Technical Report
CMU/SEI-94-TR-17
ESC-TR-94-017

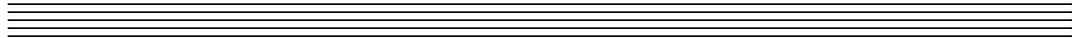
Replacing the Message Service Component in an Integration Framework

Paul F. Zarrella
Alan W. Brown

September 1994

Technical Report
CMU/SEI-94-TR-17
ESC-TR-94-017
September 1994

Replacing the Message Service Component in an Integration Framework



Paul F. Zarrella
Alan W. Brown

CASE Environments Project

Unlimited distribution subject to the copyright.

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.

Copyright © 1994 by Carnegie Mellon University.

Permission to reproduce this document and to prepare derivative works from this document for internal use is granted, provided the copyright and "No Warranty" statements are included with all reproductions and derivative works.

Requests for permission to reproduce this document or to prepare derivative works of this document for external and commercial use should be addressed to the SEI Licensing Agent.

NO WARRANTY

THIS CARNEGIE MELLON UNIVERSITY AND SOFTWARE ENGINEERING INSTITUTE MATERIAL IS FURNISHED ON AN "AS-IS" BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTIBILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.

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.

Use of any trademarks in this report is not intended in any way to infringe on the rights of the trademark holder.

Table of Contents

1 Introduction **1**

2 Background **5**

3 Message Server Overview **7**
3.1 BMS 7
3.2 ToolTalk 7

4 Getting Started **9**
4.1 Experiments with Encapsulator and BMS 9
4.2 Writing the Emulation Code 9
4.3 Developing “Support Utilities” 10

5 Adding the ToolTalk Interface **11**
5.1 Learning How to Use ToolTalk 11
5.2 Emulating BMS with ToolTalk 11
5.3 Adding ToolTalk to the Emulation Framework 13

6 Running the Experiment Scenario **15**
6.1 Modifying the EDL Scripts 15
6.2 Physical Placement of Tools and Support Software 15
6.3 Limitations of the Emulation 15

7 Replacing ToolTalk in the Emulation Framework **17**
7.1 FUSE 17
7.2 Use of FUSE in the Emulation Framework 17

8 Lessons Learned **19**
8.1 Time Requirements 19
8.2 GUI Screen Builder 19
8.3 Programming Requirements 19
8.4 Documentation Limitations 19
8.5 Applicability of ToolTalk 20

9 Summary and Conclusions **21**

10 Possible Future Work **23**

References **25**

Appendix A ToolTalk Interface Code Segments **27**
A.1 Initialization of Interface to ToolTalk Message Service 27
A.2 ToolTalk Message Pattern Creation 27
A.3 ToolTalk Message Pattern Registration 28
A.4 ToolTalk Message Pattern Deletion 28
A.5 ToolTalk Message Pattern Destruction 28
A.6 ToolTalk Message Creation and Transmittal 28
A.7 ToolTalk Message Acknowledgment and Acceptance 28
A.8 ToolTalk Message Data Argument Extraction 29
A.9 Disconnection of Emulation from ToolTalk 29

List of Figures

Figure 2-1:	Tool and Framework Structure	5
Figure 2-2:	Message Service Interface Structure	6
Figure 5-1:	BMS and ToolTalk Message Delivery	11

Replacing the Message Service Component in an Integration Framework

Abstract: In an on-going set of commercial off-the-shelf (COTS) tool integration experiments being conducted by the CASE Environments Project, we have integrated a set of CASE tools using a combination of data integration mechanisms (PCTE Object Management System (OMS) and UNIX file system) and control integration mechanisms (Broadcast Message Server (BMS) of HP SoftBench). One of the key issues addressed in our work is the extent to which the integration of CASE tools can be independent of particular integration framework technology products.

This report describes a task to examine interoperability aspects of the control integration component of the integration framework. The major conclusion from our work is that it is possible to integrate CASE tools using a message-passing approach that is independent of the integration framework product used. This report describes the activities an organization must undertake to integrate CASE tools in order to ensure this interoperation of message-passing integration products. The report also includes a set of lessons learned concerning the experiments we carried out.

1 Introduction

Over the course of the past year, the SEI CASE Environments Project has been performing a series of experiments designed to examine issues surrounding the question of what computer-aided software engineering (CASE) tool integrations are possible for third-party tool users, given the current state of commercial off-the-shelf (COTS) tools and integration technology. These experiments involve the integration of a collection of common COTS tools with environment framework technologies in support of a typical software maintenance scenario. A review of the first phase of the experiments can be found in [Brown 93].

In the experiment scenario, control integration (i.e., the synchronization and coordination of CASE tools) was accomplished through inter-tool communication via messages. In this approach, tools interact by passing messages (through a message system) that request services of each other and notify each other of their actions. This eliminates the need to duplicate functionality between tools and the need to coordinate/operate via a shared tool database. Many influential vendors have considered this approach to be a basis for control integration in CASE environment applications. For example, a message-passing approach to inter-tool communication is one of the key aspects of the Common Open Software Environment (COSE) alliance formed by Hewlett-Packard, Sun Microsystems, IBM, Unix Systems Laboratories (USL), The Santa Cruz Operation (SCO), and Univel [X/Open 93].

In fact, a number of products are already available that embody message-passing concepts. HP, for example, already uses the "control integration via messaging" approach in their SoftBench framework product [Cagan 90]. Therefore, we utilized the message-passing capabilities of SoftBench (particularly, the SoftBench Encapsulator and Broadcast Message Server (BMS)) as the primary means of tool-activity synchronization in the original experiment scenario.

While analyzing the results of the initial set of experiments, we determined that the use of BMS as the message passing component of the experiment framework could (at least in principle) be replaced by an equivalent product such as Sun's ToolTalk [Cureton 93] or DEC's FUSE [DEC 93]. This is significant in that one of the premises of the work of the SEI CASE Environments Project, and of the experiments in particular, is that different implementations of similar services can be easily interchanged to provide a degree of interoperability. The ability to interchange control-oriented integration framework mechanisms is an important part of that premise. This ability for tools to interoperate over multiple integration framework products is an essential feature of CASE tool integration that we wished to explore.

While we could have hypothesized about the feasibility of the message-service replacement by comparing and contrasting the services and their inherent functionality, our idea was to demonstrate the practicality of the operation by example. We undertook an extension to the experiment then, not to see if the message service replacement was theoretically possible, but to examine the process of actually performing (or attempting to perform) the replacement. In addition, we wanted to see if it was realistically possible to determine if one message service was "more applicable" to a specific framework/scenario than the other.

In our initial set of experiments, we concentrated on the examination of messaging systems as the basis of control integration within a development framework. With this experiment, we expected to find that message-server replacement, even with the stipulation that it be (for all practical purposes) transparent to the framework/scenario, was ultimately achievable. We also expected to find that although these different message services provide different functional capabilities in support of control integration, the choice of message service is basically application independent.

This report is presented as an introduction to the task of determining whether a message-passing-based integration framework would be better purchased for use in a CASE environment, or built using existing message-service components. The report outlines the types of activities that precede development of a specific message-based integration framework (i.e., one that emulates SoftBench Encapsulator), the considerations involved in choosing a specific messaging interface for a set of tools, and the effort involved in adapting that message interface to the framework.

The report begins with a background to the message-server replacement experiment and an overview of the BMS and ToolTalk message formats. It then discusses the development of an Encapsulator emulation for use with ToolTalk, the addition of the ToolTalk interface to the emulation framework, the execution of the original experiment scenario under the emulation, and

the replacement of ToolTalk in the emulation framework. The report concludes with a presentation of lessons learned, a summary of the experiment, and consideration of possible areas for follow-on work/experimentation.

2 Background

According to the original experiment scenario, the tools used were to provide support for a typical software maintenance function. In order to complete the scenario we used coding, metrics, testing, message server, and CM tools, along with a process control tool to manage execution of the scenario. The scenario was initially implemented around BMS as the integration framework, with PCTE as the CM data repository. In order to utilize the capabilities of PCTE, a simple file transfer mechanism was developed between PCTE and the UNIX file system. Figure 2-1 identifies the tools and framework components of the scenario.

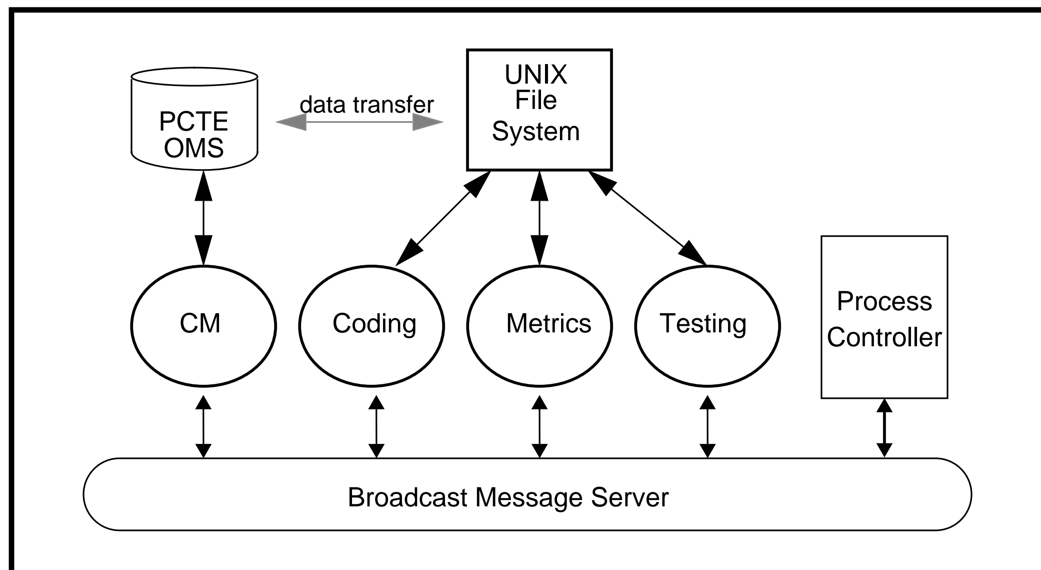


Figure 2-1: Tool and Framework Structure

In this experiment, we decided to use ToolTalk as the alternate message service as it was readily available to us (ToolTalk is available as part of Sun OpenWindows Version 3). Our first inclination was to modify the SoftBench Encapsulator to use ToolTalk instead of BMS, but we were unable to obtain the SoftBench source code. We decided instead that we would have to “reinvent” some portion of SoftBench in order to support the experiment. We also realized that the SoftBench utilities of the Encapsulator would have to be replaced by other technologies. ToolTalk does not provide the support for user interface generation available via SoftBench’s Encapsulator. ToolTalk also does not provide ready-made integrations to tools such as editors, compilers, and SCCS. These features were used heavily in our integration experiments.

Due to our limited resources, we decided to emulate only the C language interface functions of SoftBench Encapsulator for use with ToolTalk. This would allow us to focus on the integration mechanisms and would eliminate the need to restructure/redesign the process scenario. It also allowed us to use the same Encapsulator source code “scripts” without requiring devel-

opment of a script interpreter (as in Encapsulator). Again, our intent was to make the emulation and message-service replacements completely transparent to the original experiment scenario. This emulation approach is illustrated in Figure 2-2.

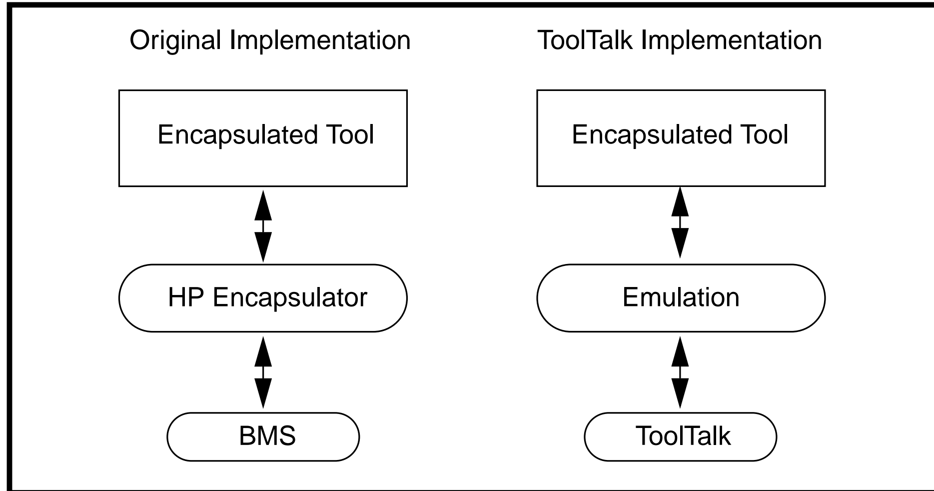


Figure 2-2: Message Service Interface Structure

It should be noted here that this experiment was not intended as an evaluation of ToolTalk, nor as a lesson in "redevelopment" of SoftBench. As such, we did not find it necessary to use all of the features of ToolTalk (e.g., object-oriented messaging, static message patterns), although we did attempt to provide a rough equivalent of Encapsulator/BMS to the original process scenario. It should also be noted that while the experiment employed framework components that are equivalent to those being integrated by the COSE alliance (i.e., SoftBench and ToolTalk), the experiment was intended to be generic in purpose, and the lessons learned from the experiment are nonetheless valid.

3 Message Server Overview

In order to be compatible with the previously developed integration scenario, the experiment attempted to emulate the messaging actions of BMS via ToolTalk. The following overview of the two corresponding message formats is presented in consideration of that effort. For an expanded description of BMS and ToolTalk, and a discussion of their functions as the message-passing component within a software development environment, see [Brown 92].

3.1 BMS

In SoftBench, the message server is known as the Broadcast Message Server (BMS). It is a component which forms the core of the SoftBench product. Messages received by BMS are distributed to all tools in the session that have registered an interest in those 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:

Sender — the name of the tool that sent the message

Message-id — a unique identifier constructed from the message number, process identifier, and host machine name

Message-type — either Request (R), Notify (N), or Failure (F)

Tool-class — the type classification of the tool that sent the message

Command — the name of the operation or event

Context — the "working area" or location of the data being processed (formed from the host machine name, base directory, and filename)

Data — a list of arguments to the command

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

3.2 ToolTalk

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.

In ToolTalk, the messages have a more complex format than SoftBench, and hence more information can be conveyed in them. Processes participate in message *protocols*, each of which 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:

Address — this identifies the potential message handlers, and can be one of *procedure* (any interested process), *handler* (a specific process), *object* (a specific object), or *object type* (a general object specification).

Class — this is the message type, and can be a *notice* (a message which provides information about an event) or a *request* (a call for some action, with the possibility of a reply).

Operation — this is the identifier for the actual event that has occurred, or the requested action.

Arguments — this is the list of parameters to the event or action.

Scope — this limits the distribution area of messages, and can be one of *session* (any process within the current login session), *file* (a particular named file), *both* (the union of session and file), or *file-in-session* (the intersection of session and file).

Disposition — this tells ToolTalk what to do with the message if a handler cannot be found (default is to discard the message), and can be one of *queue* (hold the message until a handler is started), *start* (start a process to handle the message), or *queue+start* (queue the message and attempt to start a handler process).

State — this is the state of the message as it makes the delivery circuit, and can be one of *created* (message has been created but not yet sent), *sent* (sent but not yet handled), *handled* (message has been replied to), *failed* (no handler is available or all handlers have rejected the message), *queued* (message has been queued for later delivery), *started* (a process is being started by ToolTalk to handle the message), or *rejected* (message has been rejected by a handler).

Within the defined scope of a message, the receivers of that message are obtained by matching the message attributes with the message patterns registered as being of interest to each of the processes.

4 Getting Started

4.1 Experiments with Encapsulator and BMS

In preparation for development of a framework to emulate Encapsulator/BMS, more specific information was needed concerning the functioning of encapsulated applications and of BMS. Therefore, the emulation was based not only on information provided in the *SoftBench Encapsulator: Programmer's Guide* and *SoftBench Encapsulator: Programmer's Reference* manuals from Hewlett-Packard, but on observed performance of the Encapsulator (Version A.02) as well.

Several experiments were performed to determine the appropriate functions of the encapsulated subprocess under control, the syntax and resulting format for context specifications, and the proper matching and handling of BMS messages. In addition, many other "throw-away" experiments were developed on an as-needed basis to determine the appropriate actions of the emulation under specific conditions which could arise in use (e.g., Encapsulator responses to unexpected NULL pointers, BMS handling of specific message pattern wildcards, the format of message identifiers).

Later in the development of the emulation, experiments into the workings of the user interface were performed. Again, it was intended here to provide a reasonable approximation of the Encapsulator user interface, not to identically reproduce it. However, some experimentation was necessary to determine appropriate actions based on specification of user "events", user responses to interface selections, etc.

4.2 Writing the Emulation Code

Development time encompassed coding of all the C language bindings (library routines) per those defined in the Encapsulator, including emulation of the subprocess control functions. All of the library routines were implemented, even though the experimental scenario written for Encapsulator did not make use of all of the functions available. The reasoning here was that any extensions/changes to the Encapsulator version of the scenario could be readily reflected in the emulation version.

The library routines were developed in three phases. First, the utility and subprocess routines were written to provide the "framework" for the remainder of the emulation. This included support for the basic Encapsulator data types ("string", "boolean", "event", and "integer"), and utilities to accept and handle "Application" and "System" events. Next, the message server interface and context-specific routines were written, as these were the basis of the experiment. This included all of the support necessary for "Message" events. Finally, the user interface routines were written, including support for "User" events, and the associated "object" and "attribute" data types.

The code was initially written with a “debug” interface so that the functions could be verified as they were implemented without requiring the message server and Motif user interface services. In addition, the message server interface code was developed so that specific interfaces to other message services could be (theoretically) easily added at a later date. After development of the basic portions of the emulation was completed, a Motif Interface (for “User” events) was added partially as an exercise in completing the framework, but primarily so that the scenario could be run identically as it was when using the Encapsulator.

Due to a single-user/single-system limitation imposed in the original experiment scenario, and to the time constraints of the experiment, no remote host processing capabilities were incorporated into the emulation. Also, some user-interface-specific attributes of Encapsulator (e.g., background/foreground colors, edit source type) were not implemented (because they were not used in the scenario) and, again, were deemed to be beyond the scope of the experiment. However, some other attributes which were not used in the scenario (e.g., object mapping, row/column size designation) were supported as they were easily derived from the addition of the Motif user interface.

4.3 Developing “Support Utilities”

Since we were emulating only a small portion of the SoftBench framework, some consideration had to be made as to the extent that other facilities of SoftBench would have to be incorporated in order to support the emulation. While many of the tools provided with SoftBench would not be needed, or could be substituted for, two facilities were thought to be important enough to be included as part of the emulation.

SoftBench provides a message monitor/send facility which is useful primarily for debug purposes. Such a tool was also incorporated into our experiment extension in order to facilitate testing of the interface to the message system. Although Sun includes the source for a ToolTalk message monitoring tool as a demo within the OpenWindows installation hierarchy, it was not functionally equivalent to the tool desired (e.g., the demo tool does not allow specification/display of message data arguments), and the user interfaces and message monitoring output were not compatible. As the demo tool was written for the “XView” windowing interface, it seemed easier to develop a tool specifically for the X/Motif interface (and built upon the emulation framework) than to modify the tool supplied by Sun to fit our needs.

SoftBench also contains an Execution Manager component which provides the ability to start a tool upon demand from another tool via a message request through BMS. ToolTalk provides a similar “auto-start” facility via static message pattern definition (i.e. matching message types are predefined). However, due to the dynamic messaging model of the emulation (see Section 5.2), we could not make use of the ToolTalk facility. Therefore, a tool server utility was developed for use in our experiment to provide equivalent functionality as that of SoftBench (although tangential to the scenario).

5 Adding the ToolTalk Interface

5.1 Learning How to Use ToolTalk

Programming with the ToolTalk message interface was basically a self-taught undertaking. This was accomplished primarily via reference to the documents *ToolTalk 1.0 Programmer's Guide*, and *Application Integration with ToolTalk — A Tutorial*, both of which are provided by Sun Microsystems. The tutorial provided a basic example that was used as a starting point for interface code development. Our goal at this stage of the experiment was to add a straightforward interface to ToolTalk without any bells and whistles. At this point, some basic experiments were conducted with ToolTalk (via the Application Programming Interface (API)) in order to determine the most appropriate messaging model for use in the emulation.

5.2 Emulating BMS with ToolTalk

While attempting to emulate BMS with ToolTalk, we found several limiting factors related to message delivery (see Figure 5-1). All BMS messages ("request", "notify", or "failure") are handled on a "one-to-many" basis. That is, BMS makes any message equally available to any interested tool, for as many message events (i.e., message patterns) as are matched. On the other hand, ToolTalk limits tool access by the message class ("request" or "notice"), and further by handler response to request messages ("reply", "reject", or "fail").

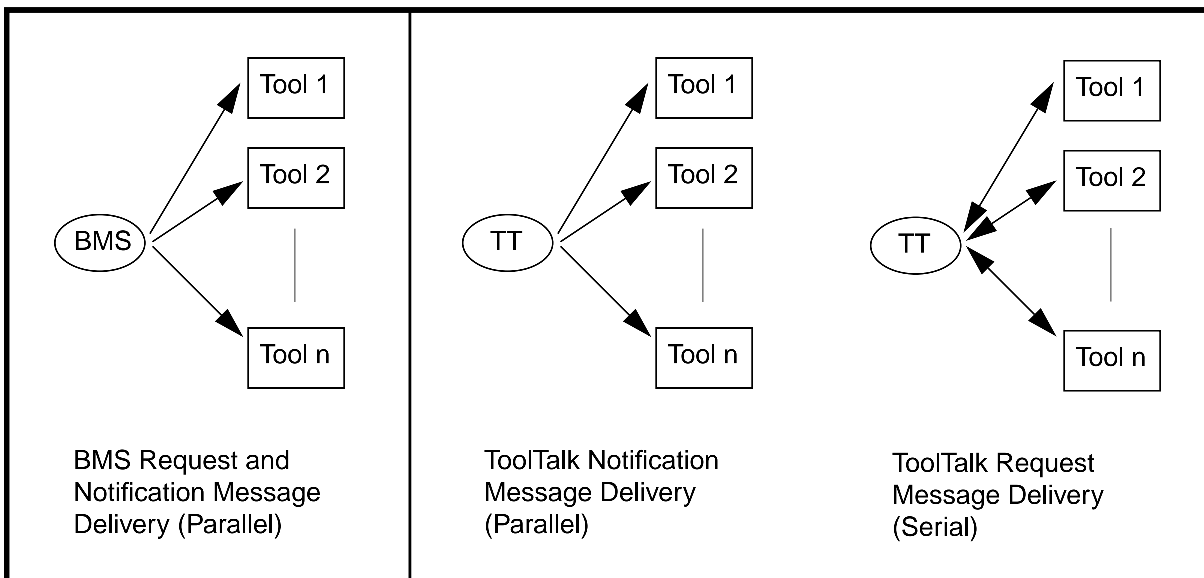


Figure 5-1: BMS and ToolTalk Message Delivery

ToolTalk request messages are sent to one handler at a time until the request has been satisfied (either positively or negatively). A specific request message will not be sent to a handler more than once, regardless of the number of message patterns that it matches for that handler. In contrast, ToolTalk notification messages are sent to all tools, and may be sent to the same tool multiple times depending on the specifics of the registered message patterns.

In order to make all ToolTalk messages available to all interested tools (exactly once), to allow the tool server utility to "hold" messages for tools that it is in the process of starting, and to accommodate wildcard message classes, we used a simple model wherein all messages were sent as requests, and each message handler released each message to the next interested tool by rejecting the message after it had examined and processed it. We later extended the model to include notification messages, and utilized a single "exact" message pattern for each user-specified notify/failure message event (to eliminate multiple message delivery).

In addition, since the message-pattern-matching capabilities of BMS and ToolTalk are not identical, and we were attempting to emulate the BMS characteristics, some of the finer-grained pattern matching function (most specifically relating to context and "wildcard" considerations) was performed by the emulation as opposed to ToolTalk.

This was achieved by having the emulation dynamically define a minimal pattern for message send/receive consisting of (basically) a generic identifier for the tool set. When sending a message, the context attributes (e.g., host, directory, file) were attached as the first 'n' data elements of the message pattern (where 'n' was constant), followed by any additional user-defined data elements. After receiving a message which met the ToolTalk pattern matching requirements (i.e., the generic tool identifier), the emulation would examine the data elements to further determine a contextual pattern match. The emulation would simply reject any message that did not meet the expanded match characteristics. This did not seem to have an adverse performance impact on the emulation.

It should be noted here that the ToolTalk message patterns could have been (and ultimately were) expanded to limit message reception based on the specific message context. However, even with the more exact receiver pattern match, the emulation still had to examine the context attributes of the message to determine which internal message event(s) had been matched and to perform the appropriate "callback" processing. ToolTalk pattern callbacks could not be used for this purpose, as they would not provide the same function as BMS for multiple message patterns that matched on the same request message (due to the "single request message delivery per handler" attribute of ToolTalk).

One other consideration when developing the emulation was the issue of "unhandled" messages. Unhandled messages may be caused by any of three conditions: a ToolTalk selected handler "blocking" message events, not having yet entered the event loop (via a "start" or "restart" command), or simply being otherwise busy (and therefore unable to handle the message). Since ToolTalk delivers request messages in a serialized fashion, a handler can block other handlers from receiving a message until it unblocks and/or handles the message (or exits). While the documentation would indicate that ToolTalk might conceivably time-out in this

situation and offer the message to another handler, limited experimentation did not identify the time-out interval. There seemed to be no way to work around this limitation in the emulation, although it did not present itself as a problem in our scenario.

5.3 Adding ToolTalk to the Emulation Framework

Once the emulation framework was developed, and the ToolTalk experimentation completed, the addition of ToolTalk to the emulation as the message server was, in itself, actually quite easy. Addition of the ToolTalk interface required the addition of less than 100 lines of code to the emulation (for comparison, there was a total of about 5000 lines of source in the entire emulation). The ToolTalk interface code developed included support for:

- initialization of the interface to the ToolTalk message server;
- message pattern creation, registration, and deletion/destruction;
- message creation and transmittal;
- message acknowledgment and acceptance (based on context);
- message data argument extraction;
- controlled disconnection from the ToolTalk message server.

C language code segments for these operations can be found in Appendix A.

6 Running the Experiment Scenario

6.1 Modifying the EDL Scripts

The scenario was originally coded for the SoftBench Encapsulator in Encapsulator Definition Language (EDL). EDL is similar in syntax to C, so each of the EDL-based encapsulations used in the scenario was easily rewritten in C. Some changes had to be made specifically to turn the EDL scripts into working C programs (e.g., creating a "main" routine, calling an initialization routine), but no more than would have been required to modify the scripts to run with the Encapsulator C language interface (provided with SoftBench).

6.2 Physical Placement of Tools and Support Software

One of the first difficulties encountered with running the scenario was the problem of having different software tools physically located on different machines. ToolTalk is included as part of the Sun OpenWindows Version 3.0 (or greater) installation and only runs on SunOS version 4.1.x, a combination of which was running on only one machine in our network. Meanwhile, PCTE, the coding tools, and the testing tools were licensed only on a different machine. As it turned out, the version of OpenWindows on the "license" machine could be upgraded without having to re-install the entire system (possibly requiring updated versions of some or all of the COTS tools used in the scenario). Although we were able to elude such configuration problems, it does point to a potential problem in the general case.

6.3 Limitations of the Emulation

As previously mentioned, the issue of replacing SoftBench supplied support utilities was considered before attempting to fully implement the emulation. The scenario made use of the SoftBench "softeditsrv" and "softbuild" tools as a visual editor and compiler, respectively. As it was deemed to be beyond the scope of the experiment to provide an interface to these specific tools, simple "encapsulations" were developed to the X11 "xedit" (in place of "softeditsrv") and UNIX "cc" (in place of "softbuild") utilities. Identical message interfaces (at least to the extent of that required in the scenario) were incorporated into these encapsulations so that no changes to the scenario were required.

In addition, the server process could not be utilized in the scenario due to a (since fixed) problem encountered in ToolTalk with respect to the mode in which it was used within the emulation (i.e., rejected messages were not being delivered to other matching handlers). Therefore, all of the processes involved in the scenario had to be started in advance and would then wait for message "instructions". This removed the "on demand" process starting capability of the scenario (available with SoftBench). It did not, however, change the scenario itself nor did it require any changes to the tool encapsulations (other than to remain installed throughout the duration of the scenario).

7 Replacing ToolTalk in the Emulation Framework

As indicated previously, the emulation was written so as to make it possible to replace the interface to ToolTalk with that of another message service. When access to DEC FUSE became available, an extension to the message-service-replacement experiment was subsequently conducted.

7.1 FUSE

In FUSE, the message server is accessed through a set of programming facilities called FUSE EnCASE. Like BMS, messages received by FUSE are distributed to all tools in the session that have registered interest in those messages.

FUSE EnCASE employs a Tool Integration Language (TIL) to specify the attributes of the messages that a tool sends and receives, and stores that information in a schema file for use by the message server. The language has the following components:

- Class — the name of the tool.
- Attributes — a specification of the tool for use by the FUSE EnCASE Control Panel and the FUSE message server, which contains such information as tool label, tool pathname, and tool grouping characteristics (“local” or “global” scope of message exchange).
- Messages — a list of names and response, parameter, and return-value types for each message type that the tool can send and receive.
- States — a specification of the tool state names, and the types of messages (both predefined FUSE message types, and those from the “Messages” list) that can be sent and received by the tool in each state.

7.2 Use of FUSE in the Emulation Framework

Adding the FUSE interface to the emulation as the callable interface and associated documentation was quite straightforward. For purposes of the experiment, the documentation used was the “DEC FUSE Reference Manual” and “DEC FUSE EnCASE Manual” (based on FUSE version 1.2) from DEC.

We chose a simple model for use with FUSE which utilized a single message-type consisting of a simple character string parameter (the BMS-type message pattern). Much of the original debug interface code of the emulation doubled as “support” code for the FUSE interface, or was modified slightly to also serve as such. Only about 20 lines of C language code were written specifically to provide the actual emulation interface to FUSE/EnCASE.

The only problem encountered in the FUSE version of the emulation was similar to that of the “unhandled messages” problem encountered with ToolTalk. In the case of FUSE, however, messages sent while message events were not being handled were simply lost, and no coding

work-around could be determined. The only tool involved in the scenario that this affected was the CM tool, which blocks messages during initialization. This process was simply pre-started, and allowed to initialize in advance of running the scenario.

8 Lessons Learned

8.1 Time Requirements

The task took approximately three staff-months to complete. This included the time needed to learn some of the more “esoteric” features of the Encapsulator/BMS and the time to learn how to program with ToolTalk and with the Xt Intrinsics package of Motif. While we had anticipated reasonably well the amount of effort required to complete the emulation, more time was expended experimenting with SoftBench/BMS than was expected, while less time was expended learning the ToolTalk interface than was expected.

Also, while the Motif-based user interface of the emulation “approximates” that of Encapsulator, it was not fully debugged as part of the emulation effort, and some simplifying changes were made to the mode of operation. These changes were user related, and none of them affected the actual scenario.

8.2 GUI Screen Builder

Access to a graphical user interface (GUI) screen builder would have made creation of the user interface for the emulation much easier. However, as this was a secondary part of the emulation, it was decided that we would spend the limited time required to learn enough of the X/Motif programming interface to support the emulationrather then purchasing such a tool. As such, we made our own “build vs. buy” decision at this point.

8.3 Programming Requirements

The amount of coding required to support the experiment, given the time involved, required a reasonable level of proficiency in C (or C++, or whatever language would ultimately be chosen). In addition, the timing of the experiment required a fairly steep learning curve for Motif (X Windows) interface programming. In contrast, development of the ToolTalk interface code was fairly straightforward once the task of learning the ToolTalk interface was completed.

8.4 Documentation Limitations

While the task of incorporating ToolTalk into the emulation would have been impossible without the documentation provided, the documentation did prove to be on a lesser level than would have been desired. While much of the information presented related to ToolTalk from a conceptual standpoint or at the “overview” level for programming considerations, the programming examples provided were limited, and the mechanics of operation were presented without enough practical application information (e.g., little mention of default settings, little discussion/explanation of message/pattern attributes and settings other than those that are required).

8.5 Applicability of ToolTalk

In this experiment, the capabilities and usage of ToolTalk were limited by the constraints of the framework into which it was being added (i.e., the Encapsulator/BMS emulation). Throughout both experimentation and documentation review, it seemed that it would have been easier (and more efficient) to fit ToolTalk it into a general application framework instead of having to emulate a specific one (SoftBench/BMS). In addition, the former case would have allowed for better use of the capabilities of ToolTalk by designing the framework and message interface specifically for that purpose. Many of the decisions made in development of the interface to ToolTalk were dictated solely by the attempt to make ToolTalk emulate the function of BMS.

9 Summary and Conclusions

As indicated in the introduction, two results were expected from this experiment: first, that the message service replacement could actually be reasonably performed, and second, that the choice of message servers was mostly independent of the application for which it provided the control integration mechanism.

The experiment showed that replacement of the message service in a controlled experiment framework is quite possible. As outlined in the report, however, such an undertaking does have preconditions to success. For example, a substantial amount of groundwork had to be performed in order to begin the message-service replacement portion of the task. Also, in our situation we were more interested in the capability and applicability aspects than in the specific level of effort expended or in the "product worthiness" of the end result.

These factors all contribute to the consideration of the level of priority that an organization must place on development time versus cost of purchase (i.e., the decision of "build vs. buy"). A product like SoftBench is relatively expensive, but requires little in the way of engineering resources to be fully utilized. On the other hand, while ToolTalk is "free" (as part of SunOS), it requires a significant investment in development of an integration framework and associated support tools.

In addition, as discussed in the "Lessons Learned" section, we discovered that replacement of the message service is not entirely independent of the framework/application. While there would seem to be less of a dependency when initially defining/selecting a service specifically for an application, there are compatibility issues to be considered (and handled) when attempting to replace the service in an existing framework where messaging characteristics (e.g., types, formats, handling) are already defined. Without messaging standards, it would appear that "plug-and-play" message service components would be impossible.

One final consideration is that when an organization decides to build their own framework (or modify a purchased one), they assume all responsibility for future extensions/compatibility considerations (e.g., adding support tools, resolving problems, incorporating new versions of tools/framework components). On the other hand, the organization also maintains control of the integration and can make changes as they see fit in order to tailor the system to their needs.

10 Possible Future Work

While the experiment proved that message service replacement in the framework could be done, there are other aspects of the experiment that could be examined. These include:

- *Incorporate object-oriented capabilities provided by ToolTalk into the emulation.* It would be interesting to see how the support framework (and the experiment scenario) might be changed with the addition of object-oriented messaging. In the “process-oriented” messaging model used in the experiment, messages are directed to process(es) for handling. In an “object-oriented” messaging model, messages are directed to objects (data) instead of processes. In this model, ToolTalk determines the handler process based on preregistered object-process addressing rules.
- *Investigate adding other support tools with ToolTalk interface to the scenario.* As the scenario changes, it might be interesting to see if new tools can also be added to the emulation version. In addition, it might be interesting to see if any new COTS tools that employ a ToolTalk message interface can be integrated into the scenario (or possibly integrated via the emulation).
- *Add interface to another message server.* It would be interesting to incorporate another message service interface into the emulation (e.g., one based on the Common Object Request Broker Architecture (CORBA) [OMG 92]). This type of experiment should again be fairly straightforward as the emulation framework has already been completed, and the interface insertion points have already been identified. What is left is to emulate BMS according to the capabilities of the alternate message service. Along these lines, it would be interesting to obtain the SoftBench BMS interface library for use by the emulation.

References

[Brown 92] Brown, A.W. *Control Integration Through Message Passing in a Software Development Environment*. Software Engineering Institute Technical Report, CMU/SEI-92-TR-35, ADA259853, December 1992.

[Brown 93] Brown, A.W., Morris, E.J., Zarrella, P.F., Long, F.W., & Caldwell, W.M. "Experiences with a Federated Environment Testbed." *Proceedings of the Fourth European Software Engineering Conference,* Garmisch-Partenkirchen, Germany, September 13-17, 1993.

[Cagan 90] Cagan, M.R. "The H.P. SoftBench Environment: An Architecture for a New Generation of Software Tools." *Hewlett-Packard Journal*. 41(3), June 1990.

[Cureton 93] Cureton, B. (Ed.) "Software Engineering on Sun Workstations." New York: Springer-Verlag, 1993.

[DEC 93] *DEC FUSE Reference Manual*. Digital Equipment Corporation, 1993.

[OMG 92] *The Common Object Request Broker: Architecture and Specification (CORBA)*. Object Management Group, Inc., 492 Old Connecticut Path, Framingham, MA 01701 USA, 1992.

[X/Open 93] *Common Desktop Environment: Functional Specification*. X/Open Company Limited (for Hewlett Packard, IBM, Sunsoft, Inc., USL), Apex Plaza, Forbury Road, Berkshire, RG11AX, United Kingdom, 1993.

Appendix A ToolTalk Interface Code Segments

The following code segments represent the C language code developed for the interface to the ToolTalk message service in the Encapsulator emulation. Note that error checking/handling on ToolTalk API calls, and other non-ToolTalk specific code has been deleted for brevity.

A.1 Initialization of Interface to ToolTalk Message Service

A.1.1 routine: init()

```
mark = tt_mark();
tt_open();
session_id = tt_default_session();
tt_session_join(session_id);
msg_fd = tt_fd();
```

A.2 ToolTalk Message Pattern Creation

A.2.1 routine: make_message_pattern()

```
event->tt_pattern = tt_pattern_create();
tt_pattern_category_set(event->tt_pattern, TT_HANDLE);
tt_pattern_class_add(event->tt_pattern, TT_REQUEST);
tt_pattern_scope_add(event->tt_pattern, TT_SESSION);
tt_pattern_session_add(event->tt_pattern, session_id);
tt_pattern_op_add(event->tt_pattern, "INCUBATOR");
tt_pattern_arg_add(event->tt_pattern, TT_IN, "string",
   equal_string(msg_tool, "*") ? NULL : msg_tool);
tt_pattern_arg_add(event->tt_pattern, TT_IN, "string",
   equal_string(msg_id, "*") ? NULL : msg_id);
tt_pattern_arg_add(event->tt_pattern, TT_IN, "string",
   equal_string(msg_type, "*") ? NULL : msg_type);
tt_pattern_arg_add(event->tt_pattern, TT_IN, "string",
   equal_string(msg_class, "*") ? NULL : msg_class);
tt_pattern_arg_add(event->tt_pattern, TT_IN, "string",
   equal_string(msg_command, "*") ? NULL : msg_command);
tt_pattern_arg_add(event->tt_pattern, TT_IN, "string",
   equal_string(msg_host, "*") ? NULL : msg_host);
tt_pattern_arg_add(event->tt_pattern, TT_IN, "string",
   equal_string(msg_directory, "*") ? NULL : msg_directory);
tt_pattern_arg_add(event->tt_pattern, TT_IN, "string",
   equal_string(msg_file, "*") ? NULL : msg_file);
while ((msg_arg = get_next_pattern_data_arg()) != NULL)
   tt_pattern_arg_add(event->tt_pattern, TT_IN, "string",
      equal_string(msg_arg, "*") ? NULL : msg_arg);
```

A.3 ToolTalk Message Pattern Registration

A.3.1 routine: add_event()

```
if (event->type == Message && event->tt_pattern != NULL)
   tt_pattern_register(event->tt_pattern);
```

A.4 ToolTalk Message Pattern Deletion

A.4.1 routine: remove_event()

```
if (event->type == Message && event->tt_pattern != NULL)
   tt_pattern_unregister(evemt->tt_pattern);
```

A.5 ToolTalk Message Pattern Destruction

A.5.1 routine: free_event()

```
if (event->type == Message && event->tt_pattern != NULL)
   tt_free(event->tt_pattern);
```

A.6 ToolTalk Message Creation and Transmittal

A.6.1 routine: send_message()

```
tt_msg = tt_message_create();
tt_message_class_set(tt_msg, TT_REQUEST);
tt_message_address_set(tt_msg, TT_PROCEDURE);
tt_message_scope_set(tt_msg, TT_SESSION);
tt_message_session_set(tt_msg, session_id);
tt_message_op_set(tt_msg, "INCUBATOR");
tt_message_arg_add(tt_msg, TT_IN, "string", tool_name);
tt_message_arg_add(tt_msg, TT_IN, "string", msg_id);
tt_message_arg_add(tt_msg, TT_IN, "string", msg_type);
tt_message_arg_add(tt_msg, TT_IN, "string", msg_class);
tt_message_arg_add(tt_msg, TT_IN, "string", msg_command);
tt_message_arg_add(tt_msg, TT_IN, "string", context_host);
tt_message_arg_add(tt_msg, TT_IN, "string", context_directory);
tt_message_arg_add(tt_msg, TT_IN, "string", context_file);
while ((msg_arg = get_next_message_data_arg()) != NULL)
   tt_message_arg_add(tt_msg, TT_IN, "string", msg_arg);
tt_message_send(tt_msg);
tt_message_destroy(tt_msg);
tt_free(tt_msg);
```

A.7 ToolTalk Message Acknowledgment and Acceptance

A.7.1 routine: handle_message_event()

```
tt_msg = tt_message_receive();
```

```
if (tt_msg != NULL && tt_message_state(tt_msg) == TT_SENT) {
   matched = check_extended_pattern_match(tt_msg);
   tt_message_reject(tt_msg);
   if (matched)
      process_message(tt_msg);
}
tt_message_destroy(tt_msg);
tt_free(tt_msg);
```

A.8 ToolTalk Message Data Argument Extraction

A.8.1 routines: message_class(), message_command(), message_directory(), message_file(), message_host(), message_id(), message_tool(), message_type()

```
data = copy(arg = tt_message_arg_val(tt_msg, offset));
tt_free(arg);
```

where *offset* is the offset (from ‘0’) of the specific message context attribute in question (i.e. class, command, directory, etc.). See Section A.6.1 for the order of attribute placement (as the initial data arguments) in the message.

A.8.2 routine: message_data()

```
num_args = tt_message_args_count(tt_msg);
mark = tt_mark();
if (n != 0)
   data = copy(tt_message_arg_val(tt_msg, n+DATA_OFFSET));
else  for (i = DATA_OFFSET; i < num_args; i++) {
         arg = tt_message_arg_val(tt_msg, i);
         data = append_arg_to_string(data, arg);
      }
tt_release(mark);
```

where *n* is the index of the data argument requested (‘n=0’ selects all data arguments) and *DATA_OFFSET* is the offset (from ‘0’) of the first non-contextual data argument in the message. In the current version of the emulation, DATA_OFFSET is ‘8’ (see Section A.6.1).

A.9 Disconnection of Emulation from ToolTalk

A.9.1 routine: finish()

```
tt_release(mark);
tt_close();
```

where *mark* is the ToolTalk API stack position marker (obtained in Section A.1.1).

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS None
2a. SECURITY CLASSIFICATION AUTHORITY N/A 2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for Public Release Distribution Unlimited
4. PERFORMING ORGANIZATION REPORT NUMBER(S) CMU/SEI-94-TR-17	5. MONITORING ORGANIZATION REPORT NUMBER(S) ESC-TR-94-017
6a. NAME OF PERFORMING ORGANIZATION Software Engineering Institute 6b. OFFICE SYMBOL (if applicable) SEI	7a. NAME OF MONITORING ORGANIZATION SEI Joint Program Office
6c. ADDRESS (city, state, and zip code) Carnegie Mellon University Pittsburgh PA 15213	7b. ADDRESS (city, state, and zip code) HQ ESC/ENS 5 Eglin Street Hanscom AFB, MA 01731-2116
8a. NAME OFFUNDING/SPONSORING ORGANIZATION SEI Joint Program Office 8b. OFFICE SYMBOL (if applicable) ESC/ENS	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F1962890C0003
8c. ADDRESS (city, state, and zip code)) Carnegie Mellon University Pittsburgh PA 15213	10. SOURCE OF FUNDING NOS. PROGRAM ELEMENT NO: 63756E PROJECT NO.: N/A TASK NO: N/A WORK UNIT NO.: N/A

11. TITLE (Include Security Classification)
Replacing the Message Service Component in an Integration Framework

12. PERSONAL AUTHOR(S)
Paul F. Zarrella and Alan W. Brown

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (year, month, day)	15. PAGE COUNT
Final	FROM TO	October 1994	30 pp.

16. SUPPLEMENTARY NOTATION

17. COSATI CODES		
FIELD	GROUP	SUB. GR.

18. SUBJECT TERMS (continue on reverse of necessary and identify by block number)
Computer-Aided Software Engineering (CASE)
Integration
Software Engineering Environment (SEE)

19. ABSTRACT (continue on reverse if necessary and identify by block number)

In an on-going set of commercial off-the-shelf (COTS) tool integration experiments being conducted by the CASE Environments Project, we have integrated a set of CASE tools using a combination of data integration mechanisms (PCTE Object Management System (QMS) and UNIX file system) and control integration mechanisms (Broadcast Message Server (BMS) of HP SoftBench). One of the key issues addressed in our work is the extent to which the integration of CASE tools can be independent of particular integration framework technology products.

This report describes a task to examine interoperability aspects of the control integration component of the integration framework. The major conclusion from our work is that it is possible to integrate

(please turn over)

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT UNCLASSIFIED/UNLIMITED ■ SAME AS RPT ☐ DTIC USERS ■	21. ABSTRACT SECURITY CLASSIFICATION Unclassified, Unlimited Distribution
22a. NAME OF RESPONSIBLE INDIVIDUAL Thomas R. Miller, Lt Col, USAF	22b. TELEPHONE NUMBER (include area code) (412) 268-7631 22c. OFFICE SYMBOL ESC/ENS (SEI)

ABSTRACT — continued from page one, block 19

CASE tools using a message-passing approach that is independent of the integration framework product used. This report describes the activities an organization must undertake to integrate CASE tools in order to ensure this interopration of message-passing integration products. The report also includes a set of lessons learned concerning the experiments we carried out.