y 2

Interconnection of software tools or applications, considered as black boxes

Adolfo Guzmán and Weiping Yin

ABSTRACT. With so many software tools available today for increasing the productivity of the programmer, there is a frequent need to integrate, interconnect or make collaborate two or more tools that, when developed, had no idea that they would be talking or connected to each other much later. To compound the problem, these tools most likely were developed by different manufacturers, or in some other form there is a resistance to share or publish source code for such tools; only object code is available. The problem to be solved is how to make such "black box" tools collaborate in new desired ways, *without having to change the code* of either tool.

Some current and new (proposed) methods for solving this interconnection problem (which is similar for tools and for applications) are given and discussed, as well as when it is convenient to make some of these tools act as providers of a service ("servers"). Certain methods rely on some degree of knowledge (at tool design time) of the probable ways to interconnect the tool in the future. This paper focuses in tools written in the same language for the same machine, but some extensions to alleviate this restriction are mentioned. Examples of the methods are given. The paper also contains some implications for tool builders that wish to integrate their tools via these methods with other yet-unknown tools.

Keywords: software engineering; tool interconnection; tool integration; software engineering environment; SEE; IPSE; applications interaction.

I The problem to solve

Tools to develop software, and in general, to help in applications programming, significantly increase the productivity of programmers and users. Unfortunately, most of these tools: (1) are designed to work in a stand-alone mode, interacting perhaps with a proprietary database or with the operating system; (2) *are available only in object code*, the source being kept unavailable by the original developer. On the other hand, there are demands for tools' collaboration: (a) complex problems use a multiplicity of tools, so that their interaction becomes necessary or natural; (b) methods† for software development require the developer to follow certain sequence of steps (hence, of needed tools); thus, the tools need to interact (be supervised) by another tool: the software engineering environment; (c) in general, a given tool is used only in a part of the development: an interpreter is used in the initial steps of fast prototyping methods; later, it is displaced by a compiler. Tools come and go. This calls for dynamic tool integration.

The problem to be addressed is: how can two (or more) tools (or applications) be made to interact, when the code of the tools can not be changed, but more code can be written around them?

1•1 Integration, collaboration and use

Some definitions follow.

Tool T_1 ***communicates*** with T_2 when T_1 provides some information to T_2. "T_1 communicates with T_2" does not necessarily imply "T_2 communicates with T_1."

Presumably, T_1 provides the information in order to request from T_2 some action or in some manner modify its behavior. Thus,

T_1 ***uses*** T_2 when T_1 communicates with T_2 in order to obtain some information or service from T_2, or to modify T_2's behavior or state. "T_1 uses T_2" does not imply necessarily "T_2 uses T_1."

Connects, ***Interconnects***, *Collaborates* or ***Integrates*** are often used in place of *uses*; they are less desirable terms since they tend to imply a commutative relation, while *uses* does not.

T_1 is a ***clear box*** if its easy‡ to modify its code. This generally means that its source code is available. An non modifiable tool is a ***black box***. There could be "gray" or "almost black" tools.

The use of T_2 by T_1 has the purpose of accomplishing some goal or service S, which neither T_2 nor T_1 can accomplish by itself. It requires in general some additional code and modifications to T_1 and T_2. If both T_1 and T_2 are black boxes, their modification is impossible; service S can only be achieved by means of additional code. This is called *glue code* or ***integration code*** when conceived as located "between" T_1 and T_2, and is called ***encapsulation code*** when it is thought to be "around" the used tool, T_2 (which is now encapsulated).

†. "Methods" is used to refer to a series of steps or algorithms; "Methodology" is the science that studies methods. In U.S.A. there is a tendency to use "Methodology" when "methods" is intended.

‡. For instance, if its inner workings, functions, arguments, etc., and the purpose of these, are known, so that its modification or detailed understanding is straightforward.

Presented at he *Fourth International Symp. on Artificial Intelligence*, Nov. 1991; Cancún, Mexico

A. Guzmán is with *Int'l Software Systems, Inc.* 9430 Research Blvd., Austin, Tx. guzman%issi.uucp@cs.utexas.edu; W. Yin is with *HaL Computers, Inc.* 8920 Business Park. Austin, Tx. 78759

Fifth International Symposium on Artificial Intelligence. Cancún. 1991

Artículo 82

1•1•1 Tool versus application

A piece of software intended for developers' use is a ***tool***; intended for end-users' use, is an ***application***. The boundary is blurred: there are "tools" like grep that are employed by programmers and end users. This distinction is not necessary here and the paper will speak of integrating tools, meaning in general tools, applications and any piece of software.

1•1•2 Tight integration versus tight coupling

- A system is ***tightly coupled*** to another when it can observe and react to changes in the state of the other quite soon after the changes occurred. The degree of coupling considers the control sequence of the tools; how tools call each other, and how they interact to achieve an expanded role. This will be the main subject of this paper.
- Two tools exchanging objects of small size are ***tightly integrated***. The degree of integration considers the kind and "level" of information they share. Information can be shared/interchanged:
  (1) At the character and gesture entry level, where two tools share the same sequence of input keystrokes and mouse clicks;
  (2) At the file level, when they operate on a common set of files; often, but not necessarily, these files contain text (ascii characters);
  (3) At the database interface level, when tools use a common notation or a common set of routines to obtain access to artifacts in the database; but it is transparent for these notation/routines how the information is actually stored in the database. Database independence is achieved, since the same notation† or set of routines‡ can access artifacts stored differently in different brands of databases.
  (4) At the database level, when tools use a unique representation (of the application objects) in secondary memory [but not necessarily in main memory],
  (5) At the memory level, where tools use the same representation (data structures) in main memory [generally implying integration at the database level, too]. Some workstations (Sun) possess *shared memory* to store the shared data.
  (1) and (2) are considered loose integration; (5) is tight integration. These forms of integrations are discussed elsewhere [8], but see also §3•1•1 below. See Figure 1•
- *Single-language environments* are used to develop the tools in a common programming language. This makes possible common in-memory structures (tight integration) and source code tool modification (to be seen in §2•1) (tight coupling). Often, the tools are developed concurrently .

1•2 Role of a software engineering environment

Some tools are integrated not with each other but into an integrating software, which acts as central or sole integrator. This integrating software is called Software Engineering Environment (SEE) or Integrated Project Support Environment (IPSE)□. For the purposes of this article, the integration of a tool into an IPSE is no different that the integration of a tool to another; hence, the IPSE is considered as still another tool: an "integration tool."

Example 1: Atherton's Software Backplane [6] is an IPSE.

II Current solutions

2•1 Integration of a tool from a clear box

The simplest way for tool T_1 to use T_2 is to modify T_1; thus, T_1 should be a clear box. T_2 does not need to be. The integration code gets placed inside T_1, and is being built with T_2 in view. This allows for tight integration or coupling. T_1 knows when to use which parts of T_2. Most of the tools developed by the same group and not intended to be used stand-alone are integrated this way.

Example 2: A compiler using its parser.

Usually, medium to tight integration (§1•1•2, (4), (5)) is done from clear boxes.

2•1•1 Pros and cons of modifying a tool to use another

✓ Being able to make modifications to the source code for T_1 makes the integration of T_2 simple. The use of parts of T_2, if needed, is straightforward.

✗ Source code, if purchased, costs more than object code, in part because code of clear boxes could be stolen or plagiarized.ø

---

†. For instance, SQL, Standard Query Language.

‡. Example: create_loan, destroy_loan, update_loan, save_loan, etc., used by a mortgage company to transfer to/from memory loans from/to an abstract database (a "loan server"), which could be implemented using different vendor databases for different sites or computers, say.

□. Some people use IPSE for an environment devoid of tools; a SEE is an IPSE populated with tools.

ø. The Free Software Foundation gives away (free) source code; most of their tools, therefore, can be tightly coupled with other tools and with programs being developed.

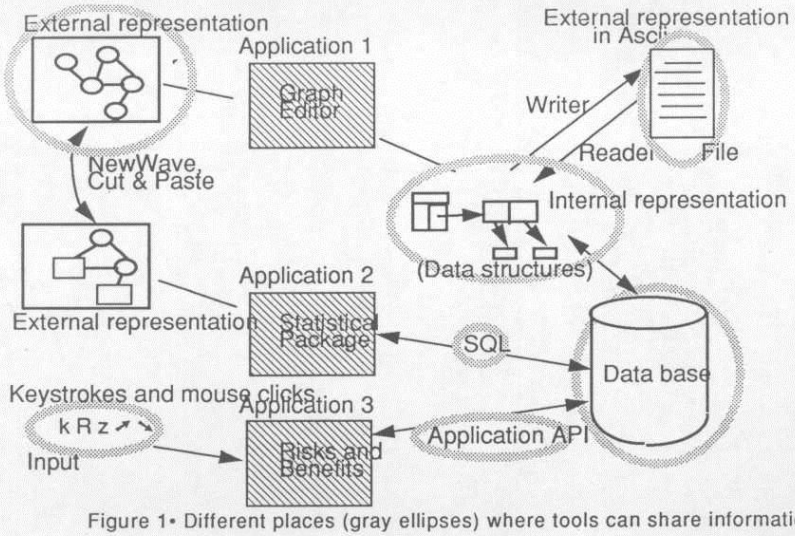

Figure 1• Different places (gray ellipses) where tools can share information.

If, in addition, T_2 is also a clear box, it can be modified to further suit the needs of T_1, if needed.

Example 3: The Proto interpreter using its graphic editor [1], [2].

✗ Careless modification of T_2 may cause errors in other programs using the non modified version, and will increase the difficulties for version control and configuration management of those systems depending on T_2.

☞ This method (modification of T_1, or clear box integration) should be used whenever possible.†

2•2 Loose coupling: Integrating a tool at both ends

A black box T_1 can use T_2 via some code that performs some work before communicating with T_2 (for instance, computing appropriate arguments for it), and some more work after T_2 finishes, such as giving to T_1 part of the values returned by T_2. Thus, T_2 is "tied" at its beginning and at its end. T_1 can not use the integration code (because T_1 can not be modified), so that the integration code ends up using T_1 and T_2. T_1 and T_2 have a more symmetrical role than in §2•1. Each call to each tool can have a preamble and a postamble, which *encapsulate* (cf. §1•1) it.

T_1 is loosely coupled to T_2, since the time between T_1 requesting some service from T_2 and T_2's response may be arbitrarily large; T_1 is not immediately aware of changes in T_2's state.

†. The hand ☞ points to considerations for tool builders that desire to integrate their tools via the methods discussed in this article.

Example 4: A user interface calls (Figure 2•) a matrix inversion package and sends electronic mail m ="job finished" to the user. The UI can also call email directly.

The code for the UI (the integration code) could be:

```
(Button "INVERT"):
  get numeric file f;   -- MATRIX preamble
  result = MATRIX (f);
                --prepare messg m with results r;
                    --MATRIX postamble
  EMAIL (m);
  return;              /* return to UI */
(Button "MAIL"):
  EMAIL ( );
  return;              /* return to UI */
```

Example 5: Atherton Software Backplane [6] uses this form of tool coupling; no two tools could call each other directly; each tool gets called by the Backplane (Figure 3•)

A tool attached at both ends to an IPSE has long periods of time (from start execution to end of execution) during which it interacts with nobody. Its coupling with any other tool is quite loose. In many cases, more frequent interactions are often needed, and the next sections address this.

2•2•1 Pros and cons of integration at both ends

✓ Simple to accomplish.

✓ Works for completely black tools.

✗ Limited in scope: more general ways of integration, tight coupling among tools, interactively sharing data, is not possible.

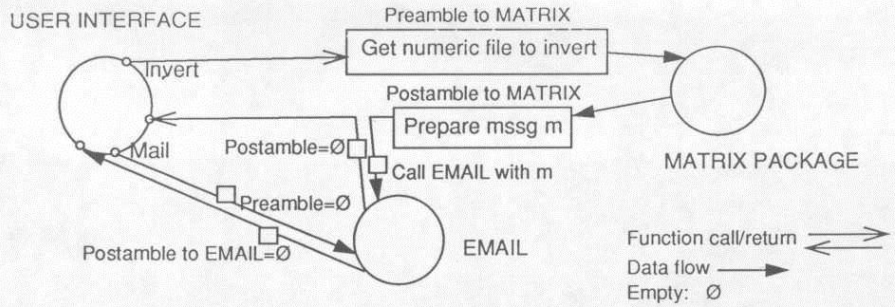

Figure 2• The Matrix Package and EMAIL were integrated into the UI through a preamble and a postamble.

☞ Simplest form of black box tool integration.

2•3 Intercepting calls meant for somebody else

Often, loose coupling like that of the preceding section is not enough: T_1 needs to know what T_2 is doing, specially if it deleted some data structure or file. T_1 needs to know the state of T_2 more frequently. But, how can T_1 read or sense more frequently (and react to) the state of T_2, *if it does not know that T_2 exists?* T_1 was written without this sensing capability. And, even if it knew the state of T_2, how could T_1 react to this knowledge? It was written without this reaction capability, and it can not be rewritten (no source code). The situation seems hopeless.

Fortunately, there exists a neat trick of great help here, which depends on the ability to substitute certain functions that T_1 uses. It is through changes to these functions† that T_1 can use T_2 in a tighter coupling manner.

But these functions are black boxes, too! How can they be modified? The answer: With a preamble and a postamble (§2•2 "Loose coupling: Integrating a tool at both ends").

The first step is to identify these functions. These functions relate to events or actions that are about to occur (performed by said functions), that may be important for other tools to be aware of. The second step is to modify them in a special manner: whatever they were doing before T_1's integration [that is the "important action" for which they are responsible], they should keep doing it‡, unless the new tool {T_1, glue code, T_2} is meant to do something different. Also, they need to react to T_2's state. For this purpose, glue code is placed around each of these functions.

Conceptually, a subroutine sub gets replaced by a new subroutine with the same old name sub, which does something additional: sense and react to the state of T_2. The new sub can also modify the behavior of T_1, by returning a modified value to it, or by performing an action that the old sub did not perform, if that action is the correct or sensible action to take given the state of T_2.

Example 6: The way the Sun Network File System [7] handles files over the network. Before NFS existed, the calls from an application program to a file function (file_open, say)

†. These functions will be named s_1, s_2,...; they are subroutines that T_1 calls; they can also be (§2•4) message servers to be sent messages from T_1. They will be called "internal API's" of T_1 (§3•1•1).

‡. Because T_1 is still expecting from them their previous behavior.

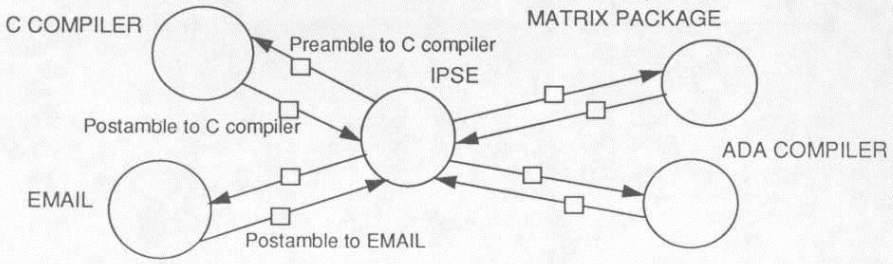

Figure 3• Black-box tools integrated to an IPSE. The particularities of the application (i. e., the specific manners in which the tools collaborate) are written "inside" the IPSE, which contains the integration code.

would start a function that opens a file in the caller's workstation. When NFS came, all it had to do was to "intercept" the calls to open_file and other file-handling functions, and then do some extra work: check where the file resides; send messages across the network and get the file; or, if the file is local, invoke the old open_file. Thus, the application program is interacting with NFS, without knowing it (Figure 4•). The application program is thus treated as a black box: no source code for it is needed.

Example 7: (a familiar case) A program for computing a matrix's eigenvalues uses a subroutine sqrt(x). This subroutine can be replaced by *another* sqrt function with, say, increased accuracy. Calls to the old sqrt (in some library L_1) are diverted to the new sqrt: same name, but in L_2.

2•3•1 Pros and cons of call interception

✓ The interceptor provides an improved service, which subsumes (extends) the former service. The application program need not change, nor the replaced functions, such as the old [local] file functions in Figure 4• All programs, new and old, automatically get the improved service.

✓ Fast. No interpretation of calls.

✗ Rigid. To connect T_1 to T_2 and T_3 (Figure 7•), requires first replacing a function s_1 by s_2 = {new function that connects to T_2, perhaps calling s_2} and then replacing s_2 by s_3 = {new function that connects to T_3, perhaps calling s_2}. Nevertheless, this is conceptually simple: s_2 can be regarded as an encapsulated s_1, and s_3 as an encapsulated s_2, from the point of view of T.

✗ Requires that the loader does not get confused by the presence of two subroutines (one old and one new) with the same name. This may require to use the dynamic option for the loader and link to shared libraries.

2•4 Tool integration via a broadcast message server

The methods discussed above also apply to tools that exchange messages. In the method discussed now, all the messages issued by T_1 to its servers s_1, s_2,..., are now routed† to a central message table or message server, accomplished by a *broadcast message server* in SoftBench [3], [4], or an inter-application message service in Sun's ToolTalk [9]. In this server, messages can be routed not only to the original functions, but to new ones of which the sender is unaware (Figure 5•). Each tool defines a set of messages that inform of important events or actions that such tool feels other tools may want to know. Also, each tool defines (to the message server) a set of messages that it wants to hear. The message table is called *blackboard* in Artificial Intelligence.

Figure 5• shows tools either taking notice of the message or ignoring it. The message server simply sends the message of a given row to all the tools wanting it (✓). The tool sending the message does not know what other tools will react to it. It only expects certain behavior as a consequence of the message sent. It is the responsibility of the message server that such expected behavior is indeed fulfilled for each message sent [that is, for each row of Figure 5•], regardless of the combination of ✓ and ✗'s in the row. With this kind of table, the coupling of the tools may be dynamically altered: for instance, changing to ✗ all the marks in the column of a given tool effectively decouples it from the system.

In this paradigm, T_1 does not talk to other tools: it does not know their names. T_1 just "talks." Precisely who is listening and reacting to its messages is not its problem. T_1 is analogous to a queen who does not address her butler or cooker "I would like to eat." She just utters "I want food x" and food appears, brought by some food servers that could change depending on the day of the week or the kind of food x is. Also, T_1 need not send messages for everything it needs done; it could just call a simple subroutine without the message server knowing it. Conversely, T_1 can issue informational messages ("I am happy" or "I am about to die") without expecting any response to them.

†. This is the principle of interception already discussed in §2•3.

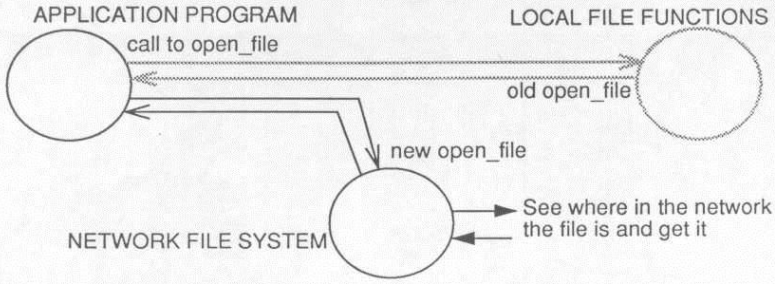

Figure 4• NFS intercepts the call to open_file, providing an extended service.

MESSAGES	TOOLS (and functions)			
	Mail	Editor	Compiler	ActivityDisplay
send_mail	✓	✗	✗	✓
edit_file	✗	✓	✗	✓
list_file	✗	✓	✗	✓
open_file	✗	✗	✗	✓

Figure 5• A message server allows other tools to eavesdrop (and possible take action) on original messages. Here, a new tool, ActivityDisplay, is listening to everybody's messages, in order to produce an activity diagram.

2•4•1 Pros and cons of a message server

✓ Simplicity, extensibility. It is *syntactically easy* to add a tool: create a new column in the message table (Figure 5•) for it; to delete a tool: delete its column; to disconnect a tool: set all entries in its column to ✗; to change the use of a tool: change some ✓ and ✗ in its column in the message table. Two tools that provide the same service but at different speeds can in parallel process a request; the first one to fulfil it can stop the other. All of this without the tools being aware of each other's presence. The tool interaction is built into the message server.

✓ Flexibility. Changing a table changes the interactions or ways of coupling among tools.

✗ It is *semantically difficult* to add a tool: it has to be done rather carefully; the response to the message in every row has to be preserved (but it can be augmented) by the addition of the new tool.

✗ Slow. Each call is intercepted, broadcasted, and manipulated, before other tool(s) can act upon it.

✗ It is easy to lose track of what is happening. Unwanted interactions among tools can take effect. Debugging the message table (Figure 5•) when many tools are involved, can be tricky.

✗ Limited. The tool interaction is in principle state-less: each tool responds to the same message in the same manner [although in practice the arguments could be used to change the behavior of the recipient]. The tool server is a finite state machine where the messages play the role of the symbols that cause transitions, and each state is an n-dimensional binary vector $[a_1, a_2, \ldots, a_n]$, where n is the number of tools and each a_i is 1 if for that state tool i is active, and 0 if not.

☞ Publicize the API's (messages, in this case) of the tools you build.

☞ If you want to use this method, consider the variant of §3•2

III Some solutions proposed

3•1 Function substitution

This is a generalization of §2•3, requiring a small amount of cooperation from the tool builder. It is also called "call interception" because the calls to some API functions get intercepted by new API functions having the same name as the intercepted functions. The functions themselves get redefined, but their interfaces (name, type of arguments, etc.) do not.

3•1•1 APIs through which the tool can be integrated

(a) The builder identifies some of the inner interfaces (those among subroutines of the tool) as possible interfaces with other tools. These interfaces (but not their source code), called API†, are made public: their names, arguments, and semantics: what each of these functions accomplishes or does, what are the roles of their arguments, etc. Common levels of interfaces are:‡

(1) at the character/gesture entry level. This API contains all the functions that obtain characters from the keyboard or clicks from the mouse and give it to the tool in question;

(2) at the memory (data structure) level. This API contains all the functions that modify pertinent□ data structures in memory;

(3) at the input/output (database) level; this API contains all the functions that interact with a database(s);

(4) at start and end (control level), or "external API level." This is the API *of the tool* itself; it contains all the functions through which this tool can be used. §2•2 used this level.

Although more than one API can be made public for each tool, the discussion here, for simplicity, speaks of only one.ø

(b) Whenever a given task (such as a modification of a data structure) can be accomplished through a published API, the tool is obliged to use it. That is, a public API defines subroutines for tasks *which are not done in any other manner* (through another function, or by direct pointer manipulation, say) in the tool.

(c) When delivering the tool (to a purchaser), the API is delivered in a separate library.

3•1•2 Method of integration

When a tool T_1 has an API defined as above, and it is decided to integrate it with another tool T_2, the following ques-

---

†. Application program interface: a set of subroutines through which a given program or tool is used.

‡. Compare with §1•1•2 and Figure 1•

□. One which is worth sharing with or serving as an interface to an external tool.

ø. Not every subroutine deserves to be published as part of an API; only those which make sense "for the world to know about;" only those which are reasonable candidates for tool interconnection.

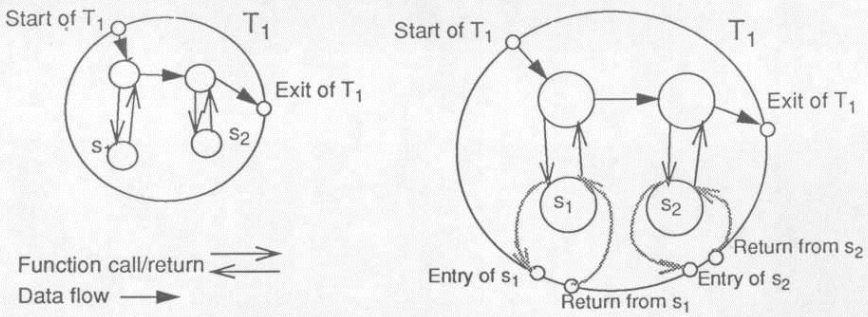

Figure 6• Subroutines s_1 and s_2 of tool T_1 (left) are made available in the external interface of T_1 (right).

tion has to be answered for each of the subroutines s_i of that API: "what should tool T_2 do when tool T_1 is calling this s_i?" The answer tells what code to write instead of the old s_i; that code constitutes the new s_i. Tool T_1 will call s_i as before, but the code of the new s_i will be executed instead. Some s_i can be left unchanged, if it is not required for T_2 to do anything when T_1 calls that s_i. Normally, the new s_i calls the old s_i (and the new code around the old s_i is part of the *integration code*), but it is not necessarily so. An example illustrates the method.

Example 8: A spreadsheet designer decides that it is appropriate to make available for interaction with other tools the functions that evaluate *the rightmost cells in each row* (which presumably contain the totals or some important value for such rows). Such API is published, containing only one function, s_1: `evalrightmost (n)`, where n is the row number. She now checks her code to see that no rightmost cell is evaluated using another function, say, eval(i, j), which evaluates *any* cell but is not part of the API. Having done that, her spreadsheet is now capable of integrating (using) a tool that, say, draws pie chart and graphs. The purpose of the integration is to redraw a pie chart each time one of these rightmost values changes. To accomplish this, the library that the tool uses (containing $_{old}$evalrightmost) is replaced by another that contains $_{new}$evalrightmost: a function that first calls $_{old}$evalrightmost and then calls the draw program to redraw the pie with the new value returned by $_{old}$evalrightmost. The code for the spreadsheet remains unchanged. It is in this form how the spreadsheet integrates the drawing program.

The method makes available (Figure 6•) "in the external interface" of T_1 the subroutines s_i of a given API. Encapsulation code is added to these s_i, to transform them into new s_i's that, among other things, call T_2. This is illustrated with the following example.

Example 9: The compiler T_1 has a public API for error handling; among these the function `error (file, line, err_num, err_messg)` signals the current error. This function currently sends an appropriate message to an error file (Figure 8•a). The modification consists in T_1 using T_2, an editor, to display the relevant source code *as the error occurs*. For this, function error is modified to first call (or send a message to) the editor T_2 to show the source file and the offending line; and then, proceed with the old business handled by the old error function (Figure 8•b).

3•1•3 Pros and cons of function substitution

✓ The tools (integrating and integrated) remain unchanged.

✓ Fast. No interpretation of calls.

✗ Changes to the API should be carefully done, to make sure that they will not interfere with other uses of the old API: old and new functions of the API should be functionally compatible (although not equivalent, since the new API extends the services that the old provided). Moreover, it could be possible for a program, if so chooses, to call the old API and not the new one.

✗ Rigid. If tools T_1 integrates T_2 and T_3, and now T_2 disappears, a new interconnection code needs to be written. Nevertheless, in the presence of dynamic loading/binding, the trick of §2•3 "Intercepting calls meant for somebody else" can be used.

✗ Requires that the loader does not get confused by the presence of two subroutines (one old and one new) with the same name.

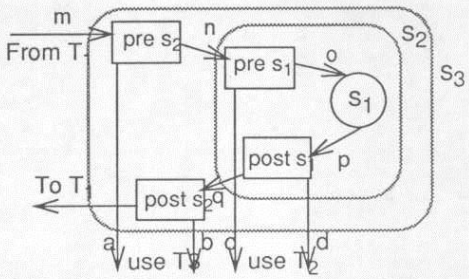

Figure 7• Call interception as a cascade of preamble-postamble pairs.

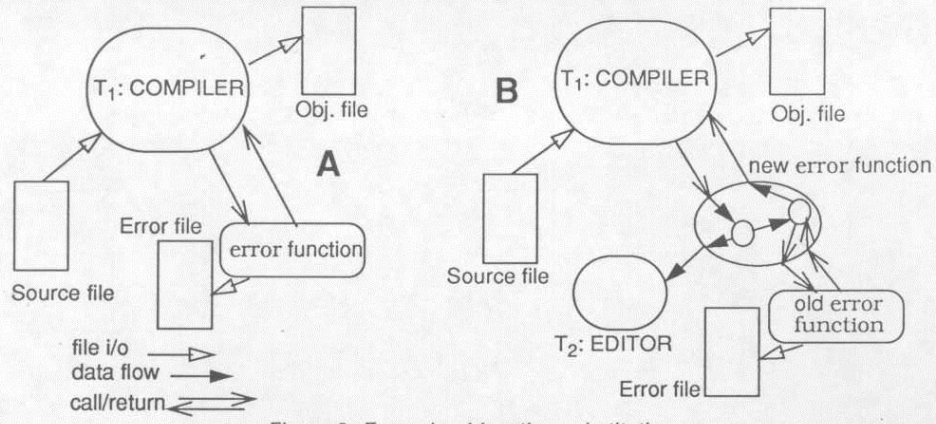

Figure 8• Example of function substitution.
In order to enable the compiler T_1 to exhibit the source code of the current error, the function error, that belongs to the API of T_1, has been replaced by a new function error, that uses the editor T_2 and then performs its old duties.

☞ Publicize the internal API(s) of your tool: those which you think are appropriate for integrating some other tool into yours.

☞ Use when a static integration is acceptable.

3•1•4 Converting call interception to message server integration

If T_1 uses T_2 via the method in §3•1, which substitutes function s_1 belonging to the API of tool T_1, then it is also possible for T_1 to use T_2 via the method in §2•4, employing a message server (which provides flexibility in place of rigid preamble-postamble encapsulation), as follows:

Figure 7• represents T_1 using T_2 and T_3 in a certain way, via the substitution of s_1 by s_2 and later of s_2 by s_3. The problem is to produce a server table like Figure 5•, for the same use. To do this, without loss of generality, s_2 can be represented with a preamble, a call (or message) to s_1 and a postamble; and similarly for s_3, as in Figure 7•

The desired table, having the same flow of messages as Figure 7•, is shown in Figure 10• This table provides the same functionality that Figure 7•, but in a more flexible manner: it is easy to dynamically change the connections among tools, by modifying some rows.

3.1.4.1 Extended notations

It is easy to simplify tables like that in Figure 10• by using a notation such as that of Figure 9•

The ✔ under function S_1 in first row now means: Yes, function S_1 will react to message m, but in this way: {$preS_2$, $preS_1$, S_1, $postS_1$, $postS_2$}; that is: first it will execute interaction code $preS_2$, then $preS_1$, then S_1 itself, then $postS_1$, and finally $postS_2$, all of this as the reaction to message m. The second row simply says: Only tool T_3 will react to the message a (which will be issued by $preS_2$), and similarly for the other rows. In other words, the dispatch server is now used as a primitive interpreter that executes the "code" {$preS_2$, $preS_1$, S_1, $postS_1$, $postS_2$} in sequence. Other notations could be used to mean parallel execution, non-deterministic execution, etc.

Another way to extend the notation is to have in each column, instead of the ✔, a pattern (string with placeholders such as *, %L, to be matched in special ways) specifying the format of acceptable messages. This is done in the Field Environment [8].:

3•1•5 Transforming a message server into call interceptions code

Conversely, each row of a message table like Figure 5• can be converted to fast code resembling calls interceptions. A given row like

	T_1	T_2	T_3	T_4 ... T_n
mess1	✔	✔		✔

says that message m goes to tools T_a, T_b, ..., T_k; such row can be replaced by the following code:

MESSAGE	S_1	T_2	T_3	T1
original messg m	✔ = {$preS_2$,$preS_1$, S_1, $postS_1$,$postS_2$}			
messg a				✔
messg b				✔
messg c			✔	
messg d			✔	

Figure 9• Extended notation

mess1: call T_a; call T_b; ...; Call T_k;

These calls could be executed serially, as shown, or in parallel. The advantage of this conversion is that all the calls are "pre-wired" and faster; the disadvantage is lack of flexibility if tool connection changes.

MESSAGE	S_1	T_2	T_3	$preS_1$	$preS_2$	$posS_1$	$posS_2$
messg m to S_1					✔		
messg a			✔				
messg n				✔			
messg c		✔					
messg o	✔						
messg p						✔	
messg d		✔					
messg q							✔
messg b			✔				

Figure 10• Message server table corresponding to calls in Figure 7•

3•2 Message server reacts to tools' state

A generalization of §2•4, this method does *selective broadcasting* of messages according to the state of its sender. The results are less message traffic, and detection of integration errors when a tool's behavior is inconsistent with its published API (as shown in its manual).

3•2•1 The state of a tool

At any given time, a tool will be in exactly one of a (usually small) set of modes or states, such as not-running, reading, modifying-memory, writing, etc. Some tools contain a global variable†, provide a function in an API or send a message to inform the world about its current state. Or the state can be deduced from the type of messages the tool sends through some of its APIs.

To make the message server (§2•4) sensitive to the tool states, the table of Figure 5• is generalized so that each row heading (representing in Figure 5• a message or API call) is replaced by a trio {T, s, m}, meaning: it is legal for tool T in state s to send message m. Trios not represented *are illegal*, and cause the message server to issue an error if it receives one at running time (Figure 11•).

Illegal combinations should not happen, if the tool and its manual agree. Nevertheless, in the presence of an inconsistency among tool and manual (error in the code or in the manual) or an inaccurate interpretation of the manual (Cf. §3•2•3, "interpretation" bugs), or of faulty integration code, some illegal combination *may* happen, but the absence of a trio for it in the dispatch table will detect this error!

On the other hand, through a bad interpretation of the tool's manual, a trio could be deemed legal and be present in the table, when in fact it is illegal. This kind of error can be discovered because, at execution, the trio never causes any firings.

TRIOS {Tool, State, Message}	T O O L S (and functions) Mail	Compiler Editor		Activity Disp.
{UI, *, send_mail}	✔	✗	✗	✔
{UI, start, edit_file}	✗	✔	✗	✔
{Debugger, editing, edit_file}	✗	✔	✗	✔
{Editor, file_present,list_file}	✗	✔	✗	✔
{ActivityDisplay,*,open_file}	✗	✗	✗	✔

Figure 11• LMessage server that is sensitive to the state of tools.

The UI can send_mail anytime, but can send edit_file only if in state start; the debugger can not send edit_file unless it is in state editing.

3•2•2 Variants of state-sensitive broadcasting

(a) The state of the receiving tool, in addition to the sender's, could be used to determine whether to deliver a message. For instance, the tick ✔ in the second row, second column (Figure 11•), which reads {UI, start, edit_file} Editor= ✔, can be generalized by specifying which states of the editor can receive the trio {UI, start, edit_file}, as follows:

{UI, start, edit_file} = {Editor, file_present}= ✔

{UI, start, edit_file} = {Editor, file_absent}= ✔

These rows say that the Editor in states file_present and file_absent can receive the message edit_file coming from the UI when the UI is in state start.

(b) If the different subroutines in the API of the receiver are known, the messages could be more specific and be targeted to a specific API of the receiver, not just to "the receiving tool."

(c) These variants should be easy to understand and rather popular, since you already have C (or any programming language) to program *any* message server.

3•2•3 Pros and cons of state-based tool interaction

As we know, this method uses selective broadcasting based in the state of the issuing tools.

✔ Provides more control over the work flow than the simple broadcast approach.

✔ Protects from bugs or cases where the behavior of the tool differs from its manual's description.

✔ Allows debugging of the integration code.

✔ Useful in a highly dynamic environment; when tools are disconnected, re-connected, etc.

†. Accessible, say, through the Common Area of Fortran.

✗ Requires knowledge of the state of the integrating tool (first component of a trio).

✗ "Interpretation" bugs are produced if the state transitions are not build with strict adherence to the tool's manual.

☞ Provide a global read-only variable or some easy manner to know the state of the tool you build.

☞ Use it when writing integrating code, and keep the states unless error messages are distracting.

3•3 Tools as servers

Under the following circumstances it is convenient to consider the conversion of an ordinary tool (that gets called and bounded into the caller) into a server that receives messages:

(a) If the tool gives a centralized service that somehow it is not convenient to replicate; for instance, a license server that allows copies of a protected program to be ran by different users;

(b) If the tool uses a single data repository; for instance, a data base server; a printer's spooler.

(c) If the tool attends several users but all of them sit in the same workstation (for instance, several users of a mainframe using the Lisp compiler);

Tools that are used intensively and interactively by each user should not be converted in servers; it is better in general for each user to have his own copy in his own workstation. Ex.: a text editor.

3•3•1 A tool serving both a user and another tool

This paper assumed that T_1 is active (has a user behind it) and that T_2 is only executing T_1's requests. In the more general case, T_2 is active, too: it has a user issuing commands to it, in addition to those coming from T_1. Or there could be more than one tool issuing commands to T_2. The solution becomes more complex, but it involves seeing tool T_2 as a server, which serializes requests of different users, be they tools or people.

3•3•2 Converting subroutine call ⇔ server message

Let us say that two copies of the tool exits: Tsub, which is called as a subroutine, and Tser, a server to which messages are sent.

To convert a call to Tsub into a message to Tser, replace Tsub by a new subroutine:

```
(new Tsub):     send a message to Tser;
                r:= value returned from Tser;
                return (r);  --return as value of new Tsub the
                               --value that Tser returned
```

To convert a message to Tser into a call to Tsub, replace Tser by a new server:

```
(new Tser):     spawn a new process P;
                send it the message and the <return address>;
(new process P): z: = call Tsub;
                send z to <return address>;
                kill (myself);
```

3•4 Interconnecting tools across machines

It is common, when one tool calls another, that both reside in the same address space, or at least, within the same file system. With remote procedure calls, tools can reside in different address spaces (possibly with different file systems) and, therefore, perhaps in different workstations.

If message passing is used, the tools can reside in different machines; the only limitations to this are the restrictions imposed by the routing capabilities of the message server (§3•2).

If the machines are heterogeneous: different operating system, different vendors, etc., the particulars of a successful connection become more complex and will not be addressed here. Nevertheless, usually among Unix systems it is possible a large degree of message exchanges and remote procedure calls.

3•5 Conclusions

The problem of connecting T_1 to T_2 (where T_1 calls or uses T_2) has been analyzed:

(a) *Clear box interconnection* is possible when source code for T_1 is available, it is easy and is the preferred interconnection method (§2•1). It involves changing the code of T_1;

(b) *Black box interconnection* (source code for T_1 is not available) is possible via loose integration (solution in §2•2), involving writing glue code around T_2.

(c) *Call interception*, which is "almost black tool integration" (source code for T_1 is not available but some information about some APIs that T_1 uses internally is available) produces much better tool integration, with tighter coupling. Cf. §§2•3 and 3•1.

(d) A *Broadcast message server* (§2•4) provides more flexibility and better control flow among tools.

(e) A *State-based message server* (§3•2) brings additional protective capabilities, but requires that the state of the calling tool T_1 be known.

The paper gives pros and cons for each case.

Probably the message server is the most interesting way to integrate tools. It changes the way programmers are used to do tool integration. It treats different tools in a uniform (client-server) manner. Tools can request the services from other tools and at the same time, provide services to others. A message server equipped with better control and management mechanisms enables both centralized and distributed computation.

Integrating different tools is akin to building a new tool. The tool integrator needs to visualize the tool integration and their control. The simpler the integration method, the better. If tools are black boxes, the less intrusion into the existing tool APIs, the better.

Tool integration is a new way to develop software, with the advantage that the components are pretty sophisticated tools and application programs.

3•5•1 Recommendations for further work

- Build hardware that knows the name and arguments of each subroutine being called (keeping a stack of symbols at running time, for instance), like the Burroughs B5500-B7700 mainframes. In this way, calls to selected subroutines (of APIs) could be intercepted much easier.
- Standardize the most common APIs that tools use. In this manner, tools will become plug-compatible. Example: There should be one single subroutine, open_database (arg1, ...) to open *any* database. Notice that the methods of this paper do not rely on any standard API; they just need that the API to be intercepted be public (hence, replaceable).
- Further along these ideas, standardize on the ways applications communicate with the world: that is, have a set of standard APIs that all (most?) applications should use. The Object Request Broker by the Object Management Group [5] is an example.

3•6 References

[1] Carla Burns. Proto — A Prototyping Tool for Requirements Specification, Analysis and Validation. *First International Workshop on Rapid System Prototyping*, June 4-7, 1990. Research Triangle Park, N. C.

[2] Carla Burns. Parallel Proto – A Prototyping Tool for Analyzing and Validating Sequential and Parallel Processing Software Requirements. *Second International Workshop on Rapid System Prototyping*. July 1991. Research Triangle Park, N. C.

[3] HP SoftBench Environment. Hewlett Packard Co.

[4] HP Encapsulator Integrating Applications into the HP SoftBench platform. Hewlett-Packard Co.

[5] Geoffrey R. Lewis. CASE integration frameworks. *SunWorld*, 73-82. July 1991.

[6] Software Backplane. Atherton Technology. 1333 Bordeaux Dr. Sunnyvale, CA 94809.

[7] Sun Network File System. Sun Microsystems Co.

[8] Steven P. Reiss. Connecting tools using message passing in the Field environment. *IEEE Software*, July 1990, 57-66.

[9] ToolTalk (Beta) Programmer's Guide. Sun Microsystems Co.