Multilanguage Interoperability in Distributed Systems: EXPERIENCE REPORT

Mark J. Maybee*
Dennis H. Heimbigner*
Leon J. Osterweil†

August 17, 1995

Abstract

The Q system provides interoperability support for multilingual, heterogeneous component-based software systems. Initial development of Q began in 1988, and was driven by the very pragmatic need for a communication mechanism between a client program written in Ada and a server written in C. The initial design was driven by language features present in C, but not in Ada, or vice-versa. In time our needs and aspirations grew and Q evolved to support other languages, such as C++, Lisp, and Prolog. As a result of pervasive usage by the Arcadia SDE research project, usage levels and modes of the Q system grew and so more emphasis was placed upon portability, reliability, and performance. In that context we identified specific ways in which programming language support systems can directly impede effective interoperability. This necessitated extensive changes to both our conceptual model, and our implementation, of the Q system. We also discovered the need to support modes of interoperability far more complex than the usual client-server. The continued evolution of Q has allowed the architecture of Arcadia software to become highly distributed and component-based, exploiting components written in a variety of languages. In addition to becoming an Arcadia project mainstay, Q has also been made available to over 100 other sites. It is currently in use by SAIC, Loral, Stars project participants, and NASA Goddard. This paper summarizes key points that have been learned from this considerable base of experience.

Keywords: Interoperability, Heterogeneity, Distributed, Software Environments, Experience.

*Department of Computer Science, University of Colorado, Boulder, CO 80309-0430
†Department of Computer Science, University of Massachusetts, Amherst, MA 01003-4610

1 Introduction

A distributed system is one that is implemented as a collection of components that interoperate with each other, but that execute in separate address spaces, and may execute on separate hardware/software platforms. Distributed systems offer a number of important advantages over systems implemented as a single process running on a single platform. Distributed systems may be more robust, as it is possible to implement key services redundantly on different hardware and/or software platforms. Distributed systems may be faster as it may be possible to effectively parallelize bottleneck jobs. Distributed systems may be more flexible and extensible, as changes may be quarantined to smaller subsystems, and may be carried out without the need for change to the entire system. Distributed systems may be more effective in reusing sizable components, as these components are less likely to require recompilation and reloading. Distributed systems may be composed of components implemented in differing languages and dialects.

These advantages are particularly important to the implementors of software environments. Software environments are notoriously large, restricting their flexibility, extensibility, and ability to reuse existing componentry. On the other hand it is essential that software environments be highly flexible and extensible as most will need to undergo continuous change and enhancement. If an environment is implemented as a distributed system, consisting of separately compiled components, the required flexibility and extensibility can be achieved, often by reconfiguration using existing components, such as off-the-shelf database systems. As many environments are still experimental prototypes, it is particularly important that they be able to rely upon support from diverse, possibly competing, components, possibly written in different languages. Distribution also facilitates this.

1.1 Motivation for Q

Q has evolved over a period of years to provide the infrastructure for distributed objects within the Arcadia environment project [21, 10, 9]. That evolution was driven by a cycle involving experience with Q leading to a crisis in handling some important problem, followed by extending and modifying Q to address the problem successfully.

Generally speaking, the problems that Q has encountered and overcome have been related to issues of heterogeneity. Arcadia systems have been built intentionally to be heterogeneous with respect to computing platform hardware, operating systems, and especially programming languages. The latter point is worth expanding upon, since many other systems (e.g., CORBA [15], OLE2 [13, 14], DCE [3]) claim to support multi-language heterogeneity. In fact, they have generally restricted themselves to C and C++. From its inception, Arcadia has been using a variety of languages, including C, C++, Ada, Lisp, and even occasionally Prolog. As a result, we have developed an extensive, tested Q library for inserting distributed object capabilities into almost any programming language.

The need for these capabilities was clear to us in 1988, and persists to this day. Currently, however, there are indications that this need is understood more widely in the community, and there are some projects that start to address these needs. CORBA and DCE, for example, are such projects. From our current perspective, however, we now see that, even had CORBA been available to us in 1988 in its present form, it

would form only a part of a solution to our problems. For example, CORBA is avowedly an attempt to provide for multilingual interoperability, but its primary thrust is most clearly and sharply towards interoperability among clients and servers written in C and C++. Also, it is oriented is toward traditional client-server architectures while Q has moved on to support peer-oriented architectures. Further, both CORBA and DCE have made assumptions about concurrency and threading that Q rejects in order to expand its ability to support more platforms.

The details will be presented subsequently. For now, however, we believe it is worth noting that the CORBA standard is a useful start towards meeting our needs, but that the sorts of experiences that have shaped the development of Q need to be duplicated in CORBA to shape its future if it is to also be successful in meeting these same needs.

2 Q Version 1

For largely pragmatic reasons the *Open Network Computing* (ONC[1]) specifications for *Remote Procedure Call* (RPC) and *External Data Representation* (XDR) [18, 17] were chosen in 1988 as a basis for the construction of our language-heterogeneous interoperability mechanism. The, then newly available, version 4.0 release of RPC/XDR from Sun Microsystems was a public domain implementation that included the source code. This made modifications easy. Also the ONC standard is the RPC which underlies Sun's Network File System (NFS). As a result, it was, and still is, the most widely available RPC system. There were also good scientific reasons for this choice of RPC system. The ONC implementation already solved some key requirements of the interoperability system: it supported autonomous components communicating across process and platform boundaries. That seemed to leave only the task of adapting the model to provide multi-language support.

ONC RPC/XDR provides the ability to exchange meaningfully typed data values between two processes. This model supports a procedure call abstraction of inter-process communication, allowing one process to make a procedure call to another process — *even across machine boundaries and independent of machine architectures* — and have ONC/RPC handle the details of data marshaling and inter-process communication.

Data marshaling is the process of arranging data in a language and architecture independent format prior to dispatching it in a message. This is to insure that the semantics of the data values are preserved[2].

ONC/RPC supports communication between two processes written in the C language. Its interfaces are written in C and make use of semantic constructs which are not supported in Ada (such as procedure variables). Additionally, its data representation does not define any mapping to assure the consistency of types when passing data between different type models.

Operating under the assumption that it would be possible to layer heterogeneous language support atop a standard communications interface, a new and improved interoperability mechanism was conceived. Figure 1 depicts the virtual machine layers of the resulting interoperability mechanism. A variety of language interfaces rest upon a

[1] ONC is a trademark of Sun Microsystems, Incorporated.

[2] This is particularly relevant when moving data between machines with differing byte-order architectures [2].

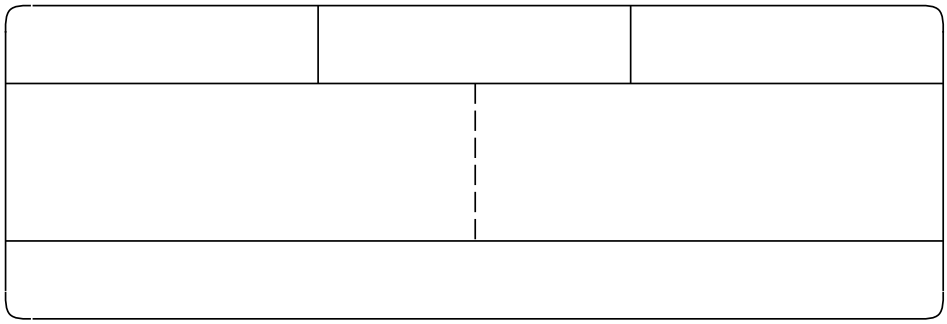

Figure 1: Virtual machine layers

standard remote procedure invocation interface (ONC/RPC) with a separate argument marshaling interface (ONC/XDR). Underlying this is a basic data transport mechanism supporting the physical message transport needs of the system.

A variety of language interfaces were constructed to explore the flexibility of the underlying support mechanism and the model in general. These languages included Ada, C, C++, Lisp, and Prolog. Most of this papers discussion of language support will center on the Ada language interface, however, because problems encountered while developing this interface prompted much of the research. Also, for entirely pragmatic reasons, it was necessary to focus attention on the Ada and C languages, as these were the implementation languages used by the bulk of the software tools being supported.

Problems were encountered with both the ONC/RPC interface and the ONC/XDR interface while trying to adapt them for use from multiple languages. Problems with the ONC/RPC interface centered around its dependence on features found in C (its implementation language) which are not present in many other languages (e.g., Ada). Problems with the ONC/XDR interface revolved around its implicit assumption that all data to be marshaled would be instances of C types. These problems, and their solutions, are discussed in detail in the following sections.

2.1 Argument Marshaling Support

In general, an application should not have to worry about the issue of data marshaling. An interoperability system's infrastructure should be able to hide this necessity from the application by automatically marshaling and unmarshaling data when necessary. However, this becomes a problem when the support infrastructure does not, and can not, know the type of data being shipped.

The ONC implementation of RPC handles this issue by requiring that the application provide the *generic* remote call procedure with explicit marshaling procedures. At each remote call, the client provides the procedures necessary to marshal the argument list and unmarshal the return value. Similarly, on the server side, the server application must *register* each service routine with a set of procedures to perform the argument unmarshaling and the return value marshaling.

While this model provides a clean procedure call abstraction for interprocess communication when all procedures are written in a language such as C, it breaks down for

procedures written in many other languages, most notably Ada. Ada does not permit procedure parameters. It is therefore necessary to incorporate a modification to the ONC/RPC model to resolve this problem.

The solution is to completely decouple the argument marshaling process from the remote procedure invocation mechanism. Doing so alleviates the need for procedure parameters in the invocation interface. Rather than having the application provide a set of arguments and a marshaling procedure to the invocation mechanism, the application instead provides a pre-marshaled set of arguments. Upon return from the remote procedure the application is handed the still-marshaled result and must explicitly un-marshal. Similarly, on the server side, the application service routine will be handed a marshaled set of arguments for explicit un-marshaling, and it must pre-marshal the return value prior to completion.

This decoupling of the argument marshaling from the remote procedure invocation was an important step to opening up the RPC model to multi-language implementations. It allowed the construction of a matched set of C and Ada interfaces. There is a function to establish a connection to a remote server, and a procedure to perform individual remote procedure calls on a server as discussed earlier. This interface is complemented with a set of matched interfaces for supporting data marshaling.

2.2 Type System Support

The above discussion of data marshaling assumes the existence of a language and architecture independent representation for data. The ONC/XDR standard was designed to address this problem. Unfortunately, this standard was principally concerned with heterogeneity with respect to architecture rather than language. Its interfaces are written in C and so provide support only for C data types. Immediate problems occur when attempting to adapt this standard for use between multiple languages. When passing typed data between C programs there is always a type correspondence. However, when passing data between two differing languages there may *not* be a type correspondence. Consider passing a fixed point data value from an Ada program to a C program. There is no fixed point type defined in the C language, and so achieving type correspondence is impossible (at the representation level).

This problem is solved by eliminating the requirement that all data types be representable in all languages supported. Instead an attempt was made at type model *concordance*. This means that whenever there is a data instance with an equivalent representation in two, or more, languages, it may be passed between components constructed in those two languages. For example, most languages define an integer type, so each interface supports a marshaling routine for that type. Integer data instances may be passed between all languages which supply this interface routine.

Type concordance, for a type X, between any two languages, A and B, is based on three factors:

1. the marshaled representation of the type X,
2. the representation of the type X in language A, and
3. the representation of the type X in language B.

Figure 2: Array Representation

For example, if the type is `integer` and the marshaled representation of an integer is a signed 32 bit value, and the representation of an integer in language A is as a 32 bit value, and the representation of an integer in language B is as a 16 bit value, then, only integers in the range $-2^{15} \ldots 2^{15} - 1$ may be passed from language A to B. The responses of marshaling routines to invalid data values are language interface specific. The Ada interface for example, raises an exception. The C interface returns an error result.

The following base types are currently supported and have the indicated marshaled representations:

Integer is a signed 32 bit value with range $-2^{31} \ldots 2^{31} - 1$.

Floating point is signed 64 bit value whose precise representation is defined in the IEEE standard on floating point numbers [7].

Fixed point is a 32 bit value whose precise representation is defined in the Ada language reference manual [23].

Enumeration is represented as an integer denoting the position of the data value in the enumeration set[3].

Boolean is an enumeration type. `False` is represented as the integer 0 and `True` is represented as the integer 1.

Character is an 8 bit value with range $0 \ldots 255$ of type ASCII[4].

String is an instance of the array type that will be discussed later.

Pointer is an abstraction represented with a boolean flag. This is actually a type constructor to be discussed later.

The base types supported by the Ada and C interfaces appear in table 1. Note that two components may not exchange a data instance of a type for which there is not a type concordance between the two languages. This approach is predicated on the assumption that there is a set of base types common to most languages. For Q, we have suggested that this is essentially the C language type system, including the integer, floating point, character, pointer, and enumeration data types.

Each language interface must provide mappings to as many of the base types as possible. Language interfaces are also free to expand their type support beyond the base set.

[3]Enumeration is actually a type constructor.

[4]To be precise, ASCII is an instance of an enumeration type.

Table 1: Types supported by C and Ada marshaling interface

C type	Ada type	Marshaled as
`int`	`INTEGER`	Integer
`double`	`FLOAT`	Floating point
	`FIXED`	Fixed point
`bool_t`	`BOOLEAN`	Boolean
`char*`	`STRING`	String
`caddr_t`	`ACCESS`	Pointer

```
GENERIC TYPE    Struct_type     IS PRIVATE;
        TYPE    Index           IS (<>);
        TYPE    Struct_array IS ARRAY (Index) OF Struct_type;
        WITH PROCEDURE struct ( qdrs    : IN Handle;
                                sp      : IN OUT Struct_type );
PROCEDURE generic_array (
        qdrs            : IN Handle;
        a               : IN OUT Struct_array);
```

Figure 3: Array constructor interface for Ada

The data model is enriched by supporting the ability to compose the base types into more complex abstract data types. Vector types are represented as an index range followed by the vector elements (see figure 2). Arbitrarily dimensioned arrays may be represented by a recursive application of this technique. Figure 3 contains the support procedure (`generic_array`) for vector marshaling provided in the Ada interface. Note that it is a generic procedure that can be instantiated for any constrained vector type.

Besides automatic array marshaling, support for pointer types is also supplied. Since passing a pointer across address space boundaries is meaningless, marshaling a pointer actually passes the data value referenced by the pointer. When unmarshaled, memory is automatically allocated, the data value is stored in it, and a pointer to it is returned.

More complex types (such as records, linked lists, etc.) must be built up out of these primitive constructors. For example, aggregates such as C `structs` or Ada `RECORDS`, can be represented as a sequence of data values each corresponding to a field of the aggregate, and represented according to its type. The aggregate is then reconstructed in the same sequence as the one from which it was emitted. In this fashion a C structure may be passed to an Ada component and reconstructed as an Ada record.

Applications must include marshaling functions for all the types that are to be used in interprocess communication. Note that while this approach provides the ability to marshal arbitrarily complex type structures, it does not provide any support for conveying the semantics associated with them.

2.3 Experience with Version 1

APPL/A [19] is a software-process programming language designed as an extension of the Ada programming language. It adds constructs to the Ada language designed to

support change management in process-centered environments. APPL/A's extensions to Ada include persistent relations, triggers, enforcible predicates, and transactions. A number of Arcadia components are written in APPL/A (Rebus, Debus, Process_Viewer, Project_Panel, etc.). In the implementation of APPL/A used by Arcadia, the scope of persistent relations is global to a cluster of different components, presumably hosted on different platforms. Multiple components that access a relation with the same name will be accessing a common instance. This requires coordinated access to shared nonlocal data. To achieve this, Arcadia provides a sharable data repository, to store the relation contents, and a global event manager, to coordinate the relation manipulation activities, to support the APPL/A implementation. Interoperability with these environment infrastructure components is provided by Q.

The *Global Event Monitor* (GEM) is designed specifically to support APPL/A in a distributed component environment. For example, the APPL/A language allows trigger procedures to be associated with any relation operation. A trigger may be invoked either before or after the relation operation, and may be specified as synchronous or asynchronous. When a relation operation is invoked the APPL/A run-time system first checks for any triggers that should be invoked before the operation is executed, all such triggers are executed. When the synchronous triggers have completed, the operation is executed. Following the completion of the operation the run-time system checks for any post-operation triggers and executes them. Only after all synchronous triggers have completed is the operation invocation completed. In a distributed environment, multiple components may be simultaneously accessing a relation and any number of them may define triggers on it. A great deal of event management is required to achieve the coordination necessary to implement the APPL/A trigger model.

The trigger model is supported in Arcadia by both GEM and a *Local Event Monitor* (LEM) embedded into each APPL/A-based application. Q is used to support the interoperation of these monitors. The triggering model is implemented by these monitors, in the form of servers, at each relation operation in the following manner:

Before each operation is performed: the application informs the GEM, which in turn informs all LEM's, that an operation is about to be performed. Each LEM executes any pre-operation triggers. The GEM awaits the completion of all *synchronous* triggers before returning control to the local application.

After each operation is performed: the application once again informs the GEM, which again informs all LEM's. This allows the LEM's to execute post-operation triggers. As with pre-operation triggers, all synchronous triggers must complete before the GEM returns control to the local application.

The execution model of components implicitly assumed by Q up to now was a single threaded one. While Q supported the notion of application components acting as both client and server, it was assumed that they would not do so simultaneously. That is, a component could act as a client or a server and alternate arbitrarily between the two roles, but would not act as both at once in a multi-threaded application. The GEM, however, was designed and constructed as an Ada application. It made full use of the multi-threading capabilities inherent in the Ada language. It embedded a server (the LEM) into each APPL/A application and instantiated it as an asynchronous task.

The GEM/LEM implementation allowed for the possibility of an application acting *simultaneously* as both a client and server. Consider the following scenario:

1. Application A initiates a relation operation R_x.
2. Application B defines a trigger on that relation operation.
3. Before B is informed of R_x, it initiates some relation operation of its own, R_y.
4. The possibility now exists that B may be engaged in some remote procedure call with the global event manager *at the same time* that its local event manager is responding to the operation R_x.

Complicating matters further, we observed that, in addition to the interactions between an APPL/A component and the GEM, APPL/A components typically also simultaneously maintained a client relationship with a data server.

2.4 Crisis

Simultaneous remote procedure call activity (due to any combination of simultaneous client and/or server activity) was not properly handled by the original Q design. Recall that a remote procedure call is built from an exchange of two messages between a client and a server. Clients await response messages from servers and servers await request messages from clients. The problem with simultaneous RPC activity becomes apparent when two or more threads of control (e.g., Ada tasks) are awaiting messages at the same time. The ONC/RPC implementation underlying Q uses the `select` system service call to await messages. The `select` system service listens for IO activity, incoming messages in this case, on a set of IO channels and returns to its caller when there is some IO pending on one of those channels. When multiple tasks are awaiting messages, multiple calls to the `select` service will be outstanding – all waiting on the same IO channels. The `select` system service is not designed to be used in this manner, its semantics under these conditions are undefined. The observed behavior of Q under these conditions was unpredictable. Sometimes remote calls would succeed, sometimes they would not, sometimes the system would hang. A re-design of the Q architecture was called for; this led to the development and implementation of Q version 2.

3 Q Version 2

Emerging environment architectures, using multi-threaded components each maintaining multiple simultaneous client and server interfaces, led to a realization that Q's language support must encompass thread support in multi-threaded languages. Significant restructuring of the Q system was necessary to support the ability to embed multiple clients and/or servers in a single process (see figure 5). The result was a new design that supported the separation of the application architecture from its process binding.

3.1 IO Multiplexing Support

In the single threaded Q model of version 1, each language interface was only a relatively thin veneer over the remote procedure call interface substrate. This interface, version

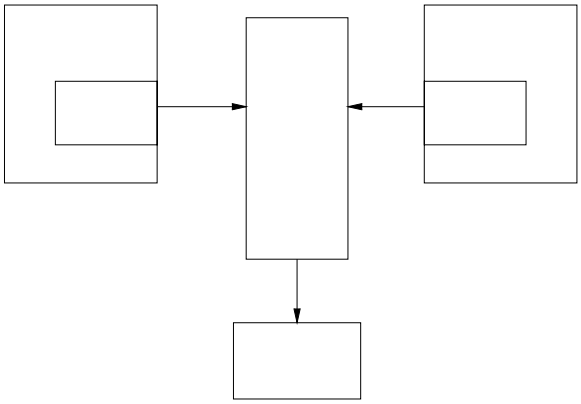

Figure 4: IO multiplexing architecture

4.0 ONC/RPC, was also based upon a single threaded execution model. As discussed in section 2.3, multiple execution threads, initiated from multiple simultaneous tasks in Ada applications, were trying to block on `select` calls simultaneously. The resulting behavior was unpredictable, and usually erroneous. What was required was an IO multiplexing capability to resolve multiple requests for IO availability into a single select call.

To facilitate this the ONC/RPC infrastructure was reengineered and extended to produce the *Augmented Remote Procedure Call* (Arpc) interface [4]. Among other things, the new infrastructure exposed a message passing interface for client/server interactions. Where previously a client made a single call to `clnt_call`, now the client called `clnt_sendmsg` followed by `clnt_recvmsg`[5].

The Ada-language interfaces were re-written atop this new interface and an Ada-level select mechanism was introduced to deal with the IO multiplexing problem. The IO multiplexing interface provided a single generic procedure: `await_io`. Each individual client and server in a component would then instantiate this procedure to interface with a central IO multiplexor (see figure 4). The central multiplexor ran as a separate Ada task. When a call to an instance of `await_io` was made the procedure would register with the multiplexor to be informed when the next message arrived for it. Utilizing the `select` call, the multiplexor would monitor the arrival of messages and inform the appropriate `await_io` procedures when their messages arrived.

A slightly simplified algorithm for client and server RPC proceeds as follows: The client initiates an RPC by sending a message to a server[6]. It then waits for a response message from the server. The `channel` variable indicates the unique communication link being used between this client and server and is defined when the client first establishes communication with the server. When the message arrives the transaction is competed with `qpc_complete` and the RPC is complete[7]. The server spends the bulk of its time in `await_io`. It waits for service requests on a set of `channels`, which are the communication links that have been established with its clients. The `svc_receive` function

[5] A message passing interface has always been exposed on the server side.

[6] `qpc_initiate` is the Ada interface name of the Arpc interface routine `clnt_sendmsg`.

[7] `qpc_complete` is the Ada interface name of the Arpc interface routine `clnt_recvmsg`.

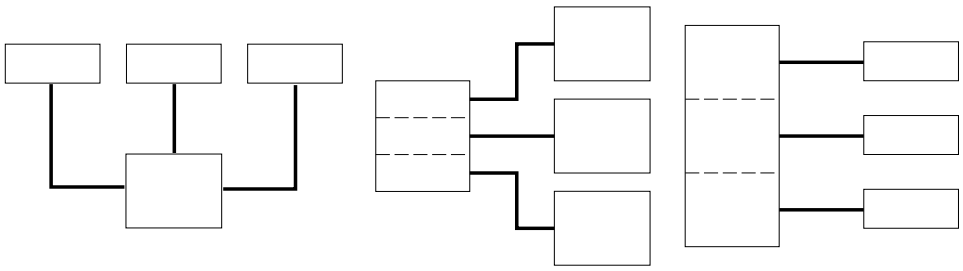

Figure 5: Logical client/server architectures

returns the service request message. The `server` routine is then invoked to execute the requested service. Upon completion, the service results are returned in a message to the client with the `svc_sendreply` routine.

3.2 General Architecture Support

The purpose of introducing the IO multiplexing facility to the Ada interface was to be able to support more general component architectures. Q was developed to support the sort of architecture depicted in figure 5a. However, experience with the APPL/A implementation demonstrated that Ada's inherent multi-tasking abilities could and would be leveraged upon in order to construct more complex application architectures than originally imagined. Already applications were combining multiple clients into single components (see figure 5b), as seen in the APPL/A-based applications discussed above. The addition of a Chiron interface, as discussed in the next section, would lead to multiple servers embedded into a single application as well (see figure 5c).

The logical progression depicted by the figures in 5 is towards increasingly arbitrary combinations of communicating clients and servers. These figures represent combinations of "pure" clients and "pure" servers communicating. This might be thought of as a sequence of *logical* architectures for collections of clients and servers. The new Q design supports this concept by allowing arbitrary mappings of these logical architectures onto component processes. The original expectation was that this binding would usually be one-to-one: that is, each client and server would occupy a single application process. But experience has demonstrated that other bindings are clearly desirable. Q has been designed to allow essentially arbitrary binding of clients and servers to processes.

Of particular interest amongst the possible mappings is the peer architecture. This is a mapping of both a client and a server into each process such that each process may either initiate, or respond to, remote procedure requests. This is required when callbacks from a "server" back to its "client" are needed. Examples of this behavior occur when a service procedure may run for an unbounded amount of time and the client does not wish to await the outcome, or when a server wishes to inform clients of events of interest to them. The first behavior was demonstrated in the GEM implementation discussed above. The second type of behavior is frequent in user interface applications, where it is desirable that the interface remain responsive even while engaged in lengthy service

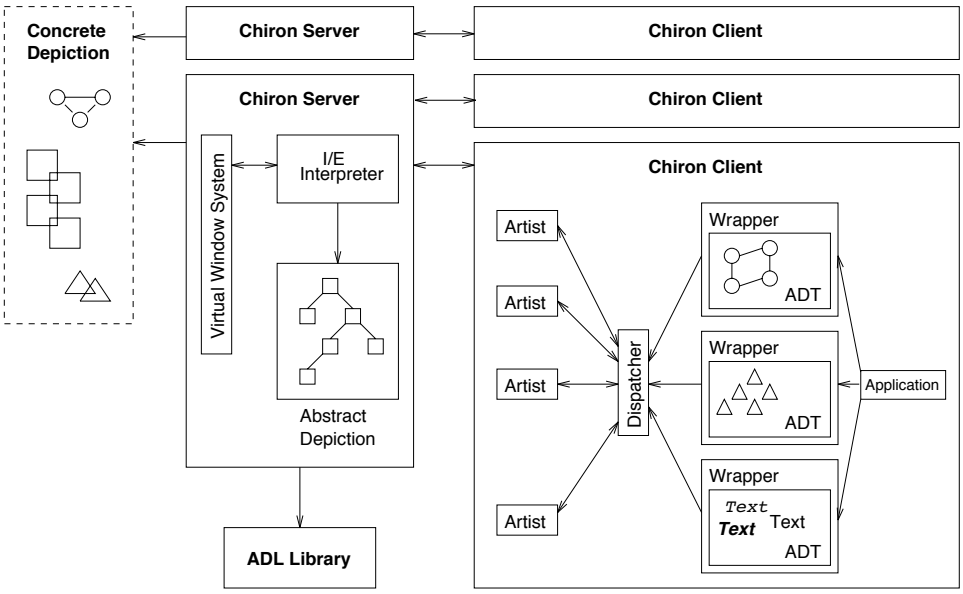

Figure 6: Chiron conceptual architecture.

operations.

3.3 Experience with Version 2

Chiron [26, 11] is a user interface development system (UIDS) supporting the user interface needs of the Arcadia environment. It emphasizes the value of separating the application from the graphical user interface (GUI) by means of a client/server split. Figure 6 depicts the Chiron architecture for achieving this separation. Applications are embedded in Chiron clients, while the bulk of the code for supporting the application's user interface resides in a Chiron server. The Chiron architecture is an attempt to separate the concerns of the application domain (model) from the presentation domain (view) by means of a sharp split between a client (application) and a server (presentation). Chiron clients and servers are implemented as separate components and rely upon Q for interprocess communication support.

The Chiron model assumes that an application's user interface can be constructed from a set of visible abstract data types (ADT) in the application. Each of the ADT interfaces is wrapped so that Chiron is involved when any interface operation is invoked. User interface designers are then expected to develop *artists* using the ADT interfaces. An artist is a code module for constructing a graphical depiction of an ADT; it can be thought of as defining a *view* of the ADT. It is possible to construct multiple artists, or views, for a single ADT. Chiron maintains consistency between artist depictions through its wrapper technology. Whenever the ADT is manipulated, via an artist interface or from some internal application operation, the wrapper intercepts the operation and *dispatches* it to any other artists for that ADT.

The artists, dispatcher, wrapped ADTs, and the remaining application code make

up a Chiron client. Chiron servers maintain the abstract depictions defined by client artists. Artists interact with the Chiron server to construct and maintain a user interface as an abstract depiction. The Chiron server, in turn, realizes this abstract depiction via some graphical depiction engine such as an X11 server. Separating the Chiron server code from the client code reduces the size and complexity of client applications. It allows the dynamic creation of additional artists within the client to support new or changing user views and interactions. In addition, it allows modifications to be made to the client or any server without having to re-compile unchanged application or server modules. Because Q allows the Chiron processes to reside on separate hardware platforms, flexibility, portability, and efficiency are all enhanced.

Chiron is based on a concurrent control model with the application and user interface simultaneously active. The Chiron server remains responsive to user interface manipulation events, passing those events back to the artist in the clients as necessary. The application embedded in the client also remains active and can continue to manipulate its ADT interfaces while the user interface is active. To support this control model Chiron uses a peer-peer architecture: clients and servers contain both a Q server and client in order to provide two-way interprocess service.

The Chiron experience with Q has lead to key changes in Q. The first involved alleviating sluggish run-time performance. (Chiron applications are real-time systems: if the result of a computation is not obtained within a certain period of time, it is considered to be incorrect regardless of value.) The other was necessary to eliminate strange inconsistent behavior in the message passing substrate.

The symptoms of the Chiron performance problem included two measures of interest: a single run of a particular application took about 11 minutes of wallclock time while consuming less than 4 seconds of CPU time. Traces of system calls using UNIX `trace` revealed that the processes typically were waiting on synchronous I/O (UNIX `select` system call).

Investigation of the Chiron design revealed a decision that clearly affected performance: time slicing was enabled. Time slicing is a commonly used technique for executing a large number of tasks on a smaller number of processors. Ada runtime executives may optionally permit time slicing of tasks on a processor by repeatedly suspending the execution of one task and resuming another. Another approach, required by the Ada standard as an alternative to time slicing, is to allow scheduling of tasks based on their predefined priorities. On each processor, a task is run until a synchronization point is reached and then the scheduling decision is reevaluated. This enables a task to run until preempted by one of higher priority in order to avoid unnecessary context switching. However, program design must ensure that a single task does not monopolize the processor. Assuring that can be quite difficult. Therefore most compilers (e.g., Verdix and Sun Ada compilers) strongly encourage the use of time slicing by making it the default.

Early versions of Q polled for input, trading off rapid response for design simplicity. As discussed above, in section 3, the multiplexing facility utilized the `select` service to await messages. This request blocked until an input message appeared. Ada tasks in a time sliced program that were waiting for messages could not progress, yet they continued to consume full time slices because the Ada run-time system did not recognize their blocked state. This was an unforeseen complication arising from Q's attempt to provide a single substrate to support multiple languages. Chiron applications were in effect spending nearly all of their execution time waiting for time slices to expire. While

this is a problem of the Ada run-time system, it exemplifies the class of problems that can occur when using synchronous IO under differing run-time systems.

The solution to this problem was to move from a synchronous I/O model to a signal-based asynchronous model. Instead of having the multiplexor poll for interprocess messages, the data channels are configured for asynchronous I/O. When a message arrives a signal is sent to the process. Therefore, processor time is only used for interprocess communication when it is known that data is pending. Most significant, however, is that time slicing can now be disabled so that blocked tasks are not scheduled for execution.

As an example of the effect of this change, the application referred to above, which consumed about 11 minutes of wallclock time to run with time slicing enabled and blocked tasks scheduled, required only 27 seconds of wallclock time with asynchronous I/O operation. This order of magnitude improvement in performance has typically been realized in representative Chiron applications.

The second Q problem uncovered by Chiron was far more subtle and insidious. The symptoms of this problem were occasional irreproducible errors in the message substrate: messages being lost, messages delivered twice, and messages apparently being delivered to the wrong recipient.

3.4 Crisis

The problem was that, while the Ada interfaces had been re-engineered to support general multi-client/multi-server (i.e., a multi-threaded) architectures, the Arpc substrate was not. The Arpc substrate is written in the C language and relies heavily on the standard C libraries supplied with all C compilers. Arpc is *non-reentrant* and, in general, so are standard C libraries. A typical characteristic of non-reentrant code is the use of unprotected global data structures. In a multi-threaded application, two threads of execution attempting to manipulate such global data are likely to produce errors. Consider the following scenario:

1. Routine X adds links to a global linked list `l`.
2. Thread A calls routine X with new link `x`.
3. X assigns `x->next` to be `l`.
4. The time-slice for thread A ends.
5. Thread B is started and it calls routine X with link `y`.
6. X assigns `y->next` to be `l` and `l` to be `y`.
7. Sometime later, the time-slice for thread B ends.
8. Thread A is restarted.
9. X completes its operation by assigning `l` to be `x`.
10. The link element `y` is now lost.

A solution to this problem was incorporated as a key new feature of version 3 of Q.

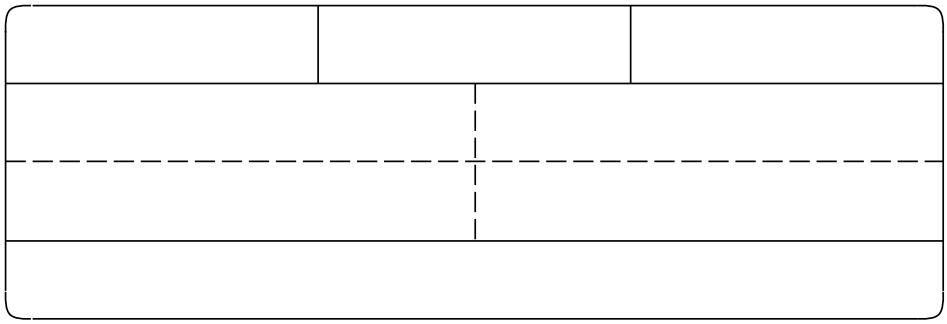

Figure 7: Q virtual machine layers

4 Q Version 3

The solution to the non-reentrant interface problem was two-fold. First, a non-blocking message passing interface was constructed between the Arpc interface and the language dependent interfaces. Second, calls into the non-blocking interface were protected against reentrant access with semaphores. The resulting Q architecture is presented in figure 7. This re-design coincided with the realization that the multi-threaded architectures that supported peer-style inter-component communication were becoming the norm rather than the exception within the Arcadia project.

Careful readers may realize that, based on the earlier discussion of the evolution of Q, the current Q substrate interface should already be non-blocking. This was largely true. The blocking interface had been isolated to the IO multiplexing interface and that had been converted from synchronous to asynchronous. However, it was at this point in the Q development that the true value of a non-blocking interface was realized and formalized.

The key to solving the reentrant problem from Ada was the introduction of semaphores to protect the non-reentrant code levels of the Q mechanism. A single semaphore, `Nonreentrant_Access`, was introduced to provide a mutual exclusion zone around all non-reentrant interface procedures. When an execution thread wished to use one of these interface procedures it first had to obtain the semaphore. Any other thread attempting to use one of the interface procedures would now be blocked until the semaphore became available again. Upon completion of the interface procedure, the thread would give up the semaphore. In this way, only a single execution thread may be active in a non-reentrant procedure at any given time.

It is important to note that the mechanism used to protect against reentrant access must be implemented as part of the language interfaces which form the top layer of the Q model. The alternative is to place the protection (e.g., semaphores) into the non-reentrant interface itself. The problem with this is very similar to the problems encountered with the `select` service discussed in section 3.3. If a caller blocks inside the lower layers, outside the language-specific layer, there exists the possibility that the language-specific run-time system will not recognize the blocking event. The result is the continued scheduling of threads that can not execute and a degradation in performance. The implication is that all multi-threaded language interfaces are required to ensure

non-reentrant usage of the underlying support layers in Q. This is not unreasonable however, as any multi-threaded language is likely to provide some form of support for exclusive access.

A second requirement is also imposed upon languages using a non-blocking RPC interface. The language layer must provide a mechanism for awaiting the arrival of messages. This issue was discussed in section 3.3. The solution was to provide asynchronous IO support with signals. Whenever a message arrives the Q substrate generates an IO signal. It is the responsibility of the language-specific interface layers to provide signal handlers.

4.1 Experience with Version 3

Arcadia's use of Q is now ubiquitous, as Q has become the foundation for interoperability in that environment. Q version 3 has also been distributed to over 100 other sites. It is in use by SAIC, Loral, Stars project participants, and NASA Goddard. The majority of sites are using Q because of its support for multi-language interoperability, and specifically its support for Ada. Q is also being used successfully in software evolution projects, where it supports the ability to interoperate with old components as large systems transition from one implementation language to another. Feedback continues to be quite positive, however there is ever increasing demand for more supported platforms and languages.

5 Summary of Experience

Much of Q's development has been driven by experience with the various application and infrastructure components in the Arcadia project. Q has become the major mechanism used to support the interoperability needs of Arcadia. Almost every component therein (Rebus [22, 20], Debus [16], PIC [24, 25], BMS, Process_Viewer, Project_Panel, ACV, Agendas, etc.) utilizes Q. Arcadia demonstrations are typically run on a network of Sun and DEC workstations, and considerably greater heterogeneity and distribution are possible. The use of Q in this distributed environment has brought to light some facilities lacking in the Q implementation. For example Q does not address the need for a meta-level component for interoperability services. As Arcadia grows so to will Q: modern software architectures appear to be more and more like the Arcadia environment. This trend implies uses for Q beyond the Software Development Environment arena which spawned it.

6 Related Work

6.1 DCE

The Distributed Computing Environment [3] is an integrated set of services designed to support distributed applications. These services include:

- Remote Procedure Call
- Directory Service

- Time Service
- Security Service
- Threads Service

The remote procedure call services were developed specifically to provide simplicity, performance, portability, and platform independence. However, they were not designed specifically to support multi-language interoperability. As a result DCE only attempts to provide direct support for the C and C++ languages. Additionally, DCE made the mistake of deeply embedding threading support into the model. As a result, it is difficult to rehost DCE onto platforms with different thread models, and even more difficult to embed DCE into languages like Ada that provide a significantly different model of concurrency.

6.2 CORBA

CORBA[8] [15] is an evolving application interoperability standard. The current version, 2.0, was released in late 1994. As an evolving standard it is a moving target. There are a number of available systems which claim to comply with this standard (e.g., DSOM, Orbix[9], ORBeline[10], etc.), however these are almost all based upon the version 1.1 definition of the standard. As new versions emerge, these systems must adapt to remain consistent with the standard.

Like DCE, CORBA is based on an RPC model of interoperability. Like other RPC mechanisms, this provides support for platform-independent interoperability. CORBA also provides what is claimed to be a target language neutral interface definition language (IDL). However, IDL appears remarkably similar to C++ and it is not clear how well it will map to other languages. Almost all known implementations of CORBA support only C and/or C++.

As the CORBA definition did not exist at the time this research was conducted, it did not have significant impact on this work. However it is now clear that there is a great deal of correspondence between the issues that CORBA is attempting to address and the issues raised in this work. Although the CORBA specification does not address concurrency, available commercial implementations have chosen to mimic DCE and embed threading into their implementation, which leads to the multi-language and multi-platform problems already discussed.

The CORBA marshaling system also introduces a number of problems. Most important is the problem of "external" or "pre-defined" types. Experience with Q indicates that it is common to take pre-existing packages and wrap them to provide distributed interfaces for the package. As a rule, these packages define a number of input and output data types. CORBA (and technically, DCE) does not support direct marshaling of such pre-defined types. Instead, one is required to define a duplicate type system in IDL and translate at run-time between the pre-defined types and the IDL types. The CORBA type system suffers from other deficiencies (See [5] for more detailed discussions).

[8] CORBA is a trademark of the Object Management Group, Incorporated.
[9] Orbix is a trademark of Iona Technologies.
[10] ORBeline is a trademark of PostModern Computing Technologies Incorporated.

6.3 OLE2

Microsoft's COM/OLE2 [14] has grown over time to provide support for distributed objects. Strictly speaking, COM (component object model) is the lowest layer in OLE2, but is the part that is most analogous to Q, CORBA, and DCE. Until COM provides complete support for distributed objects, it is premature to make direct comparisons. Recently, Microsoft has begun to address these issues by using DCE to provide the underlying naming and marshaling support. It is fair to say that this means that COM will have the same merits and demerits as DCE.

6.4 Matchmaker

When supported by the capability-based interprocess communications found in the Mach kernel [1] Matchmaker [8] provides a heterogeneous, distributed, object-oriented programming facility. Currently the Mach/Matchmaker system supports the generation of interfaces between C, Common Lisp, Ada, and Pascal. The Matchmaker language defines the type model within which the supported languages may exchange data objects. The built-in types provided by the type model are boolean, character, integer, string, communication port, and real. The type model also provides record, array, union, and enumeration type constructors. Matchmaker restricts the use of pointers, variable arrays, and unions to top-level declarations and so they may not be used when constructing other types. While this type model is restrictive, it still quite powerful and has been effectively used throughout the Mach kernel.

Most RPC mechanisms choose to use a standard encoding scheme for data values so that these values are not affected by differences in data representation across platforms. Instead, Matchmaker embeds typing information with the data values and makes it the responsibility of the receiving machine in a value transmission to be aware of any data representation differences between platforms, and perform any necessary transformations. Essentially, Matchmaker is relying on the capabilities defined in the low-level IPC support of the Mach operating system kernel to achieve multi-platform interoperability. The dependence on Mach kernel features limits the usability of Matchmaker to hardware platforms running the Mach operating system. This lack of flexibility/portability and the sparse usage of Mach make Matchmaker unsuitable as a solution to the interoperability needs of most contemporary software development environments.

6.5 Mercury

The work done at MIT on a value transmission method for abstract data types [12, 6] is designed to support communicating abstract data types that are interoperable between regions of a system using different data value representations. This method defines call-by-value semantics for communicating values over a network of different computers. A canonical representation for each type used in communications is defined. Each region of the system that uses a type that is to be communicated must define a translation between its internal representation of the type and the canonical representation. Much of the Mercury system concerns itself with supporting an extension to the remote procedure call paradigm called streams by addressing value representation issues.

7 Future Directions

While the Arcadia environment is composed of autonomous components, there are a number of interdependencies implicit in the interoperation of those components. For example, an APPL/A-based application with a Chiron interface relies on the ability to interoperate with a Chiron server component to supply its user interface needs, with a Triton server for its data persistence needs, and with a GEM server for its event management needs. In the Q system, these dependencies are expressed as explicit requests for servers from the application. These requests are explicit in the sense that they specify the explicit *name* and *location* of the desired server. The problems with this are twofold. One, this locks the server into a specific location. But, in a distributed environment it is often desirable to maintain the flexibility to move services in response to load and need. In addition, the mechanism requires that the desired server be active and available at the time of the request; this requires that the environment "predict" the needs of its clients, rather than letting it "react" to needs.

8 Conclusions

Modern software system requirements for flexibility and reuse are creating increasingly stringent demands for powerful interprocess communication mechanisms. The original design and implementation of Q were aimed at supplying capabilities for safe and effective interoperation between Ada and C programs by extending the ONC XDR/RPC model. Ada language issues posed some problems, which were handled effectively.

Continuing experience with, and evaluation of, Q revealed deeper problems arising from recognition of increasingly taxing demands. Original assumptions that time-slicing executives could be relied upon turned out to be incorrect, and the need to accommodate asynchronous communication was realized. Further, complex systems such as Chiron showed the need for support of peer-peer (in addition to client-server) interprocess communication. Is is now clear that effective, safe, deadlock-free, and efficient support for peer-peer and client-server interprocess communication between components cannot be provided by a simple RPC model. The revised Q model now meets all of these needs.

The simplicity and elegance of the remote procedure call model of interprocess communication is appealing. It is intuitive to most software programmers and is consistent with the notion of object encapsulation so popular today. However, experience in component-based software environments has demonstrated that while many individual component interactions may be client/server in nature, the larger scope of the interactions is quite a bit more diverse and typically more peer oriented. More often than not, components are implemented with multiple threads of control. Individual components act alternately as client or server in unpredictable sequences and often simultaneously. Any interoperability mechanism intended to support this type of interaction must be capable of meshing cleanly with the threading models implied by component implementations to have even a chance at avoiding deadlock and providing efficient service.

Acknowledgments

We would like to thank Greg Bolcer of the Chiron development team and Stan Sutton, the developer of APPL/A and GEM, for their contributions. Their patience in endeavoring to use a new system and their invaluable feedback is appreciated. We would also like to acknowledge Stephen Sykes, who helped architect the initial version of the Q system, and David Levine, who influenced the development of concurrency support in Q.

This material is based upon work sponsored by the Air Force Material Command, Rome Laboratory, and the Advanced Research Projects Agency under Contract Number F30602-94-C-0253. The content of the information does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

References

[1] M. Accetta, R. Baron, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for UNIX development. In *Proceedings of the Summer 1986 USENIX Technical Conference and Exhibition*, June 1986.

[2] Danny Cohen. On holy wars and a plea for peace. *IEEE Computer*, October 1981.

[3] Open Software Foundation. *OSF DCE Application Development Guide.* Prentice Hall, 1993. Revision 1.0.

[4] Dennis Heimbigner. ARPC: An augmented remote procedure call system. Technical Report CU-ARCADIA-100-94, University of Colorado Arcadia Project, November 1994.

[5] Dennis Heimbigner. Why CORBA Doesn't Cut It or Experiences with Distributed Objects. Technical Report CU-ARCADIA-108-95, University of Colorado Arcadia Project, Boulder, CO 80309-0430, 30 June 1995. ftp://ftp.cs.colorado.edu/pub/cs/techreports/arcadia/Misc/sett.ps.Z.

[6] M. Herlihy and B. H. Liskov. A value transmission method for abstract data types. *ACM Transactions on Programming Languages and Systems*, 4(4):527–551, October 1982.

[7] Institute of Electrical and Electronics Engineers. *IEEE Standard for Binary Floating-Point Arithmetic*, August 1985. ANSI/IEEE Standard.

[8] Michael B. Jones and Richard F. Rashid. Mach and matchmaker: Kernel and language support for object-oriented distributed systems. Technical Report CMU-CS-87-150, Carnegie Mellon University, September 1986.

[9] R. Kadia. Issues encountered in building a flexible software development environment: Lessons learned from the Arcadia project. In *Proceedings of ACM SIGSOFT '92: Fifth Symposium on Software Development Environments*, Tyson's Corner, Virginia, December 1992.

[10] R. Kadia. Lessons learned from the Arcadia project. In *DARPA Software Technology Conference*, Los Angeles, California, April 1992.

[11] Rudolf K. Keller, Mary Cameron, Richard N. Taylor, and Dennis B. Troup. User interface development and software environments: The Chiron-1 system. In *Proceedings of the Thirteenth International Conference on Software Engineering*, pages 208–218, Austin, TX, May 1991.

[12] B. H. Liskov, T. Bloom, D. Gifford, R. Scheifler, and W. Weihl. Communication in the mercury system. In *Proceedings of the 21st Annual Hawaii Conference on System Sciences*, pages 178–187. IEEE, January 1988.

[13] MicroSoft. *OLE 2 Programmer's Reference*. MicroSoft Press, 1994. Volumes One and Two.

[14] Microsoft/Object Management Group. *Draft Component Object Model Specification*, 6 March 1995.

[15] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, 29 December 1993.

[16] Xiping Song and Lee J. Osterweil. Debus: A software design process program. Arcadia Technical Report UCI-89-02, Department of Information and Computer Science, University of California, April 1989.

[17] Sun Microsystems. XDR: External data representation standard. Technical Report RFC-1014, Sun Microsystems, Inc., June 1987.

[18] Sun Microsystems. RPC: Remote procedure call protocol specification. Technical Report RFC-1057, Sun Microsystems, Inc., June 1988.

[19] Stanley M. Sutton, Jr., Dennis Heimbigner, and Leon J. Osterweil. Language constructs for managing change in process-centered environments. In *Proceedings of ACM SIGSOFT '90: Fourth Symposium on Software Development Environments*, pages 206–217, Irvine, CA, December 1990.

[20] Stanley M. Sutton, Jr., Hadar Ziv, Dennis Heimbigner, Harry E. Yessayan, Mark Maybee, Leon J. Osterweil, and Xiping Song. Programming a software requirements-specification process. In *Proceedings of the First International Conference on the Software Process*, pages 68–89, Redondo Beach, CA, October 1991. IEEE Computer Society Press.

[21] Richard N. Taylor, Lori Clarke, Leon J. Osterweil, Jack C. Wileden, and Michal Young. Arcadia: A software development environment research project. In *Proceedings of the IEEE Computer Society Second International Conference on Ada Applications and Environments*, pages 137–149, Miami, Florida, April 1986.

[22] Robert B. Terwilliger, Mark J. Maybee, and Leon J. Osterweil. An example of formal specification as an aid to design and development. In *Proceedings of the ACM SIGSOFT '89: Fifth International Workshop on Software Specification and Design*, pages 266–272, Pittsburgh, May 1989. SIGSOFT Engineering Notes.

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

[24] A. L. Wolf, L. A. Clarke, and J. C. Wileden. The AdaPIC toolset: Supporting interface control and analysis throughout the software development process. *IEEE Transactions on Software Engineering*, 15(3):250–263, March 1989.

[25] Alexander L. Wolf, Lori A. Clarke, and Jack C. Wileden. Ada-based support for programming-in-the-large. *IEEE Software*, 2(2):58–71, March 1985.

[26] Michal Young, Richard N. Taylor, and Dennis B. Troup. Software environment architectures and user interface facilities. *IEEE Transactions on Software Engineering*, 14(6):697–708, June 1988.