Enveloping Sophisticated Tools into Process-Centered Environments*

GIUSEPPE VALETTO AND GAIL E. KAISER kaiser@cs.columbia.edu
Rank Xerox Research Centre, 6 Chemin de Maupertuis, 38240 Meylan, France
Columbia University, Department of Computer Science, New York, NY 10027, United States

Columbia University Technical Report CUCS-22-95, July 1995
©1995 Giuseppe Valetto and Gail E. Kaiser

Abstract. We present a tool integration strategy based on enveloping pre-existing tools without source code modifications or recompilation, and without assuming an extension language, application programming interface, or any other special capabilities on the part of the tool. This Black Box enveloping (or wrapping) idea has existed for a long time, but was previously restricted to relatively simple tools. We describe the design and implementation of, and experimentation with, a new Black Box enveloping facility intended for sophisticated tools — with particular concern for the emerging class of groupware applications.

Keywords: Tool integration, workflow, computer-supported cooperative work, computer-aided software engineering

An extended abstract of this paper appears as Giuseppe Valetto and Gail E. Kaiser, Enveloping Sophisticated Tools into Computer-Aided Software Engineering Environments, *IEEE Seventh International Workshop on Computer-Aided Software Engineering*, July 1995, pp. 40-48.

1. Introduction

Process-centered environments and other task-oriented frameworks (see, e.g., the NIST/ECMA reference model [25]) usually support dialogues between external tools and the environment, which serves as a mechanism for integrating the tools according to their roles in the workflow. We identify three categories of integration methods, with respect to their approach to adapting the tools to the environment:

* This work was conducted while Mr. Valetto was a graduate student at Columbia University. Prof. Kaiser was supported in part by Advanced Research Project Agency Order B128 monitored by Rome Lab F30602-94-C-0197, in part by National Science Foundation CCR-9301092, in part by the New York State Science and Technology Foundation Center for Advanced Technology in High Performance Computing and Communications in Healthcare 94013, and in part by grants from AT&T Foundation, Bull HN Information Systems and IBM Canada Ltd. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the US or NYS governments, ARPA, NSF, NYSSTF, AT&T, Bull, IBM or Xerox.

- *White Box*, where a custom tool is developed as part of a particular environment or a pre-existing tool's source code is modified to match a framework's interface. Custom tools may be prohibitively expensive to develop. Changes to pre-existing tools can often be implemented in a straightforward, repetitive manner, but nevertheless the source code must be available — perhaps an insurmountable difficulty when integrating off-the-shelf tools from independent vendors. The White Box approach is followed by several commercial message buses, most based on either the Field broadcast message server [29] or the Polylith software bus [28]. PCTE [34] and similar framework standards probably require more effort in tool adaptation, or a priori adherence to the standard by vendors, but enable a higher scale of integration. The CORBA interoperability standard [24] is not specifically directed to environment frameworks, and seems best suited to tools explicitly organized as servers — which relatively few are at present.

- *Grey Box*, where the source code is not modified but the tool provides its own extension language or application programming interface (API) in which functions can be written to interact with the environment. Relatively few tools, aside from database management systems, provide such convenience (although see [26]). Dynamic linking coupled with replacement of standard libraries (e.g., for I/O) works for some environments, e.g., Provence [23], concerned with monitoring simple events such as file system accesses, but it seems unlikely in the general case that arbitrary tools would happen to fit the protocols of a task-oriented framework. In particular, a process-centered environment requires that task prerequisites be fulfilled prior to performing the task, so mechanisms to detect that some tool has already completed a task are inadequate [27].

- *Black Box*, when only binary executables are available and there is no extension language or API. In this case, the environment must provide a protocol whereby *envelopes* [1] extract objects and/or files from the environment's data repository, present this data to their "wrapped" tools in the appropriate format, and provide the reverse mapping for updated data and tool return values. Envelopes may also be used in conjunction with Grey and White Box methods, but are mandatory for Black Box integration.

Our primary goal in this paper is to augment enveloping concepts and technology to apply to a much wider array of tools. We concentrate on the Black Box model, since it is often the only choice as well as the most challenging.

Typical Black Box enveloping technology expects the tool integrator to write a script or program that handles the details of interfacing between the tool and the environment framework, often both to respect the environment's notion of task and to access its data repository, as well as the actual invocation of the tool with an appropriate command line and collection of any outputs and return values. In the case of a process-centered environment, the process definition determines the workflow within which such a script or program may be executed. For example, the task's prerequisites may need to be satisfied in advance and its obligations fulfilled afterwards. The state of the on-going process execution usually sets the context for providing parameters to the tool and determines what should be done with its results.

This approach works well for tools, such as the standard UNIX toolset, that accept all their arguments from the command line at invocation, read and write some files (whose file system pathnames are given on the command line), and return a simple status code. Notice this does not preclude interactive tools such as word processing and drawing systems, since the tool's own user interface appears on the user's screen when the envelope executes the tool. The user may then enter text or click menu items as desired; however, the granularity of access to objects/files from the environment's data repository is the entire tool invocation. In other words, the nature of current Black Box enveloping technology requires that the complete set of arguments from the repository is supplied to the tool at its invocation and that any results to be returned to the repository are gathered only when the tool terminates, so that the tool execution is encapsulated within an individual task.

There are numerous tools whose natural and/or convenient use doesn't fit this description, but may be highly desirable to integrate into process-centered environments, including at least the following categories. Note these classes are not disjoint.

- Tools intended to support *incremental* request of parameters and/or return of (partial) results in the middle of their execution, such as multi-buffer text editors and interactive debuggers. Although such tools by definition allow submission of an arbitrary sequence of the user's choice of commands during their execution, when run in a stand-alone fashion, current enveloping technology does not permit the sequence of commands to be guided, automated or enforced by a task-oriented environment, and often even precludes retrieval of their parameters from the environment's data repository (e.g., if the process engine controls all access to the repository).

- *Interpretive* tools that maintain a complex in-memory state reflecting progress through a series of operations: Lisp applications, such as "Knowledge-Based Software Assistant" (KBSA) systems [9], are classic examples. Such tools may require severe start-up overhead and command substantial system resources (thus we refer to them as "heavy-weight"). We are particularly concerned with permitting different users to submit tasks to the same tool execution instance, even when that tool was not designed to support multiple users. One of our goals is to extend a variety of single-user tools to (modest) multi-user operation.

- *Multi-User* tools, such as conventional database management systems that guarantee atomicity and serializability of separately transmitted but concurrently executing transactions. An important subclass is *Collaborative* tools (often referred to as computer-supported cooperative work — CSCW — or groupware), which abhor the conventional isolation model and directly support multiple users interacting with each other, such as WYSIWIS (what-you-see-is-what-Isee), IBIS decision support, Fagan-style document inspection, desktop video conferencing, etc. (see [21], [2] for more examples).

We introduce a *Multi-Tool Protocol* (MTP), where *Multi* refers to submission of *multiple* tasks to the same executing tool instance and enabling of *multiple* users to interact with that same tool instance. Tool instances may operate for an arbitrary period of time, far beyond the length of an individual task on behalf of an individual user; thus we refer to the executing tool instance as "persistent" with respect to the duration of the tasks submitted under the MTP protocol. MTP also addresses *multiple* platforms: submitting tool invocations to machines other than were the user is logged in, e.g., when operating over a heterogeneous collection of workstations and server computers but executables are available for only a particular machine architecture or even only for a specific host; and *multiple* tool instances: managing a set of executing instances of a tool, e.g., when licensing limits the number of instances that can operate at the same time, common with commercial server licenses. MTP, as currently defined, treats tools in a Black Box manner. MTP has been implemented as part of the Oz process-centered environment.

Section 2 supplies brief background information on Oz. Section 3 introduces a tool modeling notation for specifying the category and special requirements of the tool; this notation extends Oz's previous facility, but could readily be adapted to other process-centered environments with some notion of tool declaration. Then we present our main work in Section 4, covering the general ideas, persistent tool sessions for four different categories of tools, an extension of the Oz client/server architecture for managing MTP tools (intended to be adaptable to other client/server or peer-to-peer architectures), the protocol for interaction between a process or task control engine and executing tool instances, and finally the structure of the tool wrappers themselves (we will use the terms "envelopes" and "wrappers" interchangeably throughout the paper). Then Section 5 describes four tool integration experiments, one of which represents each of our categories. The paper concludes by summarizing our contributions and outlining future work plans.

2. Oz Background

Oz [7] is a process-centered environment framework that supports interoperability among autonomously defined processes, where the participating instantiated processes may reside on the same machine, the same local area network, or be geographically dispersed across the Internet or other wide area network. The processes are all written in the same rule-based process modeling language, but the workflows they implement may be arbitrarily diverse, so each instantiated process may perform completely different tasks using different toolsets. The interoperability aspect is not particularly germane to this paper, which is primarily concerned with integrating tools within a single process and leaves inter-process tool integration for future work (see Section 6).

Oz represents both product (project artifacts) and process (workflow status) data using a home-grown object-oriented database management system, a separate local objectbase for each instantiated process. An object may contain zero or more file attributes, each typed as either `text` (ASCII) or `binary`. The value of a file attribute within a local objectbase is a file system pathname into a "hidden" file system specific to that objectbase, not intended to be accessed except through Oz. Non-file attributes include the usual primitive values (strings, integers, etc.), references to child objects, and links to arbitrary objects elsewhere in the same local objectbase.

Oz's Shell Envelope Language (SEL) [15] is typical of current Black Box enveloping facilities [2], which typically involve some scripting language. The process engineer (or environment builder) writes what are essentially UNIX *sh*, *csh* or *ksh* scripts, using added constructs that a translator expands into regular shell commands to handle the details of interfacing between the tool and the environment framework. An SEL envelope is associated with each primitive task (primitive tasks may be grouped into aggregate tasks in the process definition [18]). After parameters have been bound and other preliminaries completed, Oz's process execution service directs that the named envelope be invoked on the arguments specified in the task definition, including literals and/or object attributes. When the envelope terminates, it returns a status code and (optionally) result values to the process engine, at which point the pending task assigns the result values to objectbase attributes and performs various operations based on the envelope's status (typically indicating success versus failure).

The mechanism described above is implemented within a client/server architecture, one server per instantiated process, with process execution and object management services in the shared server and user interface and envelope invocation facilities supported by each client [8]. The server sends envelope names and arguments to the client responsible for that task, and then handles other clients in a first-come-first-served manner until the tool completes and the results arrive at the front of the server's request queue. Clients are always connected to their local server, but may dynamically open and close connections to remote servers as illustrated in Figure 1. An infrastructure supports communication and coordination among clients and servers. See [5] for additional information.

3. Tool Modeling

Assuming both SEL-like enveloping and the new MTP protocol are available, the process or other task-oriented execution service needs to specify which tools require which protocol. In principle, every tool could be invoked via the new MTP protocol, but we retained SEL for Oz (or the equivalent facility for some other system) as the default because we believe that MTP is complementary to SEL on a per-tool basis: together, they address with greater specificity the peculiarities of diverse families of applications, and the choice allows minimization of overhead balanced across a

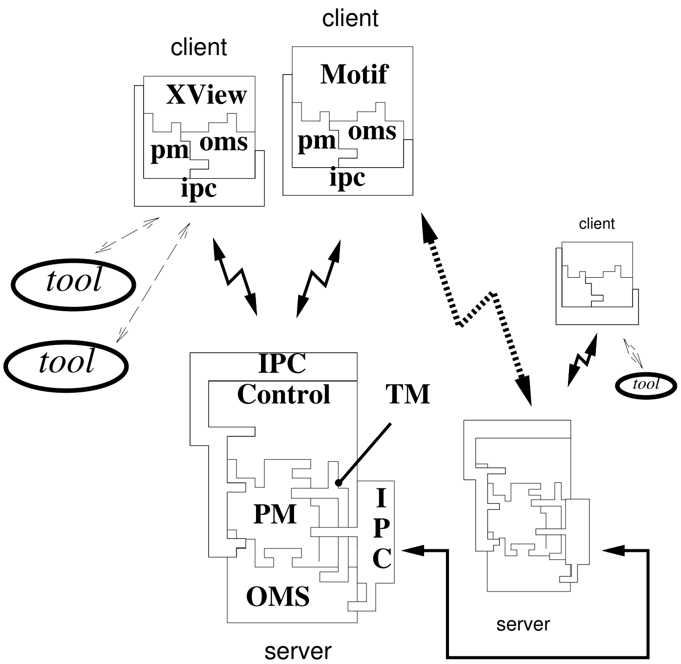

Figure 1. Oz architecture

```
<tool-name> :: superclass TOOL;
  [ protocol    : (MTP, SEL) ;
    path        : <string> ;
    architecture: (sun4, ...) ;
    host        : <string> ;
    instances   : <integer> ;
    multi-flag  : (UNI_QUEUE, MULTI_QUEUE,
                   UNI_NO_QUEUE,
                   MULTI_NO_QUEUE) ;
  ]
   <activity-name> : string =
      "<envelope-name> <parameters locks>";
   <activity-name> : string =
      "<envelope-name> <parameters locks>";
   ...
end
```

Figure 2. Modified tool definition notation

number of factors (see Section 4). In general, we believe an approach to integration based on *multiple enveloping protocols* is likely to achieve the greatest generality.

In the Oz implementation, the tool declaration notation has therefore been modified to include the new portion shown between square brackets ("[...]") in Figure 2, which is optional and may be omitted for SEL (some but not all of these fields are meaningful for SEL, as explained later, but defaults are assumed if they are not provided by the environment builder).

The new fields have the following meanings:

- `path` indicates the pathname in the file system where either the tool's executable or envelope resides (an envelope is not always needed for tool initialization when using our MTP protocol, depending on the details of the tool). For example, an envelope might prompt the user for tool parameters not managed by the environment (such as a database volume).
- `host`, an Internet address, is given when it is necessary to run the tool on a specific host because of some restriction (perhaps due to pragmatic licensing issues).
- `architecture` is used to indicate the machine architecture and/or operating system on which the tool (and its corresponding envelope) is expected to run. When the `host` is not specified, the system refers to the `architecture` specification and separate environment instance-specific configuration information, to determine a corresponding default machine on which the persistent tool (and its envelope) will be invoked.

In principle, an envelope forked on one machine could invoke a tool on another using UNIX *rsh* or other remote job control, but tracking an operating system process on another machine generally introduces an extra level of complexity in the envelope, and there may be difficulties redirecting user I/O for interactive tools. These issues are addressed in Section 4.2.

- `instances`: This specifies the maximum number of copies of the tool that can execute at the same time (0 means there is no upper limit). Independent of licensing issues, this could be used to bound the system resources allocated to a heavy-weight tool in all its instantiations.

- `multi-flag`: This determines the behavior of MTP in managing the interactions between multiple human users and a persistent tool instance. We distinguish among four categories of tools, with respect to their single-user versus multi-user and single-tasking versus multi-tasking capabilities, through the cross-product of two orthogonal dimensions:

  - **UNI vs. MULTI**: MULTI (multi-user) indicates that the same instance of the program can be shared by several users, whereas UNI (single-user) allows only for isolated work of each user on his/her own executing instance of the tool;
  - **QUEUE** vs. **NO_QUEUE**: where concurrent (overlapping) execution of multiple tasks with respect to the same tool instance is supported for NO_QUEUE (multi-tasking) but not for QUEUE (single-tasking).

  It may seem counterintuitive to think of these dimensions as orthogonal. In the case of `MULTI_QUEUE`, i.e., multi-user and single-tasking, multiple tasks on behalf of different users can share the same tool instance, but only one actually runs at a time (first-come-first-served). For `UNI_NO_QUEUE`, i.e., single-user and multi-tasking, multiple tasks can execute simultaneously in the same tool instance (perhaps in distinct "buffers" or other tool-specific contexts — the tool need not be implemented using multi-threading or parallel processing technology), but all must be on behalf of the same user.

Each of the declarations following the brackets specifies the name of a task together with the file name of an envelope, distinct from the one that started up the tool (if any). The task-specific envelope is invoked whenever the corresponding task is submitted to the persistent tool. There are likely to be several qualitatively different tasks that can be performed using the same tool, so it is expected that multiple task/envelope mappings would be listed in the tool declaration. If so, multiple instances of the same task or several entirely different tasks can be submitted to the same persistent tool execution. Formal parameters and locking information are also listed (transaction management is outside the scope of this paper, see [3], [17]). The envelope specified by the task handles the passing of arguments back and forth to/from the environment as well as the details of interaction with a tool that is already running.

These declarations appear in identical form in SEL specifications, but in that case each envelope invokes a distinct tool instances to perform the task (and envelopes may be grouped into the same tool declaration for abstraction reasons, without necessarily employing the same external program). We made no changes at all to Oz's process modeling language other than the tool declaration notation, and our approach is intended to be orthogonal to the environment framework's mechanisms for workflow definition and performance.

4. The Integration Protocol

We adopted what we call a *loose wrapping* approach, as opposed to the *tight wrapping* currently effected in Black Box enveloping schemes. The latter relies on complete encapsulation of all of the tool's actions inside the envelope, whereas the former is instead based on control of the tool's behavior (from the viewpoint of the environment), with the enveloping facility intervening only as the need arises during workflow execution and/or upon detection of some external event relevant to the environment. A typical example of the former is when the initiation of a workflow step (either automatically or through an environment command selected by a user) requires the tool to perform some task, and of the latter when a tool action saves some files that should be recorded in the environment's repository.

Control, as opposed to encapsulation, provides a means for long-lived and intermittent dialogue between external tools and the environment; meanwhile, the tools continue their execution effectively detached from the environment framework. Tight wrapping, on the other hand, governs all phases of a tool's execution, from the moment of invocation to termination; to perform multiple tasks using the same tool, it must be explicitly and repeatedly instantiated (even if on behalf of the same user) each time a task is assigned to the tool.

Our approach may be viewed as combining the advantages of conventional Black Box enveloping and event notification systems like Field and Yeast [30], where tools execute persistently but the server's concern is only for events of interest to other tools and there are no separate "environment commands" or "workflow" that control tools. The Forest extension of Field manages the propagation of event notifications among tools according to "policies" [14], analogous to Oz's process management services, and Provence is implemented on top of Oz's predecessor (Marvel), but neither has any means for requiring satisfaction of task preconditions. These systems also do not address one of our foremost requirements, to integrate multi-user tools, and few message buses are concerned with groupware or even support multiple users per bus. Buses internal to process-centered environment frameworks such as ConversationBuilder [22] and ProcessWEAVER [13] are exceptions.

Once we established loose wrapping as the overall principle on which to base our design, we analyzed the major capabilities needed to implement our tool modeling facilities (described in the previous section). We divide these functions into two categories: those generally concerned with Black Box integration — i.e., the abilities to invoke and terminate an instance of a tool on demand, to parameterize that instance according to the single environment task, to transform objects from/to the environment's representation to/from that required by the tool, to support and display the I/O flow between the wrapped program and its user(s) — and those especially necessary given the nature of the four tool categories of interest (i.e., the cross-product of UNI vs. MULTI and NO_QUEUE vs. QUEUE):

1. The ability to limit the number of co-existing (executing) copies of a given tool according to the specifications set out in the tool's declaration, and to record and service previously unsatisfied requests as soon as possible;

2. The ability to exploit the persistence of MTP-tools, in order to share their instances among multiple users — possibly emulating partial multi-user capability for programs not usually employed for groupware;

3. The ability to coordinate overlapping requests for access to an instance of a persistent tool from separate users, to avoid deadlocks and starvation on the one hand, and of unintended concurrency of several activities for programs that do not support any form of multi-tasking on the other;

4. The ability to record results of intermediate steps of the tool's processing, during the execution of each single task.

To fulfill these requirements, we have introduced several extensions to Oz's process management services. Analogous extensions could be made to other environment frameworks.

4.1. Tool Sessions

To encompass both serial and concurrent access to a tool instance, we introduce *sessions*, which define the life-span of a persistent tool: a session normally begins with an `OPEN-TOOL` command and ends with a `CLOSE-TOOL` command, as illustrated in Figure 3. A session's body is made up of a set of primitive tasks determined dynamically as the users carry out their work within the environment. Each `MTP-activity` in the figure maps to an individual primitive task. Note that although the tasks are listed in sequence, they could potentially overlap (for NO_QUEUE tools),

`tool` could refer to any tool declared as MTP. The `session` identifier distinguishes among simultaneously executing instances of the same persistent tool, so that multiple users can choose to participate in a particular session opened by another user (for MULTI tools). Both arguments are selected from menus. Users can ask to join an existing session by clicking the corresponding identifier when issuing an `OPEN-TOOL` command. The current implementation does not provide any support for access control, e.g., specifying which users are permitted to, or are required to, join a particular session; the latter is being addressed by current work in Oz process modeling (see Section 6). There is also no support for providing parameters for tool initialization from <u>within</u> the environment, which is less limiting than it

```
OPEN-TOOL tool [session]>
    <MTP-activityA> <argumentsA> <session>
    <MTP-activityB> <argumentsB> <session>
    ...
CLOSE-TOOL <tool [session]>
```

Figure 3. Tool session template

sounds since the tasks that trigger incremental interaction with the tool usually provide parameters from the environment.

Leaving a session is achieved with a `CLOSE-TOOL` command applied to a session where there are still other active users. In this case, the `CLOSE-TOOL` does not kill the tool instance, but only changes internal information about the association between the user and the session. Termination of the program follows the `CLOSE-TOOL` command of the last participant.

Besides setting the duration of a specific tool instance and providing a context for sharing an application, sessions are central in several other functions supported by our protocol. For example, they implicitly operate on what we call the *Session Queue* of a tool. This feature allows us to satisfy the constraints posed by the `instances` field of a tool declaration, accordingly limiting the maximum number of copies of the program that can be active simultaneously. (Such a restriction could be violated due to tool instances executing completely outside the environment, resulting in tool invocation failures.) When `OPEN-TOOL` is issued, the system first checks whether the request is satisfiable given this constraint. If the limit has been hit, the request is not serviced, but is recorded in the Session Queue; when an already running session is terminated, the next queue entry is extracted and automatically initiated (the user is effectively notified when the user interface of the tool pops up on his/her screen).

Our design also allows for a special case where it is possible to use a persistent tool without being compelled to issue the `OPEN-TOOL` and `CLOSE-TOOL` commands every time, via an implicit *atomic* session that consists of only a single task. Atomic sessions are instituted by the system, transparently to the user, when a user issues a task associated with an MTP tool but has not previously opened or joined a session. In that case, an implicit `OPEN-TOOL` command is automatically executed and the new tool instance is marked as *atomic* by the environment, so that no other tasks (or `OPEN-TOOL`/`CLOSE-TOOL` commands) can be directed to it. When the task finishes, the tool is shut down automatically.

Our sessions idea leads to a number of questions on how different users could, practically, participate in the same session of a persistent tool, thus exploiting the same resources and the collected state of the executing tool. In our MTP design, we stressed the facets intended to accommodate in a natural way those applications that are inherently designed for collaboration, or — in some sense an even more

```
User 1: OPEN-TOOL <tool> <session S1>
        Session S1 begins
                User 1: <tool> <MTP-activity A> <argument set A>
                       Activity A begins
                               ...
                       Activity A ends

                User 1: <tool> <MTP-activity B> <argument set B>
                       Activity B begins
                               ...
                User 1: <tool> <MTP-activity C> <argument set C>
                       Activity C is stored in Activity Queue of S1
                       (Activity B continues)
                               ...
                       Activity B ends

                       Activity C begins (automatically resumed)
                               ...
                       Activity C ends

                User 1: CLOSE-TOOL <tool> <session S1>
        Session S1 ends
```

Figure 4. Example session of a UNI_QUEUE tool

ambitious goal — to exploit in a multi-user context those tools that, even if not commonly employed in that manner, the tool integrator considers adaptable to and promising for collaborative activities.

Our four categories of tools provide a flexible solution to these problems: the valid values of the `multi-flag` field within the tool modelling specifications represent and enforce in the protocol four working models, intended to cover as widely and as precisely as possible the behaviors and requirements of various classes of persistent tools.

`UNI_QUEUE` is the most basic category: with it, we intend to accommodate applications that are strictly single-user and that would not adequately support concurrent operations deriving from simultaneous MTP activities. Therefore each instance of such tools is reserved exclusively to the user who requested it in the first place, via an `OPEN-TOOL` command, and the body of the session is made up of a simple sequence of activities that are never permitted to overlap. A generic example can be seen in Figure 4.

The most significant difference between MTP's `UNI_QUEUE` and SEL is that multiple operations can be sent to the same copy of the tool, under the complete control

```
User 1: OPEN-TOOL <tool> <session S1>
        Session S1 begins
                User 1: <tool> <MTP-activity A> <argument set A>
                       Activity A begins
                               ...
                User 1: <tool> <MTP-activity B> <argument set B>
                       Activity B begins
                               ...
                       Activity A ends

                User 1: <tool> <MTP-activity C> <argument set C>
                       Activities B, C carried out in parallel
                               ...
                       Activity C ends
                               ...
                       Activity B ends

                User 1: CLOSE-TOOL <tool> <session S1>
        Session S1 ends
```

Figure 5. Example session of a UNI_NO_QUEUE tool

of the process execution engine, by exploiting the newly introduced concept of *Activity Queues*: each `UNI_QUEUE` session is associated with an Activity Queue, which holds in first-come-first-served order the tasks waiting to take control of the tool instance.

`UNI_NO_QUEUE` is intended to satisfy more complex integration requirements and to allow for more operational flexibility. Again, each tool instance is reserved for just one user, but the full exploitation of the inherent multi-tasking (or multi-buffer) capabilities of the tool is supported, by directing to the tool multiple simultaneous activities. The outcome is exemplified by the generic session illustrated in Figure 5.

If a tool is not inherently multi-user (as is the case for most current tools), but is declared `MULTI_QUEUE`, only the most rudimentary form of sharing is possible: different users are allowed to join the same session, and therefore to access the same executing tool instance, but they must "take turns" (if they happen to issue requests that overlap in time): they are forced to wait in the Activity Queue until the previous task is finished. Note that users whose requests are placed in the Activity Queue may still execute other tasks — or decide to abort and try again later (Oz's XView and Motif interfaces allow a user client to context-switch at will among in-progress tasks, and other environments generally do likewise). We show a hypothetical session for a `MULTI_QUEUE` application in Figure 6.

```
User 1: OPEN-TOOL <tool> <session S1>
        Session S1 begins
                User 1: <tool> <MTP-activity A> <argument set A>
                       Activity A begins
                               ...
User 2: OPEN-TOOL <tool> <session S1>
    (User 2 joins session S1)
                               ...
                       Activity A ends

                User 3: <tool> <MTP-activity B> <argument set B>
                       Activity B begins (on borrowed session S1)
                                   ...
                User 2: <tool> <MTP-activity C> <argument set C>
                       Activity C is stored in Activity Queue of S1
                               ...
                       Activity B ends

                       Activity C begins (automatically resumed)
                               ...
User 1: CLOSE-TOOL <tool> <session S1>
    (User 1 leaves session S1)
                               ...
                       Activity C ends

User 2: CLOSE-TOOL <tool> <session S1>
        Session S1 ends
```

Figure 6. Example session of a MULTI_QUEUE tool

```
User 1: OPEN-TOOL <tool> <session S1>
        Session S1 begins; system component 1 dispatched to User 1
                User 1: <tool> <MTP-activity A> <argument set A>
                       Activity A begins
                               ...
User 2: OPEN-TOOL <tool> <session S1>
    (User 2 joins S1); system component 2 dispatched to User 1
                User 2: <tool> <MTP-activity B> <argument set B>
                       Activity A, B are carried out in parallel
                       by components 1, 2 respectively
                               ...
                User 2: <tool> <MTP-activity C> <argument set C>
                       Activity C is stored in Activity Queue of
                       system component 2
                               ...
                       Activity B ends
                       Activity C begins (automatically resumed)
                               ...
                       Activity A ends

User 1: CLOSE-TOOL <tool> <session S1>
    (User 1 leaves session S1); system component 1 is killed
                               ...
                       Activity C ends

User 2: CLOSE-TOOL <tool> <session S1>
        Session S1 ends; all existing system components are killed
```

Figure 7. Example session of a MULTI_NO_QUEUE tool

Although limited, this form of sharing can be usefully exploited in various collaboration scenarios, for example, by multiple users committed to take care of different sequential stages of the same complex, long and composite software task, in which all must employ the same external program. We can then think of the `MULTI_QUEUE` tool as a semi-permanent global service for these users.

The `MULTI_NO_QUEUE` class was conceived to accommodate inherently multi-user systems, including groupware, taking into account their architectural and functional peculiarities. MTP ensures in this case that every `OPEN-TOOL` command issued by some user in the context of the same session maps to the instantiation of a portion of the same multi-user system (e.g., a client in a client/server architecture), which is assigned to that user. See Figure 7 for an outline of a hypothetical session.

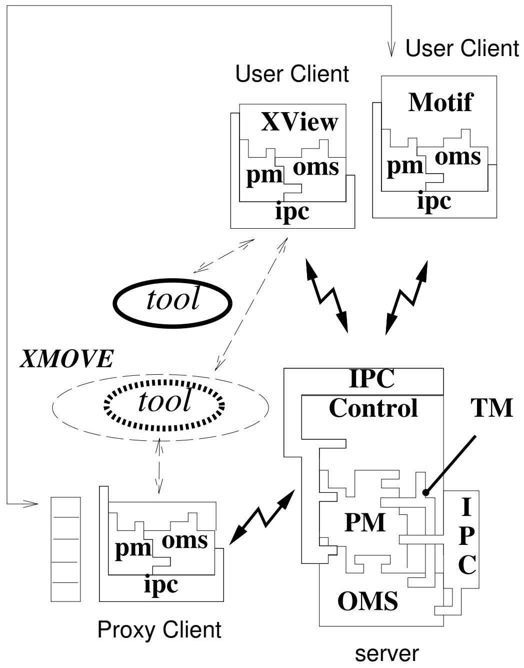

Figure 8. New Oz architecture

While MTP is in charge of directing users' work items to MULTI_NO_QUEUE tools, it is the intrinsic multi-user nature of these applications that defines whatever sharing and concurrency policies are necessary to operate in the multi-user and possibly collaborative context; the transparency or visibility — or lack thereof — among user clients with respect to their activities and data depends solely on the nature and the purpose of the tool, which may support collaboration or enforce isolation. The integration protocol, per se, is not concerned with these issues (see Section 5.4).

4.2. Architecture

The implementation architecture is necessarily specific to Oz, but we anticipate that a similar approach would apply to other multi-user process-centered environments. We divided Oz's clients into two categories, new *proxy clients* (or just proxies) and the original *user clients* [3]. MULTI_QUEUE operation (with respect to a single site) is depicted in Figure 8. Both user clients and proxy clients support SEL (see Section 6). Proxy clients introduce into the architecture a new kind of long-lived entity, with the role of spawning, managing, and achieving the integration of persistent tools. User clients are always associated with human users of the system, who invoke and close them at will, and therefore they cannot be relied on to support the life cycle of a persistent tool instance. The Oz server persists indefinitely but provides process execution and object management services and most aspects of tool management discussed in this paper, but is intentionally not directly involved with tool invocation (in part for performance reasons, see [4]).

In our design, the session management commands (`OPEN-TOOL` and `CLOSE-TOOL`) are issued by user clients on demand by human users and executed by the appropriate proxy client, installed on the machine determined by the **`host`** and **`architecture`** data in the MTP `TOOL` declaration and, if both fields are null, then on the same machine where the Oz server is running. Subsequent tasks submitted to the same application may be initiated from a user client's user interface, but are delegated to the proxy client. The same proxy manages all persistent tools executing on the same host (with respect to tasks managed by the same Oz server).

Proxy clients do not need to interact directly with any human operator, so no user interface is needed. However, they must manage the user I/O to/from persistent tools. This involves redirection of simple textual I/O between the tool and the user client, and more significantly the ability to display the tool's own graphical user interface (GUI) on the user's monitor. Most inherently multi-user tools are able to dispatch private instances of their user interface to each user, but for other tools (e.g., originally single-user tools extended by MTP to a modest form of groupware) we exploited the public-domain *xmove* utility [32], which transfers the GUI of a tool across workstations and X terminals. Resetting the X Windows *DISPLAY* variable would be insufficient, since the GUI instance has to start on one monitor for one user, then move to another monitor for a second user, etc. without reinitializing the tool.

Another job assigned to proxies is to spawn, manage, and communicate with auxiliary programs called *watchers*, each of which operates in the temporary directory for a tool instance and "notices" any files created or updated by a tool. These files are mapped to task arguments according to a configuration file constructed by the task envelope. The files can then be transferred back to the environment when the task is completed.

Besides the capability for the same tool instance to handle multiple tasks, another major difference between a SEL-like protocol and MTP's UNI cases, at least with respect to environment frameworks similar to the Oz architecture, is forking of the envelope and, indirectly, the tool by a proxy client — often not on the same machine as the user client — which could result in unnecessary communications overhead. MTP could easily be modified to default to a proxy on the same machine as the user client, and even some of the user and proxy client functions could be merged so that a separate proxy would not be needed when **`host`** and **`architecture`** specifications are not supplied and/or match the user client machine.

4.3. Task Execution

The most significant remaining issue that must be resolved to complete the design of our new protocol is the way in which the execution of envelopes is accomplished, in the manner of the *loose wrapping* concept. A typical MTP task execution steps through the sequential phases listed below:

1. A **reservation** phase, in which a tool session is acquired on behalf of the task and its associated user. This is carried out in according to the session mechanism explained in Section 4.1.

2. An **initialization** phase, in which the objects/files from the environment are made available to the tool and any other parameterization functions are performed. We have employed for this purpose a standard envelope template, which accepts as its parameters: pathnames corresponding to file attributes in OZ's object management system; the path to a dedicated temporary directory that is created when the tool is started up and within which it normally operates; and some additional information used for internal housekeeping. The filename of this envelope is given by the tool declaration in its `envelope-name` field.

   The envelope is forked by the relevant proxy client, which sets up UNIX pipes for communication. The first job of the shell script is to copy the files into the tool's dedicated directory, thus making them visible to the tool; then any series of shell commands can be inserted, to perform whatever customization is necessary; finally, via the pipes, a sequence of text messages is sent to the proxy, to be displayed to the user inside a task-specific pop-up window. These messages may include the values of primitive attributes from OZ's objectbase, and are intended to direct the loading of the files from the temporary directory into the memory of the application and otherwise instruct the user as to what to do. For example, the text presented in the window might indicate the command line or the mouse action that the user should enter to get started on the task, although the details of performing the task itself are usually left to the user's own devices.

   Although we would have preferred a totally automatic loading procedure, as accomplished by SEL, that it is hardly possible given the inherent restrictions of the Black Box model: MTP tools are already running before the execution of any envelope (therefore they cannot be initialized according to the individual tasks) and, in general, only human users can directly interact with them through their user interface; moreover, we cannot assume any special facilities on the part of the tool for simulating user input, and redirecting "stdin" is generally insufficient for GUI tools. However, the envelope, via messages to the pop-up window, may still provide assistance and guidance to the users in a practical and convenient manner.

   A Grey Box variant of MTP could overcome this drawback, since the tool's programmable facilities could act in collaboration with the envelope, producing and exchanging messages that would be interpreted as directives to be executed by the tool. (Some Grey Box experiments have been conducted using SEL; see Section 5.2.) In the White Box case, this issue can usually be avoided entirely.

3. An **operation** phase, which includes free use of the tool with all its features, including manipulation of the loaded data. There is no restriction on the use of the tool, because it is accessed directly and not through any intermediary medium. The only requirement of the MTP protocol (that cannot, however, be enforced in the Black Box case) is that the execution must not be terminated through the tool's own internal command, menu button or procedure, but only via the environment's `CLOSE-TOOL` command. In addition, both MTP and SEL assume that users do not access the "hidden" file system sereptitiously, e.g., loading files into the tool outside the workflow, although there is nothing beyond an obscure organization to prevent them from doing so.

4. One or more **data recording** phases may interleave with other actions, whenever the user saves temporary results of the work he/she is performing (the tool updates the copies of the files kept in its own temporary directory, and not those internal to the environment). Such events are monitored by the proxy client's watchers. A table of updated files is maintained in the proxy and used in the next phase.

5. The **conclusion** of the task, at which point control of the tool is released (with respect to this task). The user is required to designate the task's completion as either a `success` or a `failure`, via buttons in the pop-up window, and the data resulting from the execution is stored back in the environment only if the user considers the task successfully completed.

   SEL expects the envelope to automatically capture the return code of the tool after the user decides to close it, but in MTP the tool remains indefinitely active; therefore the only means of ending an individual task is to let the user decide when his/her work is finished and to provide a way to communicate this fact (and how to handle the results) to the envelope. Other differences are that SEL file updates are permanent, regardless of the success or failure status: actually, SEL may return any value in a range determined by the encapsulating task, each of which will result in different consequences following the task. A similar facility could be added to MTP.

4.4. Wrappers' Structure

Envelopes provide a very flexible approach to tool integration. Consisting of either standard scripts in some scripting language (as we have employed for MTP), or augmented variants of the scripting languages that provide primitives to handle interfacing to the environment and its data repository (as in SEL) — or possibly even written in a conventional programming language, wrappers offer programmable facilities that can handle the different needs and idiosyncratic properties of a wide range of external applications in a convenient and uniform way.

MTP uses two kinds of envelopes: the first is executed in response to the `OPEN-TOOL` command, whereas the second operates at the granularity of the individual task. The latter is concerned mainly with preparing and loading the data that must be processed by the program during the associated task; the former is used to perform customization of the tool, in order to present it to the user(s) in the correct state, in relation to the characteristics of the system and the work model indicated by the `multi_flag` specification. This kind of customization script is usually very simple — no more than a few lines of straightforward shell commands — but sometimes may be quite complicated, accounting for complex interactions with the environment through watchers, and sometimes even for the invocation of other auxiliary (simpler) scripts that perform supplementary bookkeeping or actions in response to particular states of the application. An example of a relatively complicated case is shown in Figures 9 and 10.

The envelope writer must be a relatively skilled shell programmer with some knowledge of the purpose and the functions of the wrapping protocol to be able to easily set up the scripts. The burden might be lowered somewhat if MTP were to extend the scripting language with special-purpose primitives, perhaps somewhat different sets to accommodate each of the four work models. However, the experience gained with SEL shows that even with such primitives the scripts are not exactly trivial, since the intrinsic specificity of the applications command ad hoc treatment for each case.

Language extension would be useful mainly to abstract and parameterize those operations that must be carried out in a repetitive manner for any application; this seems more plausible with the data interface between the tool and the environment, rather than with the adaptation of their reciprocal behavior. Consider the example shown in Figures 11 and 12: some of the shell commands, those marked with the comment `# always`, must always be present in any MTP task-related envelope; others, indicated by the comments that contain the words `FILE parameters`, are needed to handle certain types of incoming data, and are similar but not identical in all the envelopes. These two sets of commands together contribute to preparing the data involved in the activity.

The other shell commands, marked by the `# tool-dependent` comments, are concerned with operating the tool towards the goal of the task at hand. It is clear that in the general case the size and the complexity of this last set is dependent on the wrapped application, of the supported work model, and (especially if a lot of direct interaction with the user is necessary) of the task to be performed, while the former two sets are relatively independent of all these factors; hence it would be easier to invent scripting-language extension facilities to express them. However, it would also be possible (and desirable) to define some ad hoc constructs for use in those tool-dependent statements that communicate to the user the actions that he/she should perform, e.g., to carry out the loading of task arguments into the tool instance, during the initialization portion of an MTP task (see Section 4.3).

```
#!/bin/sh
#initialize variables
SERVER_PID=-1
CLIENT_PID=-1

# look if already hooked to the environment directory
FOUND=`find . -name linkfile -print`

# if environment dir. is not found
if [ "x$FOUND" = "x" ]  #no marvel_server active
then
        # prompt the user a request to
        # provide path for the environment dir.
        PFOUND=`find . -name prompt -print`
        if [ "x$PFOUND" = "x" ]
        # wait for reply and then start server + client
        then
            echo "type the path for the environment:" >> fake_prompt
            echo 1 >> fake_prompt
            mv fake_prompt prompt # create prompt file in the
                                  # tool dir. to the benefit
                                  # of the watcher
            # Watcher takes care of the prompt to obtain an
            # answer from the user; look out for that answer
            FOUND=`find . -name reply -print`
            while [ "x$FOUND" = "x" ]
            do
                    sleep 7
                    FOUND=`find . -name reply -print`
            done

            # read the reply file with the path
            # for the environment dir.
            read ENV_DIR < reply
            # create a link to the environment dir.
            ln -s $ENV_DIR linkfile
            rm reply
            # invoke marvel server and a client
            echo 1 >> clients
            /proj/oz/current/bin/marvel_server linkfile &
            SERVER_PID=$!
            echo $SERVER_PID >>server_pid
            sleep 1
            /proj/oz/current/bin/marvel -x linkfile &
            CLIENT_PID=$!
```

Figure 9. Example initialization script for a multi-user client/server tool, Part 1

```
        # prompt is already provided by another instance of
        # marvel_script; wait for it to start up the server
        # and then start up a client only
        else
                FOUND=`find . -name linkfile -print`
                while [ "x$FOUND" = "x" ]
                do
                        sleep 7
                        FOUND=`find . -name linkfile -print`
                done
                /proj/oz/current/bin/marvel -x linkfile &
                CLIENT_PID=$!
                read CLIENT_NUMBER < clients
                CLIENT_NUMBER=`expr $CLIENT_NUMBER + 1`
                echo $CLIENT_NUMBER > clients
        fi

# server is already active: go ahead and start a client
else
        /proj/oz/current/bin/marvel -x linkfile &
        CLIENT_PID=$!
        read CLIENT_NUMBER < clients
        CLIENT_NUMBER=`expr $CLIENT_NUMBER + 1`
        echo $CLIENT_NUMBER > clients
fi

CURR_DIR=`pwd`
# trap a request to kill this marvel component and
# invoke close_marvel_script to take care of this task.
trap '/u/astor/gv/THESIS/Rivendell/oz0/close_marvel_script $CURR_DIR
$CLIENT_PID; exit 1' 2
wait
```

Figure 10. Example initialization script for a multi-user client/server tool, Part 2

In Figure 12 these messages are implemented simply with `echo` commands prefixed by a common string (`#***#`); the output is redirected through pipes maintained between the envelope and the proxy client that initiated it, and the proxy is in charge of displaying the messages to the user in the pop-up window that is associated with each task. One could certainly imagine more sophisticated facilities for guiding the user, but where the tool integrator would prefer to avoid implementing them in shell.

5. Integration Examples

To test the facilities described in the previous sections, we have used several available in-house applications and off-the-shelf tools. The purpose of these tests was to gain confidence in the viability of the new MTP protocol, and in particular to challenge its ability to accommodate a wide range of variability in the nature of the wrapped applications.

Therefore, we have tried to define the degree of integration that can be reached and to identify limitations (either linked to the characteristics of the tool category under examination, or specifically to the adequacy of our support to the single cases) or unresolved problems we need to address during future development. The applications we used as examples were:

- `idraw` as a `UNI_QUEUE` tool, where tasks are queued for one-at-a-time execution (the same user may submit tasks from multiple clients, and the user interface is transferred among monitors as needed);
- `emacs` as a `UNI_NO_QUEUE` tool where steps are not queued but may overlap (typically on a single monitor);
- A Lisp-based natural language processing system called *FUF* as a `MULTI_QUEUE` tool, where steps are queued for one-at-at-time execution (and the UI is transferred among users participating in the same session as needed); and
- Marvel, the predecessor of Oz, as a `MULTI_NO_QUEUE` tool (that supplies its own clients for multiple users).

5.1. UNI_QUEUE: idraw

idraw [38] is a popular public-domain drawing tool, commonly used to develop pictures and diagrams stored in a postscript form; it provides an intuitive graphical user interface employing a well-known paradigm based on mouse movement and menu selection to operate on a virtual canvas shown within an X window. It is intended to be single-user; although *idraw* does support multiple buffers, we ignored that feature here, and treated the system as if it were necessary to save the current document before loading a different one. The limited use of *idraw* serves

```
#!/bin/ksh
#input parameters:
#                  $1 tool dir.    <-----  MTP additional parameter
#                  $2 C file       <-----  NOTE: FILE parameter
#                  $3 analyze status<----  Literal
#                  $4 analyze log file<--  NOTE: FILE parameter
#                  $5 compile status<----  Literal
#                  $6 compile log file<--  NOTE: FILE parameter
#                  $7 rule identifier<---  MTP additional parameter
#                  $8 client identifier<-  MTP additional parameter

cp $2 $4 $6 $1          # copy all FILE parameters in the tool dir.
CFile=`basename $2`                 # for all the file parameters
AFile=`basename $4`                 # for all the file parameters
CompileFile=`basename $6`           # for all the file parameters
CPath=`echo $1/$CFile`              # for all the file parameters
APath=`echo $1/$AFile`              # for all the file parameters
CompilePath=`echo $1/$CompileFile`  # for all the file parameters

F_LIST_DUMMY=$1/filelist_tmp            # always
F_LIST=$1/filetable                     # always
touch $F_LIST_DUMMY                     # always

echo $8 $7 $CFile $2 >> $F_LIST_DUMMY # for all the file parameters
echo $8 $7 $AFile $4 >> $F_LIST_DUMMY # for all the file parameters
echo $8 $7 $CompileFile $6 >> $F_LIST_DUMMY
                                      # for all the file parameters

FOUND=`find $1 -name filetable -print`          # always
if [ "x$FOUND" = "x" ]                          # always
then                                            # always
        mv $F_LIST_DUMMY $F_LIST                # always
else                                            # always
        F_LIST_CAT=$1/merge_list                # always
        cat $F_LIST_DUMMY $F_LIST > $F_LIST_CAT # always
        rm $F_LIST_DUMMY                        # always
        mv $F_LIST_CAT $F_LIST                  # always
fi                                              # always
```

Figure 11. Example task script for a multi-tasking tool, Part 1

```
# tool is emacs
echo \#***\#TYPE: CTRL-xf $CPath          # tool-dependent : load code file
if [ $5 = "NotCompiled" ]                 # tool-dependent
then                                      # tool-dependent
        echo \#***\#TYPE: CTRL-x 2 # tool-dependent : display new buffer
        echo \#***\#TYPE: CTRL-xf $CompilePath
                                   # tool-dependent : load analyzer logfile
fi                                        # tool-dependent
if [ $3 = "NotAnalyzed" ]                 # tool-dependent
then                                      # tool-dependent
        echo \#***\#TYPE: CTRL-x 2 # tool-dependent : display new buffer
        echo \#***\#TYPE: CTRL-xf $APath
                                   # tool-dependent : load compiler logfile
fi                                        # tool-dependent
```

Figure 12. Example task script for a multi-tasking tool, Part 2

as an example of the category of programs where such restrictions are inherent. From our point of view, *idraw* presents some additional features of interest since it fulfills our definition of heavy-weight tool: there is a relatively long initialization time following its invocation [4].

In our experiment, we employed a distinct task, parameterized by a file attribute from Oz's objectbase, to construct a complete figure or to allowing editing of an existing figure stored in that file, with the details of the drawing left to the creativity and expertise of the user. That is, a task's envelope would send a message to be displayed in the pop-up window, telling the user to load a file with a particular pathname, and briefly instruct the user regarding the purpose of the drawing to be constructed for that file. The user was responsible for using *idraw*'s normal command to later save that file, prior to announcing the conclusion of the task. This accounts for a simple interaction model that is common practice in the use of such kind of tools; however, it would alternatively be plausible to invent tasks and corresponding envelopes to operate at a much finer level of granularity, for example, "select the line icon and insert a vertical line two inches to the left of the triangle", but we doubt this would be useful (except perhaps as part of a tutorial in the use of a system devoted to the management of graphic documents).

The construction of the corresponding wrapper, and of wrappers for most `UNI_QUEUE` applications, is actually very simple: the only tool-dependent statements are aimed at instructing the user on how to load the input file and (optionally) on what he/she must do with it. The actual envelope is shown in Figure 13; note its similar structure to the task-specific envelope (for a `UNI_NO_QUEUE` tool) in Figures 11 and 12.

```
#!/bin/ksh
#input parameters:
#                   $1 tool dir.        <------ MTP additional parameter
#                   $2 file             <------ NOTE: FILE parameter
#                   $3 message          <------ Literal
#                   $4 rule identifier  <------ MTP additional parameter
#                   $5 client identifier<------ MTP additional parameter

cp $2 $1                      # copy all FILE parameters in the tool dir.
FileName=`basename $2`        # for all the FILE parameteres
FilePath=`echo $1/$FileName`  # for all the FILE parameteres

F_LIST_DUMMY=$1/filelist_tmp          # always
F_LIST=$1/filetable                   # always
touch $F_LIST_DUMMY                   # always

echo $5 $4 $FileName $2 >> $F_LIST_DUMMY
                                      # for all the file parameters

FOUND=`find $1 -name filetable -print`                # always
if [ "x$FOUND" = "x" ]                                # always
then                                                  # always
        mv $F_LIST_DUMMY $F_LIST                      # always
else                                                  # always
        F_LIST_CAT=$1/merge_list                      # always
        cat $F_LIST_DUMMY $F_LIST > $F_LIST_CAT       # always
        rm $F_LIST_DUMMY                              # always
        mv $F_LIST_CAT $F_LIST                        # always
fi                                                    # always

# tool is idraw
echo \#***\#TYPE: CTRL-O      # tool-dependent: accesses the loading cmd.
echo \#***\#SELECT $FileName       # tool-dependent : menu choice of file
echo \#***\#CLICK on "Open" button       # tool-dependent : performs load
# display to user the message attached to
# this instantiation of the task
if ["x$3" != "x"]                        # tool-dependent
then                                     # tool-dependent
        echo \#***\#$3  # tool-dependent : displays task-specific message
fi                                       # tool-dependent
```

Figure 13. Task envelope for the idraw application

A few words are in order regarding our intentionally restrictive use of *idraw*: we had some trouble finding a good candidate for the most basic `UNI_QUEUE` category, among the interactive tools we had on hand for testing (SEL seems adequate and completely satisfactory for non-interactive tools, such as compilers, that must be restarted for each new set of arguments anyway); *idraw* on the other hand seemed to have many of the properties that we were looking for in a `UNI_QUEUE` candidate. However, we recognize that it would normally be deemed `UNI_NO_QUEUE`, because of its intrinsic multi-buffering capability (see Section 5.2). Further, one could imagine employing *idraw* in a multi-user context, where one user starts a picture and others add to and finish it, analogous to the work model in Section 5.3, in which case *idraw* could even be designated `MULTI_QUEUE`.

Given all of the above, one may have the impression that perhaps the `UNI_QUEUE` category is not really necessary. However, we expect that environment builders will discover cases where they intend a tool to be used in a certain restricted way within the workflow, and enforcement of `UNI_QUEUE` would prove useful.

In general, `UNI_QUEUE` appears suitable to deal with those applications that do not present any multi-tasking capability and do not seem particularly adaptable to multiple users, but are most conveniently handled as persistent tools. The main advantages of persistence for this class of tools, and the most valuable improvements introduced by MTP's loose wrapping compared to tight wrapping as in SEL, is the reduction of startup overhead (since the tool need be invoked only once) and the user can run ordered sequences of tasks on the same instance of the program without losing its internal state.

5.2. UNI_NO_QUEUE: emacs

emacs [33] is one of the most readily available and widely used text editors; its sophisticated functionality and features make it a very useful tool, which nearly reaches in itself the status of a single-user programming environment. All of its commands are expressed with sequences of keystrokes, augmented with mouse pointing and selection; its latest versions also support menu selection, at least for its main features. One of the most useful properties of *emacs*, and one of the most important for us with respect to this discussion, is its buffering capability. This enables the user to operate simultaneously on multiple files, keeping several buffers in the background and switching among them on command. Coupled with the ability to split the display and hence show more than one of the buffers, this feature is of great use to perform complex and incremental editing sessions that involve as many different data sets as needed.

Many users would prefer to use *emacs* in the natural fashion available outside a process-centered or otherwise task-oriented environment framework, which is to create and kill buffers, load and save files, and cut and paste among buffers/files, as the urge arises during perhaps very long work sessions [5]. *emacs* demonstrates the most obvious limitation of conventional Black Box wrappers, in which some peculiarities of the application do not fit well with the protocol's design and are left unsupported, but it is nevertheless possible to integrate the program in some form.

MTP's `UNI_NO_QUEUE` class allows for overlapping multiple tasks that involve loading various buffers of the same executing *emacs* instance with the desired files for the user's editing sessions. MTP then employs watchers to allow mapping of each modified file to the corresponding task and hence discriminates what file attributes must accordingly be modified inside the environment at the end of the task. The use of the pop-up window during the initialization and conclusion phases of each task effectively isolates the overlapping tasks, in the sense that data flow and status are independent.

In our experiment, we employed individual tasks, parameterized by file attributes, to edit programming language or documentation files; the details of the programming or writing were the concern of the user. That is, a task's envelope would display a message on the pop-up window telling the user to load the file with a given pathname, and perhaps briefly explain to the user the purpose of the code or prose in that file. Rather than simply asking the user to edit, the envelope might request the user to repair the syntax errors found during the last compilation — by sending a file containing those error messages to another buffer as part of the same task. The complete script of an *emacs* wrapper of this kind is shown in Figures 11 and 12; it performs the loading of a C source file together with the results of the last analyzer (*lint*) and compiler runs, in case they had generated some error messages. Again, the user must give *emacs*'s normal command to save the source file. He/she may choose to indicate that the completion of the task has been successful, by committing changes to the environment's repository via the `success` button in the pop-up window, or not to save his/her work, by selecting the `failure` button (which has the effect of withdrawing whatever intermediate saves were performed during the work and noticed by the watchers). As with *idraw*, we did not consider finer-grained tasks such as "add a new floating point variable to function f and initialize it to *pi*", but the implementation supports them.

A previous attempt to extend Black Box enveloping had focused on *emacs* as a test case, and tried to resolve the problems posed by the desired incremental data exchange with the environment. This previous attempt exploited a facility not provided by most tools: an extension language. *emacs*' extension language, called *E-Lisp*, allows users to define their own new functions and commands, and thus customize *emacs* to their applications.

Ad hoc *E-Lisp* functions were coupled with an augmented version of SEL, to effect a Grey Box integration, where the environment could perform loading of additional files into the same *emacs* instance at any time and discern which files had been updated. No special effort was required by the user, in contrast to the attention he/she must pay to MTP's pop-up window. This was achieved using one wrapper for the entire session, which dealt with addition of new buffers as new tasks were submitted, rather than using a separate wrapper per task. There was a major drawback to this approach, however: only one final status result could be returned to the environment, when *emacs* and its wrapper terminated, and all files were effectively recorded into the environment's repository at this same moment. In other words, it was not possible to treat separately the different sets of data acquired throughout the work session — a central feature of MTP.

Later during the development of MTP, we looked at *E-Lisp* again to pursue Grey Box integration. Ad hoc *E-lisp* functions implemented a direct interface between *Emacs* and the watcher utility, and also completely automated the initialization phase of the tasks. The conclusion phase, particularly the choice of the `success` or `failure` return status for the separate tasks run on the same instance of *emacs*, is still an explicit responsibility of the user even under this paradigm.

In general, `UNI_NO_QUEUE` appears appropriate for applications with some internal multi-tasking, multi-buffer or multi-context capability, but still not particularly useful or desirable for multi-user access. The main advantage of persistence for this class of tools is that the user can run partially ordered tasks on the same instance of the program, again without losing its intermediate state information, and possibly allowing for sharing or splicing of intermediate results among the tasks. Note there is no explicit means for directing, from the environment, intentional cut and paste or other sharing among tasks; the tool integrator can, however, prevent such cut and paste by designating the tool as `UNI_QUEUE`. Cut and paste can be directed within a single task that simultaneously presents multiple file arguments to the tool, with the envelope's messages to the pop-up window instructing the user what to do.

5.3. MULTI_QUEUE: FUF

FUF is a sophisticated unification-based tool running on top of Lisp and is used, among other things, in the field of Natural Language Processing for the generation of sentences from corresponding syntactic data structures [12]. It defines hierarchical procedures that apply in sequence one or more separate layers of unification rules to its input structures — as well as to the new structures produced by each step of the procedure — in order to obtain as output all the valid surface forms, under the constraints posed by the language rules. *FUF* is a typical Lisp-based interpreted application, in that it that supports various kinds of interactive tracing facilities and has the option to test and execute various data and program files, by loading and swapping them on the fly. As with most interpretive tools, it maintains sufficient information in memory to reflect the progress of its elaboration through the series of commands issued to it since start-up. Moreover, like many query systems constructed on top of Lisp, there is a long startup time and it engages a considerable amount of system resources (notably main memory and swap space) and thus qualifies as a heavy-weight tool.

One of the main reasons for this choice as our exemplar `MULTI_QUEUE` tool is that it is easy to imagine a scenario in which, in order to process some data with *FUF*, multiple unification procedures are needed, each of which is the responsibility of a different member of a development group. Our paradigm could facilitate the testing and execution of the various phases of the project through a (modest) form of groupware: sequentially, each developer would load into *FUF* its own program, run it on the appropriate data and refine it as much as needed, and produce at the end an output that is also the input for the next step, also leaving the system in the correct state to begin the following task. The final outcome of the overall workflow would be produced by a single instance of the system and as the result of the collaboration of several users. Analogous collaborative work models could be applied to other programs, which outside the MTP framework could not be employed in this way.

The envelopes we devised for these kind of tasks are devoted to load within the memory of *FUF* a specific unification program, and to handle the correct system configuration for it, by asking the user to type the appropriate Lisp commands. The user might know little, if anything, about the configuration issues involved: he/she needs only to follow the instructions appearing in the pop-up window, since each envelope is specialized towards a separate portion of the group work. After this initial customization, the user is left completely free to query *FUF* and interact with it in the typical fashion of Lisp-based interpretive applications. Any files produced as result of this operation may be imported into the objectbase via the `success` vs. `failure` choice that ends the task, as already described above.

From a general point of view, the `MULTI_QUEUE` category allows the reuse of single instances of such computationally expensive programs throughout a series of tasks. Another important point in favor of supporting this class is that the information retained in the tool's memory space (and not necessarily persistently on disk) represents both the current state of the system and the history of its past performance, and is generally necessary for generating the answer to new queries. This makes even more valuable the ability of the `MULTI_QUEUE` work model to support applications with long-duration work sessions that go beyond any single task, and to ensure common access to them to any set of clients.

The most relevant consequence of the creation of this category is indeed that, by exploiting Activity Queues and the *xmove* facility that achieves passing of control over the user interface among users involved in a session, it allows us not only to conveniently integrate a vast and peculiar family of tools, but also to actually modify at the same time their intrinsic single-user nature and extend their use along the lines described above. We consider this as one of the most interesting and meaningful results of this work.

xtv [1] provides a related facility, also in a Black Box fashion, but at a finer level of granularity and without any particular consideration for workflow. *xtv* simultaneously displays the X user interface of a more-or-less arbitrary X Windows tool to multiple users, and provides its own floor-passing scheme with respect to which one user has control of the mouse and keyboard at any given moment. If we were to employ *xtv* instead of *xmove*, then most of our `MULTI_QUEUE` tools could nominally become `MULTI_NO_QUEUE` as far as MTP were concerned, but still lacking facilities for truly concurrent work.

5.4. MULTI_NO_QUEUE: Marvel

We decided to use as a testbench for this category MARVEL [18], [8], the predecessor of OZ, which is also a multi-user process-centered environment, but with the difference — not relevant to this paper — of supporting only one process at a time, i.e., of being centralized, with no notion of inter-process interoperability. The main reasons for this choice were the familiarity we have with MARVEL as a complete multi-user system, the in-house availability of the application in a ready-to-run state, and its stability compared to using OZ itself as the "tool".

MARVEL, as a typical client/server system (and unlike most applications based on peer-to-peer architectures), poses, in the most general case, the problem of treating differently the `OPEN-TOOL` command initiating a session, when it is necessary to start-up both the tool's server and a client, from those subsequently issued to join the session, which obtain further copies of only the MARVEL client. Conversely, the last `CLOSE-TOOL` command in a session must deal with shutting down the tool's server. Moreover, since one can optionally employ a daemon that automatically starts up the MARVEL server with the first client and automatically shuts it down when the last client exits, MARVEL can also be used to simulate the behavior of non-hierarchical architectures, which do not need special treatment for the activation of its first and last components.

The intrinsic difficulties of dealing with these issues were solved in the context of the envelope indicated by the `path` field of the tool declaration and invoked by the `OPEN-TOOL` command. MARVEL's initialization envelope is shown in Figures 9 and 10. The shut-down of MARVEL's server was also handled by this envelope, but MTP could easily be extended to handle a separate envelope triggered by the `CLOSE-TOOL` command. MTP, with its `MULTI_NO_QUEUE` class, is therefore able to support a generic multi-user tool, by forking and providing copies of the program to every participant in a session, as required by its structure.

During our experiment with MARVEL, we devised tasks that perform operations within an in-progress workflow (as with OZ, the product data and process state is persistent across sessions as well as tasks within a session). The wrappers instruct the user, with the usual pop-up messages, on how to use MARVEL's GUI to browse the objectbase, inspect the process definition task set, etc. It is also quite simple to ask users to initiate specific MARVELtasks (represented as rules) on certain objects; a template for such scripts is shown in Figure 14.

This raises the possibility of an OZ meta-process that controls one (or more) MARVEL process(es), effecting a form of hierarchical workflow system. This could potentially address a certain limitation in MARVEL, shared by OZ, that relationships among tasks within a process are formed only with respect to satisfying local constraints, and there is no global topology or grand view [20]. However, that grand view could feasibly be defined by the meta-process, by directing the workflow among the entry points of aggregate tasks, while the process itself directs only the workflow among primitive tasks. Further discussion of this idea is outside the scope of this paper.

```
#!/bin/ksh

#input parameters:
#                   $1 tool dir.        <------ MTP additional parameter
#                   $2 rule name        <------ Literal
#                   $3 object id(s).    <------ Literal
#                   $4 rule identifier  <------ MTP additional parameter
#                   $5 client identifier<------ MTP additional parameter

# no file involved
F_LIST_DUMMY=$1/filelist_tmp                               # always
F_LIST=$1/filetable                                        # always
touch $F_LIST_DUMMY                                        # always
FOUND=`find $1 -name filetable -print`                     # always
if [ "x$FOUND" = "x" ]                                     # always
then                                                       # always
        mv $F_LIST_DUMMY $F_LIST                           # always
else                                                       # always
        F_LIST_CAT=$1/merge_list                           # always
        cat $F_LIST_DUMMY $F_LIST > $F_LIST_CAT            # always
        rm $F_LIST_DUMMY                                   # always
        mv $F_LIST_CAT $F_LIST                             # always
fi                                                         # always
# tool is Marvel
echo \#***\#SELECT $2 rule within Rules menu
                              # tool-dependent : choose rule
echo \#***\#TYPE: $3 (as parameter for $2)
                              # tool-dependent : rule parameter(s)
echo \#***\#TYPE: return       # tool-dependent : fire rule
```

Figure 14. Task envelope for the Marvel application

In general, there are some important differences between the integrations of collaborative and non-collaborative tools, which must be taken into account when considering the capabilities of `MULTI_NO_QUEUE` integration. In the non-collaborative case, in which each user works in isolation from the rest (a multi-user database management system is a typical example) the different requests are handled by the intrinsic multi-tasking capability of the tool and conflicts among overlapping argument sets are sporadic and resolved either before the arguments are passed to the tool by a conventional concurrency control mechanism provided by the environment (Oz, by default, implements atomicity and serializability among individual or multi-step tasks delimited as transactions [17]), in the case of data from the environment's repository, or by the tool's own policies, in the case of an external repository specific to the tool (e.g., the database volume in the case of a database management system).

In the collaborative case, instead, even though most of the multi-user machinery is necessarily offered by the wrapped tool itself, the problem of shared use of data becomes more problematic. A simple example is that of a multi-user editor [10], employed in the context of a groupware task: the program itself permits and is able to deal with concurrent modification of its internal data, but from the viewpoint of environment's data repository it is necessary to support a concurrency control policy that allows multiple writers of the object containing the edited file (this is achieved in Oz by defining and loading application-specific concurrency control policies, written in a notation [16] that permits definition of "cooperative transactions" [19]). Concurrency control, per se, is not in the strictest sense a part of the wrapping facility, but is nevertheless essential in order to fully integrate this class of tools.

6. Contributions and Future Work

We have fully implemented all the facilities discussed in this paper, except as noted in the text, and support the tools we chose as test cases for MTP's four work models. Future experiments should encompass more exacting tools. Nevertheless, the completed experiments — all of which run quite satisfactorily — have demonstrated the feasibility of employing wrappers for persistent tools within a process-centered environment framework.

Further, we have introduced several useful concepts for the domain of Black Box tool integration, including a categorization of tools into families with diverse multi-user and multi-tasking capabilities, the notions of multiple complementary enveloping protocols and of loose wrapping, the idea of interfacing with already-executing persistent instances of external programs, and the ability to extend the functionality of intrinsically single-user tools to partial sharing of their data and computational resources. The support for directing tool execution to a proxy client, when the `host` or `architecture` field is non-empty, has recently been extended to SEL, since the problems of host licenses and architecture and operating system dependencies apply even to the rather mundane tools (compilers and the like) that are supported by previous approaches to Black Box enveloping.

The `MULTI_NO_QUEUE` model presented here is best suited to *asynchronous* groupware applications, where users enter and leave the tool as they please. There is as yet no notation in Oz's process modeling language to define the circumstances under which tool sessions should be automatically opened/joined and closed/left, which would still allow for asynchronous groupware but more closely integrate sessions into the workflow in a similar manner to how individual tasks within those sessions are supported. In-progress work also includes process modeling and execution support for *synchronous* groupware in which multiple users perform a task together at the same time [6]. For example, the `multi-flag` field, originally introduced for MTP, is now used within SEL to identify tools that support this kind of collaboration, so that the system can simultaneously submit the task and its arguments to the clients corresponding to multiple designated users [5].

We are also working on extending the MTP approach to exploit Oz's multi-site, multi-server, multi-process orientation. The implementation described here operates only within a single site, server, and process (i.e., a shared network file system is assumed and authorization issues are not addressed). We are working on tool modeling and infrastructure support to redirect execution to proxy clients attached to a remote server (potentially located at another Internet site), in cases where tools are available only in the remote environment (e.g., due to proprietary or licensing issues, or the need for a special-purpose machine). We have already developed process modeling notation to direct control over pending tasks to alternative users [35], which would be needed when it is inappropriate or technically infeasible for a remote user to receive the tool's GUI.

Another interesting direction, now in the planning stage, is to split off all tool management (for both MTP and SEL) from the Oz server into a separate component, to be called Rivendell, that would execute as another operating system process distinct from the server, user clients and proxy clients. This would lower the load on the server, simplify later replacement of the component within the Oz system (if desired), and ease the incorporation of both MTP and SEL facilities into other systems.

Acknowledgments

Prof. Kathy Mckeown provided the FUF application and served as the second reader for Mr. Valetto's Masters thesis [36]. Peter Skopp played a major part in designing and implementing the architectural changes needed to introduce proxy clients into Oz, a variant of which are used on a one-to-one basis to support low-bandwidth (modem) user clients [31]. George Heineman conducted the SEL Grey Box experiment involving overlapping tasks submitted to emacs, and developed the watcher utility as part of that effort. Richard Baldwin is working with Prof. Kaiser on SEL proxy clients and the other new ideas outlined above, except for the synchronous groupware facilities — which were developed by Issy Ben-Shaul.

Notes

1. The first use of the term "envelope" in this sense, that we know of, was with respect to the Istar system [11].
2. SEL and many of the other Oz facilities mentioned in this paper were originally developed for an earlier system called Marvel.
3. Proxy clients and user clients were initially referred to as Special Purpose Clients and General Purpose Clients, respectively [37].
4. About 15 elapsed seconds on a Sun SparcStation 10 workstation.
5. The second author has been known to keep the same *emacs* instance running for months.

References

1. Hussein M. Abdel-Wahab. XTV. http://www.cs.odu.edu/ waha_cit/XTV.doc/xtv.html.
2. *Transcending Boundaries: ACM 1994 Conference on Computer Supported Cooperative Work*, Chapel Hill NC, October 1994. ACM Press.
3. Naser S. Barghouti. Supporting cooperation in the MARVEL process-centered SDE. In Herbert Weber, editor, *5th ACM SIGSOFT Symposium on Software Development Environments*, pages 21–31, Tyson's Corner VA, December 1992. Special issue of *Software Engineering Notes*, 17(5), December 1992.
4. Israel Z. Ben-Shaul. An object management system for multi-user programming environments. Master's thesis, Columbia University, Department of Computer Science, April 1991. CUCS-010-91.
5. Israel Z. Ben-Shaul. *A Paradigm for Decentralized Process Modeling and its Realization in the OZ Environment*. PhD thesis, Columbia University, Department of Computer Science, April 1995. CUCS-014-95.
6. Israel Z. Ben-Shaul, George T. Heineman, Steve S. Popovich, Peter D. Skopp, Andrew Z. Tong, and Giuseppe Valetto. Integrating groupware and process technologies in the OZ environment. In Carlo Ghezzi, editor, *9th International Software Process Workshop: The Role of Humans in the Process*, pages 114–116, Airlie VA, October 1994. IEEE Computer Society Press.
7. Israel Z. Ben-Shaul and Gail E. Kaiser. A paradigm for decentralized process modeling and its realization in the OZ environment. In *16th International Conference on Software Engineering*, pages 179–188, Sorrento, Italy, May 1994. IEEE Computer Society Press.
8. Israel Z. Ben-Shaul, Gail E. Kaiser, and George T. Heineman. An architecture for multi-user software development environments. *Computing Systems, The Journal of the USENIX Association*, 6(2):65–103, Spring 1993.
9. Melissa Chase and Howard Reubenstein. An assessment of KBSA and a look towards the future. Technical Report RL-TR-92-163, Rome Laboratory, June 1992.
10. Prasun Dewan, editor. *Special Issue on Collaborative Software*, volume 6:2 of *Computing Systems, The Journal of the USENIX Association*. University of California Press, Spring 1993.
11. Mark Dowson. Integrated project support with IStar. *IEEE Software*, 4(6):6–15, November 1987.
12. Michael Elhadad. *Using argumentation to control lexical choice: a unification-based implementation*. PhD thesis, Columbia University, Department of Computer Science, 1993.
13. Christer Fernström. PROCESS WEAVER: Adding process support to UNIX. In *2nd International Conference on the Software Process: Continuous Software Process Improvement*, pages 12–26, Berlin, Germany, February 1993. IEEE Computer Society Press.
14. David Garlan and Ehsan Ilias. Low-cost, adaptable tool integration policies for integrated environments. In Richard N. Taylor, editor, *4th ACM SIGSOFT Symposium on Software Development Environments*, pages 1–10, Irvine CA, December 1990. Special issue of *Software Engineering Notes*, 15(6), December 1990.

15. Mark A. Gisi and Gail E. Kaiser. Extending a tool integration language. In Mark Dowson, editor, *1st International Conference on the Software Process: Manufacturing Complex Systems*, pages 218–227, Redondo Beach CA, October 1991. IEEE Computer Society Press.
16. George T. Heineman. Process modeling with cooperative agents. In Brian Warboys, editor, *3rd European Workshop on Software Process Technology*, volume 772 of *Lecture Notes in Computer Science*, pages 75–89, Villard de Lans (Grenoble), France, February 1994. Springer-Verlag.
17. George T. Heineman and Gail E. Kaiser. An architecture for integrating concurrency control into environment frameworks. In *17th International Conference on Software Engineering*, pages 305–313, Seattle WA, April 1995. ACM Press.
18. George T. Heineman, Gail E. Kaiser, Naser S. Barghouti, and Israel Z. Ben-Shaul. Rule chaining in MARVEL: Dynamic binding of parameters. *IEEE Expert*, 7(6):26–32, December 1992.
19. Gail E. Kaiser. Cooperative transactions for multi-user environments. In Won Kim, editor, *Modern Database Systems: The Object Model, Interoperability, and Beyond*, chapter 20, pages 409–433. ACM Press, New York NY, 1994.
20. Gail E. Kaiser, Steven S. Popovich, and Israel Z. Ben-Shaul. A bi-level language for software process modeling. In Walter F. Tichy, editor, *Configuration Management*, number 2 in Trends in Software, chapter 2, pages 39–72. John Wiley & Sons, 1994.
21. Simon Kaplan, editor. *Conference on Organizational Computing Systems*, Milpitas CA, November 1993. ACM Press.
22. Simon M. Kaplan, William J. Tolone, Alan M. Carroll, Douglas P. Bogia, and Celsina Bignoli. Supporting collaborative software development with ConversationBuilder. In Herbert Weber, editor, *5th ACM SIGSOFT Symposium on Software Development Environments*, pages 11–20, Tyson's Corner VA, December 1992. Special issue of *Software Engineering Notes*, 17(5), December 1992.
23. Balachander Krishnamurthy and Naser S. Barghouti. Provence: A process visualization and enactment environment. In Ian Sommerville and Manfred Paul, editors, *4th European Software Engineering Conference*, number 717 in Lecture Notes in Computer Science, pages 451–465. Springer-Verlag, Garmisch-Partenkirchen, Germany, September 1993.
24. John R. Nicol, C. Thomas Wilkes, and Frank A. Manola. Object orientation in heterogeneous distributed computing systems. *Computer*, 26(6):57–67, June 1993.
25. Reference Model for Frameworks of Software Engineering Environments: Edition 3 of Technical Report ECMA TR/55, August 1993. NIST Special Publication 500-211. Available as `/pub/isee/sp.500-211.ps` via anonymous ftp from nemo.ncsl.nist.gov.
26. David Notkin and William G. Griswold. Extension and software development. In *10th International Conference on Software Engineering*, pages 274–283, Raffles City, Singapore, April 1988.
27. Steven S. Popovich. Rule-based process servers for software development environments. In John Botsford, Arthur Ryman, Jacob Slonim, and David Taylor, editors, *1992 Centre for Advanced Studies Conference (CASCON)*, volume I, pages 477–497, Toronto ON, Canada, November 1992. IBM Canada Ltd. Laboratory.
28. James M. Purtilo. The POLYLITH software bus. *ACM Transactions on Programming Languages and Systems*, 16(1):151–174, January 1994.
29. Steven P. Reiss. Connecting tools using message passing in the Field environment. *IEEE Software*, 7(4):57–66, July 1990.
30. David S. Rosenblum and Balachander Krishnamurthy. An event-based model of software configuration management. In Peter H. Feiler, editor, *3rd International Workshop on Software Configuration Management*, pages 94–97. ACM Press, June 1991.
31. Peter D. Skopp. Process centered software development on mobile hosts. Technical Report CUCS-035-93, Columbia University Department of Computer Science, October 1993. MS Thesis Proposal.
32. E. Solomita, J. Kempf, and D. Duchamp. Xmove: A pseudoserver for X window movement. *The X Resource*, 1(11):143–170, July 1994.

33. Richard M. Stallman. Emacs the extensible, customizable, self-documenting display editor. In *SIGPLAN SIGOA Symposium on Text Manipulation*, pages 147–156. ACM, June 1981. Special issue of *SIGPLAN Notices*, 16(6), June 1981.
34. Ian Thomas. PCTE interfaces: Supporting tools in software-engineering environments. *IEEE Software*, 6(6):15–23, November 1989.
35. Andrew Z. Tong, Gail E. Kaiser, and Steven S. Popovich. A flexible rule-chaining engine for process-based software engineering. In *9th Knowledge-Based Software Engineering Conference*, pages 79–88, Monterey CA, September 1994. IEEE Computer Society Press.
36. Giuseppe Valetto. Expanding the repertoire of process-based tool integration. Master's thesis, Columbia University, Department of Computer Science, December 1994. CUCS-027-94.
37. Giuseppe Valetto and Gail E. Kaiser. Enveloping sophisticated tools into computer-aided software engineering environments. In *IEEE 7th International Workshop on Computer-Aided Software Engineering*, pages 40–48, Toronto Ontario, Canada, July 1995.
38. John M. Vlissides and Mark A. Linton. Unidraw: A framework for building domain-specific graphical editors. *ACM Transactions on Information Systems*, 8(3):237–268, July 1990.