Technical Report

CMU/SEI-92-TR-035
ESD-TR-92-35

Control Integration Through Message Passing in a Software Development Environment

Alan W. Brown

December 1992

Technical Report

CMU/SEI-92-TR-035
ESD-TR-92-35
December 1992

Control Integration Through Message Passing in a Software Development Environment

Alan W. Brown

CASE Environments Project

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

This report was prepared for the

SEI Joint Program Office
HQ ESC/AXS
5 Eglin Street
Hanscom AFB, MA 01731-2116

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

FOR THE COMMANDER

(signature on file)

Thomas R. Miller, Lt Col, USAF
SEI Joint Program Office

This work is sponsored by the U.S. Department of Defense.

This work was created in the performance of Federal Government Contract Number F19628-95-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. The Government of the United States has a royalty-free government-purpose license to use, duplicate, or disclose the work, in whole or in part and in any manner, and to have or permit others to do so, for government purposes pursuant to the copyright license under the clause at 52.227-7013.

This document is available through Research Access, Inc., 800 Vinial Street, Pittsburgh, PA 15212.
Phone: 1-800-685-6510. FAX: (412) 321-2994. RAI also maintains a World Wide Web home page. The URL is http://www.rai.com

Copies of this document are available through the National Technical Information Service (NTIS). For information on ordering, please contact NTIS directly: National Technical Information Service, U.S. Department of Commerce, Springfield, VA 22161. Phone: (703) 487-4600.

This document is also available through the Defense Technical Information Center (DTIC). DTIC provides access to and transfer of scientific and technical information for DoD personnel, DoD contractors and potential contractors, and other U.S. Government agency personnel and their contractors. To obtain a copy, please contact DTIC directly: Defense Technical Information Center, Attn: FDRA, Cameron Station, Alexandria, VA 22304-6145. Phone: (703) 274-7633.

Table of Contents

List of Figures

Control Integration Through Message Passing in a Software Development Environment

Abstract: Understanding tool integration in a Software Development Environment (SDE) is one of the key issues being addressed in current work on providing automated support for large-scale software production. Work has been taking place at both the conceptual level ("What is integration?") and the mechanistic level ("How do we provide integration?"). Many people see the answers to these questions as providing the cornerstone of real progress in the area.

Until recently, existing integration mechanisms have been very rigid in the support for integration that they provide. Users have been offered a fixed level of integration with little flexibility. However, one approach that has been recently implemented employs a control integration paradigm that appears to be flexible, supportive, and adaptable to a wide range of end-user needs. Implementations of this paradigm are based on the notion of "message passing" as the underlying communication mechanism between SDE services.

In this paper we examine the message passing approach to integration in an SDE, look at the general principles of the approach, describe some existing implementations, and discuss the use of such a mechanism as the basis for a more flexible environment that is open to experimentation with different approaches to integration.

1. Introduction

Controlling and coordinating tool interactions in a software development environment (SDE) requires an approach to tool integration that is both flexible and adaptable enough to suit different user needs, as well as simple and efficient. These two conditions will ensure that new tools can be easily integrated and that the productivity of the tools is not significantly impaired. Traditional approaches towards tool integration have been based on data sharing, most often through a common database in which all tools deposit their data [2]. While this approach can provide a high level of control and coordination between tools, it also imposes a significant overhead on the tools, both because of poor performance of existing database mechanisms when used in this way, and because of the necessary agreement required between the tools to define a common syntax and semantics for their data (e.g., a common data schema).

One approach to integration that has been developed lately has been called *control integration*. This approach is based on viewing an SDE as a collection of services provided by different tools. Actions carried out by a tool are announced to other tools via control signals. The tools receiving such signals can decide if the other tool's actions require that they take any actions themselves. For example, when an editing tool announces that changes have been made to a source file, a build tool may receive this information and initiate a new system build. In addition, one tool may directly request that another tool perform an action by sending it a control signal. For example, the build tool may request that the source file be compiled by a particular compiler. Hence, the primary means of coordination between tools is through the sending and receiving of control signals.

In the rest of this paper we examine the notion of control integration in an SDE, review a number of existing systems, and analyze those systems to identify their differences and to reveal interesting future directions for this work.

The reviewed systems do not represent an exhaustive examination of systems implementing a control integration approach. Rather, they are illustrative of the range of sophistication of such systems.

The paper is organized as follows. Section 2 contains a discussion on integration, illustrating its importance, and introducing the concept of control integration. Section 3 describes a control integration technique employed by a number of SDEs known as the message passing approach. Three such SDEs — FIELD, SoftBench, and ToolTalk — are reviewed in Section 4 and analyzed in Section 5. The paper concludes with a summary in Section 6.

2. Integration in an SDE

We can identify three strands of work associated with the analysis of integration in an SDE:

1. Defining new *mechanisms* for integration — database solutions, common user interface standards, intermediate data interchange formats, and so on. Examples of this work are the CAIS [17] and PCTE [14] efforts in the definition of public tool interfaces and services and the CDIF standard [9] for data interchange between CASE tools. The reference model for frameworks of CASE environments adopted by both the National Institute of Standards and Technology (NIST) and the European Computer Manufacturers Association (ECMA) as a technical report [1] fits mainly into this category, as its present focus is the description of framework services to facilitate the comparison of different CASE products.

2. Examining the *semantics* of integration — discussing what integration means, what levels of integration are possible, what the costs of integration are, and so on. Initial work by Wasserman [23], the classifications by both Thomas and Nejmah [22], and Brown and McDermid [6], and the work of Wallnau and Feiler [5] are all useful in this regard. To a lesser extent, the NIST/ECMA reference model also discusses tool integration semantics.

3. Analyzing the relationship between *integration and process* — how tool integration affects the software development process, where in the development life-cycle different levels of integration are most appropriate, and so on. Little documented work appears to be available in this area.

A general conclusion in much of this work is that the issue of integration is a much more subtle and pervasive characteristic of an SDE than may have initially been envisaged. In particular, the work points to the fact that:

- Integration is a property that may be applied to many different facets of an SDE.

- Addressing integration in one facet does not necessarily imply anything about improving integration in other facets.

- Measurement of integration within any facet should be more refined than simply stating that there is “tight” or “loose” integration — producing a suitable calibration for each facet is currently a topic for research.

- Trying to achieve the greatest amount of integration within each facet is not always desirable in a particular SDE, as there are potential drawbacks associated with tight integration (for example, the difficulties of evolving the SDE and of reusing components of the SDE [6]).

All of these points lead us to consider the nature of existing mechanisms for integration and the architectural framework within which those mechanisms can be found. As an example, consider the most widespread mechanism that is envisaged when the issue of integration is discussed — a common data repository. The classic approach to tool integration, as

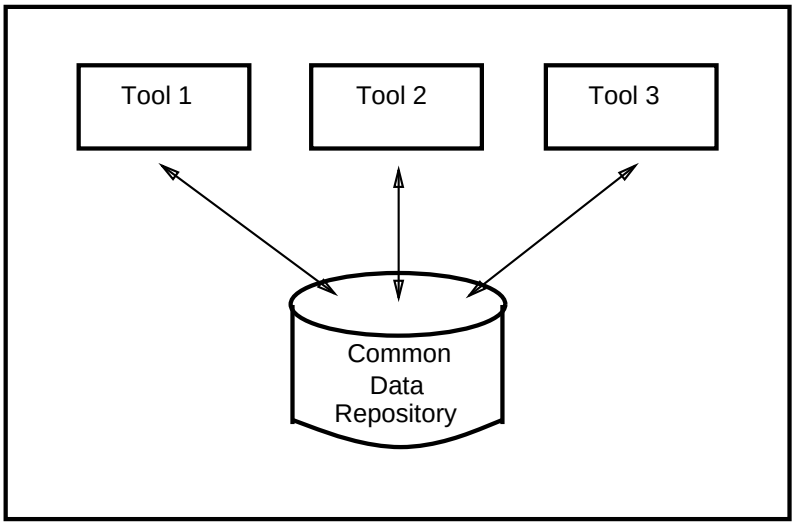

Figure 2-1 Integration Through a Common Repository

defined in the Stoneman report [7], has been integration through a common database of tool information. The use of a central repository as a means of integration can be referred to as a *data integration approach*. In this approach, tools deposit data into a common repository, and refer to the repository when information is required, as illustrated in Figure 2-1. There are clear advantages to this approach in terms of its central control of all operational data, allowing tools to share information in an unambiguous, well-defined way. However, there are also problems associated with this approach:

1. A definition of the common data structures must be agreed *a priori*. This requires the development of an understanding of both the structures and the semantics of those structures as they will be produced/used by all tools to be integrated. Difficult as this is, it is made considerably worse when, at some time in the future, a new tool needs to be integrated. In order for existing tools not to be affected, only consistency-preserving additions to the common data definitions are permitted.

2. The common data repository is often a large, complex resource that must be controlled, managed, and maintained. This can occupy a great deal of time, money and effort on behalf of the organization using the SDE.

3. To support the large common data repository, a complex software system will typically be required. This will add to the cost and complexity of the SDE itself and have an impact on overall system performance. In some documented cases, this overhead has been far from negligible, and has significantly impaired the usability of the SDE.

These factors have lead to the development of much simpler, less demanding approaches to tool integration. While these simpler approaches may reduce the level of support, the corresponding costs involved in introducing, using and maintaining the SDE are also reduced.

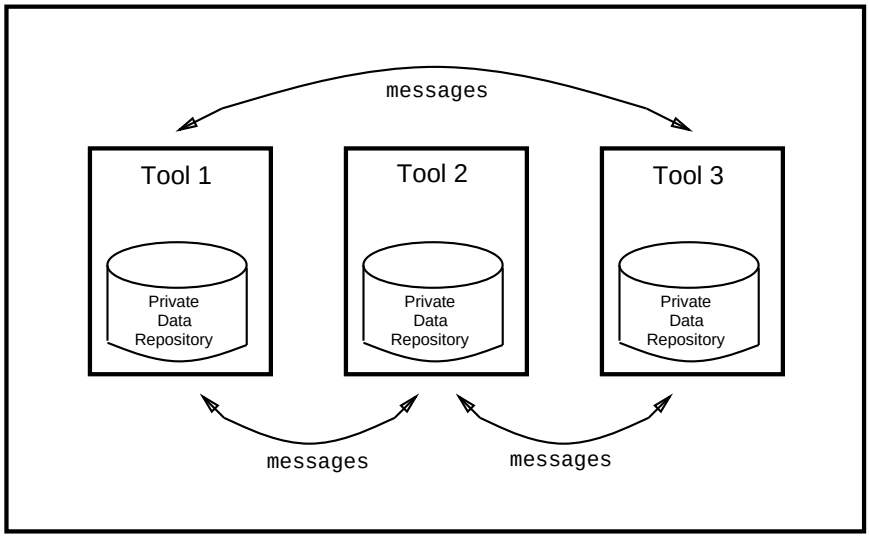

Figure 2-2 Integration Through Message Passing

One approach that is of particular interest is based on a *control integration* approach. In this approach, rather than communication between tools primarily taking place via shared data structures, tools interact more directly by requesting services of each other.[1] When one tool requires some action to be taken, it can request that another tool perform that action, rather than directly implementing (and possibly duplicating) that functionality itself. Implementing this approach results in an SDE architecture in which tools can be viewed as collections of services. Each tool performs a small, well-defined set of functions, and provides access to its services through a programmatic interface. One way to visualize the control integration approach, as illustrated in Figure 2-2, is as tools communicating via messages.

We now look in more detail at the control integration paradigm by describing an architectural approach to an SDE based on message passing.

[1] Intuitively, we can distinguish control integration from data integration by making the conceptual distinction that data integration has a single point of control to which all data is moved. In contrast, control integration allows data to be dispersed across different tools and allows control signals to coordinate the interaction of those tools.

3. The Message Passing Approach

In the message passing approach, tools in an SDE communicate by passing messages informing other tools of their actions and requesting services from the other tools. In order for meaningful communication to take place between the tools appropriate mechanisms need to be in place to allow the communication, and an agreed protocol must be established between the tools to ensure that messages are sent at the necessary times and that their meaning can be interpreted upon receipt.

The mechanistic aspects of the architecture are provided by a *message server*, which is responsible for distributing all messages between tools. The message may contain some indication of which tool, or set of tools, should receive the message, and may define a scope (e.g., local host) for which the message is applicable. Typically, however, the message server has no knowledge of the semantics of the messages.

For effective communication, tools must agree on both the syntax and semantics of the messages they send to each other. For example, between a configuration management (CM) and a design tool, the syntax and semantics of a version check-in operation, or event, may be agreed to allow the design tool to issue such an operation at the end of each design session and for the CM tool to respond in the appropriate manner. The messages themselves generally have a very simple format, each message containing sufficient information to identify the sender, recipient, and the operation being requested (with any necessary parameters such as the names of the files that are being manipulated). Achieving these syntactic and semantic agreements between tools is essential for meaningful communication.

As a further illustration we can consider a typical program editing session. Here, on invocation the editing tool would send out a message to inform other interested tools that it had started. Subsequently, when important events occur (such as the saving of the file being edited) appropriate messages are sent. On saving a file, for example, a program build operation, the collection of metrics, or the sending of electronic mail to the quality controller could all be triggered.

An important point to emphasize in this process is that the tools that are sending messages essentially broadcast their messages to all tools by sending them to the message server. The tools do not need to know which other tools are currently running in order to send messages, and hence are uneffected by changes to executing tools in the environment. It is the responsibility of the message server to selectively forward messages to appropriate tools (a process that is sometimes called *selective broadcast* or *multicast*) or to initiate execution of a tool to handle the current message.

3.1. Inserting a Tool into a Message Passing Architecture

Inserting a tool into a message passing architecture typically requires that at least the following actions take place:

- Conversion of the input to, and output from, a tool into message responses and requests. Message events must be defined that initiate some part

of the tool's functionality on receipt of a particular message and messages transmitted when the tool performs a particular action or reaches a particular state.

- Modification of the tool's user interface can also be applied to provide a common "look and feel" for all the tools in an environment. To aid the visual consistency and ease of use of the tool, a window-based interface with buttons and menus can be constructed.

The message and user interface handling routines can be provided as an envelope, or wrapper, for the original tool to enable the original tool to remain unchanged.

The only remaining task is to write the application routines that dictate the connections and sequencing of events that model a particular development scenario. For example, part of a typical software development scenario could be that the action of quitting from the source code editor initiates the following events:

1. A CM tool is called to record the saved source code as the latest version of the software.
2. A new object module is generated from the source code by invoking the compiler.
3. Assuming the source code compiles successfully, the appropriate libraries are used to build a new executable image of the system.
4. A metrics tool is invoked to collect and store information about the new source code module.
5. A test case generation tool is invoked to generate a new set of test cases so that the new executable image can be checked for correct operation.

The application routines necessary to implement such a scenario would initiate events, test the values returned in response to those events, set up the appropriate control structures, and so on.[1]

3.2. Comparison with "Point-to-Point" Tool Connection

The most prevalent way in which tools are currently interconnected is via direct "point-to-point" connection between the tools — one tool will make direct calls to the interface(s) of another tool. Where access to the source code of the tools exists (e.g., when the tool vendors themselves implement the integration) the tools are often amended to use those interfaces. Otherwise, some translation and filtering routines may be used to implement the integration.

In both cases the disadvantage of the "point-to-point" integration is that the integration is targeted specifically toward those tools and the particular interfaces provided by those tools.

[1] We have implemented a very similar scenario to this using the HP SoftBench product and a number of commercial CASE tools [3].

Thus, a design tool that wishes to integrate with three different CM tools typically offers three different versions of its integration software, each targeted at one of the CM tools. Furthermore, there is the ongoing problem of maintaining the integrations as the products evolve.

The message passing approach attempts to overcome these shortcomings by generalizing and abstracting the tool interconnection service in the form of a message server. Hence, the necessary communication between tools is more explicit, visible, and controllable. Agreements between the tools on when and what to transfer are still required, but the message passing approach provides the mechanism and forum within which such agreements can be made, documented, and allowed to evolve.

4. Examples of the Message Passing Approach

There have been a number of SDE implementations based on the principles described above. In this section we describe three such implementations — FIELD, SoftBench, and ToolTalk. Two other prominent implementation, Digital's FUSE and IBM's WorkBench/6000, are derivations of FIELD and SoftBench, respectively. Hence, for the purposes of this paper a review of those products would be superfluous.

4.1. FIELD

Developed by Steven Reiss at Brown University, the FIELD[1] environment is the origin of most of the work on the message passing approach [20, 21]. The initial implementation was available at the end of 1987.

The FIELD environment was developed for use at Brown with the following basic aims in mind:

- To establish the principle that highly interactive environments as seen with many PC-based packages can be developed for large-scale programming with equal success;
- To experiment with the extensive graphical capabilities of current workstations to enhance the quality and productivity of software development;
- To provide a platform for tool integration at Brown University capable of supporting the teaching of programming and as the basis for further research.

The two basic components of FIELD which support these aims are the use of a consistent graphical user interface as a front-end to all tools and a simple integration mechanism based on message passing.[2]

4.1.1. The Message Server

The message server, *Msg*, corresponds very closely with the general description given above. It is claimed that the power of this approach is a consequence of the flexibility that it provides. In particular, messages are passed as arbitrary length text strings. This ensures that no fixed protocol for messages is predefined, encouraging tools to form their own collaborations, sharing knowledge of message formats to provide closer cooperation. In addition, allowing the user to amend Msg easily facilitates different approaches towards creating, transferring, and parsing of messages.

[1] FIELD stands for "Friendly Integrated Environment for Learning and Development."

[2] While the graphical front-end aspects of FIELD are interesting and important to its use, in this paper we concentrate exclusively on the integration mechanisms.

In FIELD the message server is implemented as a separate UNIX process, communicating with other processes via sockets. The implementation is approximately 2,000 lines of C code comprising the server, a client interface, and a pattern matcher.

4.1.2. Messages

There are a number of interesting characteristics of the messages sent and distributed in the FIELD environment.

First, FIELD distinguishes two broad categories of message — commands and information messages. Commands are sent to a particular tool or class of tools requesting some service be performed. Information messages are typically more widely broadcast to inform all other interested tools of some event that has just occurred. Secondly, messages may be sent synchronously or asynchronously. In synchronous transmission, a tool sends a message and waits for a response. In asynchronous transmission, once a tool sends a message it resumes normal operation. As can be expected, command messages are normally synchronous messages and wait for an acknowledgement or response to be returned from the command, while information messages are asynchronous.

As a result of the above, message formats take one of two forms, corresponding to the two kinds of message.

1. **Commands** — name of recipient, command name, system name, arguments to command.
2. **Information messages** — name of sender, event causing message, system name, arguments to message.

In both of the above cases, a *system name* is used to allow the message server to distinguish between different invocations of the same tool.

4.1.3. Summary

The FIELD environment is an interesting example of the message passing approach to integration, and a number of success are claimed for it, including support for the development of a system of greater than 100,000 lines of code.

However, there are two areas of concern. One is that there are a number of unsubstantiated claims made by Reiss about the FIELD environment. For example, the statement:

> ...my experience is that the level of integration [in FIELD] is high enough for almost all applications and that complete integration is not necessary. [20]

This is a very strong statement to make without any substantive discussion. At best it must be considered an interesting hypothesis to examine. In fact, the testing of this hypothesis

may be the key to many of the problems being addressed in tool integration. In particular, it is unclear what level of integration is required, and indeed, whether an environment can be justified in providing only one level of integration, no matter which tools are being integrated, where they fit in the development life-cycle, how they are to be monitored by management, and so on. We shall return to this crucial point in the final discussion section of this paper.

It is also worth noting that FIELD has so far been directed at *programming*. From FIELD's point of view it remains to be seen if the same mechanisms can scale up to *project* support — supporting technical and managerial aspects of software development, covering much more of the development life-cycle, and supporting simultaneous multiple access to the system. Some of these aspects are described in the papers on FIELD as "work in progress."

At a more pragmatic level, Reiss recognizes the problems with performance of the current FIELD implementation suggesting various possible optimizations. However, he also points out that much of the problem lies in the layering of the FIELD environment directly on top of UNIX, and the problems of attempting to use batch-based tools in an interactive mode. This leads to the possible conclusion that tools must be designed and implemented with message passing integration in mind in order to properly exploit the message passing mechanism. If true, this conclusion has important implications for tool vendors.

4.2. SoftBench

The SoftBench environment, a product of Hewlett-Packard (HP), was expressly developed to provide an architecture for integrating CASE tools as part of HP's CASEdge initiative [8, 16]. SoftBench is based on a control, or process, approach to integration comprising three main functional components:

1. Tool Communication;
2. Distributed Support;
3. User Interface Management.

In this paper we shall concentrate on the first of those components, tool communication, which employs a message passing mechanism based on the one used in FIELD. Hence, the general approach used is as defined earlier for FIELD. In the description that follows we concentrate on the main differences from the FIELD approach.

4.2.1. The Message Server

In SoftBench the message server is known as the *Broadcast Message Server (BMS).* It is this component which forms the core of the SoftBench product. In most respects it is similar in operation to FIELD's *Msg*, with messages being received by the BMS for distribution to all tools that have registered interest in those messages. There are, however, the following points to note with the BMS:

- The BMS has the concept of a tool *protocol*. This is the abstract notion of a set of operations that are available for each class of tools. For example, the class of “debug” tools would have a protocol that included operations such as “step”, “set-breakpoint”, and ‘continue’. Any new tools which are added to the “debug” class must fully support the associated protocol. This ensures that a calling tool can rely on a client tool providing a well-defined set of operations without knowing which tool from the required class is invoked. This greatly increases tool independence, or “plug compatibility.”
- All SoftBench tools send a notification message whenever they perform an action. This approach allows triggers to be defined which are fired when notification of events is broadcast (e.g., system builds following file updates, or automatic collection of metrics data).
- Existing tools, not developed to use SoftBench, can be adapted for use with the SoftBench environment using HP’s *Encapsulator* tool. Through the Encapsulator the user develops an envelope within which the tool can execute and send/receive messages. The encapsulation is written in an Encapsulation Description Language (EDL) which acts as an interpreter for the tool.
- When executing, if a tool sends a message requesting a service which no currently executing tools can provide, the SoftBench Execution Manager will automatically look for a suitable tool and execute it. The request for the service is then forwarded to that tool.

4.2.2. Messages

In SoftBench messages are strings of text that follow a consistent format. In particular, there are three kinds of messages — request messages (R), success notification (N), and failure notification (F). Each has the following components:

- **Originator** — the tool that sent the message. This is left empty in SoftBench as all messages are broadcast to the BMS, so the originator is not required;
- **Request-id** — is a unique identifier constructed from the message number, process identifier, and host machine name;
- **Message Type** — one of R, N, or F;
- **Command Class** — is the class of tool (e.g., “debug” or “edit”);
- **Command Name** — the name of the operation or event;
- **Context** — provides the location of the data being processed. It is formed from the host machine name, the base directory, and the filename;
- **Arguments** — a list of arguments to the command.[3]

Hence, there is a single, well-defined format for all three types of SoftBench messages.

[3]Note that all data arguments are by reference to avoid having lots of data copied around in the messages.

4.2.3. Summary

The SoftBench environment is a very exciting recent development in the integrated CASE marketplace. Indeed, the product has already received enthusiastic support, with over 10,000 reported sales.

Based on the descriptions of SoftBench available we make the following critical observations:

- At least initially, SoftBench is a "program design, build, and test" system, *not* a complete software development environment. Part of the reason for this lies in the choice of tools that HP has made to integrate with SoftBench — program editor, static analyzer, debugger, program builder, and mail tool. However, it is interesting to postulate that there is a more fundamental reason than this. In fact, it may be the case that the approach works best, and tools are more readily integrated, when there is an obvious and clear relationship between the tools. The set of program development tools available fall into this category. It is not at all clear that integrating, say, technical and managerial tools, or documentation and development tools, would be nearly as "clean" or as "convenient". We discuss this point in more detail later in the paper.

- No details at all are given about the implementation of SoftBench. In particular, it would have been reassuring to have seen performance figures for the use of SoftBench. As nothing is mentioned, it must remain an open issue, particularly given Reiss's earlier comments regarding poor performance in FIELD.

- Almost a throw-away line in the description of SoftBench states:

  > ...one copy of the BMS executes to control the environment of each user. [16]

  If there is a BMS per user it is not clear how users communicate, or access tools being used by another user. The implication is that the way in which messages can pass between many users is if all the tools that users require are installed on a single host, with each user's desktop machine acting as the remote client for the tool's input and output display. The administrative burden of this arrangement given the large and complex nature of many existing CASE tools (e.g., a CASE tool may often require 32 Megabytes of RAM and over 100 Megabytes of disk space) may make it infeasible. Hence, multi-user collaborative development may be difficult in the current version of SoftBench.[4]

[4]The forthcoming release of SoftBench, version 3.0, is expected to address some of these multi-user problems.

4.3. ToolTalk

A recent offering from Sun Microsystems is the ToolTalk service, described as “a network spanning, interapplication message system” [11, 12, 13]. Initially, ToolTalk 1.0 is implemented on top of SunSoft’s ONC Remote Procedure Call (RPC) mechanism, and runs on SunOs 4.1.1 or later.

In abstract terms, ToolTalk shares many of the characteristics of the SoftBench product. Perhaps the most noticeable difference is the object-oriented emphasis that has been used in describing the ToolTalk service. For example, the ToolTalk service is said to manage “object descriptions”, the messages of ToolTalk are described as “object-oriented messages”, and one of the main advantages claimed for the ToolTalk service itself is that it provides both a solution to today’s integration problems, and a migration path to tomorrow’s object-oriented architectures.

4.3.1. The Message Server

The message server in ToolTalk is a special process called *TTsession*. Each user session has its own instance of the TTsession process.

Programs interact with the ToolTalk service by calling functions defined in the ToolTalk application programming interface (API). This allows applications to create, send, and receive ToolTalk messages.

4.3.2. Messages

In ToolTalk, the messages have a more complex format than either FIELD or SoftBench, and hence more information can be conveyed in them. Processes participate in message *protocols*, where a message protocol consists of a description of the set of messages that can be communicated between a group of processes, a definition of when those messages can be sent, and an explanation of what occurs when each message is received.

A message consists of a number of attributes. These are:

1. **An address.** This can be the address of a procedure, process, object, or object type. Thus, the receiver of a message can be of any of these types, providing a great deal of flexibility.

2. **A class.** There are two kinds of message class – notices and requests. A notice is a message which provides information about an event (such as start up of an editor, or termination of an editor), while a request is a call for some action to be taken (such as a build request). For a request message, the process making the request may continue while the request is handled (e.g., a window based application can continue to handle window events, mouse clicks, and so on), or may wait for an appropriate reply.

3. **An operation.** The identifier for the actual event that has occurred, or the requested action.

4. **A set of arguments.** Any parameters to the event or action are listed.

5. **An indication of scope.** Messages have a particular scope within which they are valid, limiting their potential distribution. The possible values for a message scope in ToolTalk are *session* (all processes with the current login session), *file* (a particular named file), *both* (the union of session and file), *file-in-session* (the intersection of session and file).

6. **A state.** Some messages are returned to the sender to indicate that the message server has, or has not, been able to find a recipient. Valid states of a message are *created*, *sent*, and *failed*.

Within the defined scope of a message, the receivers of that message are obtained by matching the message's attributes with the message patterns registered as being of interest to each of the processes (i.e., tools).

4.3.3. Summary

ToolTalk shares many of the characteristics of HP's SoftBench product, and it is difficult not to conclude that ToolTalk is Sun's reaction to the high level of interest that has been generated by SoftBench.

There are a number of observations that can be made about ToolTalk in comparison with the SoftBench product. We highlight the following:[5]

- There is no mention of any generally available *tools* that have been integrated with the ToolTalk services. While there is much discussion of the ease with which existing tools can be integrated through these services, no tools are actually named. The only example used [13] is an Electronic Design Automation (EDA) system. The tools integrated were very specialized and appear to have been integrated to support a fixed life cycle of tool interaction. The reason that little information on tools is given stems from a marketing decision that Sun have made to provide ToolTalk as a message passing layer that can be purchased in isolation from any tools. Separately available is an SDE called *SPARCWorks* — a set of tools that makes use of the ToolTalk product.

  The decision to market ToolTalk in this way may be a distinct advantage to Sun in the longer term, when customers have a better understanding of the ToolTalk product, and there are many tools and SDEs available to choose from which operate on ToolTalk. However, in the short term this decision may cause confusion and misunderstanding if application developers purchase ToolTalk only to find that no tools are provided with it to help evaluate and become familiar with the ToolTalk product.

[5] These observations are broadly consistent with a similar review carried out internally within Hewlett-Packard [19].

- The amount of work required to integrate a new tool with the ToolTalk service is also not discussed. There is no equivalent to SoftBench's Encapsulator, and Sun have not announced any plans to provide such a capability. As a result, to integrate tools into ToolTalk without amending the tools' source code it is necessary to write routines in the C (or C++) programming language. A wrapper for an existing tool would hence consist of the use of a graphical user interface generator such as Sun's DevGuide to allow a window-based interface for a tool to be constructed, and the necessary calls to ToolTalk using the Unix operating system calls of "fork" and "exec" with subsequent communication via Unix pipes.

  While it is claimed that knowledgeable Unix and C (or C++) programming personnel will find the writing of a tool wrapper for ToolTalk to be an easily managed task, the Encapsulator tool provided by SoftBench appears to be a way to make the task of tool encapsulation more accessible to SoftBench users, with the ability to write wrappers in the C (or C++) programming language if necessary.

- As with SoftBench, ToolTalk does not have a notion of groups of users collaborating on a project. Messages are sent to processes or are session-based. However, ToolTalk does have a way for users to collaborate through the notion of "file scoped messages."

  In file scoped messages a user specifies a file or directory that they are interested in. ToolTalk maintains a record of which users are interested in which scoped files. When a message is sent scoped to a file, ToolTalk forwards the message to all the user sessions interested in that file. A demonstration of multiple simultaneous editing of a shared file has been produced to illustrate the use of this scoped file concept.

5. Discussion

In this section we review a number of interesting points raised in the above descriptions and amplify some of the issues which were addressed. We first introduce a conceptual model which helps in understanding the three implementations described, then focus on some practical issues.

5.1. Conceptual Issues

It is tempting to view the three implementations of the message passing approach that we have examined as no more than three competing systems based on the same underlying principles. However, with hindsight, their differences can perhaps best be analyzed with reference to a conceptual framework developed in a previous paper [6]. Here, we briefly review that framework, and examine the relationship between the three implementations discussed in this paper in the light of that framework.

5.1.1. A Conceptual Framework

Analysis of existing tools and environments has led to a number of proposals describing a spectrum of levels of integration within an SDE. Each proposal can be considered a conceptual framework for analyzing the architecture of particular SDE implementations.

In the proposal by Brown and McDermid [6], five such levels for discussing tool integration were identified, as illustrated in Figure 5-1.

The five levels of the Brown/McDermid proposal were intended to be interpreted within the context of data sharing between tools in an SDE.[1] We can summarize the description of the levels as follows:

1. **Carrier Level** — Composing tools by enforcing a single, consistent file format.
2. **Lexical Level** — Sharing a common understanding of the lexical conventions of structures which are shared between tools.
3. **Syntactic Level** — Agreeing on a set of data structures, and rules for the formation of those structures, between a set of tools.
4. **Semantic Level** — Augmenting a common understanding of the data structures used by tools with a common definition of the semantics of those structures.
5. **Method Level** — Determining a common model of the software development process in which all tools are compatible.

[1] They provide a framework for discussing tool integration, *not* an architecture for implementing tool integration in an SDE.

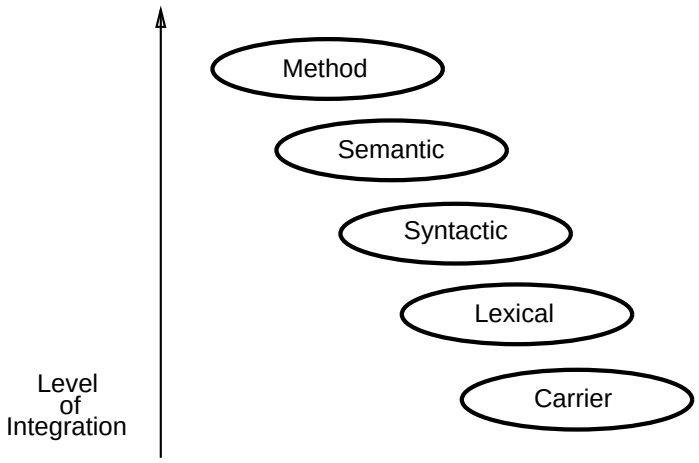

Figure 5-1 A Conceptual Framework for Analyzing an SDE

5.1.2. Applying the Conceptual Framework to the Message Passing Approach

We can now apply the five levels of the Brown/McDermid proposal to the message passing approach, with the three implementations addressed in this paper as examples. In this approach it is the *messages* that form the primary means of communication between tools and, hence, the information content of the messages that must be considered in analyzing different approaches to integration.

In their simplest form, messages are uninterpreted text strings. It is entirely the responsibility of the tools to agree on an interpretation of the messages. We could view this as *carrier level* integration between tools.

However, by building into groups of tools some common understanding of the data items (or, tokens) contained in the text strings, a *lexical level* integration can be established. All tools, together with the message server itself, know how to divide a message string into tokens such as identifiers, operation names, and so on.

Building on this level, a common syntax for the tokens leads to *syntactic level* integration. Not only can tools identify tokens in a message string, they now also have an agreed format for those tokens.

By agreeing on the meaning of the tokens in a message string, *semantic level* integration is achieved. At this level, not only has the identification of tokens been agreed, but their interpretation in terms of actions and events has also been agreed.

Finally, knowledge about the software development process in which the tools are participating leads to *method level* integration. The tools now have much more context in which to interpret the messages, having some knowledge of the operations and events that have preceded this message and those that are likely to follow.

Based on this analysis of the various levels of integration in the message passing approach,

we can assess the three implementations we have examined with regard to this classification:

- FIELD is an academic prototype system, with the goal of being a flexible, adaptable system for experimentation. As a result, messages are essentially uninterpreted strings of text, with agreement between tools necessary to interpret messages. The implementation of *Msg*, the message server, has a protocol for identifying the tokens of a message built-in. This places FIELD in the carrier level of this classification in concept, but the lexical level in practice.

- SoftBench is a commercial product with the aims of both standardizing an approach to tool integration with the SoftBench product, and to making integration of tools as straightforward as possible. Hence, SoftBench has included rules about the structure of a message in terms of the tokens and their ordering. In addition, the notion of a tool protocol has been introduced. This allows collections of tools to agree on a common set of services to allow users of those services to be independent of which actual tool implements the services at any particular time. This raises SoftBench to syntactic level integration, as the information communicated between tools is essentially through "typed messages."

- ToolTalk attempts to encode much more information in the messages it sends than either FIELD or SoftBench. It describes its messages as "object-oriented" because the ToolTalk service supports many message protocol styles, messages can be addressed to groups of tools, and message protocols can be inherited through a hierarchy relating objects in the ToolTalk service. We can see ToolTalk as an attempt at semantic level integration, based on the fact that knowledge of the message components themselves is shared between tools through inheritance.

In summary, we see that the three implementations of the message passing approach discussed in this paper can be distinguished by their differing approaches towards the information conveyed between tools in the messages transmitted. In fact, the implementations show a progression from lexical, to syntactic, to semantic levels of tool integration according to the Brown/McDermid classification.

It is interesting to speculate how the "next step" in this progression might take place — towards method level integration. In the context of the message passing approach discussed in this paper, method level integration can be interpreted as the encoding of *policy*, or *process* information within the message passing mechanisms themselves. In practice, this may mean that, in addition to tool protocols, policy protocols could be defined for a group of tools, describing the permitted interactions between tools, which sequences of messages encode a particular policy action, and so on. For example, in considering configuration management (CM) services within an SDE, a tool protocol may consist of a standard set of CM operations such as "check-in-version," "check-out-version," and "merge-versions." A number of individual CM tools may conform to this CM tool protocol by implementing those operations. A CM policy protocol, however, would encode particular uses of the CM tool protocol operations to support a particular CM process. Handling change requests, for instance, may be encoded as a CM policy by ensuring that a "check-in-version" operation is always preceded by a "QA-approval."

Such a CM policy enforces a particular use of the CM tool protocol. Further investigation of method level integration within the message passing approach to integration is under investigation [4].

5.2. Practical Issues

5.2.1. Extensibility

One of the major strengths of the message passing approach to integration appears to be its flexibility with regard to tool interactions. In particular, the use of the protocol concept as seen in the BMS of SoftBench leads to an easily extensible environment. For example, tools in execution do not need to be aware of exactly which other tools are running — they simply broadcast notification messages through the message server, or request a service from any tool in a particular tool class. Even if there is no tool of that class currently executing, the message server has enough information to be able to start up such a tool and forward the request (through maintaining an internal database of tools and classes).

Such extensibility is highly desirable in any system interested in event-based operation. The message passing approach appears to provide an ideal mechanism to support such a technique.

5.2.2. How Easy Is “Encapsulation”?

One problem that must be addressed by any SDE is the integration of existing third-party tools. Most of these tools were not designed and implemented with the SDE in mind, nor is the source code available to be able to amend them. The approach adopted by SoftBench and FIELD involves encapsulation. It is claimed for SoftBench that:

> Simple tool encapsulations can be described in two to five pages of code, which can be written in less than a mornings work... [16]

Clearly, further qualification of this statement is required to understand the work involved in tool encapsulation, but a typical encapsulation involves writing code to control four aspects of the tool — Input/Output streams, BMS messages, Operating System events, and X-Window events. Each of these aspects can be more or less complex depending on the tool to be encapsulated and the constraints on the desired results (e.g., attempting to produce a consistent graphical user interface across a set of tools).

The two major components of encapsulation in SoftBench, developing a graphical user interface and generating a message interface, are relatively independent. However, both may potentially involve significant amounts of effort, and may require a deep understanding of the tool to be encapsulated to achieve a reasonable result. More work is certainly needed to establish the ease and effectiveness of this form of encapsulation within a message passing mechanism. It is relatively straightforward to imagine the use of encapsulation for simple data

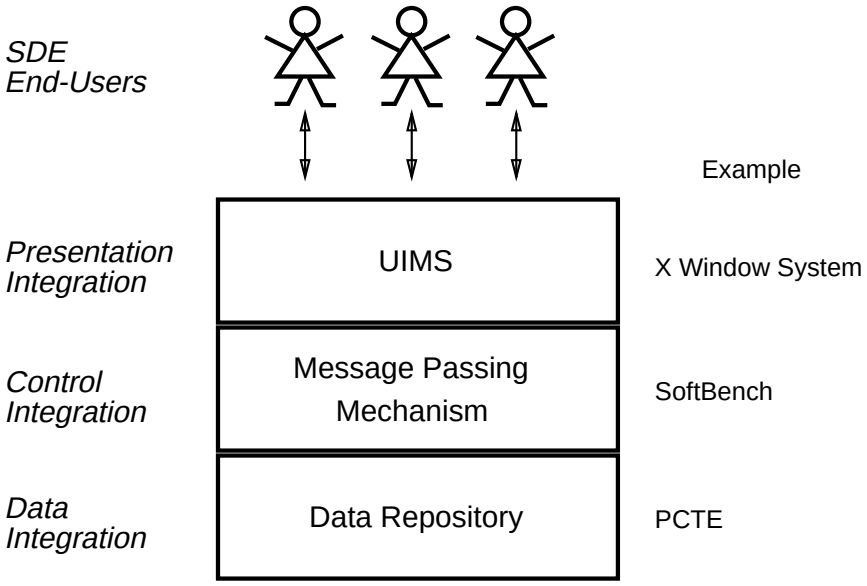

Figure 5-2 A Possible SDE Architecture?

producer/consumer tools such as simple UNIX utilities. However, for complex tools that have sophisticated interactive processing, or that assume an “egocentric” approach to integration and have been designed with little emphasis on open access to their internal services, it is much less obvious how useful this approach would be. Indeed, the SoftBench descriptions clearly state that encapsulation is intended for capturing the UNIX input and output actions of simple tools. There is a need for more work to establish a set of tool architecture classifications that relate to the ease (or otherwise) with which tools from each class can be integrated though an encapsulation approach.

Our own experiments with encapsulating tools with SoftBench have shown that for tools with relatively simple command line interfaces the Encapsulator provides a rapid way to produce a “point-and-press” interface for the tool. It is much more difficult, however, to design an appropriate message interface for a tool as it requires knowledge of both the other tools in the SDE and a well defined operational scenario in which the tool will operate [3].

5.2.3. Message Passing as Part of a Complete SDE

The use of message passing mechanisms as a component of a larger development environment is an area in need of exploration. Many people consider three aspects of integration when examining an SDE — data integration, control integration, and presentation integration. While message passing mechanisms provide control integration, it is intuitively appealing to envisage a data repository and User Interface Management System (UIMS) as providing the other two forms of integration. Hence, an SDE could be considered to have an architecture consisting of some combination of these three mechanisms, as illustrated in Figure 5-2. Taking this approach may provide the means to broaden the appeal of message passing as

a technique for integrating software development tools into a more general SDE context.

In this regard there are experiments currently taking place at HP's laboratories in the UK to re-implement the BMS of SoftBench on top of PCTE, taking advantage of the message queue facilities it provides [18]. Both SoftBench and PCTE provide X-Window System interfaces to their services to provide the user interface component of the architecture. A recent announcement by HP that they will support a PCTE-based implementation of SoftBench shows a high level of commitment to pursuing this approach towards an SDE.

Such an approach holds much promise with regard to providing a broad, flexible approach to tool integration. It will be interesting to monitor how the work develops.

5.2.4. Standard Message Protocols

The need for defining a common set of message protocols has been described earlier in this paper. In summary, we made the argument that syntactic and semantic levels of agreement between tools enhanced the quality of information that was transferred, and facilitated higher levels of integration. This argument can be made for tools from a single vendor within a message passing system, for tools from multiple vendors within a message passing system, and for tools from multiple vendors using multiple message passing systems. In particular, as message passing systems will be offered by a number of major suppliers[2] there is interest in ensuring that standards are developed that will allow third party tools to operate on different message passing systems, and to allow those message passing systems to communicate. In the past few months there have been three initiatives aimed at addressing this situation.

The Object Request Broker. As part of the work of the Object Management Group (OMG) they issued a request for proposals to specify an Object Request Broker (ORB). The ORB provides the mechanisms through which objects may transparently make requests and receive responses in an object management system. Having worked on separate proposals, a consortia consisting of DEC, HP, Hyperdesk Corporation, NCR Corporation, Object Design Inc., and SunSoft Inc. worked together to submit a joint proposal to the OMG [15]. This proposal, while still in draft form, has the potential to be an important route to providing open access to services provided by different tools using different object based message passing systems. It would be expected, for example, that an ORB would allow a request for an action from a tool working in a SoftBench environment to be serviced by a tool working in a ToolTalk system.

CASE Communique. A group led by HP, IBM, Informix, and CDC has held a number of meetings over the past year with the aim of eventually developing standards for CASE tool communication based on HP's SoftBench product [10]. This group, known as "CASE Communique," has recognized that if a common set of messages could be defined for each

[2] HP and Sun offerings have been described in this paper. Both IBM and Digital have control integration products. In 1990 Digital licensed FIELD and have productized and extended it under the name of FUSE. IBM recently licensed SoftBench technology from HP, and ported it to their RISC System/6000 under the name SDE WorkBench/6000.

class of CASE tool, then tools within a SoftBench environment would be more interchangeable within each class. For example, with an agreed set of messages for all CM tools, an editor tool could make calls to standard check-in and check-out operations with the knowledge that any CM tool in the environment would be able to act on those requests.

The CASE Interoperability Message Set. Digital, Silicon Graphics and SunSoft have produced a proposal for a set of message standards that describe semantic (not syntactic) sets of messages. The intention is that these message definitions be offered to one of the American National Standards Institute (ANSI) committees for standardization. The aim has been to provide a number of message sets for particular application areas and scenarios, with the encouragement to other vendors to help in refining those message sets and in proposing other such sets.

While these three initiatives are clearly in their infancy, the results from the work of these groups have the potential to aid tool writers in providing a set of messages to guide their implementation and tool users to allow greater choice over tools used in an environment. A further aim of these initiatives is to influence the direction of future development of message passing products. This should ensure that future versions of such products are more in tune with the needs of tool writers and tool users.

6. Summary

In this paper we have examined the message passing approach to tool integration as exemplified by the FIELD, SoftBench, and ToolTalk environments. Both FIELD and SoftBench have been very successful in practice — FIELD as a research vehicle that has stimulated a great deal of interest, and SoftBench as a product which is said to have sales of more than 10,000 seats. As the most recent product of the three, it remains to be seen how widely accepted the ToolTalk service will be within the large Sun user community. However, a number of issues and questions of the message passing approach have been raised. In particular, how much this approach is appropriate for *project* (as opposed to *programming*) support is a matter for debate and further investigation. Similarly, it is unclear whether the necessary syntactic and semantic agreements between tool vendors are yet in place to allow meaningful interaction between different tools.

While a number of open issues and shortcomings of the approach have been identified in this paper, there is clearly evidence to support further investigation of the message passing approach as the basis for providing an SDE architecture that is more open to the addition of new tools. In particular, the simplicity and flexibility that the approach provides appear to facilitate experimentation with different levels of integration between tools. As a result, it may well provide the ideal platform for experimenting in some of the most crucial aspects of SDE technology:

- **Tool Integration** — an examination of different semantic levels of tool integration in an SDE, how they can be supported in the same architecture, the relationships between the levels, the benefits and drawbacks of integrating tools at different levels, and so on.
- **Process vs. Data vs. User Interface integration** — an analysis of how independent (or otherwise) tool integration in each of these dimensions is in practice.
- **Open Tool Interfaces** — an opportunity to learn about encapsulation of existing tools into an SDE to provide knowledge about the ease of providing access to third party tools, the amount of work involved in such integration, and to determine the characteristics required of tools to ensure integration in an SDE is both practical and efficient.

We have also pointed towards possible future directions for the work on the message passing approach by describing the latest moves towards standardization of requests between object-based message systems and message protocols in the SoftBench product and through the application of the Brown/McDermid classification of tool integration levels in an SDE. Experience with current approaches to control integration in an SDE, coupled with experimentation leading to support for *method level* concepts within an SDE, appears to be an interesting direction for further work. It is our hope that in the near future we will examine each of these areas in more detail, focusing in on one aspect of tool support which pervades many areas of an SDE — configuration management.

Acknowledgements

My thanks to Peter Feiler at the Software Engineering Institute (SEI), Kurt Wallnau at PARAMAX, Ant Earl of Mark V Systems and Jack Bond of the U.S. Department of Defense for their comments on earlier drafts of this paper. Feedback from Richard McAllister of Sun on the ToolTalk product has also been very helpful.

References

[1] European Computer Manufacturers Association. A Reference Model for Computer-Assisted Software Engineering Environments. *ECMA Report Number TR/55*, January 1991.

[2] A.W. Brown. *Database Support for Software Engineering*. Chapman and Hall, London, England, 1990.

[3] A.W. Brown, W.M. Caldwell, F.W. Long, E.J. Morris, and P.F. Zarrella. Experiences with a Federated Environment Testbed. Technical Report In Press, Software Engineering Institute, Pittsbugh, PA, 1992.

[4] A.W. Brown, S.A. Dart, P.H. Feiler, and K.C. Wallnau. The State of Automated Configuration Management. In *Annual Technical Review*, Pittsbugh, PA, 1992. Software Engineering Institute.

[5] A.W. Brown, P.H. Feiler, and K.C. Wallnau. Understanding Integration in a Software Development Environment. In *Proceedings of the 2nd International Conference on Systems Integration*, pages 22–31, Morristown, NJ, June 1992. IEEE.

[6] A.W. Brown and J.A. McDermid. On Integration and Reuse in a Software Development Environment. In F. Long and M. Tedd, editors, *Software Engineering Environments '91*, Chichester, England, 1991. Ellis Horwood.

[7] J.N. Buxton and L. Druffel. Requirements for an Ada Programming Support Environment: Rationale for Stoneman. In *Proceedings of IEEE Conference on Computer Software and Applications (COMPSAC 80)*, Chicago, IL, October 1980.

[8] M.R. Cagan. The HP SoftBench Environment: An Architecture for a New Generation of Software Tools. *Hewlett-Packard Journal*, pages 36–47, June 1990.

[9] EIA. CDIF: Organization and Procedure Manual. *Report Number EIA/PN-2329*, January 1990.

[10] H. Fischer. Notes from CASE Communique Meeting, 17th October 1991.

[11] R. Frankel. *Introduction to the ToolTalk Service*. Sun Microsystems Inc., Mountain View, CA, 1991.

[12] R. Frankel. *The ToolTalk Service*. Sun Microsystems Inc., Mountain View, CA, 1991.

[13] R. Frankel. *ToolTalk in Electronic Design Automation*. Sun Microsystems Inc., Mountain View, CA, 1991.

[14] F. Gallo, R. Minot, and I. Thomas. The Object Management System of PCTE as a Software Engineering Database Management System. *Proceedings of 2nd SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments*, pages 12–15, December 1986.

[15] Object Management Group. The Common Object Request Broker: Architecture and Specification. *Draft 91.8.1*, 26th August 1991.

[16] R. Ison. An Experimental Ada Programming Support Environment in the HP CASEdge Integration Framework. In F. Long, editor, *Software Engineering Environments*, number 467 in Lecture Notes in Computer Science, pages 179–193, Berlin, Germany, 1990. Springer-Verlag.

[17] P.A. Oberndorf. The Common Ada Programming Support Environment (APSE) Interface Set (CAIS). *IEEE Transactions on Software Engineering*, 14(6):742–748, June 1988.

[18] H. Oliver. Adding Control Integration to PCTE. In A. Enders and H. Weber, editors, *Software Development Environments and CASE Technology*, number 509 in Lecture Notes in Computer Science, pages 69–80, Berlin, Germany, 1991. Springer-Verlag.

[19] H. Oliver. Private Communication, September 1991.

[20] S.P. Reiss. Connecting Tools Using Message Passing in the FIELD Environment. *IEEE Software*, pages 57–99, June 1990.

[21] S.P. Reiss. Interacting with the FIELD Environment. *Software – Practice and Experience*, 20(S1):S1/89–S1/115, June 1990.

[22] I. Thomas and B. Nejmah. Tool Integration in a Software Engineering Environments. Technical Report SESD-91-11 Revision 1.1, Hewlett-Packard, Software Engineering Systems Division, Sunnyvale, CA, June 1991.

[23] A. Wasserman. Tool Integration in Software Engineering Environments. In F. Long, editor, *Software Engineering Environments*, number 467 in Lecture Notes in Computer Science, pages 138–150, Berlin, Germany, 1990. Springer-Verlag.