Experiences With an Object Manager for a Process–Centered Environment

Dennis Heimbigner

CU-CS-484-91 Revised 6 September 1991

University of Colorado at Boulder

Technical Report CU-CS-484-91
Department of Computer Science
Campus Box 430
University of Colorado
Boulder, Colorado 80309

Experiences With an Object Manager for a Process–Centered Environment

Dennis Heimbigner

Revised 6 September 1991

1 Introduction

By now, there is a general understanding that object management[1] is central to software engineering environments. It serves as one of the primary means for integrating components of the environment by providing a common set of data structures (schema) and a shared repository for persistent objects. A sufficiently dynamic object manager is also important in providing extensibility in an environment by allowing incremental extensions to the schema and hence to the range of tools that can share information.

The Arcadia project[22] is constructing an environment which is one of the first of a new class of process–centered (or process–driven) engineering environments. A process–centered environment is one in which the programmer is guided in the task of producing software according to some methodology. Such an environment extends the more traditional tool–oriented environment by adding the capability to specify the process by which software is to be constructed. This is in contrast to a typical tool based environment in which the programmer is presented only with a collection of tools and is given no help in deciding how to apply those tools to produce a software product.

It is assumed that a process-centered environment will be controlled by a model of the process written in some formalism. Osterweil[13] has proposed the use of an executable programming language as that formalism. Such a language is called a process programming language (PPL). Arcadia uses a process programming language approach as the basis for its environment. Until recently, our (only) PPL was APPL/A[20].

Arcadia has been active in object management since its inception. The Arcadia approach has consistently been to use existing database systems and to augment them with innovative features necessary to support a process-centered environment. We have taken this approach because we have found that database research systems provide many, but not all, of the capabilities needed to support a process-centered environment. Modern database systems are very large and complicated pieces of software, spending tremendous amounts of effort to furnish reliable, secure, efficient storage management and concurrency control. Different research database

This material is based upon work sponsored by the Defense Advanced Research Projects Agency under Grant Number MDA972-91-J-1012. The content of the information does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

[1]We will generally use the term object management and object manager rather than terms such as database management, but the various terms should be considered essentially interchangeable.

systems provide, in addition, such features as schema dynamism, long transactions, and very large object management. Arcadia has no wish to duplicate these capabilities. Unfortunately, no single database system provides a combination of these capabilities that is sufficiently complete to support a process-centered environment. Thus our approach has been to augment existing systems in order both to provide increasingly satisfactory object management in Arcadia and to gain clearer understanding of the requirements for object management in a process-centered environment.

Triton is one of the object managers in use in Arcadia. It is primarily intended to support the process-related activities within the Arcadia environment[2]. Triton may be briefly characterized as a serverized object repository providing persistent storage for typed objects, plus functions for manipulating those objects.

Triton uses an existing object manager, Exodus, to provide much of its functionality (Basic type model, buffering, persistence, etc.). Exodus may be considered a persistent programming language system rather than a true database system. It consists of a storage manager and a persistent programming language named E [15]. E may be considered as a persistent version of C++, and like C++ it has the C type system augmented by classes with behaviorally defined methods. Triton adds value to Exodus by adding features needed in Arcadia, but not directly provided by Exodus. Some of these additions, such as triggers, are driven by the needs of process support and some by the nature of persistent programming languages.

In this paper, we will explain the requirements for Triton, and describe its architecture. We will then show some of its features: heterogeneity, the interface, late binding, process language support, and triggers. Finally, we will describe our observations, insights, and lessons gained in the process of constructing and using Triton.

2 Requirements

The initial design of Triton was influenced by four general requirements that we felt were essential to support any process-centered environment.

- Efficient support for the wide variety of software artifacts used within Arcadia: IRIS graphs, requirements and design nodes, configuration management graphs, test cases, documentation, and so on.
- Support for process coding languages–especially APPL/A. This requires a system supporting at least relations and triggers, or some equivalent form of event notification.
- Standard database concurrency and recovery mechanisms.
- Some form of behavioral object-orientation.

Obviously, a number of desirable capabilities are missing from this list. But we concluded, after examining what was available, that no system that was obtainable at the time would completely satisfy even this minimal set of requirements. We could find systems that had, for example, transaction management and triggers, but that was only accessible through a fixed programming language, or had difficulty with dynamic type creation. We decided that our only recourse was obtain a database manager offering a close match to our needs and to modify it.

[2]Arcadia uses several object managers for a variety of purposes. See [25] for information about another object management activity within the Arcadia project.

As the Triton project progressed, our understanding of the problems of object management deepened. The general requirements were elaborated and additional requirements were added to reflect our increased knowledge.

The original and still primary requirement for Triton is to support the process programming languages used in Arcadia. In practice, this reduces to direct support for the APPL/A process programming language and its features (see section 7).

Multi-language interoperability is another important requirement for Triton. This requirement has become increasingly important as Arcadia has evolved and we now view this requirement as a central one. It has been a driving factor in the architecture of Triton.

Multi-language interoperability covers two capabilities. First, we require the object manager to be accessible from programs written in a variety of programming languages. Currently, Arcadia has components written using Ada, C, C++, Lisp, and Prolog.

Second, it must be possible for programs written in various languages to share data. There are two typical ways to achieve this: (1) pair-wise conversion or (2) use of a common data model. We rejected choice one as being ultimately too time consuming and chose instead to use the common data model approach, even at the expense of such problems as incomplete model mappings.

Late binding of schema elements is another requirement for Triton. In many database systems, this requirement would seem to be automatically provided. Unfortunately, Exodus, like many persistent programming language systems, did not have this capability, and so it was an additional problem to be addressed.

A catalog (or meta–database, or data dictionary) is a necessary corollary of dynamic schema definition. Again, Exodus did not have this feature because of its relatively static nature.

In addition to purely technical requirements, there was a requirement to reuse as much existing software as possible. If constructed from scratch, Triton would have taken too many resources to be practical. So from the outset, it was important to avoid re-implementation. As a consequence, Triton was constructed using as much existing database technology as possible. It was important to focus the Triton effort onto those features essential to Arcadia and to reuse those components that were properly the domain of the database community.

3 Triton Architecture

Figure 1 shows the architecture of Triton. It is a client-server architecture in which the client communicates with the server using a Remote Procedure Call (RPC) protocol. In this case, we used Q[11], which is a variant of the Sun RPC/XDR protocol that has some adaptions for multi-language interoperability. The architecture of the client shown in the figure will be deferred to section 7. Suffice it to say that it communicates using RPC to call the interface functions provided by the server.

The server has five major components.

1. The *server interface* handles the details of receiving requests from clients (there may be more than one), invoking the appropriate local procedure to field the request, and returning any result back to the client.

2. The *catalog* component is a meta-database written as a collection of E types. It records the structure of the schema currently known to the server.

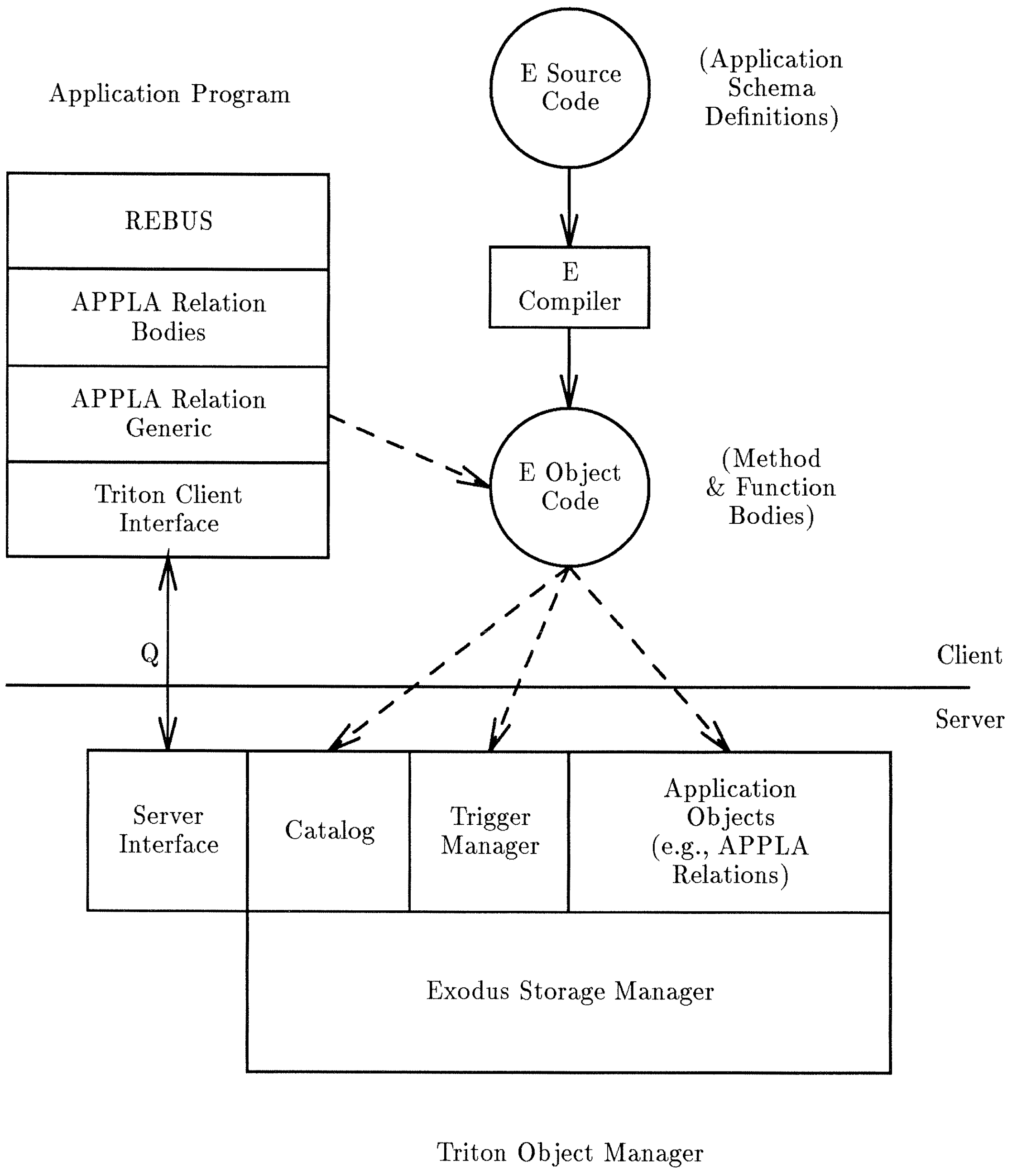

Figure 1: Triton Architecture.

3. The *trigger manager* interacts closely with the catalog and manages the attachment of triggers to various schema elements and their subsequent invocation.

4. The *application objects* are instances of the schema elements comprising the data used by an client application.

5. The *Exodus Storage Manager* provides for the persistent storage of objects from the other components.

The requirement for multi-language interoperability was critical in determining this final Triton architecture. It soon became clear that we could not count on being able to place any portion of the code for the object manager (including method code) in the client (see section 9.4 for more on this issue). As a result, we have settled on a client-server architecture with a Triton server residing in one Unix process and each client residing in a separate Unix process. This solves many of the multi-language problems by placing the Triton system in one address space and restricting other language programs to separate address spaces. In principle, any language that can support RPC can use the Triton server.

Multi-language data sharing is achieved using a common data model. A subset of E, the Exodus persistent programming language is used as the common data model. A major problem in using a common data model is data model mapping. For any given type in the client language there must be a mapping to some equivalent E type(s).

Late binding of schema elements proved to be a difficult problem because of the inherently static nature of a persistent programming language like E (see section 9.2). In Triton, late Binding is provided by allowing compiled E code to be dynamically loaded into the Triton server. Section 6 will discuss this process in some detail. In Figure 1, this is shown by the column of figures in the upper right.

In order to support dynamic loading, it was essential that Triton have a catalog to record information about schema elements (e.g., class and method definitions) and for linking to dynamically loaded pieces of E code representing methods and triggers In its intended use, E/Exodus had no need of a catalog and so it was necessary to construct one as part of the Triton server.

4 Heterogeneous Access to Triton

Triton is designed to operate in a heterogeneous environment. Heterogeneous in this case refers to differing machine architectures and/or differing client side programming languages. Minimally, it is assumed that it is possible to have TCP/IP (or UDP) connections between the client and the Triton server. In order to understand some of the Triton interface, it is necessary to understand how Triton manages heterogeneity.

A common remote procedure call protocol is used on top of TCP/IP to provide a more structured access method. Remote procedure call operates by *marshalling* the inputs to the procedure on the client side. That data is sent to the server along with some handle specifying the remote procedure to be invoked. The server *unmarshals* that data, and calls the appropriate procedure. It then takes any result, marshals it and returns it to the client.

Triton uses a standard intermediate format for sending data between the client and server. During marshalling, the input data is traversed and converted into a canonical linear form suitable for transmission over a thin wire connection. During unmarshalling, the linear form is processed to create a copy of the original data. The term XDR (eXternal Data Reference) is used to refer to the canonical linear form. The terms linearize and delinearize are used to refer to the translations between the XDR format and some local data structure.

In Triton, linear XDR strings are stored in data buffers typically called *XDR_buffers* and pointed to by *XDR_Handles.*

5 The Triton Interface

The Triton server presents a procedural interface to its clients. That is, to a client it "looks" like a library of procedures for manipulating schema elements and objects. Triton makes significant use of *handles*, which are references to objects in the server. The client can only get handles from server, copy them around, and send them as arguments back to the server. The client has no knowledge of the internal structure (if any) of the handles.

Manipulating and accessing the Triton catalog represents a significant portion of the operations provided by the server. The Triton Catalog provides two major capabilities.

Schema Definition: These operations allow a client to dynamically define schema elements into the catalog, and many return a handle to the defined schema element. The definable schema elements are classes, methods, functions, and formal arguments to methods and functions. There are corresponding schema operations to *destroy* elements, but their semantics are not well-defined.

Name Space: The space of schema elements in the catalog is almost flat. At the top level are uniquely named classes and functions. Classes "contain" named methods, and methods and functions "contain" named formal arguments. The name space operations allow clients to convert a name of a schema element into a handle for that element.

The primary activity of the interface is to receive requests from clients to invoke methods or functions defined in the catalog. The operations for doing this are shown in Figure 2. The normal E operation of invoking a method (roughly speaking, *results = Instance.method(inputs)*) is mirrored in the arguments to the *evaluate_method* interface operation:

EPOINTER instance: This is a handle to the instance object.

T_method_p m: This is a handle to the method to be evaluated for the specified instance.

XDR_Handle inputs: This is a pointer to an XDR buffer (see section 4) containing the linearized inputs, if any.

XDR_Handle results: This is a pointer to an XDR buffer into which any results from the method evaluation will be inserted.

The *evaluate_function* operation is similar, but no *instance* argument is required.

The *create_instance* operation (Figure 2) is used to create instances of objects. It takes a handle to a class, an XDR buffer of arguments to the class constructor, and a handle to a *collection* object. It returns a handle to the created object. In E, a collection type is a built-in aggregate type. Instances of collections are automatically persistent. Objects must be created as a member of some collection object in order to be persistent. Technically, this operation can also be used to create "non-persistent" objects; they will have a lifetime as long as the server. But to date, no use has been made of non-persistence.

```
int evaluate_method(EPOINTER instance, T_method_p m,
                    XDR_Handle inputs, XDR_Handle results);

int evaluate_function(T_function_p f,
                    XDR_Handle inputs, XDR_Handle outputs);

EPOINTER create_instance(T_class_p cl,
                    XDR_Handle inputs, EPOINTER collection);
```

Figure 2: Method and Instance Manipulation.

6 Late Binding

In E, a method (or function) schema element is defined behaviorally. That is, it is associated with a piece of code that is executed when the method is invoked via, for example, the evaluate_method server interface operation. Methods may be dynamically defined in the catalog and so Triton must have some method of dynamically defining the code associated with the method.

Our approach is to dynamically load compiled E code into the Triton server. For this to work, Triton requires some substantial support from the operating system loader, such as the standard Berkeley Unix loader, *ld*. With this loader, it is possible with take a piece of relocatable binary code, pass it (plus some extra information) through the loader and get in return a piece of absolute code. This piece of absolute code can then be read into the data space of the Triton Server and executed as required. Thus, to completely define a method, it is necessary to define it in the catalog and then to load the corresponding compiled method body into the server and associate the code with the schema element.

Unfortunately, in Triton there are two definitions of the structure of, for example, a class type. One is the structure defined in the catalog. The other is the structure implicit in the compiled method code. It is possible to have inconsistencies between the two definitions, and this is, of course, deprecated. When and if the server takes more control over the source code for methods, this inconsistency will no longer be possible.

The process is actually rather more complicated than described. In addition to handling the code, the server must be prepared to search the symbol tables associated with the code. Searching these tables allows the server to find the correct entry point for invoking the method code. In practice, one combines a whole group of methods into one piece of code, loads all of the code at once, and allows the server to extract out the entry points for all the methods from that piece of code.

Triton also provides a limited form of code removal. The pieces of loaded code act like a stack. Loading a piece of code "pushes" the code onto this stack. A client can request the stack to be "popped", which will remove the most recently loaded code and cause any methods in that code to be marked as undefined.

Debugging of dynamically loaded code is quite difficult. Even if the debugger is in control of the server, it is difficult to get it to recognize the existence of the dynamically loaded code.

If the code is seriously defective, then it can cause the whole server to fail. If it is only "mildly" incorrect, it can be unloaded from the code stack, corrected, recompiled, reloaded, and retested. This is often a time consuming and frustrating process.

7 Triton Support for APPL/A

APPL/A [20, 21] is a prototype process programming language. It is defined as an extension to Ada [24]. Ada provides several general-purpose capabilities to address the special needs of software process programming. The principal extensions that APPL/A makes to Ada include programmable persistent relations, triggers on relation operations, optionally-enforcible predicates on relations, and several transaction-like composite statements.

APPL/A assumes that its relations are persistent and so requires access to some form of object management systems such as Triton. APPL/A, as the first process programming language used in Arcadia, has had a significant influence on the structure of Triton. For example, APPL/A (since it contains Ada as a subset) is obviously is very different from E. Thus it introduces the problem of heterogeneous access. Additionally, APPL/A has relations and triggers, and this imposed a requirement on the object manager to support those two features. Note, however, that Triton does not provide support for the APPL/A transaction statements[3].

This section will discuss some of the details of the Triton support for APPL/A relations. The discussion of triggers is deferred to section 8. In APPL/A, a relation looks much like a *task* in that it defines the structure of the relation tuple and a limited set of operations: insert, delete, update, and find. This last operation (find) is used to provided a combination of tuple-at-at-time access and associative retrieval. As with Ada tasks, a relation has two pieces in the form of a specification and a body. The body is expected to provide implementations for the interface operations defined in the specification. See [20] for details.

It is fortunate that both Ada and E provide support for generics; the support for relations heavily uses generic components both on the APPL/A (client) side, and on the E (server) side. Basically, both the client and server sides define a generic relation unit ("package" in ada, "dbclass" in E) parameterized by, among other things, the tuple type. In order to define a relation, one defines the tuple type and instantiates the generic relation using that tuple type.

The generics on both sides (client and server) are matched in that the body of the generic on the client side "understands" the interface of the generic on the server side. When a client side entry is invoked, it can in turn invoke appropriate entries on the server side using, for example, the evaluate_method interface operation. Each generic provides entries matching the entries of the standard APPL/A relation interface: namely, insert, delete, update, and find.

Additionally, each generic is parameterized by a type representing the structure of the tuple. It is critical that the tuple types on each side be compatible. This boils down to requiring the definer of the relation to specify the correspondence between an APPL/A tuple type and an E tuple type.

The other critical parameter for each generic is an XDR procedure for linearizing and delinearizing tuples. It is important that the two procedures match in their understanding of the structure of tuples. Given a tuple on the client side, it uses its procedure to linearize the

[3]Unfortunately, no existing Object Manager is capable of supporting these statements as yet.

tuple into a canonical form for sending to the server. The server must then use its procedure to delinearize the tuple into its local form.

Given this generic, a relation body is created by instantiating this generic with an appropriate tuple type and XDR procedure. The relation body entries are then defined to invoke the corresponding operations in the generic. The body of the generic understands how to communicate with Triton, and with the corresponding relation located there.

As with the APPL/A side, a Triton relation is created by instantiating the E generic with an appropriate tuple type. The Triton interface is then used by the client side relation body (via the APPL/A generic) to access this relation instance. The actual relation contents are stored on Triton in association with this relation instance.

Referring back to Figure 1, we can now examine the client architecture. The client shown there reflects the various layers required by an APPL/A program to communicate with the server. The top level application (called REBUS in the figure) is defined in terms of a collection of APPL/A relation specifications. These specifications are implemented by relation body code. These bodies use the APPL/A generic relation interface, which is in turn implemented by a generic body. This generic body is defined using the Triton client interface library.

8 Triggers

Triton has augmented the E capabilities with a simple form of trigger. A trigger is a piece of code that is invoked whenever some *event*[4] The key feature of a trigger is that the generator of the event need not explicitly invoke the trigger. Rather, the underlying system implicitly invokes the trigger when the event is detected.

In Triton, the events that can invoke triggers are (1) method or function invocation and (2) method or function completion. There are important restriction on trigger attachment.

- Only methods or functions invoked via the evaluate_method or evaluate_function interface calls can cause triggering. Thus, an internal call from one function to another will not cause triggering.
- Instance specific triggers are not provided. If a trigger is associated with a method, then every invocation of that method for all instances will invoke the attached trigger.

These restrictions have limited the utility of triggers. The original intent to support APPL/A failed because it required trigger code to be rewritten from APPL/A to E and, more importantly, there was no mechanism for triggers to communicate back to clients asynchronously.

9 Some Lessons from the Triton Experience

Constructing Triton has been an enlightening experience for us. It has done much to make clear what the actual requirements are for object managers when they are expected to support a process-centered environment. The following sections describe some of the lessons that we learned from the Triton effort.

[4]A state-based approach, as in AP5[4], is also possible in which the trigger is invoked when some defined *system state* is reached. Triton does not support this style of trigger.

9.1 Performance

The performance of Triton leaves much to be desired. On a Sun3, using Sun's RPC/XDR and UDP, performing an evaluate_function on a function with an empty body and with no input or output takes about 20 milliseconds for a round trip. The primary costs are in (1) RPC communication, and (2) access to the catalog (necessary for triggers and to locate the function code). There are number of known optimizations that can be performed to reduce this number. With some effort, it appears that a round trip in the range of 1-5 milliseconds is possible. It is clear that for some applications, this cost is still too high. Achieving substantial additional speedup will require sacrificing something: the use of RPC, heterogeneous access, or catalog mediation.

9.2 Using A Persistent Programming Language

We seriously underestimated the amount of effort that it would take to use a persistent programming language system as the basis for Triton. In recent years in the database community (and in the commercial word as well), many new research efforts have assumed a persistent programming language as their basic architecture[8, 15, 23]. The assumption is implicit in this approach that all the programs that access stored data will be written in whatever persistent programming language has been chosen. Even in the cases[8, 23] where multiple languages are, in theory, supported, there is no obvious provision for sharing data between those language.

This language-specific approach turns out to be completely incorrect for an environment such as Arcadia. Multiple-languages sharing data is the norm, not the exception. As we have seen with Triton, converting a language specific system to a more general system has a number of painful consequences:

1. One is almost compelled to use the persistent programming language as the basis for the common data model for the system. As with E, the type system of this language may be much more complicated than is necessary. Additionally, some of the features of the language (such as generics) have complicated implementations that are difficult to model in the catalog. One is then forced to subset the language, which is often difficult. In E, for example, many of the features of the language are interwined, so one may be forced to remove features that might be better left in the subset.

2. Even with a reasonable language subset, the resulting type model may be inappropriate for use as a common type model. We have been fortunate that the E subset is not too bad in this aspect. But if we had been forced to use, for example, Lisp, we might have found that communication with Ada and C was difficult.

3. Overgeneralizing somewhat, it is typical for language specific systems to assume a relatively static type schema. It is often assumed that all the schema information is compiled into programs. Converting to a late binding system appears to require some form of dynamic loading of compiled code. This in turn requires significant support from the compilers, loaders, and even possibly the operating system. Porting Triton out of a narrow range of Berkeley Unix class of systems, for example, would be a daunting task.

9.3 Relation Support

Languages such as APPL/A rely heavily on relations as a major structuring element in their type system. The Triton type model in its pure form (i.e. E) does not directly support relations, and so the Triton interface was substantially augmented to provide better support for the definition and manipulation of relations.

Often, this concern for relations is dismissed by proponents of pure object-oriented (O-O) languages. The claim is usually made that the user can just define relations using the O-O type system. Technically this is correct, but practically it is irrelevant. Significant mechanism is needed to support relations in an O-O model. This is especially true if one wants the users to have concise and convenient access to the relational structures.

Extrapolating from the APPL/A support provided by Triton, there is a minimal set of components needed to support relations:

1. A standard interface to relations. This interface must allow for some form of insertion, deletion, and update of tuples. It must provide some form of navigational access (e.g., enumerating the tuples in a relation) and ideally it should allow some form of associative retrieval based on at least equality tests for tuple field values.

2. A standard interface for tuple types. Depending on the relation interface, this will include some form of equality test over selected fields of tuples and some form of selective update for tuple fields.

3. Simple creation of relation types in terms of user-defined tuple types. Typically this implies a *generic* definition and instantiation capability in the O-O type model.

4. Convenient support for sets or (preferably) bags of tuples. Set maintenance will typically be tied to the relation definition mechanism. The set feature should also support the necessary navigational and associative retrieval features of the relation subsystem.

9.4 Separate Client and Server Address Spaces

In our experience, it is a serious mistake to assume that one can load any component of the object management system in the same address space as client code. Especially if the client code is written in a language different from the language used for the object manager. Run-time systems often make assumptions about their control over such things as signals, memory allocation, and file descriptors. Our sad conclusion is that mixing run-time language support systems in the same address space will fail more often than not. Perhaps in some distant future, there will be standards for run-time systems, but until then, code mixing is fraught with peril.

This dictum also applies to loading behavioral methods into the client address space. It has been proposed[10] that the server should keep either interpretive versions of method bodies, or per-language versions of compiled method bodies that can be loaded into the client as needed. The use of interpretive models might work, although it might require the construction of an interpreter written in each supported language. We are sceptical about the use of any form of compiled code in a client. This requires the inclusion of dynamic loading mechanisms into each client, and we believe that run-time contention will cause this to fail.

The alternative used in Triton keeps the object manager code plus the method body code in a separate address space. We recognize some of the costs involved in this approach; there are significant performance hits in transporting method inputs to the server and retrieving the outputs. This may in some cases be offset by the more efficient execution of the methods since they are closer to the data.

As mentioned, we have chosen the Sun Remote Procedure Call mechanism as our means of client-server communication. In order to do this, we need to load the RPC libraries into the client, and this, of course, runs the risk of run-time interference. So far, this has not been a problem; the Sun code is relatively self-contained, but the potential for problems still exists. The alternative is to rewrite the RPC libraries into each supported language.

9.5 Common Data Model

Triton achieves multi-language interoperability by using a common data model. Client data is converted to the common model and stored in the server. On retrieval, the data in common format is converted back to the client model. In spite of our problems in using the rather ugly C++ type model, we were pleasantly surprised at how well this approach worked in practice. We hypothesize that the client-server architecture was the "cause." That is, inherently when a client sends data to the server, it must convert the data to a standard linear form the (eXternal Data Reference standard in our case) in order to ship it over a thin-wire connection to the server. Adding a little additional complexity to convert to and from the common data model is not a large burden. Given a shared memory model of client-server communication, the cost of conversion might seem more onerous.

9.6 Event Management

Event management has become an increasingly important feature of modern software environments. Control integration via events (as in Field[14] and HP-SoftBench[6]) is rapidly becoming the norm.

It is important to note that event systems are distinct from the database notions of triggers and rules. In an event system, there is a event server, also called a dispatcher, to which programs send, as messages, postings of events. Other programs may register with the dispatcher to receive events that match some specified pattern. As events arrive the dispatcher forwards them to registered programs as appropriate. The key feature here is that the dispatcher has very little knowledge about the sender and receivers of events and even about the semantics of the events themselves. This results in a very flexible system in which new kinds of events may be posted dynamically and senders and receivers of events may come and go quickly.

Databases, on the other hand, often assume that only a fixed set of actions (e.g., object insertions and modifications) can generate events. Further, it is assumed that the receiver of the event is known to the database system. With the possible exception of HIPAC[12], database systems appear ill-prepared to export their events to the outer environment and to register for externally generated events.

Arcadia already has several dispatcher systems: APPL/A triggers, the Amadeus measurement system[17] and the Chiron user interface system[7]. Many of these systems need to be aware of changes occurring within Triton. As a result, we needed to add event capabilities to Triton. As described in section 8, our initial attempt was based on triggers. This has proven insufficiently flexible and so we are in the process of moving to a more general event mechanism.

10 Status and Future of Triton

At the moment, a version of Triton without transaction features is running on Sun 3 and Sparc machines running Sun OS 4.1.1 or later. We are investigating the possibility of a Mach port.

Triton is used within Arcadia to support our current process programs, such as REBUS[19]. It has been exported to some external groups such as the STARS project.

Triton is undergoing A number of relatively short term enhancements:

- The next version of Exodus provides transaction management facilities and we are currently re-hosting Triton to use these features.
- We are working to integrate the Triton event dispatcher with other Arcadia dispatchers.
- We are exploring alternative, and simpler, common data models to replace E.
- We are exploring performance enhancements.

11 Related Work

At the time that Triton was first conceived, there were only a limited set of acceptable choices on which to base the effort. Licensing issues prevented consideration of commercial systems and so only research vehicles were considered. Of those systems, we chose the Exodus systems [2, 3] from Wisconsin as the basis for Triton. In retrospect, that decision turned out to be a good one because Exodus was quite robust and support was reasonable (given the inevitable limits associated with any research effort).

At the current time, there are a number of systems that, with more or less work, could serve as replacements for Triton. O2[8] and the Texas Instruments OODB[23] are similar to Exodus in that they support one or more persistent programming languages. Presumably, with some work, these systems could be used in Triton in place of Exodus by using one of their languages as a common model and adapting the dynamic loader to work with that language.

POSTGRES[16, 18] could also be used as a replacement for Triton. It obviously supports relations well, it has a catalog, and it has a form of trigger, but it is not clear how to convert its triggers to the more general event mechanism that now seem required.

Two other systems, GemStone[9] and PCTE+[1], seem much more promising as replacements for Triton. GemStone is derived from Smalltalk[5] and its interpretive nature would seem to make it possible to augment the system with the exact event mechanisms required. Interpretation would also simplify the dynamic loading mechanism. It is unclear how hard it would be to add heterogeneous access. Additionally, getting the effect of generic relations might be a little difficult since Smalltalk does not support that kind of polymorphism.

PCTE+ is very promising. Its data model (objects, links, attributes) is probably adequate for a process environment. The original PCTE had no notion of a trigger, but PCTE+ extends PCTE with a concept of notifiers, and it would appear that notifiers can be adapted to the needs of triggers and events. As with GemStone, heterogeneous access is still untested. Since PCTE+ does not support behavioral methods, dynamic loading becomes impossible. Unfortunately, it may not be irrelevant and some effort would be needed to overcome this deficiency.

12 Summary

Triton is one of the first attempts to provide comprehensive object management support for process-centered environments. It provides a behavioral object-oriented type model capable of supporting process programming languages. Triton also provides explicit support for heterogeneous interoperability in the form of multi-language access as well as shared data using a common type model. Implementing Triton has increased our understanding of the requirements for such object managers and we are now in a better position to determine which new object managers are appropriate candidates for inclusion in a process-centered environment.

Acknowledgments

We wish to thank the members of the Arcadia Consortium, and especially Lee Osterweil and Mark Maybee for their help. We also wish to acknowledge the help of the Exodus project in providing a very useful system. This paper may occasionally seem overly critical of Exodus, but this is misleading. Exodus has performed well for us. Remember that its goals were not our goals and so mismatches should not be surprising.

References

[1] G. Boudier, F. Gallo, R. Minot, and I. Thomas. An overview of pcte and pcte+. In *Proc. of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 248–257, 28-30 November 1989. Boston, Mass.

[2] Michael J. Carey, David J. DeWitt, Daniel Frank, Goetz Graefe, Joel E. Richardson, Eugene J. Shekita, and M Muralikrishna. The architecture of the EXODUS extensible DBMS: a preliminary report. Technical Report Computer Sciences Technical Report #644, University of Wisconsin, Madison, Computer Sciences Department, May 1986.

[3] Michael J. Carey, David J. DeWitt, Joel E. Richardson, and Eugene J. Shekita. Storage management for objects in EXODUS. In *Object-Oriented Concepts, Applications, and Databases*, chapter 14. Addison-Wesley, Reading, Massachusetts, 1988.

[4] Don Cohen. *AP5 Manual.* Univ. of Southern California, Information Sciences Institute, March 1988.

[5] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.

[6] Hewlett-Packard. *HP Encapsulator: Integrating Applications into the HP SoftBench Platform*, 1989. HP Part No. B1626-90000.

[7] R. K. Keller, M. Cameron, R. N. Taylor, and D. B. Troup. User interface development and software environments: The chiron-1 system. In *Proc. of the 13th International Conference on Software Engineering*, pages 208–218, 13-17 May 1991. Austin, Tx.

[8] C. LeCluse, P. Richard, and F. Velez. O2, an object-oriented data model. In *Proceedings of the SIGMOD International Conference on Management of Data*, pages 424–433, 1–3 June 1988.

[9] D. Maier, J. Stein, A. Otis, and A. Purdy. Development of an object oriented dbms. Technical Report TR CS/E-86-005, Oregon Graduate Center, April 1986.

[10] David Maier. Re: Looking for definition of oodb (an oodb manifesto). Message to comp.object news group, 28 August 1991.

[11] Mark Maybee and Stephen D. Sykes. Q: Towards a multi-lingual interprocess communications model. Arcadia Document UCI-89-06, Department of Information and Computer Science, University of California, Irvine, Irvine, February 1989.

[12] Dennis R. McCarthy and Umeshwar Dayal. The architecture of an active data base management system. In *Proc. of the ACM SIGMOD International Conf. on the Management of Data*, pages 215 – 224, 1989.

[13] Leon J. Osterweil. Software processes are software too. In *Proc. Ninth International Conference on Software Engineering*, 1987. Monterey, CA, March 30 – April 2, 1987.

[14] Steven P. Reiss. "Connecting Tools Using Message Passing in the Field Environment". *IEEE Software*, pages 57–67, July 1990.

[15] Joel E. Richardson and Michael J. Carey. Programming constructs for database system implementation in EXODUS. In *Proc. ACM SIGMOD Conf.*, pages 208–219, 1987.

[16] Lawrence A. Rowe and Michael R. Stonebraker. The POSTGRES data model. In *Proc. of the Thirteenth International Conf. on Very Large Data Bases*, pages 83 – 96, 1987.

[17] Richard W. Selby, Greg James, Kent Madsen, Joan Mahoney, Adam Porter, and Doug Schmidt. Classification tree analysis using the Amadeus measurement and empirical analysis system. In *Proc. Fourteenth Annual Software Engineering Workshop*, November 1989. NASA/Goddard Space Flight Center, Greenbelt, Maryland.

[18] Michael Stonebraker and Lawrence A. Rowe. The design of POSTGRES. In *Proc. of the ACM SIGMOD International Conf. on the Management of Data*, pages 340 – 355, 1986.

[19] S. M. Sutton Jr., H. Ziv, D. Heimbigner, M. Maybee, L. J. Osterweil, X. Song, and H. E. Yessayan. Programming a software requirements specification process. In *Proceedings of the First International Conference on the Software Process*, Redondo Beach, CA, October 1991.

[20] Stanley M. Sutton, Jr. *APPL/A: A Prototype Language for Software-Process Programming.* PhD thesis, University of Colorado, August 1990.

[21] Stanley M. Sutton, Jr., Dennis Heimbigner, and Leon J. Osterweil. Language constructs for managing change in process-centered environments. In *Proc. of the Fourth ACM SIGSOFT Symposium on Practical Software Development Environments*, pages 206–217, 1990. Irvine, California.

[22] Richard N. Taylor, Frank C. Belz, Lori A. Clarke, Leon J. Osterweil, Richard W. Selby, Jack C. Wileden, Alexander Wolf, and Michal Young. Foundations for the Arcadia environment architecture. In *Proc. ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 1 – 13. ACM, November 1988.

[23] Craig Thompson and David Wells. Report on DARPA open OODB workshop I: Preliminary architecture workshop. Technical report, Information Technologies Laboratory, Computer Science Center, Texas Instruments Incorporated, May 1991.

[24] United States Department of Defense. *Reference Manual for the Ada Programming Language*, 1983. ANSI/MIL-STD-1815A-1983.

[25] Jack C. Wileden, Alexander L. Wolf, Charles D. Fisher, and Peri L. Tarr. Pgraphite: An experiment in persistent typed object management. Arcadia Document UM-88-05, Software Development Laboratory, Computer and Information Science Department, University of Massachusetts, Amherst, Massachusetts, 1988.