# Building A Demo: A Comparison Of Three Software Integration Mechanisms

Daniel J. Barrett

Computer Science 791N
Jack Wileden, Professor

December 15, 1993
(printed February 6, 1995)

**Abstract**

This paper describes my experience in using **ToolTalk**, **Polylith**, and **Q** to integrate the components of a large software system. This experiment uncovered practical details about these integration mechanisms that are not obvious from their literature.

## 1 Introduction

The problem of high-level communication between concurrently executing programs has been addressed by many ([4], [5], [10], [11], et. al.). Recently, the term *megaprogramming* [2, 14] has been invented to describe the integration of large, diverse software systems or *megamodules* [14]. Whether existing software integration mechanisms can handle the rigors of megaprogramming has yet to be seen.

As an experiment, three such integration mechanisms — remote procedure calls (RPC) [13], **ToolTalk** [5], and **Polylith** [10] — were each used to integrate a large software system called the Precise Interface Control (PIC) Demo [15]. This paper discusses each mechanism and evaluates its suitability for the task.

## 2 Definitions

For the purposes of this paper, an *integration mechanism* is a software entity that is used to permit two or more programs to communicate. The unit of data communicated from one program to another is called a *message*. Messages have a *name* and zero or more *parameters*.

There are two steps to integrating several programs. First, the programs are given the ability to participate in communication. Second, the nature of the communication may be specified: connections between particular programs, a message language understood by the programs, actions to be taken on receipt of particular messages, etc. Some mechanisms clearly separate these two aspects, and some do not.

## 3 The PIC Demo

The SDL PIC Demo is an on-line demonstration created by the Software Development Lab (SDL) of the Computer Science Department at the University of Massachusetts. The Demo consists of seven C and Ada programs, comprising 100,000 lines (4 megabytes) of source code, running concurrently on a Sun SPARCstation. These programs work together to demonstrate Precise Interface Control (PIC) [15], a paradigm which gives a software engineer control over the import and export behavior of source code modules.

The Demo follows a client-server architecture in which are clients and an Ada program is the server. The programs involved in the Demo are:

- `Server`, written in Ada, which manages (sequentializes) access to a data repository.
- `Chooser`, `Epic`, `PIC_Edit`, and `PIC_Results`, all written in C, comprising a graphic user interface for `Server`.
- `Populate`, written in C, for "initializing" the Demo: it induces `Server` to create the data repository.
- `Monitor`, written in C, for tracing the progress of the Demo.

The C programs are collectively called the *PIC clients*, the *C clients*, or simply *clients* when the term is unambiguous.[1]

The use of several languages to create the Demo was not a carefully planned tactic, but a necessity thrust upon the designers due to software and language constraints. `Server` was created long before the other programs. It was written in Ada because that is the primary language supported by SDL. The clients were written in C because of the need for a graphic interface, available via the **X** window system.

Four classes of communication are used in the Demo:

1. The PIC clients need repository information from `Server`. The information includes strings and large arrays of records.
2. The PIC clients need to share information about the state of the Demo: the current source code module name, occurrences of events, etc. In addition, `Server` occasionally announces new state information.
3. The PIC clients need to invoke one another. For example, `Epic` has menu items for invoking an editor or a results browser.
4. `Server` communicates with a repository.

Class 1 is point-to-point. An individual client directly requests data from `Server` and receives it synchronously. Data types range from simple integers and strings to large arrays of records. Class 2 concerns information that all the clients need to know "simultaneously," so an asynchronous broadcast mechanism was deemed appropriate by the Demo designers. The data being shared here are strings only. Class 3 requires the transmission of strings to the UNIX shell. Finally, Class 4 is handled entirely within `Server` and is not discussed in this paper. See Figure 1, page 4. Light arrows represent Class 1 communication, and bold arrows represent Class 2 communication.

From a megaprogramming point of view, the PIC Demo may be viewed in several ways. On the one hand, each of the seven programs may be considered a "megamodule" [14] since it is a separate program. However, the PIC clients are not very large and have relatively simple purposes compared to `Server`. So an alternate way of viewing the Demo is to consider `Server` as one megamodule, and the set of clients as another. The clients have

[1]We later discuss other kinds of clients, such as "**BMS** clients."

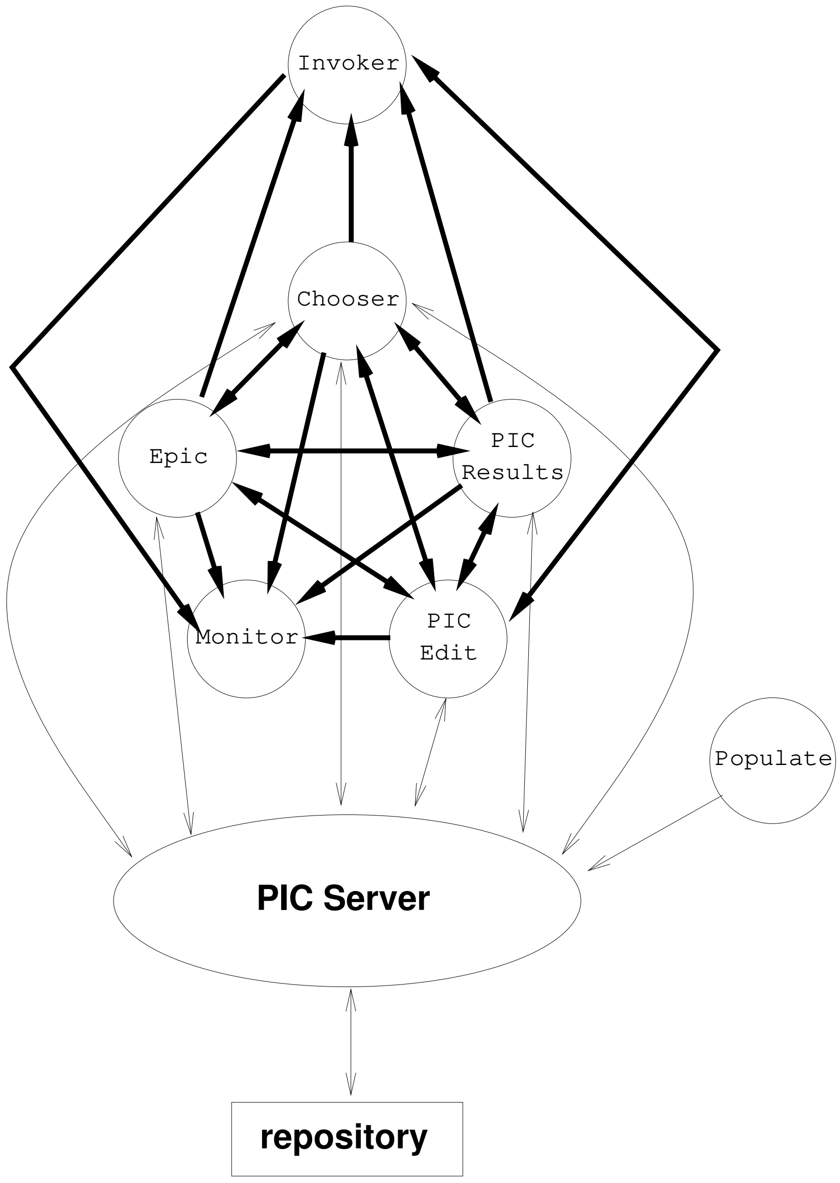

Figure 1: The Communication Architecture of the PIC Demo.

a shared ontology [14] with their view of the server and their broadcast communication system.

The PIC Demo was originally integrated using an RPC mechanism called **Q** [6], which allows communication of data values between C and Ada programs. Using **Q** as a foundation, a broadcast message server called **BMS** [1] was created to handle communication among the clients. For communication between the clients and `Server`, a software interface called **PICPocket** was created, again using **Q**. An evaluation of this integration mechanism is found in Section 5.

## 4 Assumptions

Due to specification changes in the SDL Ada support libraries, PIC's `Server` source code cannot be compiled until it is updated to reflect these changes. Since this update would be fairly major and time-consuming, I wrote a "fake" PIC program to stand in for the real thing. It is written in Ada, and mimics PIC's message interface, but does no actual calculations nor repository management. Its responses are hard-coded. Other than that, it appears to the clients as the real `Server`.

When `Server` is once again compilable, I will update my work to use it so the true Demo can be run. I expect this update to be very little work.

## 5 Sun RPC, Q, and the SDL BMS

**Q** [6] is an RPC mechanism closely patterned after Sun's XDR [12]. It is nearly function-call-compatible with XDR; but unlike XDR, **Q** provides a programmer's interface for Ada. This makes **Q** a potentially suitable choice for integrating the SDL PIC Demo.

**Q** RPC works in the following way:

1. The program declares a "handle" variable.
2. Individual primitive data values (integer, char, string) are packed into the handle, one at a time.

3. An synchronous function call transmits the handle to a second program, and optionally receives a second handle containing results from the second program.

4. The program unpacks the results, one at a time, in FIFO order.

There is no direct support for passing structured data such as arrays and records. To pass such a value via **Q**, its individual fields must be packed into (and subsequently unpacked from) the handle one at a time.

## 5.1 Integration

**Q** provides no "broadcast" mechanism, and one is needed among the clients. So I wrote **BMS**, a string-broadcasting message server. **BMS** is an executable program written in C, and it has the following abilities.

- Programs that wish to use broadcast message service must *register* with **BMS**, and *unregister* when they are no longer interested in **BMS** messages. A registered program is called a *BMS client* (to be distinguished from a PIC client). Registration and unregistration are *dynamic*, accomplished by any **BMS** client while it is executing.

- **BMS** accepts two kinds of messages: *broadcast* and *invoke*. A broadcast message is a string that is transmitted to all registered clients. An invoke message is a string that is passed to the UNIX shell and executed. Invoke messages are used so PIC clients can invoke each other.

- **BMS** delivers messages in the same order that it receives them. Broadcast messages are automatically queued until retrieved by each registered client. Every **BMS** client is guaranteed to receive all messages, and only those messages, transmitted while the **BMS** client is registered. (However, since **BMS** clients poll for their messages, it is possible that the client will choose not to examine all its available messages.)

Integrating the PIC clients was fairly straightforward. I wrote a small library in C of client-related functions for registration, message sending, and message receiving. A client invokes the register function and enters an event loop, waiting for incoming broadcast messages from **BMS**. Upon message receipt, some action is taken which may involve broadcasting other messages.

`Server` itself needed to be a **BMS** client, so I wrote an Ada interface nearly identical to that for the C client library. The main problem encountered here was the incompatibility between Ada and C string formats. Use of SDL's DynamicStrings Ada abstract data type eased these problems somewhat, but some hacks were necessary, including filling unused Ada string elements with ASCII nulls, and truncating Ada strings. There were also some problems involving C NULL strings and Ada empty strings.

Providing the point-to-point interface between `Server` and the clients was fairly straightforward. Analogous C and Ada record types were defined, and C and Ada subroutines were written to pack arrays of records into handles transmit them via **Q**, and unpack them. The software interface responsible for these tasks is called **PICPocket**.

## 5.2 Pros & Cons

**BMS** has some inherent advantages:

- Dynamic registration and unregistration.
- From time to time, `Server` needs to ignore incoming messages (turn off its event loop) and cause incoming messages to be queued during this time. This would be easy to implement, but how can one inform `Server` (via a message) to resume processing messages, if `Server` is not listening to messages? It seems we have a Catch-22 situation. But **Q** has a feature which causes ordinary messages to be ignored, but special *priority messages* to be received. By making the "resume processing" message a priority message, the problem was solved.
- **Q**'s handle-passing function is synchronous: a handle is sent and another handle is received in a single operation. Other mechanisms such as **ToolTalk** do not have such a feature.

Some negative aspects of the **Q/BMS** approach include:

- **BMS** limits messages to be strings.

- Each client is responsible for parsing its own message strings to add and/or retrieve parameters. **BMS** does not assist in this process in any way.
- **BMS** does no "filtering" of messages. Every registered **BMS** client receives every broadcast message, even those that it sends.
- **Q** has a static limit on the amount of data that may be transmitted via a handle. The SDL PIC Demo exceeded this limit regularly, since the arrays of records managed by **PICPocket** were very large.

## 6 ToolTalk

**ToolTalk** [5] is an integration mechanism created by Sun Microsystems. It is a library of over 200 functions for message passing. Its operation is similar to that of **Q**: data values are packed into messages that get transmitted. However, **ToolTalk** has several notable differences from **Q** and **BMS**:

1. All **ToolTalk** message passing is asynchronous.
2. Clients may register not only for general **ToolTalk** service, but also to receive only particular classes of messages. This feature provides message *filtering*, so clients needn't receive every broadcast message.
3. Registration for message classes may be done statically (at client compile time) or dynamically (at client run time).
4. **ToolTalk** clients may be automatically invoked when their services are needed. (However, this auto-invoke service is too limited: see Section 6.2, page 10.)
5. Message parameters may be added or retrieved by random access (the index of the parameter). **Q** allows message parameters to be accessed by FIFO only.
6. **ToolTalk** has no Ada interface.

## 6.1 Integration

Integrating the C clients provided several challenges. The first was deciding whether to use static or dynamic registration. Static registration provides several additional features, such as the auto-invocation mentioned in the previous section. However, experiments with the static registration mechanism revealed two serious limitations.

1. **ToolTalk** uses a flat namespace for messages. That is, if two completely unrelated programs just happen to use the same message name, there is a silent (and undocumented) conflict with unpredictable behavior.

2. **ToolTalk**'s auto-invocation mechanism can invoke only one copy of a given program.

As a result of these limitations, I decided to use dynamic registration. For consistency in message names, I wrote a small library of functions for **ToolTalk** registration, broadcasting, and message receipt, and all PIC clients used it. To handle program invocation, I wrote a program called `Invoker`, a **ToolTalk** client that passes its incoming messages to the UNIX `execvp()` function. `Invoker` was largely a reworking of the invocation code from **BMS**.

Once the initial work above was done, integrating the PIC clients with **ToolTalk** became as easy as with **BMS**. In fact, the source code of a **BMS**-integrated client could be translated into a **ToolTalk**-integrated client almost automatically. (This was a happy coincidence, since at the time I wrote **BMS**, I was not familiar with **ToolTalk**.)

The next challenge was to create an Ada interface for **ToolTalk**. This was feasible thanks to the Ada `pragma interface` feature, which allows invocation of, and parameter passing to and from, functions written in other languages. An important implementation decision was whether to write pragmas for the individual **ToolTalk** functions, or for higher-level, more complex functions that invoked **ToolTalk** functions. I chose the latter approach to simplify the parameter passing, since I did not know the internals of **ToolTalk** messages and did not want to write parallel **ToolTalk** datatypes in Ada. In retrospect, this decision may have been the wrong one, since I found myself modifying my complex functions numerous times to accommodate new, unexpected features and parameters.

Creating this **ToolTalk** Ada interface was a royal pain, requiring much low-level "bits 'n bytes" experimentation since documentation was scarce. Transmitting strings between C and Ada was particularly difficult, but the Ada *C_Strings* package helped a lot. Variable parameters also caused problems and necessitated heavy use of the Ada `'address` attribute.

To handle the point-to-point communication between `Server` and its clients, I essentially reimplemented **PICPocket** with **ToolTalk** instead of **Q**. However, there was one main difficulty in doing this: all **ToolTalk** message passing is asynchronous, whereas **Q** had a synchronous send-and-receive-handle function. As a result, I created two **ToolTalk** communication channels per client: "in" and "out." `Server` registers to receive messages on all "out" channels, and each client on its "in" channel. To request data from `Server`, a client sends a request on its "out" port and then busy-waits for a response on its "in" port.

An unexpected problem was encountered after the point-to-point interface was implemented. For some unknown reason, certain messages were not being received. The problem was traced to the fact that each client was registering for **ToolTalk** service twice: once for broadcasting, and once for point-to-point. The second registration was being ignored: it is semantically incorrect to register twice. (This point is undocumented.) So unlike **Q**, **ToolTalk** does not support multiple sessions.

## 6.2 Pros & Cons

Advantages of using **ToolTalk** to integrate the Demo include:

- Dynamic registration and unregistration for clients.
- Dynamic registration and unregistration for individual messages, reducing message traffic in the Demo.
- Relatively easy to use.

Disadvantages include:

- No synchronous message passing.
- No Ada binding.

- A flat namespace for message names. If two different clients have messages with the same name, a message sent to the first client might be routed to the second incorrectly.

- Only one instance of a program can be auto-invoked.

- Installing static registration files ("ptype files") involves the use of arcane UNIX commands (such as sending a "kill -USR2" signal to the `ttsession` process).

- Clients must provide their own method for waiting for the next message, since **ToolTalk** does not provide one. It is expected that one will use an **X** toolkit function for this purpose. I used simple busy-waits for the purposes of this experiment.

- I could discover no way to implement the "suspend" and "resume" `Server` operations. A brute-force method — "suspend" causes `Server` to unregister for all messages except the "resume" message — failed for unknown reasons. The semantics of unregistering in **ToolTalk** are not well documented.

## 7 Polylith

**Polylith** [10] was created by Jim Purtilo of the University of Maryland. It is an implementation of the *software bus* concept [9] in which programs may communicate point-to-point via named "bus channels."

Program integration with **Polylith** is accomplished in the following manner:

1. One or more *bus channels* are invented by choosing arbitrary names.

2. Each participating program's source code is modified to insert function calls to read from and write to specific bus channels. These programs are then compiled and linked with a **Polylith** library.

3. Using a language called *Module Interconnection Language* (*MIL*), each program's bus input and output (that is, the number and types of the parameters) are specified. The resulting specification is called a *service*. A service may be thought of as a wrapper around a program, specifying its **Polylith** input and output interfaces.

4. Using MIL, services are instantiated to create *tools*.
5. Using MIL, connections between different tools' input and output channels are specified.
6. The MIL specifications are compiled to produce a program which can be interpreted by the **Polylith** bus program, `bus`. `bus` invokes all of the desired tools and manages the communication among them.

These MIL specifications are largely independent of the programs they refer to, so program interconnections may be changed without any modification to their source code. At the same time, unlike **Q** and **ToolTalk**, **Polylith** requires this interconnection information to be specified statically. This fact has important ramifications that will be explored in the next section.

Version 2.1 of **Polylith** was used for this experiment. See Section 7.3, page 14 for important information about **Polylith** version 3.0.

## 7.1 Integration

**Polylith** was the most challenging of the integration mechanisms to apply to the PIC Demo. Primarily, the Demo has numerous dynamic features:

1. Clients invoke other clients as needed.
2. Multiple instances of clients (namely `PIC_Edit`) are needed, but the number of instances is not known before runtime.
3. Arrays of structures are passed between PIC and its clients, but the sizes of these arrays are not known before runtime.
4. The PIC clients use command-line arguments. However, in the current implementation, **Polylith** "steals" all command-line arguments for its own purposes. This is a conflict. **Polylith**'s *object attribute* features, which are often used in place of command-line arguments [7], unfortunately must be declared statically in a program's MIL specification.[2]

[2] There is reportedly another **Polylith** feature called *polyargs* which is dynamic enough to use for the PIC Demo. I am awaiting information about it.

All of these properties make the integration of the PIC Demo with **Polylith** very difficult. In fact, one could go as far as to say that **Polylith** version 2.1 is the wrong mechanism for the job. Nevertheless, I accomplished some integration tasks, which I describe.

First, I adapted the C clients to broadcast message strings via **Polylith**. This was challenging because **Polylith** 2.1 does not support multicast: it is strictly a point-to-point mechanism. To get around this problem, I used the following brute-force implementation:

1. For each of the $N$ clients, create $N-1$ incoming and $N-1$ outgoing bus channels.
2. Specify connections from every client to every other client via unique bus channels.
3. Write a custom "multicast" function which, given a message, iterates through all the incoming bus channels and transmits the message into all of them except the sender's own channel.
4. Use **Polylith**'s `mh_readselect` for message receipt, so the client need not have the identity of the sender hard-coded.

In addition, I had to specify the number of copies of multiply-invoked clients (`PIC_Edit`) statically, each with its own uniquely named bus channels. All in all, this method requires the explicit specification of $O(N^2)$ interface bindings which is both time-consuming and (I found) error-prone to maintain.

In any event, I was able to get **Polylith** to invoke all the client programs simultaneously (ignoring the issue of dynamic invocation for the moment). to broadcast messages. However, I immediately encountered the "command-line argument" problem described above (dynamic feature # 4 on page 12). The clients would execute, display "usage" messages, and quit. This caused communication to hang, since (presumably) `bus` does not expect its tools to terminate abnormally.

At this point, acknowledging the current difficulties and aware that further problems (e.g., the array length problem) lay ahead, I abandoned the use of **Polylith** to integrate the PIC Demo. This should not be seen as a denigration of **Polylith**'s capabilities, but rather as a realization of its inappropriateness for this particular application. **Polylith** has been shown to be useful for other real-life programming problems [8].

## 7.2 Pros & Cons

**Polylith** has some important strengths:

- Once application programs have been modified to contain bus read and write calls, it is very convenient to create and modify connections between programs using MIL. This property makes **Polylith** good for rapid prototyping. It also eased my **Polylith** learning curve, and aided in debugging, since it was trivial to write small test programs to communicate with the actual programs in the PIC Demo.
- The explicit MIL specification of inter-program communications may aid global static analysis of a **Polylith**-integrated system. I have not heard of any work in this area, but it seems like a logical step to take.
- There are bindings for several languages including C, Ada, and Lisp.

**Polylith** 2.1's disadvantages include:

- The lack of dynamism in registration/unregistration, program invocation, array size, etc.
- Lack of multicast, and the $O(N^2)$ problem it implies.
- The command-line argument problem.
- Bus channel names must be identical in a program's source code and its MIL service specification, but **Polylith** provides no way to enforce this constraint automatically.

Several of these disadvantages have been addressed in **Polylith** 3.0.

## 7.3 About Polylith 3.0

As this document went to press, Jim Purtilo sent me a draft of the manual for **Polylith** 3.0 [3]. This version contains some significant changes that will make **Polylith** much more suitable for the PIC Demo and other applications with similar requirements for dynamic behavior. New features include:

- Multicast.
- Dynamic registration and unregistration for receipt of multicast messages.
- Dynamic modification of inter-program communication links while the programs are running.
- A language binding for LISP.

Once version 3.0 is officially released, I plan to re-evaluate it with respect to the PIC Demo.

## 8 Analysis

The following table summarizes some of the significant features of the three integration mechanisms studied:

| **Feature** | **Q/BMS** | **ToolTalk** | **Polylith** |
|---|---|---|---|
| *Modify Source?* | yes | yes | yes |
| *Program Registration* | dynamic | static, dynamic | static |
| *Message Filtering* | no | yes | no |
| *Program Invocation* | dynamic via **BMS** | static | static via bus |
| *Communication* | point-to-point, broadcast | multicast | point-to-point |
| *Sender/Receiver Synchronization* | synchronous, asynchronous | asynchronous | synchronous, asynchronous |
| *Structured Parameter Handling* | manual | manual | built-in |
| *Parameter Access* | sequential | random | random |
| *Specification Of Communication Links* | no | no | yes |
| *Access Control* | no | no | yes |

"Modify Source" indicates whether or not the source code of programs must be modified in order to participate in communication using the given mechanism. All three mechanisms surveyed require source code modification. Other mechanisms such as **SoftBench** [4] do not, but they have other limitations.

"Program Registration" and "Message Filtering" indicate how (if at all) participating programs register for integration mechanism services in general, and for receipt of particular messages, respectively. **ToolTalk** was the most powerful of the mechanisms in this regard, although its static registration implementation is seriously flawed (as described in Section 6.1, page 9). It seems that registration for individual messages does not benefit individual programs, since they still must parse the incoming message to determine what it is, and what action to take. However, this registration does benefit the system as a whole, since filtering means less message traffic.

"Program Invocation" indicates how participating programs are invoked by the mechanism. **BMS** provides an "invoke" message. **ToolTalk** provides automatic invocation of one copy of a statically-declared program on first receipt of a message. (I wrote `Invoker` to get around the limitations of this method.) **Polylith** runs all its participating programs at once.

"Communication" and "Sender/Receiver Synchronization" indicate the underlying communication protocols available to a programmer using the mechanism.

"Structured Parameter Handling" indicates the mechanism's support for passing arrays and records. Only **Polylith** provided built-in support for these types. **Q** and **ToolTalk** require the participating programs to consider arrays and records as collections of primitive types and pack the primitive values into "boxes" or "handles" for transit to the receiver. The receiver must then unpack the boxes manually. "Parameter Access" indicates the order in which participating programs may insert or examine parameters in a multiple-parameter message.

"Specification Of Communication Links" indicates whether the mechanism has support for an explicit specification of the connections between programs. **Polylith** does, but **BMS** and **ToolTalk** do not, making their architecture and behavior harder to analyze.

Finally, "Access Control" indicates whether the mechanism enforces any kind of protection of the data so unauthorized or unregistered programs cannot read nor modify it. Only **Polylith** has access control, and of a rudimentary sort: since `bus` invokes all participating programs and controls their interactions as indicated in the MIL specification, no outside program has access to the data being passed. **BMS** and **ToolTalk** both allow any program to register to receive any or all data being transmitted.

## 9 Conclusions

None of the surveyed integration mechanisms is ideal for the SDL PIC Demo, but each has its particularly helpful features. The **BMS/PICPocket** solution was custom-made for the Demo, so it supports exactly the communication paradigms needed, but it lacks message filtering and carries along the limitations of **Q**. The **ToolTalk** solution addressed the limitations of **Q**, but its lack of synchronous messaging and limitations in static registration are real drawbacks. Finally, **Polylith** provides some features that **Q** and **ToolTalk** are missing — notably structured parameter passing and explicit specification of inter-program communication links — but its inherently static nature makes it unsuitable for a project with so much dynamic behavior.

In the future, I would like to continue this experiment by using **Polylith** 3.0, HP's **SoftBench**, and DEC's **FUSE** to integrate the PIC Demo. I would also like to use the actual PIC server instead of my "fake" one, once PIC has been adapted to use SDL's new Ada support libraries.

## References

[1] Daniel J. Barrett. SDL BMS: A Simple Broadcast Message Server. Arcadia Document UM-93-03, University of Massachusetts, Software Development Laboratory, Computer Science Department, October 1993.

[2] Barry W. Boehm and William L. Scherlis. Megaprogramming (Preliminary Version).

[3] Charles Falkenberg, Christine Hofmeister, Chen Chen, Elizabeth White, Joanne Atlee, Paul Hagger, and James Purtilo. *The Polylith Interconnection System: Programming Manual for the Network Bus.* University of Maryland, College Park, 3.0 edition, September 1993. Draft.

[4] Colin Gerety. HP SoftBench: A New Generation of Software Development Tools. SoftBench Technical Note Series SESD-89-25, Revision 1.4, Hewlett-Packard, November 1989.

[5] Astrid Julienne and Larry Russell. Why You Need ToolTalk. *SunEXPERT Magazine*, pages 51–58, March 1993.

[6] M. Maybee, L. J. Osterweil, and S. D. Sykes. Q: A Multi-lingual Interprocess Communications System for Software Environment Implementation. Technical Report CU-CS-476-90, University of Colorado, Boulder, June 1990. Currently undergoing review for publication in Software Practice and Experience.

[7] James Purtilo, December 1993. Personal communication.

[8] James Purtilo, Charles Falkenberg, Elizabeth White, William Andersen, and Tess Ollove. An Exercise With Prototyping Technology. Technical report, University of Maryland, 1993. Draft.

[9] James Purtilo, Richard T. Snodgrass, and Alexander Wolf. Software Bus Organization: Reference Model and Comparison of Two Existing Systems. Technical Note 8, DARPA Module Interconnection Formalism Working Group, November 1991.

[10] James M. Purtilo. The Polylith Software Bus. Technical Report UMIACS-TR-90-65, University of Maryland, May 1990.

[11] Steven P. Reiss. Connecting Tools Using Message Passing in the Field Environment. *IEEE Software*, July 1990.

[12] Sun Microsystems. XDR: External Data Representation Standard. Technical Report RFC-1014, Sun Microsystems, Inc., June 1987.

[13] Sun Microsystems. RPC: Remote Procedure Call Protocol Specification. Technical Report RFC-1057, Sun Microsystems, Inc., June 1988.

[14] Gio Wiederhold, Peter Wegner, and Stefano Ceri. Toward Megaprogramming. *Communications of the ACM*, November 1992.

[15] 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.