# Enveloping Sophisticated Tools into Computer-Aided Software Engineering Environments (Research Paper)

Giuseppe VALETTO
Rank Xerox Research Centre
"Le Quartz"
6, Chemin de Maupertuis
38240 Meylan
FRANCE
giuseppe.valetto@xerox.fr
+33 76615086
fax: +33 76615099

Gail E. KAISER
Columbia University
Department of Computer Science
500 West 120th Street
New York, NY 10027
UNITED STATES
kaiser@cs.columbia.edu
(212) 939-7081
fax: (212) 666-0140

CUCS-029-94
21 November 1994

**Abstract**

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

**keywords:** Computer-Supported Cooperative Work, Environment Frameworks, Tool Enveloping, Tool Integration



[1]This work was conducted while Mr. Valetto was a graduate student at Columbia University. Prof. Kaiser is supported in part by Advanced Research Project Agency under Contract F30602-94-C-0197, in part by National Science Foundation Grant CCR-9301092, 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 Government, AT&T, Bull, IBM or Xerox.

# 1 Introduction

CASE environments usually support dialogues between commercial-off-the-shelf (COTS) tools and the environment framework. We identify three categories of integration methods, with respect to their approach to adapting the external applications to the environment:

- *White Box*, where a custom tool is implemented as part of a particular environment or a pre-existing tool's source code is directly modified to match the environment framework's interface. Custom tools may be prohibitively expensive to develop. In the latter case, the changes can often be implemented in a straightforward, repetitive manner, but nevertheless the source code must be available — perhaps an insurmountable difficulty when integrating COTS tools from independent vendors. The White Box approach is followed by several commercial message buses. PCTE[19] and similar standards probably require more effort in tool adaptation, but enable a higher scale of integration.

- *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 [18]. But relatively few tools provide such convenience. In principle, dynamic linking coupled with replacement of standard libraries (e.g., for I/O) might work, in principle, but it seems unlikely that arbitrary COTS tools would happen to fit a framework's communication protocols (for instance, a framework may expect tools to ask permission before accessing objects, so that a policy tool or concurrency control can be considered, and/or to notify a broadcast message server when updates have been completed, for propagation to other tools [14]).

- *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* [7] extract objects/files from the internal representation in the environment's repository, present these objects/files to their "wrapped" tools in the appropriate format, and provide the reverse mapping for updated data and tool return values. (In the sequel, we use the terms object and file interchangeably, since some CASE environments represent software artifacts as objects and some as files. Both objectbase and file system are referred to generically as the environment's data repository.) Envelopes can also be used in conjunction with Grey and White Box.

Our goal is to augment enveloping to apply to a much wider array of tools than previously. We concentrate on the Black Box model, since it is often the only choice as well as the most difficult.

Oz's Shell Envelope Language (SEL) [10] is typical of current Black Box enveloping facilities. [1] A tool integrator writes what are essentially shell scripts, using added constructs that handle the details of interfacing between the tool and data integration and repository services. (We employ the terminology of the "toaster" reference model [8].) An SEL envelope is associated with each primitive task (primitive tasks may be grouped into aggregate tasks). After parameters have been bound and other preliminaries completed, Oz's task management service directs that the named

[1] SEL and many of the other facilities mentioned in this paper were originally developed for Marvel, Oz's predecessor.

envelope be invoked on the appropriate arguments, including primitive values and/or files from the data repository and possibly also user-supplied literals. When the envelope terminates, it returns the results to task management, at which point the pending task continues. This is implemented in a client/server architecture, with task management, data integration and data repository services in a shared server and user interface and envelope invocation facilities supported by each client [6]. The server sends envelope names and arguments to the relevant user client for execution, and then handles other clients in a FCFS manner until the tool completes and its result arrives at the front of the server's request queue.

The SEL approach works well for UNIX utilities that accept all their arguments from the command line at invocation, read and write some files (whose names are given on the command line), and return a simple status code. SEL can also handle tools that return a sequence of literal values and/or subset the list of file names provided on the command line. Notice this does not preclude interactive tools such as word processing and drawing systems, since the tool's own user interface pops up on the user's screen when the client starts executing the tool. The user may then enter text or click menu items as desired, but the granularity of access to objects in the environment's data repository is the entire tool invocation.

There are numerous tools that don't fit this description, but may be highly desirable to integrate into CASE environments, including at least:

- 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.
- *Interpretive* tools, which typically maintain an in-memory state reflecting progress through a series of operations: Lisp applications, such as "Knowledge-Based Software Assistant" (KBSA) tools, are classic examples. Such tools may involve severe start-up overhead and command substantial system resources (we refer to them in the sequel as "heavyweight"). 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 single-user tools to (modest) groupware tools.
- *Multi-User* tools, such as conventional database systems. An important subclass is *Collaborative* tools, which directly support multiple users interacting with each other, such as WYSIWIS (what-you-see-is-what-I-see) utilities, IBIS decision support and Fagin-style document inspection tools, desktop video conferencing systems, etc. (see [12, 1] for more examples).

In this paper, we introduce a *Multi-Tool Protocol* (MTP), where *Multi* refers to submission of *multiple* tasks to the same tool instance and enabling of *multiple* users to interact with the same tool instance. Tool instances may execute 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 remote machines, e.g., when operating over a heterogeneous collection of workstations and servers but executables are available for only a restricted subset of the architectures 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

```
<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 1: New Tool Definition Notation

that can operate at the same time, common with COTS server licenses. Like SEL, by default MTP treats tools in a Black Box manner.

Our initial implementation has been completed as an extension to Oz [5]. Oz is a geographically distributed process-centered CASE environment that supports interoperability among autonomously defined processes.

## 2 Tool Modeling

The task management service needs to specify which tools require which protocol. In principle, every CASE tool could be invoked via the new MTP protocol, but we retained SEL as the default because we believe that MTP should be seen as complementary to SEL on a per-tool basis: Together, they address with greater specificity the peculiarities of diverse families of applications. We believe an approach based on *multiple enveloping protocols* is likely to achieve the greatest generality.

Oz's tool declaration notation has therefore been modified to include the new portion shown between square brackets ("[...]") in Figure 1, which is optional and may be omitted for SEL (and is ignored for SEL in any case).

The main fields included between brackets in the optional section 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. One use of an envelope during initialization might be to prompt the user for the pathname to an external database.
- `architecture` is used to indicate the machine architecture on which the tool (and its corre-

sponding envelope) is expected to run. [2]

- `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). When the `host` is not specified, the system refers to the `architecture` specification and separate configuration information, to retrieve a corresponding default machine on which the persistent tool and/or its envelope will be invoked.

- `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 given persistent 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 multi-user and multi-tasking capability, through the cross-product of two orthogonal dimensions:

  – **UNI vs. MULTI**: where MULTI indicates that the same instance of the program can be <u>shared</u> by several users, whereas UNI allows only for isolated work of each user on his/her own executing instance of the tool;

  – **QUEUE** vs. **NO_QUEUE**: where <u>concurrent</u> (overlapping) execution of multiple tasks with respect to the same tool instance is supported for NO_QUEUE but not for QUEUE, whether UNI or MULTI.

  It may seem counterintuitive to think of these dimensions as orthogonal. In the case of MULTI_QUEUE, multiple tasks on behalf of different users can share the same tool instance, but only one actually runs at a time (FCFS); for UNI_NO_QUEUE, multiple tasks can execute simultaneously in the same tool instance, 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 (locks and transaction management are outside the scope of this paper, see [6, 2]). 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.

We made no changes at all to Oz's task definition syntax [11], and our approach is intended to be orthogonal to the environment's mechanism for task definition.

[2] In principle, an envelope forked on one machine could invoke a tool on another, using UNIX `rsh` or a similar mechanism, but this would make it impractical to track the tool.

## 3 The Integration Protocol

We adopted what we call a *loose wrapping* approach, as opposed to the *tight wrapping* exemplified by SEL envelopes. The latter relies on complete encapsulation of all of the tool's actions inside the envelope, while the former is instead based on control of the tool's behavior (from the viewpoint of the environment), with the enveloping facility intervening only upon detection of some event relevant to the environment. Typical examples of such events include the invocation of an environment command that requires the tool to perform some task and a tool action that saves some files that should be recorded in the environment's data 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, rules 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 intermediate between conventional Black Box enveloping and a broadcast message server such as Field [15], where tools execute persistently but the server's concern is only for events of interest to tools and there are no separate "environment commands" that control tools. The Forest extension of Field controlled the propagation of event notifications among tools according to "policies" [9], analogous to Oz's task management services, but had no distinct environmental front-end. It also did not address our foremost requirement, to support multi-user tools, and few message buses support multiple users — ConversationBuilder's Mbus is a notable exception [13].

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 provide it with objects from the data repository, to connect to the user interface of the environment the wrapped program and to support and display the I/O flow between it and its user(s) — and those especially necessary given the nature of the four main tool categories of interest (i.e., the cross-product of UNI vs. MULTI and NO_QUEUE vs. QUEUE), on which we mainly concentrated:

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 persistency 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 coming from separate users, to ensure avoidance of deadlocks and starvation on the one hand,

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

Figure 2: Tool Session Template

and of unintended concurrency of several activities for programs that don't support any form of multi-tasking on the other hand.

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 task management services. Analogous extensions could be made to other CASE environment frameworks.

### 3.1 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 begins with an `OPEN-TOOL` command and ends with `CLOSE-TOOL`, as illustrated in Figure 2. A session's body is made up of a set of primitive tasks determined dynamically as the users carry out their work within the CASE environment. Each MTP-activity in the figure maps to an individual primitive task. Note that although the MTP-activities 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 selecting the corresponding identifier from their menu when issuing an `OPEN-TOOL` command. The current implementation does not provide any special support for access control, e.g., specifying which users should join a particular session; this is being addressed by current work in Oz process modeling. There is also no support for providing parameters for tool initialization from within the environment, which is less limiting than it sounds since the tasks that trigger task-specific envelopes do indeed accept parameters from the environment's data repository.

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 in the session.

Besides setting the duration of a specific tool instance and providing a context for sharing an application, sessions are involved 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 **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 CASE environment, resulting in tool invocation failures.) When an `OPEN-TOOL` command is issued, the system first checks whether the request is satisfiable given this constraint. If the boundary has been hit, the request is not serviced, but is recorded in the Session Queue; when an already running session is terminated by a `CLOSE-TOOL` command, 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 killed by an implicit `CLOSE-TOOL` command.

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 collected state of the executing tool, and even collaborating when this is feasible. In our MTP design, we stressed those facets intended to accommodate in a natural way those applications that are inherently designed for collaboration, or — in some sense an even more ambitious goal — to exploit in a multi-user and/or multi-tasking context those tools that, even if not commonly employed in that manner, the tool integrator considers adaptable to and promising for collaborative CASE activities.

Imagine that `USER1` opened `SESSION1` for persistent tool `TOOL1` and is executing `MTP-activityA`; a bit later, `USER2` requests `MTP-activityB`, also on `SESSION1`.

- If `TOOL1` is UNI_QUEUE or UNI_NO_QUEUE, `USER2` cannot join the same session but can start a new session if the number of concurrent sessions for that tool would not exceed the number of `instances` allowed. These two categories are designed for utilities deemed by the tool integrator to be inappropriate for sharing of resources or data, so that the environment can enforce this constraint among its users.

- If `TOOL1` is not inherently multi-user (as for most CASE tools), but is declared MULTI_QUEUE, only the most rudimentary form of sharing is possible: different users are allowed to access the same executing tool instance, but they must "take turns". In our example, the system holds `USER2`'s request in an *Activity Queue* until `USER1` completes `MTP-activityA`. `USER2` is not stuck, waiting for this request to be processed, but can still execute other tasks — or decide to abort and try again later. Oz's GUI already allows a user client to context-switch at will among in-progress task sequences.

  Albeit limited, this form of sharing can be usefully exploited in various collaboration scenarios. For example, consider multiple users of a CASE environment who are committed to independently 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, that would facilitate the execution of the whole project in all of its phases.

A "heavyweight" interpretive system exemplifies this sort of application, since the information retained in its memory space represents both the current state of the tool and the history of its past performance, generally fundamental to generating the answers to future queries. In such a case, each developer would execute serially his/her own subtask on the appropriate data, producing at the same time the input(s) for the next step(s) and putting the system in the correct state to begin the following procedure(s). At the end, the developers would have reached the desired final state and obtained the results based on the overall multi-step processing of the original data.

- If `TOOL1` is multi-user (whether or not collaborative), it would be declared as MULTI_NO_QUEUE. Then, `USER2`'s request is handled by the normal multi-tasking nature (e.g., client/server or peer/peer) of `TOOL1` — and `USER1` and `USER2` work simultaneously subject to whatever sharing and concurrency control policies are supported by `TOOL1`.

### 3.2 Architectural Issues

We divided Oz's clients into two categories, new Special Purpose Clients (SPCs) and the original General Purpose Clients (GPCs). SEL continues to be supported by GPCs. SPCs introduce into the architecture a new kind of long-lived entity, with the role of spawning, managing and achieving the integration of persistent tools by realizing MTP envelopes. GPCs 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 task management and data integration and repository services, and intentionally is not concerned with tool invocation (in part for performance reasons).

In our design, the session management commands (`OPEN-TOOL` and `CLOSE-TOOL`) are issued by GPCs on behalf of human users and executed by the appropriate SPC, installed on the machine determined by the **host** and **architecture** data in the MTP TOOL declaration. Subsequent tasks submitted to the same application may be initiated from a GPC's user interface, but are delegated to the SPC. The same SPC manages all persistent tools executing on the same host (with respect to tasks managed by the same Oz server).

SPCs' do not need to interact directly with any human operator, so no user interface is needed. However, they must manage the user input/output to/from persistent tools. This involves redirection of simple textual I/O between the tool and the user, and more significantly the ability to make the tool's own graphical user interface (GUI) available to the GPCs executing tasks in the context of tool sessions. Most inherently multi-user tools are able to dispatch private instances of their interface to each user, but for other tools (e.g., originally single-user tools extended by our approach to a modest form of groupware) we exploited the public domain **`xmove`** utility [17], which transfers the GUI of a tool across workstations and X terminals. Resetting the X `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 task assigned to SPCs is to spawn, manage and communicate with auxiliary programs called *watchers*, which "notice" any files created or updated by a tool and map them to task arguments according to a configuration file constructed by the envelope. These can then be transferred to the environment's data repository.

### 3.3 Task Management Issues

The most significant remaining question is how the wrapping itself is accomplished. By this we mean the execution of wrappers; techniques for developing tool-specific wrappers is outside the scope of this paper, but should be assumed analogous to any enveloping protocol (e.g., the tool integrator must understand the communication requirements of the tool, but not its internal implementation). A typical task execution goes 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 by the session mechanism explained above.
2. An **initialization** phase, in which the objects gathered from the environment's data repository are fed 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 file system paths (corresponding to file attributes in Oz's data repository), the path to a dedicated temporary directory that is created at the same time 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 SPC, which sets up 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 SPC. This information is used to assist the loading of the data files from the temporary directory into the memory of the application and is displayed to the user inside a task-specific pop-up window. For example, the text presented in the window might indicate the command line or the mouse action that the user should use to tell the tool to conduct the loading.

   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 along these lines have been successfully conducted using `emacs`' extension language.) 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.

4. One or more **data recording** phases may interleave with other actions, whenever the user wants or needs to save temporary results of the work he/she is performing. (Recall the tool updates the copies of the files kept in its own temporary directory, and not those in the data repository.) Such events are monitored by the SPC's watchers. A table of updated files is maintained in the SPC and is used in the final 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 by to designate the task 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's repository 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.

# 4 Conclusions

We have fully implemented all the facilities described in this paper. Example applications (to date) have included:

- `idraw` as a UNI_QUEUE tool, where tasks are queued for one-at-a-time execution (the same userid 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 natural language processing system on top of `commonlisp` 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 single-process local-area-network predecessor of Oz, as a MULTI_NO_QUEUE tool (that supplies its own clients for multiple users).

Future experiments should encompass more exacting tools, and we are in the process of obtaining various licenses. Nevertheless, the completed experiments — all of which run quite satisfactorily — have demonstrated the feasibility of our approach to enveloping persistent tools within a CASE 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.

In-progress work includes process modeling of collaborative tasks suitable for COTS groupware applications, related concurrency control policies, and delegation of tasks among users. Finally, the generic monitoring paradigm at the core of our loose wrapping mechanism seems feasible, if opportunely scaled up, to be applied outside of CASE environments to componentized systems that need to share data or to collaborate in some fashion.

## Acknowledgments

Prof. Kathy Mckeown provided the NLP application. George Heineman conducted early experiments involving overlapping tasks submitted to `emacs`, and developed the "watcher" utility. Israel Ben-Shaul has extended Oz's task definition and execution facilities to support collaborative tasks [3]. Peter Skopp played a major part in designing and implementing the architectural changes needed to introduce SPCs, which will also be used for supporting low-bandwidth (modem) clients [16]. Several other members of the Programming Systems Laboratory provided useful input. [4] outlines our in-progress research.

## References

[1] *Transcending Boundaries: ACM 1994 Conference on Computer Supported Cooperative Work*, Chapel Hill NC, October 1994. ACM Press.

[2] 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.

[3] 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, 1995. CUCS-024-94. In progress.

[4] 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*, Airlie VA, October 1994. IEEE Computer Society Press. In press.

[5] 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.

[6] 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.

[7] Mark Dowson. Integrated project support with IStar. *IEEE Software*, 4(6):6–15, November 1987.

[8] Anthony Earl. Principles of a reference model for computer aided software engineering environments. In Fred Long, editor, *Software Engineering Environments International Workshop on Environments*, volume 467 of *Lecture Notes in Computer Science*, pages 115–129, Chinon, France, September 1989. Springer-Verlag.

[9] 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.

[10] 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.

[11] 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.

[12] Simon Kaplan, editor. *Conference on Organizational Computing Systems*, Milpitas CA, November 1993. ACM Press.

[13] 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.

[14] 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*, volume I, pages 477–497, Toronto ON, Canada, November 1992. IBM Canada Ltd. Laboratory.

[15] Steven P. Reiss. Connecting tools using message passing in the field environment. *IEEE Software*, 7(4):57–66, July 1990.

[16] 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.

[17] E. Solomita, J. Kempf, and D. Duchamp. Xmove: A pseudoserver for X window movement. *The X Resource*, 1(11):143–170, July 1994.

[18] Kevin J. Sullivan and David Notkin. Reconciling environment integration and component independence. In Richard N. Taylor, editor, *SIGSOFT '90 4th ACM SIGSOFT Symposium on Software Development Environments*, pages 22–33, Irvine CA, December 1990. ACM Press. Special issue of *Software Engineering Notes*, 15(6), December 1990.

[19] Ian Thomas. PCTE interfaces: Supporting tools in software-engineering environments. *IEEE Software*, 6(6):15–23, November 1989.