structure of the Organization Engine

The central toolkit is responsible for converting these operations into a smaller set that communicate back to the actual source of the data. There are only six operations at this end: initiate a connection with a data source, get a record (from the data source into the Organization Engine), create a new record, replace the contents of an entire record, replace the contents of a single field of a record, and close the connection to the data source. The toolkit communicates with the data source by specifying an identifier for records which previously came from that data source — just as the user interface communicates with the toolkit itself.

There is one important feature to notice about these interfaces: the side in possession of the data always specifies the identifier for the data. Thus, the user interface can only use identifiers that it received from the Organization Engine toolkit (or special tokens), and the Organization Engine toolkit can only use identifiers it received from the data source (or special tokens). The Organization Engine does *not* merely pass on these identifiers unchanged. Instead, it combines the identifier with a marker that allows it to identify the data source that produced the identifier. We refer to the identifiers passed between the Organization Engine and the data source as COOKIES, and the identifier passed between the Organization Engine and the user interface as OBJECT IDs. Thus, an object ID consists of two parts: a data source identifier and a cookie belonging to that data source. This technique resembles dynamic data typing, tagged records, and some object-oriented programming implementations.

The toolkit is intended as a body of code that must be extended to be useful. For each new data source, two things must be provided to integrate into the Organization Engine framework. The first is a service identifier to be used by the Organization Engine to refer to data from the new source of information. This can be either allocated by some form of universal naming authority (such as the ISO unique identifier name space) or simply be local to this installation of the Organization Engine, as the implementor prefers. The second is a body of code, typically small, which implements the six operations required by the Organization Engine toolkit of the data source. The next two sections provide some guidance in designing and building this code.

3 Communications and Interconnection

The single most important property of the communication mechanism is that it must be invisible to the user. There is probably no component of an overall system less interesting to the typical healthcare professional than the networks on which it is based. There is no set of problems less interesting to that professional than difficulties arising from interconnecting systems or networks. These mechanisms are very properly viewed as the portion of the system that the programming staff should hide.

The Organization Engine prototype gives only a small amount of help or guidance here. First, the protocol used to connect to data sources is deliberately simple: six commands (open, get, create, reply, modify, close). The protocol makes no commitment to the location of the client/server split: any reformatting work can be done either at the user's workstation (where the Organization Engine toolkit runs) or at the data source. The names of fields in individual records can be coded on either the client or server — for data sources where the format is fixed it is reasonable to put this burden on the user's desktop, for those where records are not predictably formatted it must go at the data source. The network transmission protocol is not specified by the toolkit, so this choice, too, can be made as part of the per-data-source integration code.

The current Organization Engine prototype has already adopted two different interconnection styles: one uses Remote Procedure Call to connect the toolkit to the data source, and the other uses DECnet task-to-task. Each of these mechanisms has advantages depending on the nature of the network and the data source. RPC is preferred when the network environment is multi-platform or multi-protocol, or if large amounts of data must be moved between the Organization Engine client and the data source server. DECnet task-to-task works well between two machines that both support DECnet and where the data to be moved is always in small amounts of predictable format.

4 Server Interface Models

The hardest part of integrating a new data source using the Organization Engine toolkit is designing the overall structure of the server. This section describes two broad classes of server API architectures, the TRANSACTION-ORIENTED model and the DATA MAP PUBLICATION (DMP) model, and provides a comparison of the two that can assist in choosing between them.

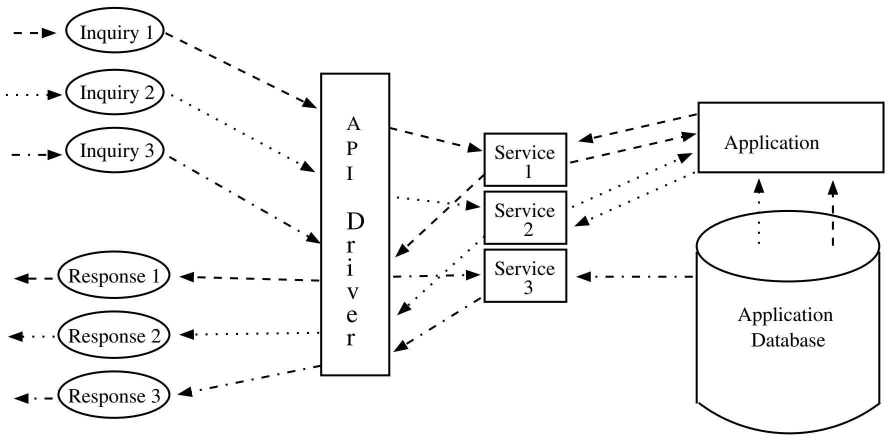


Figure 2: Transaction-Oriented Server

A transaction-oriented server allows information to be retrieved through a specific set of inquiry and reply transactions. Each inquiry message specifies the key values needed to retrieve the data; the type of the message determines the nature of the data retrieved. These servers are implemented by a top-level driver that determines the type of inquiry and dispatches routines to gather and format the information into the correct response. Such a server contains a separate code path for each inquiry message. Figure 2 shows the components of a transaction-oriented server.

A DMP server makes use of a "map" of its application dataset which is distinct from the application itself. The data map represents a publishable subset of both the data elements in the application dataset and the relationships between those elements. The server uses this information to allow clients to browse at will among the published information. The top-level server code based on this model is more complex than the corresponding code for a transaction-oriented server. However, making the map explicit allows the server to be independent of the number and nature of data elements accessible through it. Figure 3 shows the components of a DMP server.

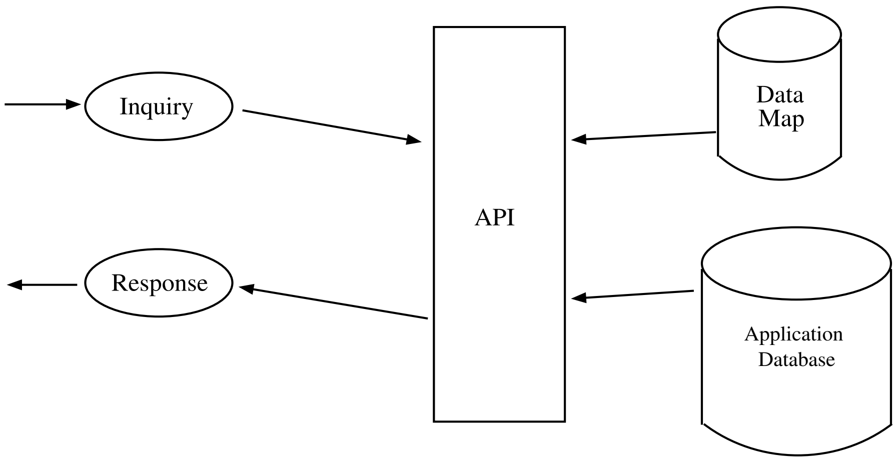


Figure 3: Data Map Publication Server

The choice between these two models is a trade-off: neither is clearly superior to the other. The decision must be made along three distinct dimensions, and evaluated for each data source to be integrated:

Initial vs. continuing development cost.
The transaction model is easily built, and corresponds directly and naturally to many existing applications. Extending this model "across the network" to desktop computers is relatively easy, and this was precisely the approach taken by the Fox Chase Cancer Center in building their medical computing system. The DMP model requires a more complex initial system, since it requires code to implement and interpret the data map. Over time, however, the flexibility inherent in simply changing a data structure (the map) as opposed to writing new code (for new transactions) can become a dominant factor in development costs. In fact, the DECrad integration allows the data map to be changed while the interface is in use, and desktop users may see the effects of the change as soon as their next interaction.

Control vs. flexibility.
If two different transactions consult the same underlying table, a transaction-oriented server can easily implement distinct access controls for the operations, while a DMP server would be almost powerless to enforce the distinction. On the other hand, the DMP model allows clients to browse the data in any order that they find useful.

Performance A transaction-oriented server can achieve better throughput than a comparable DMP server, since the former dispatches retrieval and formatting code depending on the input message, while the latter must "interpret" the map. The performance difference depends largely on the complexity of the path that must be interpreted by the DMP server, and to a smaller degree on the complexity of the map itself.

The transaction-oriented server model is easily implemented and well understood. The data map publication model, however, is less familiar and bears closer examination. At first, it may appear that the data map publication model requires access to metadata, a feature frequently unavailable in existing systems. Yet even in a database that supports metadata, the published map may or may not be derivable from the standard metadata. The remainder of this section describes the general requirements a DMP server places on the data source.

The key concept in the DMP model is the data map, and this in turn depends on the concept of DATA NAMES: the association of a unique name with a particular class of information. Given a dataname and any prerequisite information for that particular dataname, it must be possible to locate the method for retrieving all values (instances) of that class. Instances of information classes may be either singly- or multiply-valued. This definition of the dataname concept accords with the usage of the term generally accepted in the context of fourth-generation data dictionary environments.

In addition, a data map describes an application dataset by associating with each dataname the attributes and values necessary to navigate the dataset. With respect to any attribute, the dataname to which it pertains may be termed its SUBJECT. Similarly, any values the attribute assumes for a given subject may be termed its OBJECTS relative to that subject. The following list describes several of the most important attributes used in constructing data maps:

HAS-DEPENDENT : object data name is a DEPENDENT of subject data name; i.e., the value of the object data name cannot be evaluated without knowing the value of the subject data name. The value of the object of a has-dependent attribute "becomes available for inspection" when the subject dataname's value is known.

REFERENCE for datanames with a single value, the object of this attributes specifies a retrieval method.

NEXT-VALUE-METHOD for datanames with multiple values, the object of this attribute specifies a method for iterating through all of the values.

POINTER-TO indicates this dataname is an "invisible link" between two other datanames. When the subject of this attribute is encountered as a dependent, its value should be assigned to the object dataname, and the object's dependents should be evaluated instead of the subject's.

It is convenient to represent the datamap as a three level hierarchical index in which subjects point to attributes, which point to values. This representation scheme is an instance of the entity-attribute-value data model. In such a hierarchy, either all attributes for a given subject, or all objects for a specific attribute of a subject, can be retrieved quickly. For example, when an Organization Engine user selects an object, this scheme allows the DMP server to retrieve quickly all data names to be evaluated for display. The DMP server accomplishes this task by retrieving all objects of the HAS-DEPENDENT attribute for the data name corresponding to the selected object.

5 Observations and Measurements

Experience to date has been far better than expected for a first attempt at combining a research prototype with pre-existing systems. The low investment required (eight programmer days for two systems) confirms that the virtual data integration approach is an important approach to solving a major problem in healthcare information systems. The prototype is more than adequate for its intended purpose as a demonstration system. Initial indications are that the prototype might be useful in solving existing healthcare integration problems. With the low cost of integrating with other systems, the Organization Engine provides an interesting base for experimental user interfaces as well.

Quantitative performance characterization confirms that DMP servers can pay in performance for their superior flexibility. In an informal benchmark, the DECrad server retrieval logic, the only prototype component easily thus characterized, exhibited a nearly linear relationship between number of entities retrieved and elapsed CPU time. A MicroVAX II CPU (0.8 VAX Units of Processing (VUPs), or approximately 0.8 MIPS) required an average of 0.26 VUP-seconds per retrieved entity (with a standard deviation of 0.036 VUP-seconds).

Qualitative results are mixed. The MicroVAX II system measured objectively provided unacceptable user performance: on the order of fifteen elapsed seconds to retrieve a moderately-sized patient record! Results were quite different, however, on the faster VAX 4000-500 (12 VUPs). There, the prototype exhibited apparently instantaneous response when retrieving similar patient records, despite the addition of a number of items that would tend to impede performance: retrieval included the invocation of the Organization Engine client RPC stub, two-way DECnet transmission on a moderately-loaded network, activation of the DECrad RPC server, and internal task-to-task communication between server processes, in addition to the retrieval logic tasks tested in the trial described above. It is important to note that both the qualitative and the quantitative results presented here were obtained without the benefit of any system tuning, and should not be used to predict operational characteristics of live systems.

6 Conclusions

A data map publication (DMP) server, which allows client applications to browse the public parts of a data set, provides a flexible base for virtual data integration. Building a DMP server is initially more difficult than building a transaction-oriented server, but the evolutionary path is simpler. It is possible to implement a DMP server for any application dataset that possesses the following characteristics:

- the ability to obtain a value from a dataset, given a singly-valued dataname and a list of prerequisite datanames and associated values
- the ability to iterate (without duplication) over the values for a multiply-valued dataname, with prerequisite information
- the ability to store a the published data map, including prerequisite (dependency) information, and invisible pointers.

Because a DMP server inherently requires the ability to interpret data map paths at runtime, it can be less efficient than a transaction-oriented server. It is not yet clear how serious this performance difference is in practice. Future work will both quantify this difference and reduce its current level.

Underlying this work is the toolkit provided by the Organization Engine. The structure of this toolkit has proven quite robust and is the main reason our modest labor investment in integration has been so effective. The Organization Engine provides a strong modularity boundary between the user interface and the underlying integration toolkit, allowing the replacement of the user interface without an overhaul of the entire system. The integration toolkit at the heart of the Organization Engine provides little more than the glue necessary to connect this user interface API with the simpler API used to connect with a data source. The user interface boundary deals with object identifiers (a data source identifier and a "cookie" belonging to that data source), and provides an abstraction of records with fields. At the data source interface, the abstraction is one based on "cookies" and six fundamental operations: get, create, modify, replace, open connection, and close connection.

The virtual data integration method represents an effective, low-risk means for healthcare institutions to improve dramatically the accessibility of critical information without jeopardizing existing technology investments. The system described in the preceding sections is a prototype, not a finished product. Several issues remain to be addressed in future prototypes and pilot implementations, including security, multiplatform clients, inter-institution integration, client modification of server data, and others. However, the Organization Engine prototype presented here is exciting evidence that virtual data integration may help to solve some of the most critical information problems in healthcare today.