Expanding the Repertoire of Process-based Tool Integration

— MS Thesis —
Technical Report CUCS-027-94

Giuseppe Valetto
Department of Computer Science
Columbia University
500 West 120th St.
New York, N.Y.
10027

Thesis Committee: Professors Gail E. Kaiser and Kathy McKeown

November 23, 1994

Abstract

The purpose of this thesis is to design and implement a new protocol for Black Box tool enveloping, in the context of the Oz Process Centered Environment, as an auxiliary mechanism that deals with additional families of tools, whose character prevents a thoroughly satisfactory service by the current encapsulation method. We mean to address interpretive and query systems, multi-user collaborative and non-collaborative tools, and programs that allow incremental binding of parameters after start-up and storing of intermediate and/or partial results. Our goal is to support a greater amount of interaction between multiple human operators, the tools and the environment, in the context of complex software development and management tasks. During the realization of this project, we introduced several concepts related to integration of Commercial Off-The-Shelf tools into Software Development Environments: an approach based on multiple enveloping protocols, a categorization of tools according to their multi-tasking and multi-user capabilities, the ideas of loose wrapping (as opposed to the usual tight wrapping) and of persistent tools (with respect to the duration of a single task), and a functional extension of some intrinsically single-user applications to a (limited) form of collaboration.

1 Introduction

Software Development Environments (SDEs) typically rely on a set of tool programs in order to provide their desired functionality and to perform those operations necessary to the development and maintenance of software artifacts. While the principles on which such systems are built upon, their architectures and their use can greatly vary, the need to integrate tools and exploit them for the users' purposes is a common trait of SDEs [52] [14].

Integration issues include among others: building a facility for the environment, in order to invoke the tools at will and/or when needed; establishing a dialogue between the tool and the system, in order to issue commands and fully access the tool's functionality; providing a mechanism to extract selected objects, including files, from the environment, manipulate them from within the tool and return them to the SDE data management facility; retrieving files that are newly created by the tool and integrating the new objects into the environment's data repository; mapping and conversion between the environment's and the program's own data models.

Tool integration can be achieved in many different ways, largely dependent on the requirements on the environment's functionality and, consequently, on its design.

Some of the most widely used approaches are:

- The definition of a tool family dedicated to the environment, accordingly to its specifications and structure; this custom toolset is intended to provide in itself most of the power, functions and flexibility of the SDE. Such a choice involves a tradeoff between a high degree of efficiency, simplicity of design and compactness and uniformity of the system on the one hand, and limited generality of use and high cost of upgrade and expansion of the environment on the other hand.

- The modification of pre-existing external programs, in order to equip them with an interface that enables dialogue between the tools and the SDE. Such changes to the programs' structure are less expensive than the implementation of an "ad hoc" tool family, account for higher flexibility and easier expandibility of the environment and are usually achieved with limited and repetitive modifications of their code. Nevertheless, the sources must be available to the SDE developers — a perhaps insormountable hurdle when integrating commercial products from independent vendors. A very popular approach along these lines is the use of a message passing protocol managed by a dedicated centralized server, which controls and dispatches the data sent by different parts of the environment and by the modified tools; all of these components are augmented with a module that appropriately creates, sends, receives and processes such messages. In such systems, the message-passing facility and the interfaces modules (commonly referred to as a *message bus* or *broadcast message server*) constitute the component of the system in charge of the integration. In other systems, procedural calls inserted in the code of the external programs play the same role as the message bus and enable communication with the SDE.

- The encapsulation of generic, unmodified commercial off-the-shelf (COTS) tools with an *envelope* or *wrapper*, a mechanism provided by the SDE and able to interface the external programs with the environment and its users, possibly in a transparent fashion both for the users and the tools. Envelopes, whatever their implementation may be, must conceptually provide the ability to extract data from the internal representation of the SDE, present them to the wrapped program, manage control and data input/output to and from the tool and execute the desired activities. The concept of envelopes has been introduced and exploited for the first time in the Istar SDE [13]. Such an approach, which in theory allows for the greatest flexibility, is commonly referred to in the SDE community as tool enveloping and the generality of the wrapping mechanism is its crucial trait; however, it is highly unlikely to come up with a general-purpose method that can encapsulate any chosen program. Most systems employing envelopes rely on some assumptions on the nature of the tools that are useful and needed for that environment, to guide their conceptual design and practical implementation.

The main concern of this work is to investigate some options for tool enveloping in the context of Process Centered Environments (PCEs) [43] [44]. These are a class of SDEs that rely on a built-in process modeling formalism (e.g. a dedicated language) to define, enforce and support a variety of customizable software processes. PCEs offer the modeling facility and a framework to carry on the functionalities needed by those activities that are part of the range of processes they can describe. Once any process is designed and installed in a PCE, it becomes the core of the environment and defines its goals, functionality, interaction with the data and the users and its boundaries. When a new process is loaded, the behavior of the environment changes accordingly, giving to the PCE class enhanced flexibility over SDEs that maintain a fixed built-in model of operation.

In the following sections, we discuss in more detail tool enveloping for PCEs; moreover, we outline some issues relevant to building wrappers for various classes of tools and examine the different options; we also present an implementation of envelopes for long-lived, large size, interpretive and multi-user tools in the context of the Oz [4] [3] PCE, being developed by the Programming Systems Laboratory of Columbia University.

2 Motivation

Among the characteristic features of PCEs is their modularity and most specifically the replacibility of the process model loaded inside their central engine. Typical PCEs can be seen as made by a frame consisting of a number of components that offer the primary, kernel functions and services in a uniform fashion, and by a customizable core, the process, that can be designed, loaded and enacted by some of the facilities included in the abovementioned frame.

By modifying the process definition, the system can be used in many different scenarios and for various software activities. However, it is recognized that the ability of a PCE to support numerous classes and instances of processes depends primarily on the number and the types of tools that can be employed: the more freedom is given in employing any generic program, the more powerful and universal the environment is, since different processes might need tool sets of the most diverse natures.

In order to extend the process-supporting capability of a system, it is therefore evident that the tool integration service must be as general and flexible as possible. However, this strife for generality has its limits: first of all, it is practically infeasible to design a unique integration facility to accommodate any chosen tool in the same fashion. PCEs usually still assume a set of generic properties that must be satisfied by their tools; these assumptions are dependent on and in turn define what is often called their *domain*, i.e. the class of problems and activities addressed by them.

Moreover, exceedingly generic approaches can be also impractical and inconvenient, since different tools serve indeed different tasks and are consequently specialized to do their work as efficiently as possible. Forcing very diverse programs to employ some kind of uniform interface to dialogue with the environment often results in poorly exploiting some of their unique and therefore most valuable features. Especially when dealing extensively with COTS tools, whose code is rarely accessible or modifiable, an attempt to achieve large-spectrum generalization often becomes completely unrealistic.

In the context of PCEs and considering all of the above, we believe that tool enveloping is one of the most promising options to address the integration problem. It certainly accounts for the necessary flexibility at a very low cost, because it realizes what we call a *Black Box* [26] protocol for integration, since it is not concerned with the internal structure and nature of the wrapped program and no modifications to its code are required. (On the opposite side, we qualify an approach as *White Box* if the code needs to be manipulated, as for example in most message-passing systems).

However, a single type of envelope still does not overcome the abovementioned problem of using too generic an approach. It is clearly impossible, for example, to efficiently "wrap" in the same fashion an interactive query system (say, a database) and a text editor, or a multi-user collaborative tool and a single-user, small-size utility. Either unnecessary overhead or undesired simplifications would occur and hinder the use of those programs.

To extend the boundaries of a PCE's domain without imposing too much uniformity, the enveloping mechanism itself must be flexible and versatile; a possible answer is the ability to provide different kinds of envelopes, each designed to interact with one or more separate classes of COTS tools using the most convenient protocol.

Such a capability is most easily obtained in an incremental way, by creating the enveloping component of the system and providing it with a basic kind of wrapper for a broad domain.

By using the PCE, designing multiple processes and therefore exploring the boundaries of the initial domain, the need for expansion to new software activities will arise and carry along the need to accommodate new tools and to expand the set of envelopes.

This is the approach followed also in the case of our PCE, Oz, which has been equipped with a protocol for envelopes inherited from the Marvel system [27] (also a project by the PSL of Columbia University), to which it is intended to be the successor. While the original protocol (named Shell Envelope Protocol, or SEL [20]), adequately services in a rather simple and elegant way a wide range of conventional Unix utilities (as explained in the Section 5), we found it falling short when it comes to several categories of programs; therefore we tried to address some of these shortcomings with a new mechanism, complementary to SEL. On the basis of our experience, we decided to focus upon the following issues, which we consider particularly important and interesting:

- Tools that require the allocation of a large amount of resources, either at the invocation or incrementally during their work session (for example because they must keep track of the current state of the system and of all the data used within the program, in order to satisfy further requests), and/or support rather long work sessions. We refer to them as large size, long-lived programs.

- Tools that support heavy interactive dialogue with the user and frequent data exchange with the environment during their sessions. Typical examples are tools based on interpretive query systems, such as KBSA programs written in Lisp, or databases. Note how the similarities between the classes of heavy-interaction and large size, long-lived programs are rather numerous.

- Tools whose instances can be shared among different users, either in a isolated or collaborative way, and either sequentially or simultaneously. In the context of a multi-user PCE such as Oz is, sharable tools represent a class that can have endless uses and supporting them is an issue of utmost importance.

- As an extension of the previous point, we would like to come up with a mechanism that allows to convert certain tools, that are designed for and usually employed in single-user scenarios, to a shared use, even if necessarily restricted and partial. We see potential for this in very different families of applications, from multi-buffer text editors to non multi-threaded interpretive and query systems.

Thus, large size, long life, heavy interaction and sharability represent the dimensions of interest of this project and of the intended expansion of the integration domain of Oz.

The purpose of this research is therefore threefold:

1. To experiment with a technique for tool integration based on multiple enveloping protocols;

2. To investigate a protocol able to deal with large size, long-lived interpretive and sharable tools;

3. To provide Oz with an instance of such an enveloping mechanism, pursuing the goal of enhancing its flexibility and widening the domain of processes that can be modeled and supported by our system.

3 Background

All the work presented in this paper has been carried on in the context of the **Oz** project. Oz is a multi-user PCE that employs a rule-based approach to realize the process description and stores all the data, the software components and their mutual relations in an object-oriented repository, called the *objectbase* [38].

The PSL at Columbia University has been working on PCEs for several years and Oz is its most recent effort. The system benefits of course from our previous experience, most specifically the one gained in developing and testing the **Marvel 3.x** PCE [5] [1], to which Oz is intended to be the successor.

While Oz inherited from Marvel most of its main features, some crucial enhancements have been planned and implemented. One of the most important is the modification of its client-server architecture from a single-server to a multi-server structure, in which servers can be geographically distributed and can each support its own process and data model (see Figure 1). Also, different process models can be partially shared among servers, via an import-export mechanism, which allows to define common sub-processes and to specify what collaboration is permitted between the processes that import rules from others and the ones that make them available.

Some of the characteristic features of Marvel that are retained by Oz are:

- **Object-Oriented Data Model**: software artifacts are stored as instances of user-defined classes. Multiple inheritance is supported. In addition to hierarchical relations, objects are connected by directed links, to allow arbitrary semantic connections between objects in unrelated subtrees of the objectbase. Also informations about tools is abstracted by objects of the special class TOOL.

- **Rule-Based Process Model**: the process (and consequently the behavior of the environment) is described by a set of rules. Each rule can either be invoked by an human agent or automatically instantiated by the process [25], and consists of several different parts (see Figure 2):

  - **The signature**: a name and a list of typed parameters the rule accepts; the user invokes the rule by name and provides all its arguments with correct types;

  - **The condition section**: first, additional objects related to the ones bound via the rule parameters are gathered from the objectbase; then the rule processor verifies if some specified properties hold for the bound objects. On the basis of this check it is then decided if the conditions are satisfied;

  - **The activity section**: a rule-specific operation is performed on the collected data. It is in this context that Black Box tool integration takes place and envelopes are employed. In the wrapping protocol inherited from Marvel, exploiting the abovementioned SEL language, envelopes are implemented as augmented versions of normal shell scripts [32]; they handle the passing of parameters to the envelope from the environment, invoke the tool inside the script, customize its execution using the parameters as arguments and return to the system the results of the

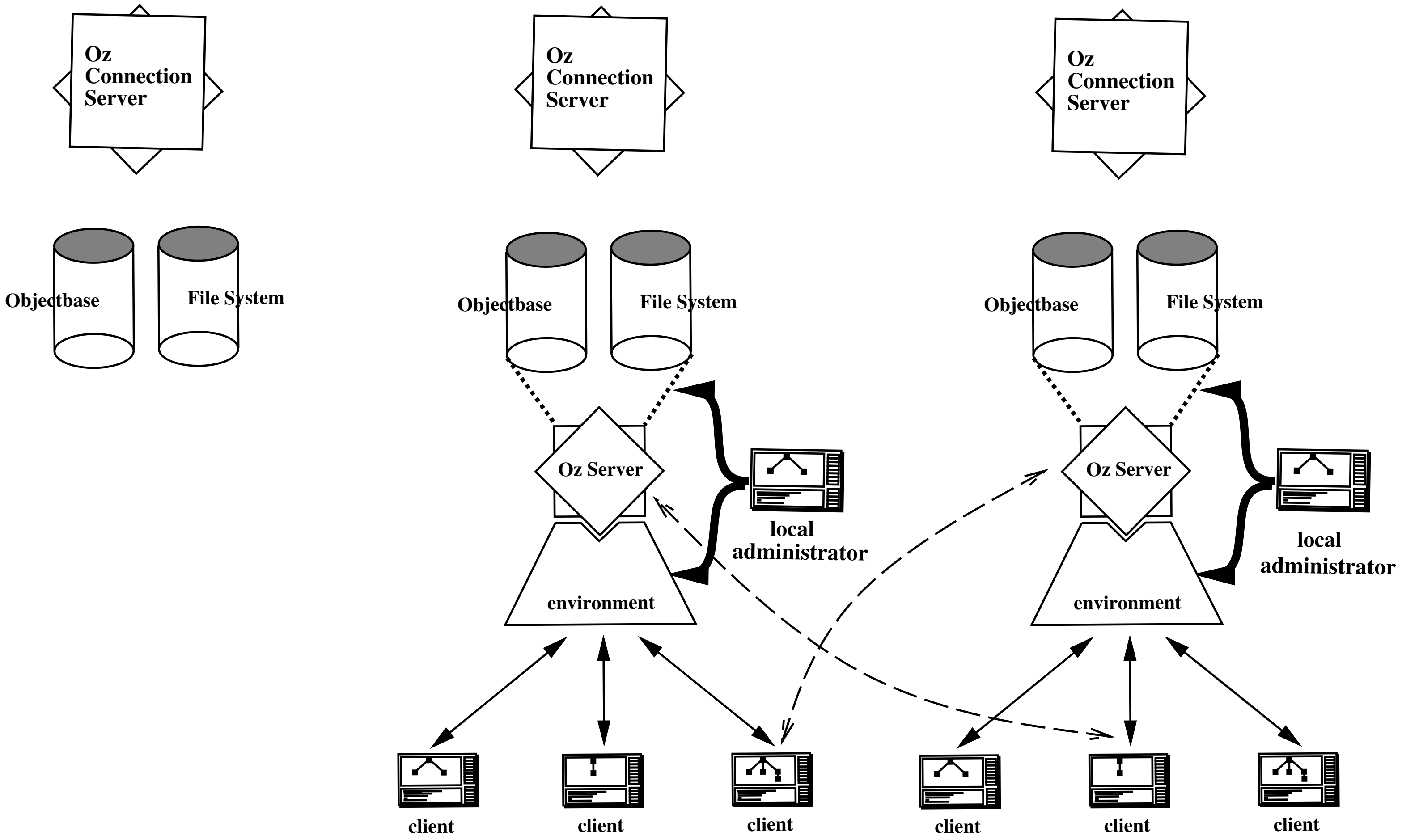

Figure 1: The Client/Server structure of Oz

```
<rule-name> [param-1, param-2, ...]:
           condition section
           {
             activity section
           }
           effect 1;
           effect 2;
           ...
```

Figure 2: A generic Rule in MSL

execution plus a status code. In SEL, each activity follows a rather straightforward input — execution — output sequence.

- One or more mutually exclusive **sets of effects**, to be chosen in accordance with the status code returned by the activity: the effects are statements used to assert the results of data manipulation by the envelope into the objectbase and to modify the state of the process.

The process model and the data definition are written by the process designer (also referred to as the *Administrator*) using the Marvel Strategy Language (MSL) [34].

- **Rule Chaining**: This is the process' assistance model, the way in which process enaction is carried on. Backward chaining takes place when the conditions of a rule are not entirely satisfied: in this case, the system selects and tries to fire those rules, whose effects could fulfill those requirements. Forward chaining may take place after the effects of a rule are asserted: following the modifications incurred in the environment, the system automatically fires those rules whose conditions match the new state of the process. Both backward and forward chaining are recursive and provide the PCE with automation facilities to enforce the process' policies and its desired behavior.

- **Built-In Commands**: it is a set of basic commands used to browse through and manipulate the objectbase. They can be seen as a source of direct access to the data, as opposed to the indirect access provided by rules. It is possible to incorporate such low-level interaction within the process model, since MSL allows to specify conditions and effects also for the built-in operations.

Besides the project aiming to the construction of a new tool integration protocol that is the object of this thesis and to which we gave the name of RIVENDELL, other new features implemented or planned for Oz are support for disconnected and low bandwidth operation [46], automatic maintenance of user agendas as "to do" lists [53], delegation of process steps as a collaboration facility between users, and an advanced rule-based language for specifying concurrency control policies [24].

4 Related work

As we pointed out in the previous sections, tool integration is of central importance to every effort to build efficient and practical SDEs; therefore many studies have concentrated on defining and exploring the meaning and the dimensions of the term *integration* as applied to SDEs. Wasserman [54] for example identified five different kinds of integration:

- **Platform**: it is concerned with interoperability of tools, achieved through the use of a common set of system services, such as networking and operating system facilities;
- **Presentation**: the stress is on giving to a toolkit the same "look-and-feel", via common GUI concepts and design;
- **Data**: it requires the abilities of sharing data between different tools and handling the data relationships among objects produced by them;
- **Control**: it is concerned with monitoring the tools' operation, and using such information to guide the development process.
- **Process**: it realizes the support of a well-defined software development process, by defining and tracking its steps.

(According to this categorization, the work presented in this thesis would be categorized mainly as Control integration.)

Moreover, Earl [14] proposed a well known reference model for Computer Aided Software Engineering Environments (CASEEs, another term for addressing SDEs), sometimes referred to as the "Toaster Model", in which a lot of emphasis is on the issues of portability and interoperability of tools. Generic tools, possibly coming from many independent sources, should be integrated by providing them with common data integration, data repository and operating system services on the architectural end, and with task management services, on the user's end; the latter should abstract from the user the details relative to the peculiarities of each individual tool. (Our work is specially concerned with task management issues.)

In the attempt to fulfill the various requirements and definitions of tool integration, and to overcome its inherent difficulties, the SDE community has developed a large spectrum of different approaches, to investigate new options as well as to propose general solutions. Systems and methods are quite numerous, even when one decides — as we will do in the rest of this Section — to neglect the wide category including all the organic collections of tools that (as for example in the case of UNIX [29]) are sometimes claimed as being SDEs in themselves and that mostly realize only platform integration.

Many methods embrace the White Box paradigm, even if there is a lot of variation among them, for example with respect to the amount of tool code that must be generated or modified to achieve integration.

An extreme approach in this sense is the realization of a set of custom tools, all managed by a common framework designed ad hoc; typical and well-known examples of such frameworks are the language-based editors in Gandalf [22], which integrate those programming activities specifically oriented towards the coding and building of a complex software project, or interpretive systems such as Smalltalk [21] or Refine [40], in which all the tools are combined together at run-time in the memory space of the interpreter.

For many other SDEs, the common framework realizing a form of White Box integration of their toolset — focused on the data dimension — is represented by the database where the results of all the development activities, in their intermediate and final stages, are stored and shared. The tools are on the one hand forced to be closely related, since they must be able to use the same data formats, and on the other hand benefit in terms of performance, because they can reuse data produced by other utilities during previous operation. Some examples are GRAS [30], based on an extension of the classic Entity-Relationship data model, and Damokles [11], that employs schemas in the form of attributed graphs. Adele [15] [2] enhances this methodology by implementing a system of triggers connected to the state of the database, so that data modification by one tool is recognized and may cause the invocation of other ones, leading to further action.

The idea of assigning the role of the main integration principle to a common object oriented data repository has been employed quite widely, for example also by several of the projects aimed to define *standards* for building generic tools with a high degree of portability and interoperability, and therefore widely reusable, even if only under the standard's specifications. PCTE [50] [17] is probably the most representative and generally accepted example of such standards. The goal of PCTE is to create a set of services and facilities, called a public tool interface, complete enough to support tool writers in very different situations and domains; many SDE prototypes and projects [51] [6] [19] in Europe as well as in the USA already exploit this facility. Another proposed standard that exploits an object oriented repository for its integration mechanism is the Ada-specific CAIS-A [36].

A different approach to the White Box paradigm, that is intended to be more cost-efficient than building custom toolsets around a given framework is represented by the class of systems based on event notification, whose stress is on control integration. One of the first and most well-known examples is Field [41], developed at Brown University: its basic principle is the addition of interface modules that send and receive codified messages to the code of generic tools. The messages produced by a tool are sent to a centralized component, known as the Broadcast Message Server (BMS), to inform it about the actions performed during the work session. The server elaborates them and produces further information that is sent to other tools, in order to coordinate their operation according to a given working model kept in the server.

Another system using a form of event notification is Yeast [42]: it has a client-server structure, in which the server process accepts from the clients event pattern definitions linked to action specifications; it is also able to recognize the occurrences of events, in a computer system, such as time passing, timestamp modifications etc., or can be notified of such occurrence, either interactively by users or automatically by tools. In response to an event recognition YEAST takes the actions that have been previously associated to that event.

Polylith [39] combines an event-driven approach with another technique in the spectrum of White Box integration: tool fragmentation. While whole external tools can be incorporated in Polylith, by relinking with the provided libraries that support the interface to the system's kernel, more often tools are identified with simpler *services* — or modules or subroutines — whose structure is declared in a service database, and whose free combination and communication is used to obtain the performance of various complex, full-fledged applications and to carry out all the tasks in the environment.

Tool fragmentation is the basic integration principle of several systems, like RPDE [23] [37], Odin [7] and IDL [47] [48]. RPDE maintains tables that represent its tool fragments as the cross-product of objects (i.e. structural components that can be manipulated by applications) and roles and methods (i.e. procedural components used to act upon objects). Odin has a very similar concept of objects and of the tool interactions that manipulate them; it also provides a language to specify tasks and composite tools, whose operators are represented by tool fragments and where objects play the role of their operands. Similarly, IDL proposes a notation to define the structural and functional features of its tools, each of which can be seen as a "building block" with a front-end for input, a composite structure defining its algorithm, and a back-end for its output. IDL declarative statements also describe how to connect several of these components into composite tools.

While White Box, in all of its flavors, is the kind of integration most frequently implemented by SDE developers, less work has been done on *Grey Box* methods. The Grey Box paradigm does not require any code modification to the tools, which instead must provide an extension language or an application programming interface (API), so that functions can be written to interact with the environment. Unfortunately, relatively few commercial applications are equipped with features that allow to build arbitrary functional interfaces to the engine of an SDE. An attempt to address this limitation is presented by Notkin and Griswold [35], who proposed a mechanism to dynamically and incrementally extend the functionality of generic software systems, without modifying the underlying source code.

We maintain that Black Box integration, via *tool wrapping* (also referred as to *tool enveloping*) is probably the most flexible and general methodology — even if it inherently presents a considerably high degree of complexity — since its conceptual aim is the encapsulation in the environment of external tools with no changes to their code, nor need for other kinds of functional extensions.

It is generally recognized that the initiator of studies along these lines has been the ISTAR [13] [12] system. While it provides its own toolkit and development and integration facilities to help building new dedicated programs according to the needs of each environment, ISTAR also allows use of third-party applications, simply by encapsulating their calls into the code of ad hoc written tools (the ISTAR envelopes), that also provide the correct interaction with the ISTAR's database and user interface.

As we already pointed out in Sections 2 and 3 Marvel and Oz both employ shell-script envelopes for executing their activities (i.e. process-related tasks) and abstractly represent external programs as objects in a toolbase.

Another example is offered by ProcessWEAVER [16], a commercial system embracing Black Box integration and combining together a broadcast message server and a process engine. Also in ProcessWEAVER tools are modeled as objects of class TOOL and envelopes have the form of interpreted procedures with a syntax similar to UNIX shell scripts.

5 Requirements

The current SEL protocol for tool enveloping in Oz is very simple and quite useful: it can adequately and efficiently support a wide range of conventional tools. However, by analyzing its structure and features, it is possible to recognize both its strongest points and its limitations and therefore to outline the functionality that is necessary or desirable for a complementary wrapping mechanism, in order to augment the integration capability of Oz to extended and/or new domains.

We can recognize in SEL envelopes three separate sections that are executed in sequence:

- Acceptance of external parameters (representing the input of environment data to the wrapped tool);
- Execution of a utility program (the tool), customized by and operating on some or all of the envelope parameters, which take the role of arguments provided to it at the time of tool invocation;
- Following the end of the tool operation, output of data to the environment: they may include modified versions of some of the input parameters or new objects, plus a status code which defines one among the possible set of statements in the effect portion of the rule, that must be executed.

The simplicity of this paradigm is together its greatest strength and limit: most Unix utilities, for example, accept all of their arguments at invocation time from their command line and return simple status information at the end of the execution; SEL successfully replicates and exploits such features, thus allowing easy integration of a huge family of tools.

However, there are numerous classes of tools that don't fit this description and typically allow or require greater or more complex interaction with the user or the environment in which they are invoked. Examples are endless and fall in many different categories:

- Interpretive and query systems are designed to accept a series of functions, each having its own parameters and result data. If the tool invoked were such a system, in SEL each of these functions would necessarily map to a different instantiation of the same program. This would be not only highly impractical, but also sometimes unacceptable, because the user would operate each time on a separate instance of the tool and fail to use one of the typical properties of an interpretive system: the ability to keep track of its internal state and of the state of the loaded data, due to past queries, and to use this information to produce the output to the current one.
- Many common tools allow the users to interact freely, loading and saving data or part thereof on the fly, at any moment during their work session. Since the envelopes are

the only interface between the data repository of Oz and the tool and because of their simplistic input — execution — output interaction model, it is not possible to support incremental input and output to these tools. In the past and in the context of the Marvel system, we have conducted experiments and tried to overcome this specific problem, at least for a limited class of software products, by using a *Grey Box* approach [1], that does not require modifications to the source code of the tool, but is limited to tools that can provide their own extension facilities. As a test case, we selected the GNU Emacs text editor and exploited its E-Lisp extension language. While partially successful, our experiment could not satisfactorily address some issues (among which the potential interaction and interference between multiple rule chains generated by different argument sets and the separate management of their possibly different return status codes), and lacked the necessary generality and robustness.

- The life span of each tool is limited by SEL to the duration of the activity phase of the rule. Following invocations of the same rule will map to new instances of that tool. This is possibly computationally expensive, for tools that need to allocate a lot of resources at start-up. Such overhead could be avoided, if the tool could outlive the single rules that exploit it.

- The SEL paradigm generally assumes that rules fired by each single user and their associated activities have limited or no impact on the work of others: Oz's concurrency control mechanism [24], tailorable for each rule signature, controls the sharing of the data among users; beside that, Oz assumes that each activity is carried on substantially in isolation. This limits its domain, with regard to the ever-growing category of multi-user collaborative systems: a first extension, that is already supported by our system, is the possiblity to invoke from the same envelope N copies of a multi-user system on behalf of N users at the same time; however SEL has no way to handle asynchronous participation in groupware task, i.e.several users connecting to and leaving a multi-user session on a long-lived external system at different times.

[1]This research has been carried out within PSL by George Heineman, a Ph.D. student at Columbia University.

By considering the above issues, the most important characteristics of a theoretical new kind of envelopes for Oz become evident: in the first place, we are interested to extend Oz's domain by integrating tools that are not properly handled by SEL because of their large size, their support for complex interaction patterns, their interpretive nature, or their support for sharability and/or cooperation. In order to achieve at least partially the above goals, the new protocol must have a series of features that differ from SEL; we think we identified its most relevant requirements in the following list:

- To allow a longer life span for the instances of its tools; this would minimize overhead at invocation and better exploit their peculiar properties, like in the case of query and interpretive systems. A tool instance that outlives a single rule and can be reused by consecutive ones is qualified from here on as *persistent*. A new protocol for invoking and closing persistent programs must consequently be conceived; moreover, the environment must be able to dispatch its user interface to different displays, since persistent tools can be used during their life time by users on different machines from the one that originally started them up.

- To support incremental feeding of data into the tool and the retrieval of partial results while in the middle of its work session; we need to introduce more flexible and free interaction patterns between the tool, the user and the data repository of the environment. To do this, tight encapsulation of all the tool's operations inside a shell script must turn into a looser way to monitor its execution and its input and output. It is noticeable that this problem is similar to what we tried to address with the abovementioned Grey Box experiment.

- To build a facility that allows users to share the same instance of certain tools, by using a method specifically conceived to convert and extend those designed for isolated work to some form of teamwork capability, and by providing easier integration and accomodation in Oz of systems that are intrinsically multi-user. We are interested in sharable tools that support both isolated and collaborative multi-user operation and we therefore consider the management of persistent tools, which can be exploited in turn by multiple rules, in principle fired by any user, as the first necessary step in this direction. Extending this concept, we can then generally think of a multi-user collaborative program (in the RIVENDELL sense, at least) as a persistent tool which allows the environment to run multiple activities on it at the same time, instead of only sequentially. A mechanism to distinguish between tools that have this property and others that don't (and a way to deal with possible overlapping requests for the latter category) is therefore needed. It is clear that such a scenario carries along numerous problems that are not limited to the integration facility of a PCE, but involve other main components, as the concurrency control policies and the process model: we did not mean to investigate such issues thoroughly in this work and consequently upgrade Oz, but by implementing a way to handle in principle this class of tools, we opened the way to their on-field analysis and provided real test cases.

To achieve all of this, we planned to introduce in Oz what we called a **Multi-Tool Protocol** (**MTP**) for integration, where *Multi* refers to the submission of *multiple* activities and to the interaction of *multiple* users with the same tool instance. Moreover, we decided to address *multiple* platforms (in the case in which our system operates over a heterogeneous collection of workstations and servers, but executables are available for only a restricted subset of the architectures) and *multiple* tool instances (i.e, the management of a set of executing instances of a tool, e.g., when licensing limits the number of instances that can operate at the same time, a common situation with COTS server licenses).

While the requirements discussed above clearly outline MTP as an integration facility that is quite different from the existing SEL protocol, they do not necessarily guarantee that more generality would be achieved by switching to it. Instead, they merely try to solve the problems arising in integrating some rather specific classes of tools, mostly disjoint from the ones already successfully dealt with by SEL, or in some cases similar to them in nature but used differently. We therefore decided that the new protocol should not replace the old one, by taking over its functionality and by adding new features on top of it, but rather be complementary and alternative to SEL. Thus, as a supplementary design requirement, we need a way to define in the process model the integration method of choice among SEL and

MTP, on a per-tool basis. The process administrator would be in charge of analyzing the character of the tools and of the two separate options and to choose which fits best.

To reach these goals we realized it was necessary to redesign many of the components of Oz (some of them extensively, other ones only marginally), as well as to conceive and implement some new ones that perform tasks and provide features that were not previously supported.

6 Design

In order to realize our new MTP mechanism and to implement in it the features we consider necessary for an easier integration of large size, interactive and collaborative tools, we recognized that changes and upgrading were necessary to the general architecture of Oz and to several of its components, beginning with its client-server structure, the corresponding executables and their communication interface, the process modeling language, the rule processor in the servers and its direct counterpart in the clients, the Activity Manager, which is specifically in charge of tool integration.

All the main modules of the system needed to be modified to a certain extent and this consideration influenced some general choices in the overall software design; we decided to introduce these changes in a modular way, by keeping as our basic working platform the version of Oz supporting only SEL and by adding on top of it the code necessary to implement MTP, but keeping it functionally isolated from the rest as much as possible.

The problem was to be able to maintain complete backward compatibility between processes written to use only the old wrappers and the new version of our PCE: to guarantee this, support of the usual SEL protocol has become the *default* behavior of Oz and during the execution of the corresponding rules no part of the new enveloping mechanism comes into play at all; it is invoked only for those activities that actually benefit from it on a per-rule basis. Such an approach was beneficial in the sense that for MTP only the minimal necessary functionality has been added on top of pre-existing core code, which is in common with the SEL protocol; on the other hand, a lot of care had to be given in this reusing process, in order

```
<tool-name> :: superclass TOOL;
        <activity-name> : string = "<envelope-name> <parameters' locks>";
        <activity-name> : string = "<envelope-name> <parameters' locks>";
        ...
end
```

Figure 3: Original TOOL declaration in MSL

to adapt the newer code to the old one nicely (for example deciding where the new modules had to be "plugged in" and when they should be activated) and to preserve efficiency.

We believe that such attention to modularization has been of great impact on the final outcome of the RIVENDELL project and also that it is in general a crucial trait, in order to obtain a functional multi-envelope integration facility for an SDE.

6.1 Modifications to the Process Modeling Language

The definition of the process, actuated in Oz by the Administrator of the environment via the dedicated language MSL, includes the description of instances of the **TOOL** class. These originally had only the functions of capturing the correspondence between an activity name, as stated in a rule and an envelope name (the tool-wrapping script written in SEL) to be executed in that context, and of specifying optional locks on the objects selected as the parameters of the activity (that is, the arguments passed to the script), in order to implement the appropriate concurrency control policy for that rule. The syntax used in a TOOL declaration in MSL was originally the one dysplayed in Figure 3.

By introducing an integration mechanism based on multiple enveloping protocols, the Administrator must also decide when it is most appropriate to use each of the various methods. This should be done on a per-tool basis, since each different approach is designed and tailored to accommodate families of tools with very specific properties. Therefore it was natural to plan to extend the process language by giving a new syntax and adding more information to the TOOL class declarations, to distinguish between SEL-tools and MTP-tools. To preserve backward compatibility to processes written using the unextended version

```
<tool-name> :: superclass TOOL;
     [ protocol     : (SEL, MTP) ;
       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 4: TOOL definition augmented for Rivendell

of the language, we designed such addition to MSL as optional. When a TOOL instance lacks it, the system assumes as a default that SEL is to be used for all the activities listed under it. Besides the choice of the appropriate protocol, we found out that more information can be necessary at this same level, at least when MTP is involved. The final format of a TOOL class definition is shown in Figure 4.

The optional part is included between the brackets and contains the following data:

- **protocol**: this flag is used by the Administrator to actually distinguish between SEL-tools and MTP-tools, so that the appropriate wrapping method is employed. To practical effects and for the current version of RIVENDELL, omitting the whole optional specification and assigning to **protocol** of the value **SEL** are equivalent; in particular, when **SEL** is assigned the rest of the information between brackets is ignored. With further development this might change, since some of the other fields can provide useful information also for an augmented or improved version of the SEL protocol.

- **path**: it indicates the full pathname in the file system where the executable for the tool resides; this might be either its binary code, or (as is usually the case for complex systems) a script that invokes it and also manages some customization of it, to be executed at start-up, in relation to the characteristics of the PCE and to the context of the process. Of course, no default value can be provided for this field.

- **architecture**: it is used to indicate the architecture of the machine on which the tool and the corresponding envelope must run; the default value is **sun4** while other machine configurations can be easily specified [2]

- **host**: it is used alternatively or complementarily to the previous field, when it is necessary to run the tool on a unique machine, whose Internet address is given here. When this information is not specified, the system refers to the **architecture** specification to retrieve the corresponding default host (specified by the administrator in a service file kept within the environment's directory) that supports it, on which the persistent tool will be run. However selected, the host must be located within the same Internet domain, sharing the same network file system (NFS) of the server where the process model resides: for reasons connected to how MTP envelopes are implemented that will be clarified in Section 6.3.2, the tools cannot be located remotely with respect to the server's NFS, containing the environment's data repository. This imposes limitations on the availability and usefulness of MTP in the context of the import-export mechanism provided in Oz to allow sharing of process rules among servers: since they can be geographically distributed and thus have limited or null visibility of other Internet domains.

- **instances**: it defines the maximum number of copies of the tool that can be simultaneously active in the environment; a value of 0, which is also set as default, means that an arbitrary number of instances can co-exist. It is used to comply with "floating" license restrictions which limit the availability of copies of a software product that can be used by a licensee at any moment. MTP cannot however in its current version take into account those instances running outside the environment, but of course still counting with regard to the "floating" license limitations; this is a serious flaw in its performance, that must be addressed in future development, via some sort of exception handling mechanism.

[2] This multi-architecture capability has not been implemented in Oz, yet.

- **multi-flag**: this piece of information is of great importance to decide the behavior of MTP in managing the interactions between multiple human users, with their requests to and their operations on the environment, and the persistent tools, as will be explained later on. It distinguishes among four categories of tools, with different properties with respect to their multi-user and multi-processing capability. They represent substantially the cross-product between two orthogonal dimensions:

  - **UNI vs. MULTI**: where **MULTI** indicates that the same instance of the program can be shared by several users, while **UNI** allows only for isolated work of each user on his/her own tool copy;
  - **QUEUE vs. NO_QUEUE**: for **QUEUE** tools, whether UNI or MULTI, simultaneous processing and execution of multiple activities is not supported, while that is allowed by **NO_QUEUE** ones, both UNI and MULTI.

  The basic class, representing the chosen default, is UNI_QUEUE.

This set of data, specifically added for MTP, could have been included in the most usual way as a subset of the attributes for the class TOOL; from many points of views, including generality and modularity of all the definitions in the process model, this would have been the optimal and most conceptually correct solution, since it would have fully exploited the object oriented nature of Oz's data definition mechanism. However, we decided to consider it instead as a separate, special feature that is exceptionally attached to the syntax and the semantics of the TOOL class in the MTP case. Once again, the reason is mainly backward compatibility to the previous versions of MSL, which is preserved by our choice, because of the optional character of our addition, which would have been lost otherwise: TOOL is currently a root class for MSL (as is **ENTITY**, from which all the classes of objects stored in the data repository descend) specially conceived to capture the concepts of envelope and tool into the process model in an object-oriented fashion, and it was not originally designed to contain any attributes beside the envelope names. In the future we plan to modify this structure, providing Oz with a complete toolbase, in order to better describe the character of the external applications to be integrated in the processes and of the acitvities that involve them.

6.2 Modifications to the Architecture

Oz inherited from Marvel its client-server structure, enhancing it with the capability of having multiple geographically distributed servers talking to each other and to the clients. Each server retains and actuates its own process model, by managing the corresponding objectbase and by dealing with and controlling the rule processor and the concurrency control policies; the clients realize the access to the environment for human users, via a graphical user interface that provides them with views of and a query facility on the different objectbases associated with all the servers and allows to manipulate objects via Oz's built-in commands and to fire environment's specific rules; moreover, clients include the Activity Manager that, in connection with the Rule Processor in the server, deals with running and wrapping external tools. A modular socket-based component is of course present in all the servers and the clients, to provide the communication channels and protocols needed to reach and talk to all the other active parts in the system; there are also, particularly in the servers, numerous other layered modules taking care of different functions, that are not essential for our discussion at this point (See Figure 5).

This kind of structure models quite well a situation in which all the active agents in the environment (represented by the client programs) are connected to and dependent on human users (which in the Oz case manage and actuate fragments of the software process interacting with their own client's user interface) and are permanently relying on a set of always available centralized services provided by the servers: while the life-span of clients depend only on the length of the work session of their users, a server must be up at least as long as any client is open and attached to it; the existence of a certain environment (or part thereof, in a multi-server scenario) is tied to the duration of the server in which it is loaded.

In the situation pictured by the requirements for RIVENDELL, we need to introduce new long-lived entities that provide services to any client and therefore must have in this sense the same behavior of the servers: the persistent tools. We therefore also need to assign to

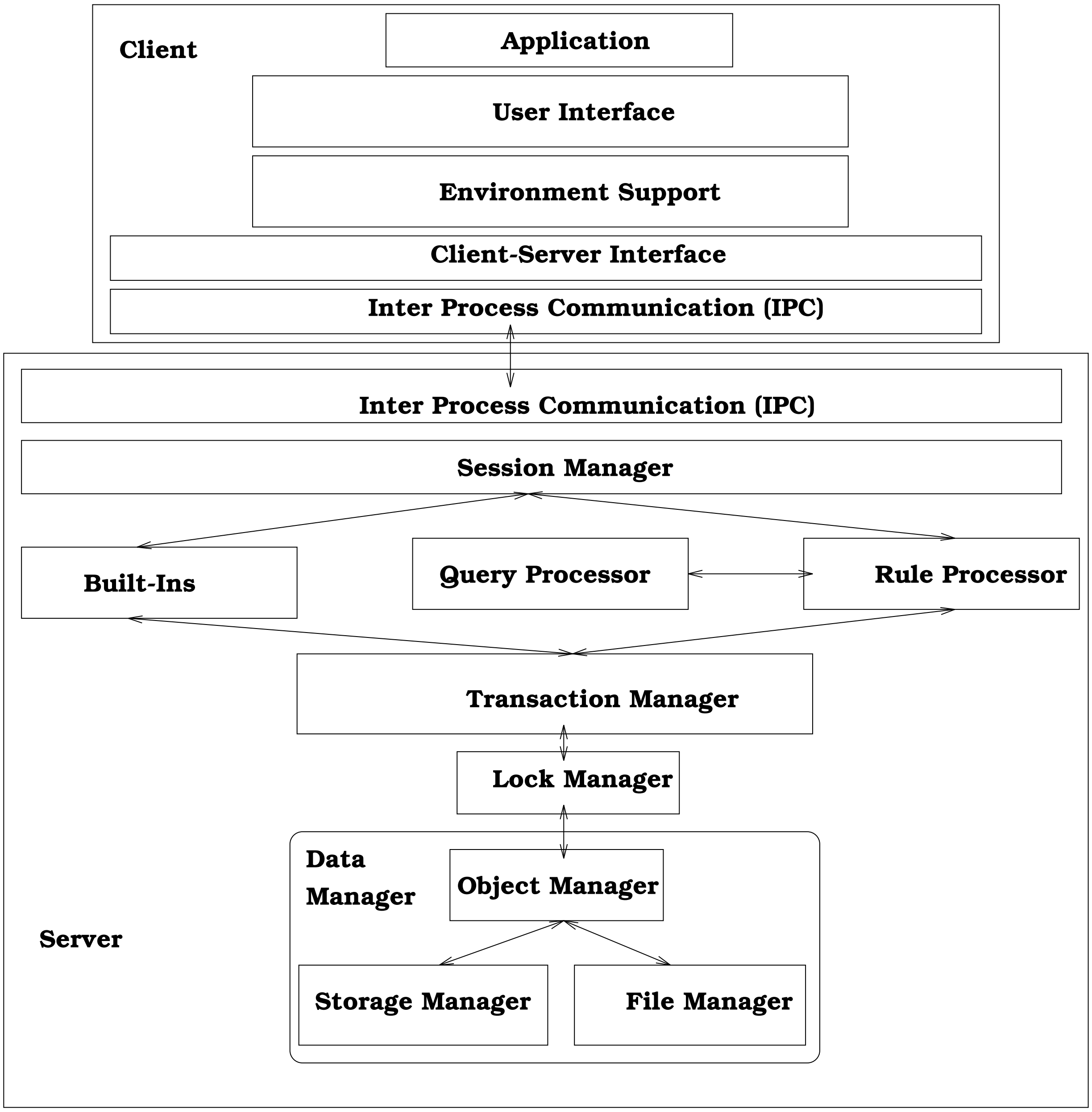

Figure 5: A View of the Client and Server Componentizations

some component of the client-server architecture the role to spawn and manage them and to achieve their integration by realizing MTP enveloping.

One of the most natural approaches to this problem could have been to use the servers as our tool providers; this would have guaranteed that MTP-tools had the same kind of persistency as the servers and that, once started up, were always available inside an active environment. However, we considered that a direct hierarchical dependency of persistent tools from servers is in contrast with a modular design: Oz servers, among their components and functionalities, don't contemplate anything that is in charge of forking new processes and exchanging with them information in the envelope fashion. To achieve that, we should have considerably changed their layered architecture, also replicating much of the code taking care of such tasks that is already present in the clients.

On the other hand, the limited duration of normal clients could not support a persistent tool's instance, since this might be needed well after the client that has forked and taken care of it is closed by its associated user.

Therefore we felt the need to introduce the concept of a new kind of component in the architecture, with the only specific purpose of dealing with persistent tools for MTP: it should be invoked automatically when the first instance of a tool on a certain host is needed; then it should deal with all the persistent programs running on the same machine, on behalf of the users of the environment, and it should run for an indefinite amount of time, until a server shutdown closes the environment that it services.

To reach this goal, we relied on a new kind of client, which is being developed by our team for a different project in the Oz context, aiming to support low-bandwidth and disconnected operation by the environment's users [45]. On that experience we built what we qualify as *Special Purpose Clients* (**SPCs**), as opposed to Oz's usual ones (that we will call from now on *General Purpose Clients* — or **GPCs**), because their only functionality is MTP enveloping. We briefly outline here their most important features:

- they are mainly a convenient mechanism to handle the needs of the new protocol, designed to introduce as little change as possible in the Oz system;
- they don't need any GUI and run on the same host as the tool they provide in the

background, since no human interacts directly with them;

- the only useful components typical of GPCs that remain inside SPCs are the communication module and a modified version of the Activity Manager that realizes the core of the new wrapping approach;

- their invocation and management, including communication and shutdown, are carried out by the server, that sees and services them mostly as it does normal clients.

6.2.1 Inter-Process Communication

The SPC idea is conceptually simple and elegant, but needs a lot of insight into inter-process communication issues. Usually, during the execution of a rule we have a dialogue between the client that fired it and a server: to the latter is assigned the evaluation of the conditions (that may initiate recursive backward chaining) and, after the end of the activity phase, the assertion of the effects and the evaluation of their consequences, leading possibly to forward chaining; the former deals, if the conditions are satisfied, with running the activity, forking the external tool, wrapping it with a SEL script and exchanging relevant data with this envelope.

This complicated mechanism implies frequent message passing between the two processes; in MTP, by introducing with SPCs a new component that is in charge of managing the activity, on behalf of the GPC, we furthermore increase the complexity of such communication. The SPC can be seen here as a proxy for the GPC during the activity phase, since it must completely replace it in the dialogue with the server, but must also correctly redirect the I/O between the GPC's GUI and the tool. The communication pattern becomes quite intricate, also because the current Oz architecture does not support direct client-to-client communication; clients always talk and listen either to their own *local* servers (i.e., a server that runs on the same subnet and shares the same software process definition), for anything that is related to rules' execution, or to remote ones, when accessing the browsing facility and the built-in commands of other processes.

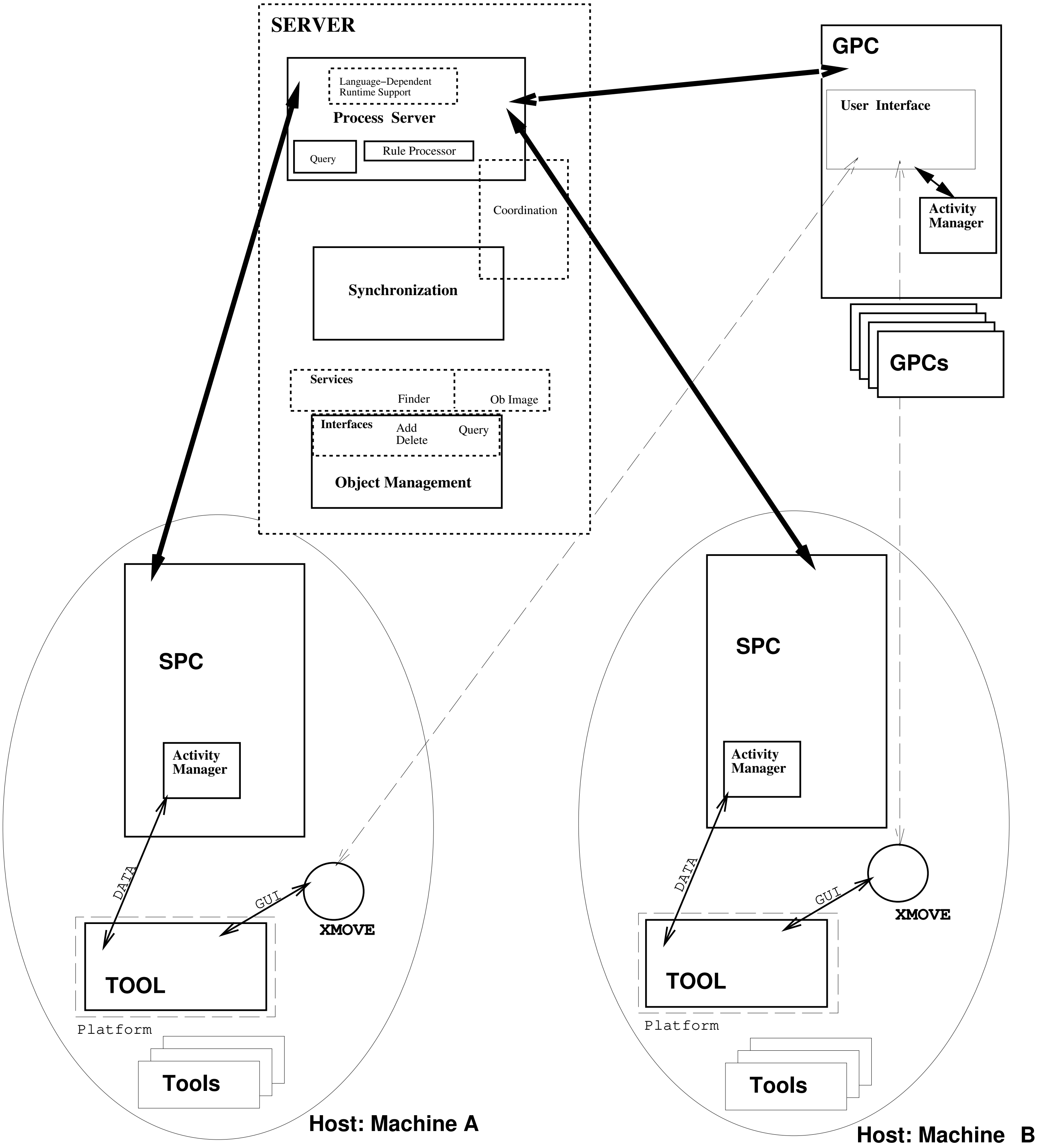

Figure 6: Architecture for MTP Operations

We decided to solve the problem by using the server as an intermediary and a message dispatcher between the two kinds of client and we conceived a mechanism, under which at any point during the operations following the firing of an MTP-rule, both the user's GPC and the SPC used as its proxy only talk to the same server, which in turn is faked to believe that all the messages received from and sent to the SPC are actually exchanged with the GPC, as during a normal SEL activity. The presence of a new party becomes thus completely transparent to the server and the basic communication model of Oz is preserved, by paying the price of some additional implementation complexity (on which we will further expand in Section 7.1, together with a discussion on the beneficial and detrimental facets of our solution).

6.2.2 Dispatching the Tools

As we can see from Figure 6, an SPC lives, together with the tools it provides, on a certain host, while the GPCs it services may run on different ones, on the same LAN. This presents us with a problem, for all the tools that come with their own X-based GUI: in the first place, we are not guaranteed that an X server is running on the SPC's machine, but also in that case we need to redirect the program's GUI to the display of the machine where the GPC resides, in order to allow the associated human user to interact with it.

To obtain this, we employed a utility, also developed at Columbia University in the context of a separate project, called **xmove** [49],

which allows the GUI of a program to be transferred across hosts and terminals. Also xmove has a client-server structure, in which the server accepts commands from the client and performs the corresponding operations on the windows it controls.

The xmove server is in our system spawned by the SPC and operates on all the programs forked on its own machine: as a default, when the tool is initially forked and in the case no X server is active on that host, all the GUIs are sent to a machine specified in the initialization parameters of the xmove utility; later on, to move to the appropriate monitors the GUI of various tools every time an activity uses them, we use a small component we implemented inside the SPCs, that serves as the xmove client and issues the necessary commands.

This way we can dispatch the tools' user interfaces to those clients that at any time need to operate on them, while their programs actually keep running on the machine assigned to them in the MSL specifications. This mechanism implies that users must take turns in accessing most utilities, since xmove does not support duplication of the GUI, but only its transfer; note however that this problem does not involve those multi-user and multi-threaded systems that can create multiple copies of their interface components in response to overlapping requests of access by different agents. The xmove clients implemented wothin our SPCs are able to discriminate between the applications that need GUI redirection and the ones that don't.

It is noticeable that, by using xmove, we not only solve the crucial problem of dispatching the interface from the original machine where MTP-tools run to the hosts where they have been invoked, posed by our proposed architecture; we also can extend the use of certain tools, that are single-user by nature, but can now be shared to a certain extent and in rudimentary sense also become collaborative. For example, a single-user interpretive system with its own user interface can be invoked persistently by User A, who works with it as long as he/she needs it and then relinquishes its control, leaving it in a certain state; at this point User B, a different agent, can fire a rule on that same tool and operate some more on it using its transferred user interface, possibly even carrying on a different part of the same job initiated by User A, since the internal state of the application has not undergone any change in the meanwhile. We codified this possibility by introducing, as will be discussed in detail in Section 6.3.3, the **MULTI_QUEUE** class of tools for our new protocol.

6.3 The Protocol

The abovementioned changes to the process language and to the overall architecture of the Oz system are the necessary prerequisites for the realization of the new integration approach, since they respectively define the parameters of its use (the augmented TOOL class definition) and provide a platform to actuate and manage it (the SPCs and xmove). Anyway, by themselves they do not provide any insight into what should be the working protocol of MTP, and only basic support for its machinery.

In the SEL case we could implement with its shell script facility a true envelope that completely encapsulates the external program from the moment of the invocation and throughout all the execution. Using SEL's augmented features we can provide the tool, on a per-activity basis, with arguments taken from the objectbase and to manage all the I/O with the environment, including the modification to its internal data and the return of the final status code. In MTP it is not feasible to employ the same approach, based on very *tight* wrapping, given the persistent natures of the tools in play. Since tools outlive activities and possibly also the users that invoked them in the first place, we need for them a more flexible view of the envelopes.

This leads to what we call *loose wrapping*: here the principle of *encapsulation* is substituted in most circumstances by the one of *control* and the envelope *per se* may not exist as a single, explicit and well defined component of the system, but its role may be played by numerous different pre-existing parts of it, each acting during a specific phase of the tool's life cycle and operation and performing some of the many functions that are needed. The advantage we see in control, as exerted by these implicit envelopes, over encapsulation provided by explicit ones, is that it makes possible in the context of persistent tools to produce a kind of permanent and generic way to dialogue with them, always active within the environment; entities like our SEL envelopes, instead (even if it were possible to extend them to take care of persistent programs), would take control in an exclusive fashion of the external utilities, on behalf of only a single user.

To design our new envelopes, it is important to outline all the tasks that an explicit wrapper for MTP must carry on, analyze them, separate them and assign each of them to the most appropriate component in our architecture, always trying to preserve its overall structure and modularity. In RIVENDELL we recognized a number of separate basic functions, that we can separate into two main sets: the ones that replicate, with some obvious differences, part of the general functionality of any enveloping system and, more specifically, of SEL, and the additional ones that are a direct consequence of MTP's own needs and purposes. We list these requirements here, according to the above categorization, and intend to discuss below how they have influenced the building of the new protocol and how they have been inserted in it.

- **General-purpose enveloping services**:

  1. The ability to fork and invoke an instance of a persistent tool at the user's will and the complementary ability to conclude its work session when it is no longer useful.
  2. The ability to bind and customize the already running instance of a tool to an MTP-activity, including the ability of providing the set of environment data, derived from the parameters of the corresponding rule, on which the program will operate;
  3. The ability to interface the envelope for the activity with the GPC that is running it, in order to support the textual I/O flow between them;
  4. The ability of recognizing the end of an activity and of selecting one of the multiple possible set of effects, among the ones expected by the associated MSL rule;
  5. The ability of returning to the environment's repository partial and final results of the tool's elaboration on the parameters.

- **Peculiar services of MTP**:

  1. The ability of dispatching the GUI of an MTP-tool to the displays of the GPCs that are executing a related activity;
  2. The ability to limit the number of co-existent copies of a given tool according to the specifications set out in MSL and to record and service the unsatisfied requests as soon as the resources become again available;
  3. The ability of exploiting the persistency of MTP-tools, in order to share their instances among multiple users, partially emulating multi-user capability also for some programs that are not usually employed in a moderately collaborative scenario;
  4. The ability of coordinating overlapping requests to access an instance of a persistent tool coming from separate users, so as to avoid deadlocks and starvation on the one hand, and unintended concurrency of several activities for programs that don't support some form of multi-tasking on the other hand.

```
Condition portion of the rule

Activity begins
        activity invocation: <tool> <envelope> <argument set>
        execution of commands
        return of <return argument set>
Activity ends

Effect portion of the rule
```

Figure 7: Structure of an SEL Activity Within a Rule

6.3.1 Tool Sessions

In the current SEL approach, external tools are instantiated and terminated from inside the envelopes, every time an activity is executed: there is a 1-to-1 mapping between each instance of a tool, with its duration, and each activity, that we can conceptually represent as in Figure 7.

SEL provides with its wrapping mechanism the loading of the initial **argument set** into the tool and the retrieval from it of the results in the **return argument set**, while among the commands in the body of the activity there is the invocation of one or more external utilities, used to manipulate this data. The PCE does not need to explicitly deal with the tools themselves (i.e., to use dedicated primitives for managing their invocation, management and termination), since all of that is abstracted from its point of view and hidden inside the envelopes.

This cannot be achieved in the case of MTP, since the execution of instances of the tools and their life cycle are not anymore contained inside any wrapper that tightly encapsulates them; to the contrary, they must be explicitly dealt with by a new component of the system's architecture, namely the SPCs. It is therefore clear that we need to establish a way to manage persistent tools, defining for this purpose some new concepts in the PCE and supporting them with a set of new commands (issued by the users and executed by the SPCs), dedicated to their management.

To achieve this, we introduced the idea of *sessions*, in order to represent the life span of a

```
OPEN-TOOL <tool> <session-name>
      <tool> <MTP-activity A> <argument set A> (by User 1)
      <tool> <MTP-activity B> <argument set B> (by User 2)
      <tool> <MTP-activity C> <argument set C> (by User 1)
      <tool> <MTP-activity D> <argument set D> (by User 3)
                              ...
CLOSE-TOOL <tool> <session-name>
```

Figure 8: A Tool Session in MTP

persistent tool: a session is begun by an explicit **OPEN-TOOL** command, its body is made of multiple activities, with their own sets of parameters and the related operations, and it is closed by the complementary **CLOSE-TOOL** command (See Figure 8). To each session is also assigned a *name* in order to distinguish it from other sessions involving different instances of the same tool. The use of named sessions is also relevant in the integration of multi-user applications, as will be shown later on.

The body of a session can therefore be seen as a collection of separated activities that use the pre-existing resource of the persistent utility, on behalf of one or more users of the environment (depending on the nature of the tool, as specified in the extended MSL definition).

The OPEN-TOOL and CLOSE-TOOL primitives deal with persistency and are used to limit the duration of a specific tool instance, since their main purpose is to respectively fork and kill it, although several other accessory functions are necessarily dependent on them. For example, they implicitly operate on what we call the **Session Queue** of a tool, a feature that allows satisfaction of the constraints posed by the **instances** field we introduced in the MSL class definition for TOOL, accordingly limiting the maximum number of copies of that program that can be simultaneously active in an environment. When an OPEN-TOOL is issued, the system must first of all check if that request is satisfiable under those constraints. If the boundary is hit, the request is not serviced, but gets recorded in the Session Queue for that tool; when already running sessions are terminated by a CLOSE-TOOL command, queued ones are sequentially extracted and automatically initiated.

Also, OPEN-TOOL commands must lead to the initiation of our SPCs when and where they are needed: in our design the actual forking of a certain persistent program on its assigned host must be performed by the resident SPC, but none is installed at the start-up of the environment. Therefore, when an OPEN-TOOL involves a certain machine for the first time, the server must create an SPC and establish a connection to it, as a preliminary operation to the instantiation of that session; only then the original OPEN-TOOL (and possibly other commands referring to the same host that could be received and queued meanwhile) can be serviced correctly.

Another feature we considered important to add is the ability to fire a rule involving an MTP-activity without having previously opened a copy of its associated tool. We wanted to replicate the usual behavior of rules employing SEL, which don't need any preliminary operation in order to be used, and to allow agents to select all rules in the easiest and most natural way. (This feature can be precious also in the case users they ignore or don't want to worry about the definition of the underlying process model, and most specifically about the integration protocol chosen by the Administrator in each single case.) To support this convenient facility, we introduced what we qualify as *atomic sessions*. When the rule is fired, the system checks if the associated agent has registered for any active session involving a copy of the necessary tool. If not, the rule is momentarily suspended, while an implicit OPEN-TOOL command is executed by the server; the resulting tool session is marked as *atomic*, so that no other unrelated rule can exploit that copy of the tool. After the end of the original MTP-rule and of all the possible chaining deriving from it (since it might need to reuse the same instance of the tool within its rules), the session is automatically closed by an implicit CLOSE-TOOL. All of these operations are transparent to the human user, except for the fact that atomic sessions must still comply with the constraints posed by the **instances** field, so that the request may be sometimes "frozen" in the Session Queue of the invoked tool, while waiting for the end of another session. (It would be possible to avoid this behavior, with a more flexible policy on the matter — possibly on a per-tool basis and only when there are no licensing restrictions, by leaving to the Administrator the evaluation of how crucial the resources reserved by each program, even momentarily, are for the efficiency of the overall system — so that atomic sessions are in certain cases allowed to break the boundary on the number of instances. However, in general even atomic sessions are not necessarily brief, given that the usage of many of the tools that we intend to integrate with MTP — i.e., interpretive systems — can be indefinitely long, and there is little chance to foresee for how much time they will be holding computing resources.)

6.3.2 MTP Activities

While sessions are the MTP mechanism used inside the environment to provide and get rid of persistent copies of external programs, they don't give any insight on how the wrapping itself must be carried out. This point must be addressed at the level of granularity of the activities, i.e. of each single component of the sessions' bodies.

As we already outlined, we decided to employ for MTP loose wrapping, as opposed to tight encapsulation, exemplified by SEL scripts. The idea we followed, in order to exert some control on the operation of the tool and to implement the needed form of interaction and dialogue among the environment (and in the first place its data repository), the human agents who employ MTP activties and the corresponding tools, is that of equipping the system, and in particular the SPCs in charge of the new integration approach, with a set of internal procedures or external accessory software artifacts, sometimes of general use, other times partially or completely customizable. Each of these components has a specific role in the integration process and comes into play in one or more of the various sequential phases listed below, which we identify as constituent parts of an activity using the MTP approach:

1. A **reservation** phase, in which a copy of the desired utility is acquired on behalf of the activity and its associated user, or, if none is active in the environment, yet, one is instantiated inside an atomic session. This phase is entirely carried on in the context of the session mechanism explained above.

2. An **initialization** phase, in which the data gathered by the conditions of the rule are passed from the objectbase to the tool and other customization functions are performed if needed. We employ for this a simple shell script, which accepts as its parameters file

paths corresponding to objects in the data repository, the path to a dedicated temporary directory, that is created at the same time the tool is started up and on which it operates, and some more data used for internal housekeeping. The filename of the script is kept inside the TOOL class definition, in the **envelope name** field and is forked by the SPC, that maintains a set of pipes to communicate with it. The first task of the script is to copy the files into that directory, thus making them visible to the external application; then any series of shell commands can be used to perform whatever customization is necessary; finally, via the pipes, the script sends to the SPC a series of special messages, used to initiate the loading of the data files from the temporary directory into the tool. The SPC recognizes such messages and displays their associated textual information to the user inside a special pop-up window: its purpose is to inform the human agent attending to the activity of all the necessary steps to complete the acquistion of its arguments; for example, the string in the window might indicate the command line or the mouse action that should be used to perform the loading.

Although we would have preferred to come up with a totally automatic loading procedure, as is accomplished by SEL, we realized that it is hardly possible, if we want to reconcile persistency and a Black Box approach. In SEL, tools are not persistent, but live only inside the envelope that invokes them from the shell; therefore it is easy to specify on the same command line all of their arguments (on the other hand, SEL is of little help in several cases, like tools that don't necessarily accept *all* the arguments from the command line, but use other conventions and means, or the ones where incremental loading of additional data after start-up is acceptable and desirable). In MTP tools are already running and therefore we cannot use the same command line approach, and only human agents can directly interact with them, through their user interface. However, the environment, using the messages and the pop-up window, may still provide assistance and guidance to the users, in the loading operations as much as in other contexts, in a practical way. Moreover, MTP activities can this way provide a more refined and detailed level of guidance, telling users what they must do <u>within</u> the tool, i.e. what specific button to push. and so on.

We also thought of and experimented with a different mechanism, that falls in the Grey Box category and is therefore useful only for those applications equipped with their own extension language or programming interface: in this case it is possible to augment their functionality so that textual messages issued from the shell script (containing the path of the files to be loaded) would be captured directly by the tool, which could then automatically acquire them using a loading procedure of its own. A promising test case we worked on involves once again the Emacs text editor. Although more elegant than mere assistance to users via a pop-up window, this method lacks generality, because of the limited range of tools to which it is applicable and because it presupposes the ability and the will of the environment designers to customize the integrated applications by implementing, possibly in a different fashion for each single case, the functional extension to obtain such automated loading; for this reason we decided not to support it in the context of RIVENDELL, even if we plan to include it in some test environments and keep exploring its potential.

3. An **operation** phase, in which the agent who fired the rule freely uses the tool with its features and manipulates the data that have been made available to it in the previous phase, exactly as he/she would have done from outside the PCE. There is no difference or limitation, because the tool is accessed directly and not through any wrapper. The only requirement of the protocol (that cannot however be automatically enforced in any way) is that the execution must not be terminated through its provided internal commands, menu buttons or procedures, but only with the environment primitive for handling sessions (CLOSE-TOOL).

4. One or more **data recording** phases may occur during the operation phase, whenever the user wants or needs to save intermediary results of the work he/she is performing. When saving, a tool updates the copies of the files kept in its own temporary directory, rather than the ones in the objectbase. We use an ad hoc small utility program, a *watcher*, spawned by the same SPC that handles the tool and at the same time, to keep track of what files involved in the MTP activity are modified and to communicate

such information to the SPC, via a set of pipes. There is a watcher keeping under surveillance each of the tools' temporary directories, while a table of the updated files is maintained in the SPC and is used in the final phase of the activity.

5. The **conclusion** of the activity, in which the control of the tool is released, one of the possible effects defined in the rule is selected and the data resulting from the execution is stored back in the objectbase. While in SEL it is up to the envelope to catch the return code of the tool after the user closes it, in MTP the tool remains indefinitely active; therefore the only solution is to let the operator decide when his/her work is finished and to provide a way to communicate this fact to the client. We use for this a couple of buttons, inserted in the window provided by the GPC for all activities. The meaning and consequences of selecting each button are opposite: one causes the watcher to save all the files that were updated during the activity into the objectbase, by copying them back from the tool directory, and maps then to a given set of effects; the other assumes that the work done has for some reason corrupted part or all of the data and performs no action on the copies in the objectbase, but only discards the temporary ones. Also, a different set of effects is chosen. MTP presents to its users a classical *commit vs. rollback* option, that currently limits MTP-rules to have only two sets of effects, while MSL would support an arbitrary number [3]. In principle, it would certainly be possible to modify RIVENDELL to accept and support rules with an indefinite number of effects, as we have in SEL, by providing a menu with a button for each of them; the only problem is that the naive user might not always know the differences among them (in SEL rules they are often subtle and clear only to the Administrator, since this kind of complexity is kept transparent to the clients, but can nevertheless lead to the instantiation of very diverse chains and process fragments), unless a satisfactory and self-explanatory label or caption (maybe provided as an adjunct to the effects' syntax) is attached to them.

The structure of an MTP activity, as seen above with its features and its procedures, is successful in integrating generic tools in a semi-automated fashion. Human intervention is

[3] However, in practice, there are few cases in which more than two are used, since the choice between success and failure is generally sufficient to represent the outcome of most tools.

necessarily heavier than in the SEL approach, since persistency leads to looser wrapping and this gives to the system less information and less control over the state and the operation of the tools. Another drawback is the need to perform a transfer of the files involved in the activity into the tool's directory and back. This is done in order to allow our watcher to keep under control the operation of the program and its consequences on the files, as well as to provide a mapping of intermediate steps in the processing of the data during a long duration activity, without risking in the meanwhile the integrity of what is stored in the objectbase. On the other hand, it imposes a serious limitation, because the tool instance and its SPC must reside on the same local area network of the environment in which the MTP tool is defined: if a user imported a part of a process from a remote server and then tried to fire one of those rules on its local objects, the transfer operations would fail, since the designed SPC and the server would run under two different file systems.

6.3.3 Sharability and Multi-Processing Capability

The purpose of RIVENDELL is not only to propose a new Black Box integration protocol that is alternative to SEL, but also to actively support with it some specific families of tools with their own peculiar properties. One of the most important subjects of our research is to study different ways to allow multiple users of the same environment to work together, exploiting the same resources and even collaborating, when this is applicable. Therefore, in MTP we put a lot of stress on those facets and features that can be used to achieve such a goal, by trying either to accommodate in the most natural way those applications that are inherently designed for teamwork, or to conceive a way to exploit in a multi-user and multi-processing context those tools that, even if not commonly employed that way, we consider specially useful and also flexible enough.

A crucial role for this issue is played by the categorization of tools into four classes, associated with the valid values of the **multi-flag** specification in the extended version of MSL: UNI_QUEUE, UNI_NO_QUEUE, MULTI_QUEUE and MULTI_NO_QUEUE. Each of these labels represents and enforces in the protocol a different working model, when overlapping activities occur on the same copy of a tool, in response to requests issued either by the same user or by various ones. Moreover, together they are intended to cover as widely as possible the spectrum of the possible behaviors and needs of tools involved in multi-user and multi-tasking processing in connection with a PCE [4].

UNI_QUEUE This represents the basic category: with it, we try to describe and accommodate the behavior of applications that are strictly single-user and that would not adequately support concurrent operations derived from simultaneous MTP activities. Therefore, each copy of such tools is reserved to the same unique user, the one that opened it in the first place, and the body of the associated session is made up of a simple sequence of activities that never overlap. To guarantee this last constraint, we introduced the concept of *Activity Queues*, in which we record and keep the requests to initiate activities that are issued while the current one is being processed, until its end. A work session for a UNI_QUEUE utility can be seen in Figure 9.

The most important advantage of the activities which exploit the UNI_QUEUE facility of MTP over the ones using SEL, is that multiple operations can be sent to the same copy of the tool, under the complete control of the process engine. In SEL, additional operations afer the first one, which is specified within the original rule, can be invoked only outside such control. The user would have to understand the format of the hidden file system internal to Oz's data repository to access any other files; moreover, the system would not be automatically notified of the modifications that occurred to those files.

UNI_NO_QUEUE This class represents the following step, after UNI_QUEUE, with regard to the complexity of the requirements for integration that must be satisfied. Here again each tool instance is reserved for just one user, but he/she is permitted to freely run on it multiple activities at the same time, thus fully exploiting its multi-tasking capability. The basic assumption the Administrator needs to verify when labeling a program as UNI_NO_QUEUE is that the nature of the tool can support the manipulation of multiple

[4] Note that, since in the case of Oz it is possible to have different GPCs active at the same time under the same user id, we must always distinguish between the term *user* and the term *client*: when deciding how to service requests for single-user programs, we must take therefore into account that it is not enough to make single-user programs accessible only by the client from which the corresponding OPEN-TOOL command has been issued, but a more refined policy is needed.

```
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 the 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 9: Example of Session of a UNI_QUEUE tool

```
User 1: OPEN-TOOL <tool> <session S1>
         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>
                               Activities B, C carried out in parallel
                                       ...
                               Activity C ends
                                       ...
                                       ...
                               Activity B ends

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

Figure 10: Example of Session of a UNI_NO_QUEUE Tool

disjoint data sets (the arguments of each activity), without causing any interference among them. Since the misclassification of a program without some kind of multi-tasking capability as UNI_NO_QUEUE may lead to the accidental loss of the work done during an activity, when an overlapping one is issued, the administrator should take special care in validating his/her declaration, via thorough tests on dummy data, before inserting them into live environments.

A session can map to something similar to Figure 10.

In this case, we assume that the machinery needed to deal with multi-tasking, like Activities B and C in Figure 10, is provided by the tool. The only basic difference from the approach used with UNI_QUEUE is that we need not employ for it an Activity Queue; what we do is actually only seconding the nature of the utility, without exerting any control on the sequence of the requests it services, so that UNI_NO_QUEUE can be seen as the "lazy" version of UNI_QUEUE.

The previous two categories are designed for utilities that don't allow the sharing of processing resources or of the data among multiple environment's users and enforce this constraint. Those tools that can be adapted to or present some form of multi-user capability fall either in the MULTI_QUEUE or in the MULTI_NO_QUEUE class.

MULTI_QUEUE From the point of view of the protocol, these tools have a lot in common with UNI_QUEUE ones, with the important difference that they are not restricted to service only the user that opened them. This class allows the most basic form of sharing of the tools: users, from their GPCs, can only take turns in using them, being forced to wait in the Activity Queue until the previous MTP activity is finished, and then exploiting the essential xmove utility to acquire the GUI on their display, every time they issue overlapping requests. Given its properties and its implementation, this class is suitable to efficiently achieve one of the most interesting goals we posed for our research: extending the use of certain single-user applications to a rudimentary, but nevertheless useful form of sharability and teamwork. It is easy to imagine multiple agents working one after the other on different stages of the same complex task, by taking alternate control of a MULTI_QUEUE tool.

There is however an important detail that cannot be left aside: users need a way to specify when they want to use for their activities the same copy of the tool that is already active in the system, rather than a new one. The problem comes directly from the concept of atomic sessions we introduced above; this is often an effective and convenient way to simplify the use of MTP rules, but presents us with the following situation: say **User 1** has initiated a session on the MULTI_QUEUE tool **Tool A** and **User 2** fires a rule that employs the same tool. Since User 2 never issued an OPEN-TOOL command for it, the associated activity would map to an separate instance of that tool, enclosed in an atomic session. On the other hand, by issuing an OPEN-TOOL User 2 would begin a new, separate session with its own tool copy.

The solution we chose relies on the restriction that a user cannot participate in more than one session for any tool at any time and on the concept of named sessions, through which we are able to give a more articulate semantics to the OPEN-TOOL and CLOSE-TOOL commands, when applied to MULTI tools. Therefore in Oz, to get access to a sharable pre-existing application, those users that did not invoke it in the first place must use OPEN-TOOL, specifying the name of that same session as a parameter; we define such an action as *joining a session.* Users *leave a session* when they select, as a parameter for CLOSE-TOOL, the name of a session that they previously joined and that has currently more than one participant; this operation does not necessarily kill the MULTI tool instance, but may only change internal information about the association between that user and that named session. Termination of the program is allowed only for the last participant in each session (who is not necessarily its initiator).

We also added one more level of complexity to the session mechanism that can be employed when MULTI_QUEUE tools come into play, mainly with the purpose of achieving an optimal use of the often costly processing resources employed by many persistent programs and to try to limit as much as possible the number of their copies active at any moment in the environment: when an MTP rule using a MULTI_QUEUE tool is fired outside any session and a copy of that tool is already available (in the context of a non-atomic session, because in that case it is reserved only to the original user), and currently not involved in any other activity, we allow the system to use it, automatically attaching and detaching the activity, and thus its rule and the deriving chain, to that session. We define this situation as a *borrowed session* and it constitutes an alternative to an atomic one, with the advantages of avoiding the processing overhead of starting up a new instance and of saving a substantial amount of resources. However, this feature presents also some disadvantages: for example when the internal state of the tool is relevant to subsequent operations, the borrowed activity and, even more important, the following ones by the "legitimate" users might give impredictable outcomes; moreover, a user borrowing a copy of a tool for a lengthy operation could greatly hinder the participants in the session. It would probably be useful in the future to implement a mechanism to decide, on the base of the nature of each MULTI_QUEUE tool and/or according to the will of the users who join a particular session, whether to allow the "lending" of a tool instance or not.

Given all of the above, it is easy to see that the MULTI_QUEUE class provides a lot of flexibility and various diverse coordination models for a hypothetical team of Oz users; the possible combinations are much more numerous than for the UNI tools and also more complex. (See Figure 11.)

MULTI_NO_QUEUE This is the class that has been explicitly conceived and designed to accommodate inherently multi-user systems, taking into account their peculiarities and especially their architectures. We decided to address mainly two families of such applications: the ones that use the very popular and well-known client-server hierarchy, and those that support a non-hierarchical structure, which resembles the model of a network connecting separated and independent components, which have the same importance and play identical roles (and that we for brevity qualify from now on as *nodes*) [5].

From these two different architectures derives an asymmetry that must be resolved by our protocol and that involves mainly the forking and killing of such tools. It is most common that the first agent to invoke a client-server system brings up first of all a server process, which then provides him/her with a client, where usually the user interface plus other decentralized services dedicated to a single user reside. When other agents want to join that same system and invoke it, it is necessary only to create copies of the client process, which are dispatched to them and connected to the pre-existing server. In a similar way, users that decide to close their clients don't influence the life span of the server, except possibly the last one, since terminating the last client usually means that the server's functionality is not needed anymore. This should be taken into account by the primitives of our PCE in charge of starting up and closing such tools.

All of these distinctions disappear if the structure is not hierarchical while others, since each invocation maps to a different copy of the same node program, which itself has the

[5]We identify as instances of such a structure either systems employing a simple peer-to-peer architecture as well as some instances of the ones built on bus architectures, since in the latter some hierarchy among the different nodes may or may not be present

```
User 1: OPEN-TOOL <tool> <session S1>
         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 11: Example of Session of a MULTI_QUEUE Tool

ability to build links to the others, already activated, which constitute identical parts of the same system. Also, in the network-like architecture the killing of any of the nodes can change the way the overall system works and communicates, but does not influence its duration. Therefore no special treatment is actually required in this case, either during the start-up and termination phases.

It is clear that our OPEN-TOOL and CLOSE-TOOL commands should act very differently in each of the two scenarios above, and in order to decide their own behavior they would also need to acquire some additional information that is not relevant for other categories of tools. However, our conclusion was that to impose further complexity on such primitives was not desirable nor practical, since it should have been done most likely at the process language level, asking the Administrator to supply a more detailed description of the nature of each MTP tool than is already required.

The flexibility of MTP and specifically the option to use customizable shell scripts to take care of tools' invocation, rather than directly forking them, was seen as an alternative way to solve this problem. We found that is usually possible to come up with scripts which take in account the peculiarities of both these multi-user architectures and automatically perform whatever is necessary to service correctly either an OPEN-TOOL or a CLOSE-TOOL; for example they can exploit the tool's dedicated directory to store and retrieve temporary data, like the number of clients attached to a server after each OPEN-TOOL and CLOSE-TOOL, the port numbers used to dialogue with the server and so on. Resorting to scripts is therefore quite easy and useful, but in many senses shows some precise limitations of our design and of the functionality of RIVENDELL, since they are currently not an actual, intrinsic component of it, nor a built-in customization method, but rather a "back door" we use for its convenience and which requires on the part of the process designers some inside knowledge of both the tools' structure and of MTP machinery; we believe this information should instead in principle be abstracted from the Administrator in some way and future development should go in that direction, trying however to establish a satisfactory trade-off with the amount of information that should be kept in the process description for each MULTI_NO_QUEUE tool. An interesting approach could be to formalize the scripts as inherent components of our protocol, to augment them (similarly to what we did with SEL) by coming up with a convenient syntax and the associated semantics to describe in a general way a wide range of tool customization operations and to consequently offer within our PCE a new powerful utility for the tool modelling that is necessary during the process design phase.

Anyway, in its current stage MTP is able to ensure (via the abovementioned execution of the shell scripts in response to every OPEN-TOOL command), either in the case of the client-server or of the network-like architectures, that all the participants to the same named session access the same instance of the multi-user system. From the Oz clients' point of view, each OPEN-TOOL maps to the instantiation of a new portion of the system, which is dispatched and made available to the GPC, exactly in the same fashion as the other categories of tools (note that, according to the architecture of RIVENDELL, also in this case the node or the client processes actually run on an SPC [6], while only their correspondent GUIs are sent by xmove to the hosts where the GPCs reside). Most of the complexity of dealing with such composite systems and with their structure is effectively kept hidden from the naive user and involves only the Administrator, since the only important notion that users must recall is that each time they select a new name for their sessions they start up a separate multi-user system, while by choosing an existing session they obtain a new node or client process for an already active one, thus joining other agents within the environment who are sharing its resources and its data.

One of the most interesting facets of this model is the fact that, to a great extent, we are able to deal with each single part of such a complex system as if it were an independent software artifact. This at the same time has favorable consequences and presents us with some intriguing problems. On one side, once a multi-user system is initiated, control of one of its portions is obtained by each GPC joining that session; the GPC then takes active part in the session by firing rules completely on its own. The transparency or visibility to other GPCs and their associated users of such activities depends exclusively on the nature of the tool,

[6] While running each component from its own GPC would be desirable and even possible from many multi-user systems, it would require the replication in the GPCs of a lot of the specific SPC functionality, especially in order to deal with the sessions and the related primitives.

which may either support collaboration or not. The integration protocol is not concerned about such issues and leaves up to the tool's functionality their management and to the PCE's concurrency control component the definition of a policy which must appropriately resolve conflicts that could arise on the data (if it is stored in the PCE's repository).

On the other side, GPCs consider a node or client process assigned to them as a stand-alone application and this leads possibly to a need for further categorization internally to the MULTI_NO_QUEUE class, with respect to multi-tasking capability. For example, a client-server system is commonly able to comply with simultaneous responses to commands coming from *separate* clients, but, depending on the structure and implementation of the client processes, it might or might not allow processing of operations which are issued on the *same* client and which overlap in time, a scenario that might certainly occur, if the user of the environment decided to fire multiple rules on the same tool.

A possible solution would be again more fine-grained information provided at the process definition level (say, an additional **QUEUE/NO_QUEUE** flag to be looked up when dealing with MULTI_NO_QUEUE applications and referring to the nature of their clients or nodes), even if it is possible to imagine systems that would present us with this problem at multiple levels and, in principle, even recursively. We however decided again to limit the detail of the tool description in MSL and to adopt for now, a rather conservative approach to this matter: we disallow GPCs to execute overlapping activities on their assigned component of a multi-user system and we use the mechanism of the Activity Queue, imposed on every single node or client, to enforce this constraint, even if this choice may sometimes seriously limit the exploitation of the power of some tools.

The schema of a hypothetical MULTI_NO_QUEUE session can become quite complicated, as in the example of Figure 12.

```
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 session S1); system component 2 dispatched to User 1
               User 2: <tool> <MTP-activity B> <argument set B>
                      Activity A, B are carried over 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 12: Example of Session of MULTI_NO_QUEUE Tool

7 Implementation Issues

7.1 Tripartite Message-Based Communication During MTP Rules

In their current implementation, Oz clients cannot directly talk to each other, while servers don't have this lmitation. All Marvel and Oz past versions relied on the fact that (at least) one of the two ends of a communication channel must be a server. Therefore, client-to-client communication may happen only through a server, which is employed to pass the information from one end to the other.

During recent development,we recognized that, since the complexity and the number of messages exchanged in Oz constantly grows while new features, especially distributed or delegated ones, are designed and built, a bottleneck problem could occur at some point. This is possible particularly if we consider that work shared among agents and groupware are becoming increasingly crucial issues in SDE technology. As an experimental step in that direction and with the introduction of SPCs, MTP has been one of the first scenarios that presented us with the need for a more complex interaction during rules' execution, and possibly for direct client-to-client message exchange. However, we judged such upgrading not to be in the scope of the Rivendell project, being rather a networking problem, to be faced in parallel on-going lines of research. Therefore, we chose to exploit the current architecture as well as we could, by performing only functional modifications on top of the communication protocol, rather than structural ones to its core.

In SEL, if a user fires a rule from his/her own GPC, a message is sent to the server, where the conditions are evaluated by the Rule Processor (potentially causing recursive backward chaining); when the activity part of the rule is finally reached, a message goes back to the GPC, causing it to fork a process executing the envelope that encapsulates the external tool; the relevant input to and output from its work session is completely managed by the envelope and the GPC's user interface, while the final status information (together with the results of the processing) is passed again to the server that performs the evaluation of the effects and, when it is appropriate, forward chaining. (See Figure 13 [7].)

[7] The numbers associated to the arrows give a temporal order to the various phases.

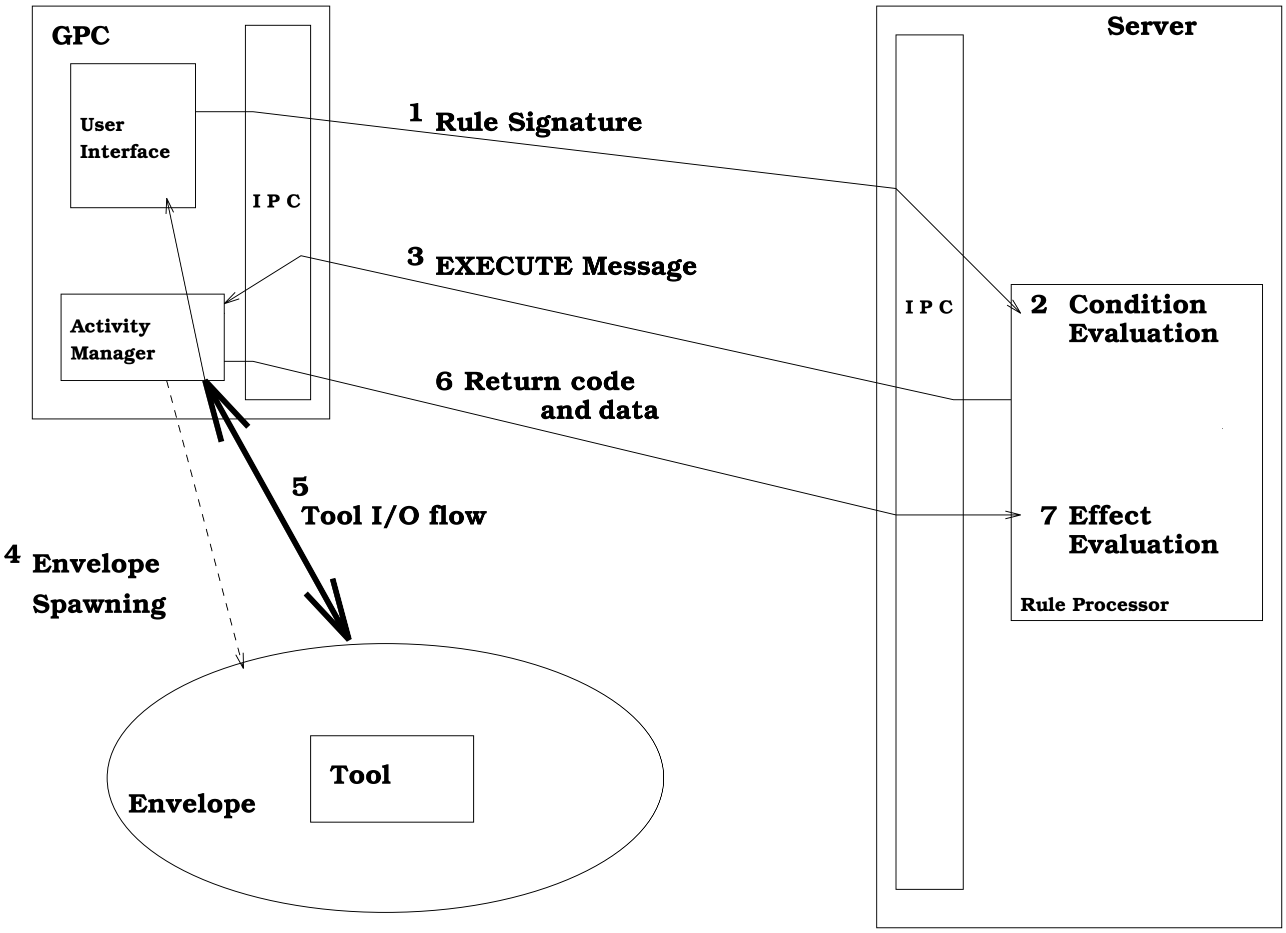

Figure 13: The Sequence of the Events and Messages for the SEL Protocol

In MTP things become much more complicated, since to SPCs is delegated a part of the work that in the usual integration mechanism is performed directly inside GPCs: once again, the initial message that fires a rule and instantiates a process fragment is sent from the GPC to the server, but the server, after the evaluation of the conditions must select the SPC on the right machine and ask it to perform the corresponding activity, according to the MSL specifications stored in the TOOL class. The associated tool output must be sent to the GPC and be displayed inside the window dedicated to each activity, while input from the user, captured by the same component of the GUI, must go to the SPC and then be fed into the wrapped program; the status code that concludes the activity is produced by the GPC and must be sent to the SPC again through the server, thus causing the end of the MTP activity, the registration of the results into the objectbase and finally the beginning of the effect phase of the rule.

As it is evident by looking at Figures 13 and 14 the pattern of communication becomes much more complicated in the MTP case. To handle this without too much additional machinery we needed to reconcile it as much as possible with the usual two-ends scheme. The principle we embraced is to maintain in a table which SPC is proxying for a given GPC during each activity. This table is built and kept in the server and when messages arrive from the SPC their signatures are changed so that they are processed as if they were coming from the GPC. The only exceptions are special messages that carry the I/O of the external program: while in SEL the server is not concerned at all with such data, in MTP it must just redirect it from the SPC to the corresponding GPC and vice versa, by looking up the pair of clients in the table.

This way, the complexity of the new communication model is almost completely hidden from the servers, as much as from the two kinds of client; they all behave almost completely accordingly to the old model: the clients always talk to their local server and the server is "fooled" into believing that it always communicates with the GPC that fired the rule in the first place.

However, there is an obvious drawback: this model introduces some communication overhead, compared to a hypothetical one where SPC and GPC interact freely, since those

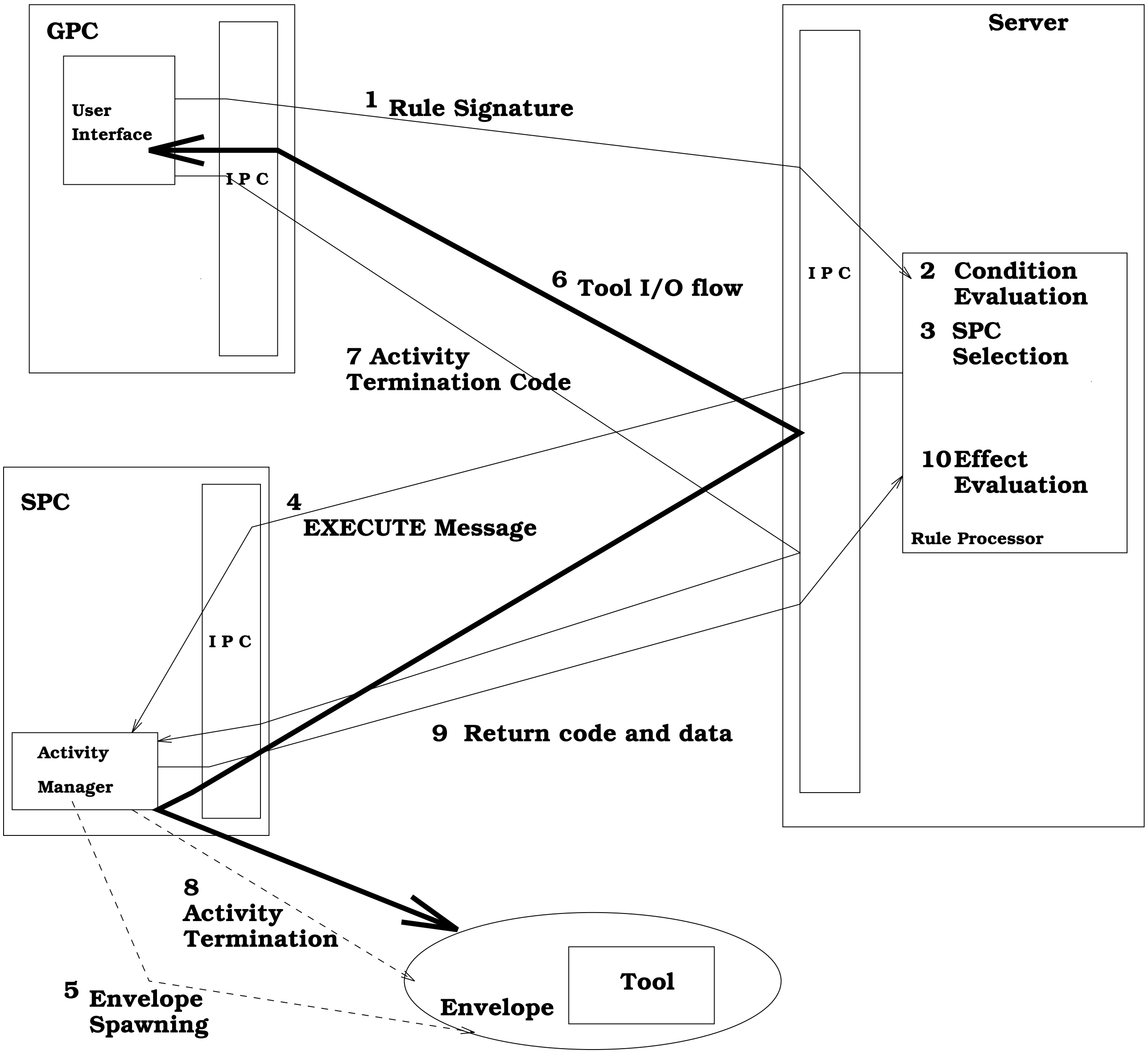

Figure 14: The Sequence of the Events and Messages for the MTP Protocol

messages that have to pass through the server must undergo additional processing and are often also sent across the network twice. Also, some new coordination problems and race conditions, due to messages coming from the two different sources, can arise in the server which takes care of an MTP rule and must be taken into proper account in the current implementation. This problem has been analyzed and avoided, by adequately designing and testing the whole tripartite message protocol between GPC, server and SPC and by carefully trimming the procedures in the server's code which poll in the correct sequence all the sockets connected to the two kinds of client. For this reason, whenever an extension becomes necessary and new message types are added to support more features, developers must to deal with the complexity of this part of the communication subsystem and must be careful to implement their enhancements in a compatible way. We foresee that in the future the whole subsystem will probably be redesigned, e.g. using a bus architecture.

7.2 Startup of MTP Tools

The property of persistency we decided to give to programs integrated by our new protocol is clearly responsible for the most extensive and important modifications introduced into the overall structure of our system, beginning with the creation of SPCs. Persistent tools are automatically instantiated via a specific primitive and by a dedicated piece of software and the corresponding operations involve some complex implementation details, specifically when dealing with two "special cases": initiation of tools residing on hosts where no SPC is already active and atomic sessions.

7.2.1 Initiation of SPCs

The new MSL definition of the TOOL class provides enough information to determine the host on which each MTP program must reside. This specification has a twofold purpose: first of all we consider important to share the load of running possibly computationally-expensive applications, like the ones we plan to integrate with RIVENDELL, among different machines; moreover, it can effectively address those cases in which a given tool is for any reason (e.g. license agreements) restricted to run only on certain hosts. Note how this feature is of general value and could benefit also a future enhanced version of SEL.

Accordingly to our design, Oz employs a distinct Special Purpose Client to manage all the instances of persistent tools assigned to a given host; the SPC must also reside there, since it is its duty to fork them. When the environment is initiated, however, no SPCs are present: an SPC is only created when needed for the first time, that is, whenever the first instance pertaining to a given machine is requested by an OPEN-TOOL command.

To service that request correctly, a sequence of operations must be performed: the system must first of all start up and initialize the SPC and then ask it to fork the tool, thus initiating its session. Therefore, when receiving an OPEN-TOOL command from a GPC, each server looks up its *client table* to see if any SPC is active on the machine where the tool must run. If this is not the case, the server saves the OPEN-TOOL with all its associated parameters and issues a request to the *Oz daemon*, an autonomous component of the system, in charge of initiating both servers and SPCs on their designated workstations. We use this approach for two main reasons: in the first place, in our architecture the clients are not dependent on their local servers and are never forked by them, but rather live independently and are linked to them only via sockets; moreover, after the server has passed its request to the daemon, it can go ahead and perform other operations, namely the processing of messages and commands coming from other clients and servers, while the daemon works on its own to satisfy the server and fork the SPC. This is clearly more convenient than to engage the server itself in the creation of SPCs, thus delaying the work of all the other agents working in the environment and dependent on it.

When a new SPC is up on the chosen host, it retrieves on its own the communication address to talk to its local server, information kept in a file inside the root directory of the environment; then it issues a special message to inform the server of its existence and a registration protocol is executed by the two parties, after which the communication channel is firmly established and the new client takes its place in the table maintained by the server, so that it can be regularly polled for service.

It is only at the end of the registration procedure that the original OPEN-TOOL can be resumed and satisfied, since now the system is ready to comply with it. We also take into account that in the meanwhile more commands depending on the same SPC might have been issued (for example, MTP rules in the context of the session just initiated, or more OPEN-TOOLs for different instances on that host, and so on) and we are able to deal with them, by buffering them momentarily until the SPC is ready. Moreover, we associate a timeout to each of these frozen requests, in case a failure happens in the forking or in the registration procedure, so that the server can get back to the GPCs waiting for a reply and inform them that some problem occurred, preventing the processing of those commands.

7.2.2 Atomic Sessions

Every rule involving an MTP activity must be part of a tool session, since in our model the session is in charge of establishing a relation between persistent applications forked with an OPEN-TOOL command and the process fragments using them. When an MTP rule is fired outside any session and no borrowing is applicable (See the paragraph on MULTI_NO_QUEUE in Section 6.3.3), we need to enclose it in a dedicated and atomic one, whose body is composed exclusively of the associated activity. This is necessary in order to provide to the rule the tool instance it needs to operate on its arguments and is accomplished by automatically executing OPEN-TOOL and CLOSE-TOOL commands before and after processing the rule respectively, transparently to the user who fired it.

This task is performed by the server, since it keeps in a table information about all the active sessions, with indication of the tools involved and of the users who joined them; when an MTP rule is fired, it is easy to check, by looking up that table, if the user is already registered for a copy of the tool described in the rule's activity part. If this is the case, the server checks the type of the tool and if it is already engaged in some other operation, then acting accordingly, either by queueing the activity or executing it. Otherwise, it is necessary to "freeze" the execution of the rule and save its signature in a dedicated data structure, while an implicit OPEN-TOOL procedure is carried on by the server and by the appropriate SPC: nobody else can join such an atomic session, which also does not accept other requests, except the ones possibly deriving from chaining from the original rule.

When the tool is finally available the conditions of the rule are re-evaluated, since the state of the environment might have been changed in the meanwhile. This implies that backward chaining from the original rule could be initiated and performed from *inside* the atomic session. In the same way, after the end of the activity, the automatic CLOSE-TOOL command is not issued until the effects and consequential forward chaining are executed. The choice of extending the duration of an atomic session to all the chaining deriving from an MTP rule is suggested by the empirical evidence that the rules in the chain may often use the same external program, specially when dealing with complex, long-lived systems. In that case it is certainly more correct and efficient to permit the use of the atomic instance also during the chain, than to open new ones for each rule in the chain.

This may be still necessary, however, if the rules in the chain involve other kinds of tools and would be again realized using atomic sessions for them. The importance of the concept of atomic sessions is here once again evident, because they permit to preserve also in MTP complete automation of the process actuation via chaining, which is one of the most valuable characteristic traits of the Oz PCE.

7.2.3 Parametrizing MTP tools

The OPEN-TOOL primitive accepts only one explicit argument, a name for the session; this may be insufficient in several cases, for example to correctly invoke certain applications that require some information at start-up (e.g. a database system might need to know which particular database it has to load), or to provide command line switches that change in one way or another the behavior and the use of the tool. Our model has therefore a serious limitation in its power to describe persistent tools and their properties.

We bypass this problem at the implementation level, by relying on the customization shell scripts we often invoke at OPEN-TOOL to initialize the copies of the tools, coupled with our watcher utilities (on which we expand further in Section 7.4).

The approach is reasonably simple: as its first operation, the script creates into the tool's directory a dedicated file, which contains a prompt for the user, asking for the necessary pieces of information; the script then enters a loop, waiting for a response file to be created in the same directory [8]. Whenever the prompt file appears, it is detected by the watcher, that encapsulates its textual content within a special message sent to its associated SPC, and from there to the core of the Oz system. The user who issued the OPEN-TOOL is then requested to provide the parameters, inside a little dialog window. His/her response is passed back to the watcher, that is in charge to write it in the response file. Such an event is in turn caught by the customization script, that retrieves the information in the file and then can invoke the instance of the MTP tool with the correct arguments.

This method realizes for the persistent tools an alternative way to access data, outside the protocol defined by Oz for its environments; therefore none of the control procedures of our system and of the user-definable policies can be enforced. This does not represent a serious data integrity problem as long as the method is used to parametrize the tools only with data comlpletely external to the environment's repository (e.g. within a <u>separate</u> tool's repository).

7.3 Queueing Sessions and Activities

We saw above how in various different occasions MTP employs queueing or buffering mechanisms to momentarily stop processing certain requests, while the system provides the resources or realizes the conditions which are needed to comply with them. The management of such features has some interesting facets and offers some implementation challenges we had to face:

- Some of the categories of tools we defined for MTP support Activity Queues, maintained by the server in the context of each session. Problems can arise when a user tries to close a tool instance while its Activity Queue is not empty, since it is necessary to resolve the conflict among the rules in the queue (which still need to access the copy of the external application) and the CLOSE-TOOL primitive. Various policies are possible to deal with such situations:

[8] All of this can be achieved by exploting common Unix commands.

- Immediate servicing of the CLOSE-TOOL, while each single rule still in the queue is enclosed in an atomic session and executed separately from the others. This solution has many drawbacks: first of all, it possibly causes huge start-up overhead, because the system must provide as many copies of the tool as the number of entries in the Activity Queue. Moreover, if this number is greater than the **instances** value, we have to deal again with queueing some of the requests, in the form of atomic sessions, for an indeterminate amount of time. Last but not least, this approach can be often logically wrong, since the queued rules might have been issued specifically for that instance of the tool, in order to benefit from its state and from the operations previously carried on by it. We face what can be defined as a *session mismatch*.
- Delayed service of the CLOSE-TOOL, after all the rules waiting for that instance have been serviced. Even if this option seems to address successfully some of the weak points of the previous approach and specifically session mismatch, it is not a feasible solution. It is possible that while the process fragments deriving from queued rules are performed, other rules involving that same instance are fired by agents who are unaware of the impending CLOSE-TOOL, thus further delaying its execution indeterminately. To alleviate this problem it is possible to mark the session in a special way, so that no more activities are dispatched to it. This is potentially confusing for the users who have joined that session and who try to access it after the CLOSE-TOOL, but before the processing associated with the last queued rule is terminated: formally the session is still up, but user are "thrown out" of it, most likely forced to employ atomic sessions instead.
- Immediate service of CLOSE-TOOL and transfer of the whole Activity Queue to another existing session for the same tool, or, if none is available, to one created for this purpose which remains invisible and unaccessable to clients. Even if this approach still does not address properly the important problem of session mismatch, it avoids unnecessary overhead, realizes prompt response to the CLOSE-TOOL command, is not confusing for participants to the session who have not issued the CLOSE-TOOL and also keeps a partial logical consistency, since the rules in the

queue, that were meant to be processed on the same instance still are, even if this could have little meaning in case they were strictly dependent also on previous operations. Anyhow, we decided to follow this line of thought in the current version of RIVENDELL, being aware that some more sophisticated effort (e.g. an agreement mechanism when closing a session with multiple participants) is desirable for future implementations.

- Rule execution is suspended in MTP basically when the necessary resource represented by the tool invoked by an activity is not available. When the system verifies this is the case, the rule has actually already been instantiated and its first section, with verification of conditions and binding of objects from the data repository, has taken place. During this phase the system is in charge of placing locks on the objects, as specified in the MSL tool definition and according to the concurrency control policy established by the Administrator. Locks are usually released only at the end of the rule execution, but since that event is delayed for a potentially long period, we decided that "frozen" rules cannot keep their locks. This avoids interference with other rules that might need and be prevented to access a subset of the same data. This, besides the fact that the state of the process and the data might have changed in the meanwhile, is another reason why it is necessary to re-evaluate the conditions of those rules and to re-acquire locks when they are finally resumed.

- Another issue strictly linked to the Activity Queues above is the decision about when extraction from them should take place. The answer may seem obvious: when the activity currently using the tool is over. It turns out instead that other approaches may be more efficient. As we argued above, it is quite reasonable that chaining generated by an MTP rule involves again the same external application, perhaps in different ways and with diverse purposes. It is after all a natural consequence of our strife to integrate either certain families of programs like large size systems with numerous functions often related to each other, or query-based ones, since they encourage to perform complex procedures that are composed and actuated by a sequence of steps, each executing a

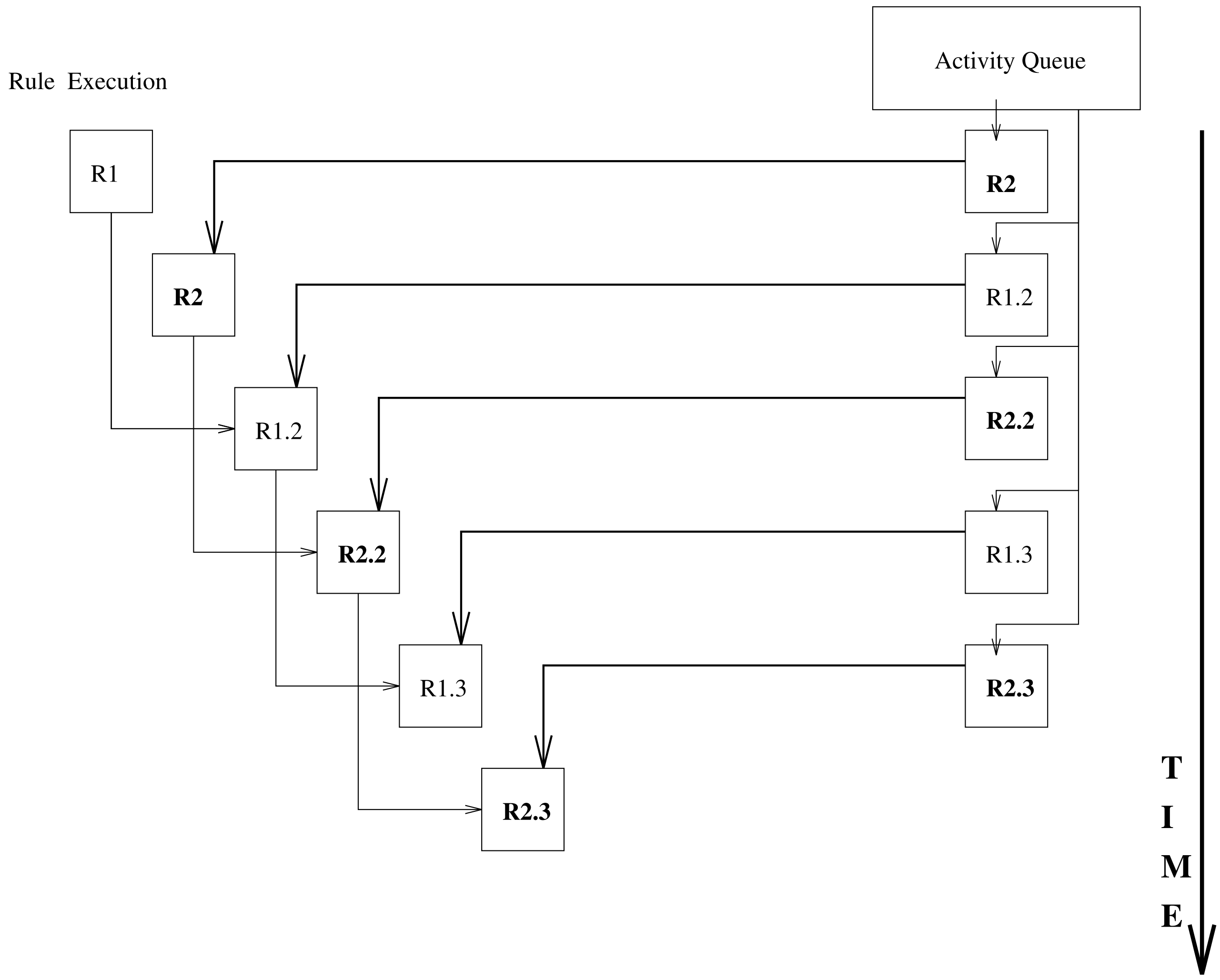

Figure 15: Sequence of rules' execution in the hypothesis of chains' alternation

part of the whole processing and dependent on the previous ones. It is therefore likely that if we extracted a request from the Activity Queue while forward chaining from the original rule is in progress, some of the elements of the chain would finish up in the queue themselves, of course until the end of the resumed activity. This would lead to an alternation between two (or more) chains of rules, taking turns with the same program (see an example in Figure 15). It is easy to see that such events are highly undesirable, since it would take a long while to conclude each of the chains, with unpredictable consequences on the efficiency of the environment and possibly on the consistency of

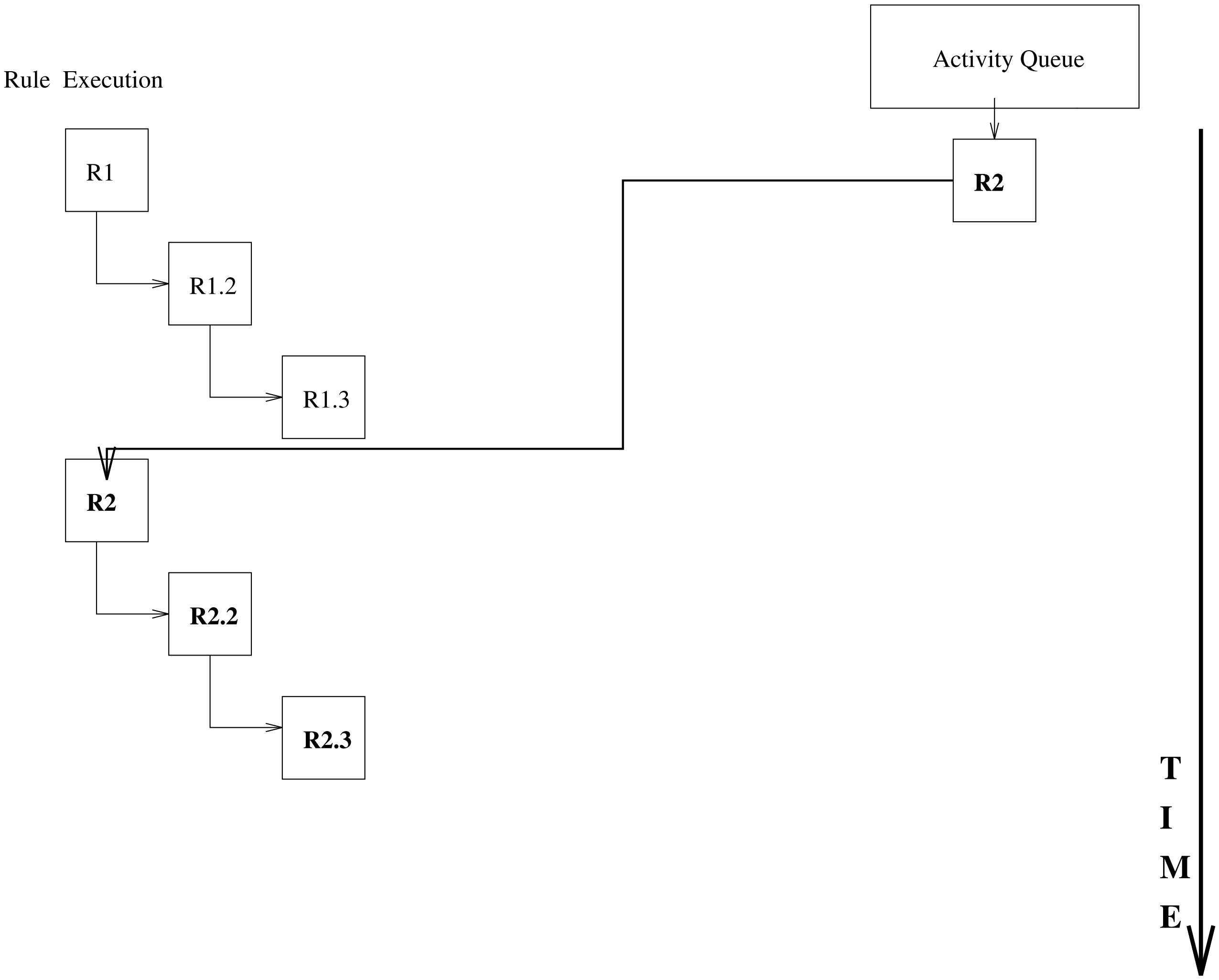

Figure 16: Sequence of rules' execution as it is in RIVENDELL.

the state of the whole process, given the considerations we made above on object locks [9]. For these reasons, we decided to perform the extraction only after the end of the whole chain, originating from the activity currently holding the tool instance. This avoids all the possible conflicts, even if forces longer waiting times on the queued requests.

7.4 The Watchers

These utilities play a fundamental role in our integration mechanism, since they constitute the devices in charge of interfacing between the environment and the tools , taking over — together with the scripts executed during the initialization pahse of the activity (See Section 6.3.2) — much of the functionality offered by SEL envelopes; MTP is conceived in such a way that external programs run pretty much in isolation after they are forked by their corresponding SPC and after any necessary initialization is performed by their customization scripts; they have no direct connection to any component of the system and dialogue with the environment's users via their own user interface (as it is provided and supported by xmove), just as if they had been invoked outside of Oz.

It is up to the watchers to inform the SPCs of all the actions of the external applications which are relevant to the PCE and especially to its data repository; they are in charge of maintaining control over the temporary directories where the copies of the objects on which the activities operate are located (see Section 6.3.2) and to notify the corresponding SPCs every time the objects are modified by the tools. When the activities are terminated, they then perform the transfer of all the final results back to the objectbase. Moreover, they can be involved in the customization phase, whenever a tool instance is started up and requires to obtain from the user some parameters which are not available through the OPEN-TOOL primitive (see Section 7.2.3).

Most of these tasks are accomplished by using *service files*, which are also kept in the tools' directories, each encoding or storing different kinds of information: the most important is perhaps the *filetable*, which contains a correspondence between each of the data files present

[9] In Figure 16 we show what actually happens to rule's execution and the Activity Queue under the current implementation.

in the directory and objects of the environment, plus markers to distinguish to which activity of those currently active in the tool session they pertain [10]. Other files with reserved names are as well internally used by the watchers to exchange with other parts of the system, via the SPC, various kinds of information needed by the watchers in different phases (see Figure 17).

Watchers revolve around a simple principle: they are able to keep under constant control the tools' directories and to recognize whenever a file is created or updated there. Following such an event, they decide what action to take, depending on wether the file name corresponds to one of the entries in their filetable or to one of their reserved ones. In the first case, they need to inform the SPC that the tool has performed an intermediate saving of some of its arguments, otherwise they operate on their own to extract data from that specific service file and perform whatever is appropriate, given the

reserved name of the file and its content. (Again, an example of the use of such service files is the procedure used to parametrize tools, illustated in Section 7.2.3.)

Their simplicity is also a source of flexibility: it is possible to implement several different flavors of watchers, and to modify the services they provide as accessory functions to the main one (that is, the control of file events inside a directory); we realized their potential and the range of tasks they can accomplish in collaboration with other parts of the system is indeed wide [11]. Our Grey Box integration experiments, for example, rely on a slightly more complex version of the watcher, coupled with an ad hoc function written for the tool in its extension language and loaded at start-up by its initialization script. Whenever a new environment file is written for the first time in the controlled directory, this event is easily caught by the watcher, which in turn notifies the tool's extension module. This is then in charge of automatically executing its standard loading procedure to acquire the file, thus bypassing the need for the user to issue it manually.

[10]This kind of data is crucial with special respect to the saving operations at the end of an MTP activity

[11]The central idea around which our watchers are built has been conceived by George Heineman, who also has done much of the work aimed to explore their potential.

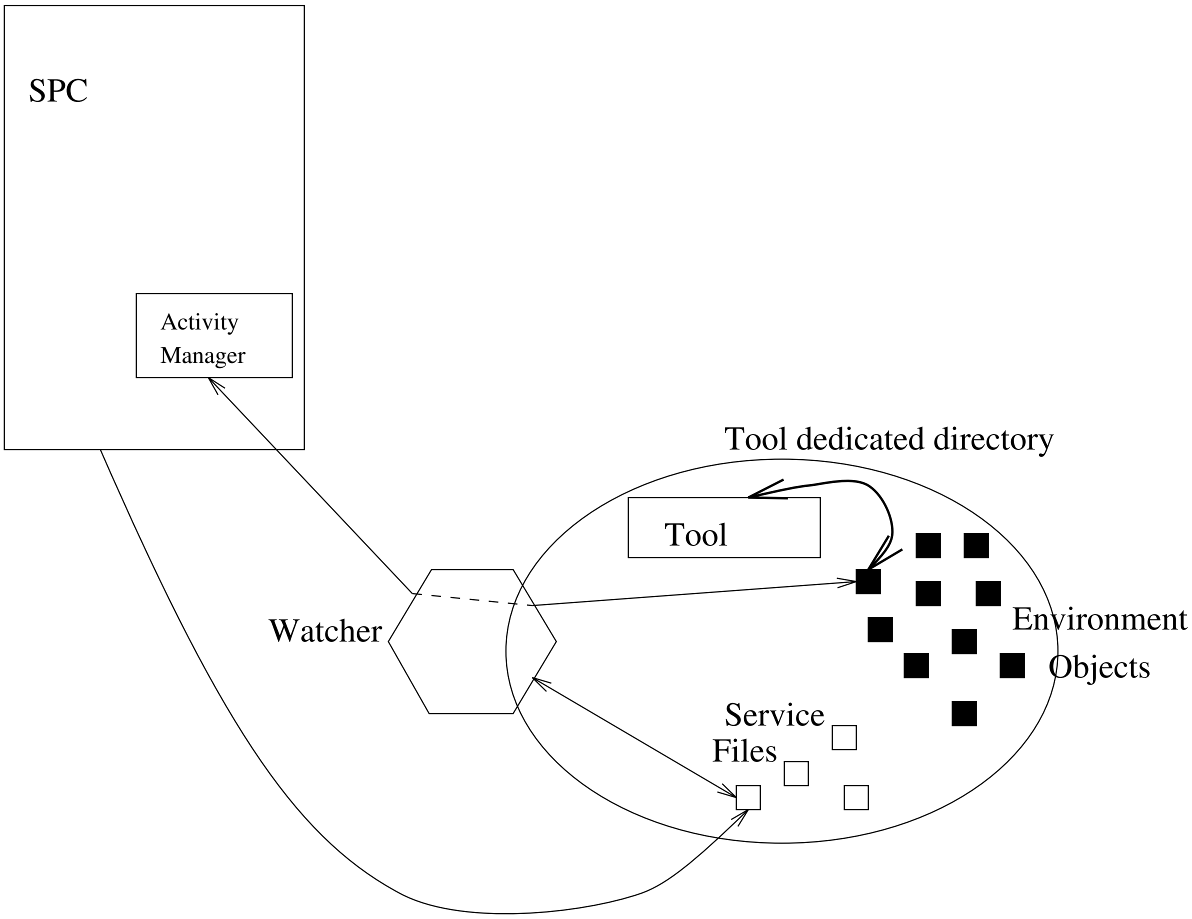

Figure 17: The work context for RIVENDELL watchers

8 Examples of Integration

MTP tries to offer to process designers a wide range of options, which have been studied to address the variability in the nature of COTS applications and to face adequately the different challenges they pose to Black Box integration.

Part of the flexibility of our approach is due to the use of customizable shell scripts for the initialization of the tool instances following the OPEN-TOOL primitive, and, at the activity level, to prepare and load the associated data which are then processed by the tool. Their goal is to permit a satisfactory and smooth wrapping of the external programs, in relation to the characteristics of our system and above all to the category assigned to the tool via the *multi_flag* specification in its MSL definition. An Administrator who is also a shell programmer with average expertise and with some knowledge of the purpose and the functions of these components inside the overall protocol would be able to easily set up the scripts. Indeed, such classification, even more than the abovementioned customization scripts, by capturing the different properties of each of its four supported categories, is the part of the system where most of the integration power of MTP resides. Even if it surely does not fully represent the possible diversity in the structure and the functionality of generic COTS tools, we believe it allows enough flexibility to accommodate a wide range of them with relatively few constraints and limitations. It certainly was beneficial that in Oz MTP is coupled with the SEL approach, which can adequately service a complementary and huge portion of a hypothetical toolset: this way the MTP design could neglect some related issues, its domain of interest remained limited and it could more closely be adapted to specific families of applications and to their peculiarities.

For each of the four classes described in the tool specifications, we have carried on various experiments, in order to define the degree of integration that can be reached and to identify limitations (either to the generality and the characteristics of the domain successfully addressed, and specifically to the completeness of our support to single meaningful cases) and unresolved problems we hope to address in future development of our multi-envelope system. We exemplify our work here, presenting some interesting results for each category.

8.1 UNI_QUEUE: Idraw

Idraw is a popular public domain drawing tool, commonly used to obtain pictures and diagrams in a postscript form, that provides an intuitive graphical interface employing an approach based on mouse movement and menu selection to operate on a virtual sheet shown inside an X window. It is strictly single-user and has the limitation of working on only one picture at any time: it is necessary to save the current project before loading a different one, since no buffering is supported. Idraw is in this sense an example of quite a large category of programs with similar properties and limitations. From our point of view, it presents some additional features of interest: it engages a considerable amount of system resources (large size) and needs a relatively long initialization time [12] following its invocation.

For all of these reasons, MTP integration as a persistent tool, via the UNI_QUEUE paradigm, seemed the most appropriate way to deal with the nature of Idraw, since it adapts well to the simple conceptual structure of this application. Moreover UNI_QUEUE, by disallowing multiple overlapping activities, preserves at any moment through its Activity Queue mechanism the integrity of the current data and of the work performed on it, hence enforcing on the process the respect of the intrinsical limitations of the wrapped program.

In general, UNI_QUEUE appeared suitable to deal with all applications which, like Idraw, don't present any specificity with respect to multi-processing or multi-user capability and to interpreter-like interaction, but are however more conveniently handled as persistent tools because of their size. Together, the cnovenience inherent in having such programs persistent and the queueing capability actually constitute the most valuable improvement introduced by UNI_QUEUE integration, with respect to SEL: they allow to run sequences of activities on the same instance without losing its intermediate state information and to manage all the incoming requests in the correct fashion.

[12] On a Sun Sparc 10 workstation, it amounts to about 15 seconds

8.2 UNI_NO_QUEUE: Emacs

Emacs is one of the most common and widely employed text editors available; its sophisticated functionality and the huge number of options and features provided make it a very useful tool, which reaches in itself the status of a small single-user programming environment. All of its commands are expressed with sequences of keystrokes, augmented with mouse pointing and selection; its latest versions present also menu selection at least for its main functions.

Among the properties of Emacs, one of the most important for us is its multi-buffering capability, through which it can deal at the same moment with multiple files (or other kinds of data displayable in its window), keeping all but the current buffer in the background and switching among them at the user's command. Coupled with the ability to split its display and hence show more than one of the loaded buffers, this feature is of great use to perform complex and incremental editing sessions involving as many different data sets as needed.

Our SEL protocol has satisfactorily integrated Emacs, but without being able to exploit its multi-buffering feature: each editing session from within the environment, since tightly wrapped by our scripts and accepting arguments only at the beginning, refers to a single file. Emacs has for a long time represented one of the typical cases of partial success of our Black Box SEL protocol, in which some peculiarities of the application don't fit well in the wrappers' design and are left unsupported, but it is nevertheless possible to integrate the program. As seen in Section 5, several attempts to extend our enveloping mechanism have therefore taken this tool as a testbench, trying to resolve the problems posed by incremental data exchange with the environment.

In MTP, we also gave a lot of attention to such issues and came up with the UNI_NO_QUEUE class, which represents our proposed solution. Our mechanism allows to run multiple activities overlapping in time, during the life-span of a copy of the editor and hence to load Emacs buffers with all the desired data for editing. When saving intermediate results, our watcher utility is able to map each modified file to the corresponding activity and hence to discriminate what objects must be modified inside the environment at the end of each of them. Compared with previous attempts based on extensions of SEL, one major improvement is the capability of handling each of the overlapping activities completely in isolation from the others, especially in its data recording and conclusion phases.

Another very important feature presented by Emacs is its extension language, called E-Lisp, that allows the users to define their own new functions and commands, customizing the program to their needs. We exploited it while experimenting with the Grey Box version of MTP, which accounts for a nicer, totally automated input of the activity arguments into the tools, achieved with little modification to our watcher and an ad hoc E-Lisp function.

Emacs is of course only an example of the programs which can fit in the UNI_NO_QUEUE frame. In principle, all the tools that can support simultaneous and independent service of multiple requests fall in this category, since it is easy to attach to each activity and its associated rule a separate internal processing thread that takes care of it.

8.3 MULTI_QUEUE: FUF

FUF is a sophisticated unification-based tool running on a Lisp platform, used, among other things, in the field of Natural Language Processing for the generation of sentences from syntactic data structures. It allows to define hierarchical procedures to accept and manipulate the input structures, by applying one or more layers of unification rules, in order to obtain as output at the end of the overall processing the valid surface forms, under the constraints posed by the rules. FUF is a typical Lisp-based interpreted application, 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 living on top of Lisp, it requires a great deal of resources and can certainly be defined as a large size tool.

The MULTI_QUEUE category allows in the first place the reuse of each copy of such computationally expensive programs, which can this way be shared by various users in a sequential way. Another important point in favor of this class is that the information retained in the tool's memory space and representing both the current state of the system and the history of its performance is fundamental to generate the answer to new queries. Because of this, it is possible and often desirable from the process point of view to consider a MULTI_QUEUE tool as a semi-permanent global service for agents in the environment engaged in some complex and composite software task. This makes even more valuable the ability of such a class to support long duration work sessions that go beyond any single activity and to ensure common access to them to any set of clients.

Coming to our testbench application, we can certainly 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 would facilitate the testing and execution of the various phases of the project through a sort of (although modest) teamwork. Sequentially, each developer would load in FUF its own program and run it on the appropriate data, at the same time producing with its output the input for the next step and putting the system in the correct state to begin a new procedure. Of course, the same situation and collaborative model could be referred to other programs, which outside the framework of Oz could not be employed in this way: from other interpretive tools, like AI ones running under Lisp, or single-user databases, to more canonical utilities, like document production ones, just to exemplify some applications of the same principle.

The most relevant consequence of the creation of this class is indeed that it, by exploiting Activity Queues and the xmove facility providing control of the user interface to any user involved in a session, 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. It is a completely new feature in our system, which is solely provided by MTP, and we believe it is one of its most interesting and meaningful results, because it effectively widens the spectrum of scenarios and tasks for which our SDE is suitable and of the options it offers to process developers and users.

8.4 MULTI_NO_QUEUE: Marvel

We decided to use as a testbench for this category the predecessor of Oz, which is also a multi-user PCE with a client-server structure, but with the important difference of being single-server. The main reasons for this choice are the familiarity we have with Marvel as a complete multi-user system and our interest in making as easy as as possible the development of processes for the Administrators, since the software task of building, testing and validating complex Oz processes is often non-trivial. [13] To achieve this goal, we need to be able to use Oz instances as tools, internally to P-Oz, while representing with rules and objects respectively all the process definition phases and the data like the environments' own directories, the MSL specifications, the SEL envelopes, the MTP scripts and so on. Obviously, a first step in this direction is the ability to use P-Oz to build and run Marvel processes, which in many senses can be seen as proper subsets of the more complex Oz ones.

Marvel, as a typical client-server system, poses in the most general case a problem we already outlined in the discussion on the MULTI_NO_QUEUE category in Section 6.3.3, that is, the need to treat differently the OPEN-TOOL primitive initiating a session (when it is necessary to start-up both the tool's server and a client) from the ones subsequently issued to join the session and obtain further copies of the Marvel client. Conversely, the last CLOSE-TOOL command in a session must deal with shutting down the tool's server. Moreover, since in our system it is possible to set up and employ a daemon, in order to bypass the manual invocation of the server at the start-up of the environment and to automatize this operation, Marvel can also be used to simulate the behavior of non-hierarchical architectures, which don't need special treatment for the activation of its first component. The technicalities of initiating and killing both flavors of multi-user systems are usually easily dealt with by the initial customization scripts invoked at OPEN-TOOL.

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; however, there are some important differences between the integration of collaborative and non-collaborative tools, that must be taken into proper account. In the latter case, in which each user works on his/her own, in isolation from the rest of the system (typical examples are represented by most databases or similar query processors) the different requests are

[13] PSL developed in the past a P/Marvel environment for Marvel, that can be used to build and maintain Marvel processes. It proved to be an interesting and useful utility, that added value to the project. We plan to realize a similar environment for and within Oz.

handled by the intrinsic multi-tasking capability of the tool and conflicts due to overlapping argument sets are sporadic and anyway resolved either before the activity phase of the rules, by the default concurrency control mechanism of Oz, or by the internal policies of the tool. In the former case, instead, even if it is once again safe to assume that most of the multi-user machinery is offered by the wrapped tool itself, the problem of shared use of data becomes more systematic. A simple example is that of a multi-user editor, employed in the context of a groupware task, as the one described by Dewan and Riedl for their FLECSE [10] toolset; the program itself permits and is able to deal with concurrent modification of its internal data, but for the environment's data repository managed by Oz it is necessary to come up with an ad hoc concurrency control policy that allows multiple write locks on the object containing the edited file.

Such flexibility must be provided by the PCE with some feature (in Oz it can be achieved by defining and loading appropriate control rules for the environment [24]); this is not in the strictest sense a part of the wrapping facility, but is nevertheless essential, in order to be able to take advantage of the integration of this class of tools.

The approach used within MTP to support access to a multi-user program for different agents is of special importance in the case of asynchronous systems. As we already pointed out in Section 5, it is currently possible to run N components of a multi-user synchronous system at the same time, using SEL and a specia system support mechanism [14]. However, the very nature of SEL envelopes does not allow the handling of asynchronous situations, which are instead easily dealt with by the concepts of persistent tool, through the session mechanism. The necessary research and work to integrate these two separate approaches will be pursued as one of the future extensions to Oz.

8.5 Lessons Learnt and Suggested Future Experiments

The examples mentioned above represent only a few of the possible cases that can be effectively addressed by RIVENDELL. However, they already have given us valuable feedback on what are the strong points and the limits of MTP and suggested future research lines.

[14] This feature was developed by Israel Ben-Shaul for his PhD thesis [3].

For example, we could see that our method still falls short — under certain points of view — with regard to an easy Black Box integration of multi-user systems via the MULTI_NO_QUEUE paradigm. One of the most evident proofs is that the envelope scripts to be executed at the moment of the OPEN-TOOL commands are usually much more complicated than for any other class [15]. The problems mainly reside in the mismatches that occur because of the client-server vs. network archtectures and the open-session vs. join-session cases. We realized that a more detailed description of the operations to be performed in each of these situations is needed at the level of the session-handling primitives, which should be probably enriched both in their syntax and semantics.

Another important addition to such primitives would be their parametrization, that would allow to provide them with arguments coming from the environment. Supply of arguments to OPEN-TOOL is currently allowed only through a "back door" mechanism, that employs the watchers and the SPC and holds no relation to the data definition in the objectbase, nor accounts for any control on what data is accessed by the process. A serious drawback descends: it is not possible to enforce any concurrency control, a problem that is again particularly serious in the case of MULTI_NO_QUEUE tools, in which the participants to the same session most likely share a relevant number of objects. By formalizing the parameters' acquisition during the customization phase of a persistent tool, we could overcome this problem, by using the same coordination mechanism already provided at the rule level.

Another improvement that is clearly needed is a mechanism to write envelopes (either the customization or the activity ones) in a more user-friendly and automated way. SEL supports a language providing some primitives that augment common shell script and that are compiled into shell commands by an ad hoc utility. This is very convenient to hide many repetitive sets of statements and to abstract burdensome levels of details, thus making the task of writing envelopes easier for the Administrator. MTP would greatly benefit from such a facility in the same way as SEL does, since repetitive and internal housekeeping commands are quite frequent and at the present stage must still be carefully and manually inserted into

[15] In our experience, we could see that the complexity of the envelopes usually gives an empirical measurement of the difficulties found by the process designer to adapt the tool to the constraints imposed by the integration protocol.

the script by the process designer.

To explore the potential of our integration method and find the best ways to improve it, we also plan to continue experimenting with an increasing number of tools that fall the categories of interest of MTP. Among the ones we have been considering, there are:

- Relational and object oriented databases, as typical query systems, and at the same time as large-size tools that are quite demanding with respect to computational resources (single-user databases can be instances of UNI_QUEUE or MULTI_QUEUE, while multi-user ones can be classified as non-collaborative MULTI_NO_QUEUE).

- Single groupware applications aimed to the production of documents or of program source code, as for example DistEdit [31] that deals with group editing, or GroupDesign [28] that is oriented towards drawing in structured graphics, as is the LBL Whiteboard, a public domain product (all of these collaborative applications can be well representative of MULTI_NO_QUEUE).

- Tool-kits that support collaboration within some set of software activities, like the abovementioned FLECSE, which employs various dedicated groupware tools, all built on the common framework provided by Suite [9].

- Complex and large-size knowledge-based CASE tools and systems, as Concept Demo [8], that employs AI planning formalisms to guide the process of developing software artifacts, or Refine, that proposes a programming language, a data repository in the form of a knowledge base, and a set of dedicated utilities, all aimed to the production of Lisp applications.

9 Contribution

9.1 Outcome of This Study

Our work has been totally implemented and integrated within the code of Oz; the PSL of Columbia University plans to conduct further research along these lines, to address the most problematic issues and to pursue the most promising facets presented by RIVENDELL.

In the context of Oz, MTP has contributed to expanding the spectrum of COTS tools that can be efficiently wrapped and serviced within an environment, increasing the use of our SDE, as well as its conceptual generality.

We believe we have introduced with RIVENDELL a few original and useful concepts in the domain of Black Box tool integration for PCEs; among them, in our opinion the most interesting are:

- The idea to employ multiple enveloping protocols for Black Box integration, with the purpose of increasing the flexibility and the modeling power of an environment already exploiting tool enveloping.

- The presentation of a protocol that tries to address the needs of classes of COTS tools that are rarely included in generic SDEs, at least in a Black Box fashion, partially because of the intrinsic difficulties in their integration — given some of their characteristics, as large size, long duration, and complex interaction models (interpretive and/or collaborative).

- The introduction of a categorization of those tools accordingly to their multi-tasking and multi-user capabilities, with the purpose of facilitating accommodation and most complete exploitation of their most valuable features, while ameliorating the complications inherent in their peculiarities.

- The realization of a new flavor of client process for our PCE, maintaining no direct interaction with the human agents in the environment, but to which can be delegated certain specific operations, that are not feasible or convenient for the normal interface clients [16]. We believe this is a contribution that is relevant beyond the scope of the RIVENDELL project, since it is an architectural feature that can be useful in the most general context. For example, the delegation to an SPC of a certain process fragment

[16] The design and implementation of this system component has be carried out in close collaboration with Peter Skopp, of the PSL at Columbia University, who has been developing a "proxy" client for Laputa [46]. The main difference is that an SPC can service different GPCs in their tool-related operations while each proxy is dedicated to a single client, as auxiliary in low-bandwidth operation.

can overcome architectural walls (e.g., it is possible to ask to SPC running on a DEC host to perform a compilation for that architecture, even if the user is currently operating with his/her own GPC from a Sparc workstation, or vice versa).

- The new concept of *loose wrapping* in tool enveloping, as opposed to the usual envelopes that completely encapsulate the external programs during their entire processing cycle and life span. Loose wrapping does not modify or limit in any way the use of the program, since it is only interested in checking its operation at some meaningful stages. By embracing this approach, it can also be possible to conceptually identify what such important events are, from the point of view of a system requiring Black Box integration (Some work addressing similar theoretical issues and involving Marvel has been carried out in the context of the *Provence* [33] project.)

- In connection with loose wrapping, the idea of interfacing our envelopes with pre-existing, long-lived instances of the programs (i.e. persistent tools) rather than with new ones, initiated ad hoc for the processing needs of the single activities.

- The extension of some intrinsically single-user tools to partial multi-user operability, performed by our protocol. This allows to describe, explore and exploit new collaborative scenarios for SDEs, without the need to acquire and successfully integrate complex systems explicitly designed for groupware.

- A new mechanism to effectively exploit full multi-user COTS products (especially asynchronous ones, see Section 8.4) from within our PCE, opening the field to the discussion and implementation of collaborative process models, which could be based on this work and rely on our approach to integration and could concentrate on different related issues as concurrency control, delegation, and the like.

9.2 Open Research Opportunities

The area of research over tool integration inside SDEs is certainly much wider of what is covered in this work. Even if we limit our scope to the Black Box protocols and to the families of tools we addressed in RIVENDELL, we can easily recognize that numerous alternative points of view and approaches exist, that are at least as valid and worthy of exploration as ours. Moreover, we can see that also within the lines of our research there is room for great improvement and extension, and for the investigation of a number of interesting issues.

Among them, as we have already mentioned, an important role would be played by the augmentation of the session-handling primitives and in certain senses of the session concept altogether, within the context of the definition of a software process. A fuller integration of sessions with the other facets of Oz could lead to interesting consequences: for example, the ability to attach agendas to an open session (i.e. to define a series of tasks that users willing to join or leave a certain collaborative session must perform); another possibility could be the instantiation of (possibly even delegated, i.e. created on behalf of different agents within the environment) sessions as an effect of rule chaining or also of the selection of the OPEN-TOOL and CLOSE-TOOL commands themselves (actually we already have proved, by handling Session Queues and atomic sessions, that the above is practically feasible).

Another interesting issue is in our opinion the loose wrapping approach to enveloping, that could be seen as an hybrid between conventional envelopes and a broadcast message server, as in Field, or its proposed extension Forest [18] [17], in which the attention is shifted from the external program as a whole, to single meaningful events caused within the environment by its use. This kind of monitoring can be a precious information resource not only for a specific SDE, but in general. Therefore it holds promise as a mechanism that could be used to support the interface between very diverse CASE systems that nevertheless need to share data or to collaborate in some fashion, with the only requirement that they must be able to dialogue with the monitoring components [18]. To achieve this, utilities sporting a more sophisticated design based on both theoretical considerations (i.e. what classes of

[17] The most relevant difference is that in generic message bus systems the tools are expressly and accurately designed or modified to create, send, receive and interprete messages informing other components of their activities, while in our approach it is up to the watchers to capture the meaningful events by "spying" on the tools, which are not in any way conscious of their presence and purposes.

[18] An example of a different CASE system that could be interfaced to Oz via *Rivendell* is ProcessWEAVER; we believe that certain similarities we found in it, such as the presence of a toolbase where the external applications are declared and modeled, and the wrapping approach carried out by interpreted envelopes, could be quite useful in this respect.

tool-specific events are interesting from the points of view of integration and cooperation) and on architectural and implementation-specific issues, should substitute our unambitious "lightweight" watchers.

10 Appendix

10.1 Rivendell Administrator Manual

10.1.1 Definition of MTP Tools

SEL is the default for tool categorization in MSL. In order to create an MTP tool, the process Administrator must explicitly insert specific information in the MSL TOOL definition. As shown in Figures 3 and 4 additional data is required for MTP, in the form of six parameters enclosed in square brackets.

The setting of the parameters should comply with the following guidelines:

- **protocol :** MTP
- **path :** String indicating the complete path in the environment's file system for the tool's *customization envelope* (See Section 10.1.2), or, if no specific operation needs to be performed at start-up, the complete path to the tool's executable. No default is provided for this parameter, that must always be provided when the value of **protocol** is "MTP".
- **host :** String containing the complete Internet address (IP form) of a machine located in the same Internet domain and sharing the same file system as the Oz server where the process definition and data are stored. It is used to designate the host where the SPC handling the MTP tool must run (and, as a consequence, where all the instances of that tool will live). When this specification is missing, the address is deduced by looking up the **architecture** field; in case that information is also missing, the default host machine is the same where the GPC asking for the MTP tool runs. **Note:** It is perfectly legal to assign the same machine to more than one MTP tool; however, for the

sake of efficiency, the Administrator should always be aware of the computational load implicit in handling each of the different applications and try to set the parameters in such a way to keep it as low and balanced as possible.

- **architecture :** The default value is "sun4". String values for other architectures that support Oz can be specified. This parameter comes into play if no value for **host** is given, but a restriction on the architectures on which the tool is available is present (e.g. due to licensing or structural issues). In this case, the Administrator should create into the environment's directory a text file, named **.MTP_default**. Its content should be in the following form:

```
<architecture A> : <host machine A>
<architecture B> : <host machine B>
<architecture C> : <host machine C>
...
```

  and indicates what is the default machine to be chosen for each supported architecture. For example:

```
sun4 : foo.bar.columbia.edu
dec : foodec.bar.columbia.edu
...
```

- **instances :** Non-negative integer, specifying the maximum number of instances of the tool allowed to be running in the environment at the same time. The value 0 is legal and it is used to indicate that no upper limit is enforced. The purpose of this parameter is to deal with floating license restrictions, as well as to limit the burden posed on the tool's dedicated host machine, especially by computationally expensive persistent tools.

- **multi_flag :** The four legal values are: UNI_QUEUE, UNI_NO_QUEUE, MULTI_QUEUE, MULTI_NO_QUEUE. Each of these categories is handled differently by the MTP protocol, since they presuppose diverse multi-tasking and multi-user capabilities. What follows is sketchy descriptions of each category and of the properties they model and support; refer to past Sections 6 and 6.3.3 for an extensive and detailed discussion of their peculiarities and of the corresponding work models, as supported by RIVENDELL.

- **UNI_QUEUE** It is intended to encompass tools which have neither outstanding multi-tasking nor multi-user capabilities, but are intrinsically "heavyweight", with regard to initialization time and amount of allocated resources. This category allows the issuing of multiple overlapping activities on the same instance of the tool, but services them in a sequential order.
- **UNI_NO_QUEUE** It applies to single-user applications that support the separate and simultaneous handling of disjoint data sets without interference. In this case, multiple activities — each with its own parameters — can be directed to the same instance of the tool and are serviced in parallel by the protocol. It is important that the Administrator validates his/her choice to include a given application in the UNI_NO_QUEUE category by preliminary testing the functionality of the tool and its effective ability to manage overlapping threads; applying the UNI_NO_QUEUE work model on tools that do not support multi-tasking might result in the loss of part of the work previously accomplished during the tool session.
- **MULTI_QUEUE** It presupposes or enforces some form of sharability of the tool's resources among multiple human agents, in a sequential way. It allows different users to require service to the same copy of an application, and "passes around the token" in a FCFS fashion; the tool is dispatched to the next user as soon as it is free, i.e. after the end of the currently executing activity. Typical candidates for this category are exemplified by interpretive or query-based applications, or single-user databases.
- **MULTI_NO_QUEUE** It is intended to describe complete multi-user systems, made up of multiple components, each of which is assigned to a different user, and all concurrently operating, either collaboratively towards completion of a common task, or independently, but nevertheless sharing some centralized computational

resource. The protocol dispatches the components to all clients who request them, by joining a specific session; each session (and not each single component) maps to an instance of the multi-user tool. Submission of multiple activities to each component is allowed, but they are serviced sequentially, exactly as if the component were a UNI_QUEUE tool on its own. Multi-user databases, client-server systems, collaborative tool suites are some examples of applications that fall into this category.

Important: The listing of activities, each of which is executed during a rule and maps to an envelope, with the optional additions of lock information on the envelope's parameters, maintains for MTP the same syntax used by SEL. However, the semantics are different: the position, of the activity declaration, i.e. the tool it is assigned to in the MSL TOOL class definition, was simply a convention for the SEL protocol, but is crucial for MTP. This happens because each instantiation of that activity will be directed to and executed on an instance of that persistent tool, as defined by the six parameters enclosed in square brackets we discussed above.

10.1.2 MTP Envelopes

MTP introduces two separate kinds of wrappers that are currently composed as plain shell scripts, following in certain cases fixed and repetitive command patterns. Besides scripts invoked and executed during each activity, which correspond to SEL envelopes, there are optional *tool customization* ones, that are run at invocation of an instance of a persistent tool.

Customization Envelopes The Administrator has the choice to assign to the **host** field in the TOOL declaration either simply the path to the binary code of that application, or the path to a customization envelope, that is used to perform preliminary operations and that invokes the tool itself from inside its body.

The reasons for such a choice can be very different: for example the script can be used just to invoke a tool with some parameters defined ad hoc, or extracted from the Unix

```
#!/bin/sh
...
# invocation of the program ''MTP_tool'' with some parameters
#and recording of its Process ID
        MTP_Tool -x -w &
        CLIENT_PID=$!
...
# trap a request to kill this tool instance after the execution of a
#script ''Close_Script'', containing a series of ad hoc commands.
       trap ' Close_Script $CLIENT_PID; exit 1' 2
```

Figure 18: Excerpt of a Tool Customization Script: Command To Be Executed Before Exit

environmental set-up of the machine that executes it; in other cases, they are used to define what is to be performed whenever the tool is closed, as clean-up actions or the like, via a trap mechanism (See Figure 18 for an example); in more complex situations, typically for multi-user tools, a series of preliminary checks might need to be performed and data must be retrieved, in order to correctly instantiate the copy of the tool. (See Figure 19.)

Moreover, we implemented in customization envelopes a way to request to the user invoking the tool to provide additional parameters to the script — and the tool's instance — in an interactive way. The excerpt of an hypothetical envelope in Figure 20 shows the sequence of shell commands to be performed for each argument [19].

Activity Envelopes An activity envelope is invoked every time an MTP activity is executed by the system. The envelope files are kept in the environment's directory and follow the same name convention as SEL compiled envelopes, that is "¡activity-name¿.env". Their purpose is to parametrize the persistent tool with the arguments of the correspondent rule and to load them in the application's memory in a semi-automatized way, as explained in Section 6.3.2. Differently from SEL envelopes, they <u>are not</u> concerned with returning results to the nevironment at the end of the activity, since this task is accomplished by the watcher program employed by MTP (See Section 7.4) .

[19]Notice that the file names **prompt** and **reply** found in the excerpt are crucial, since they are service files recognized and manipulated appropriately by our watcher utility (See Section 7.4).

```
#!/bin/sh
#initialize variables
VARIABLE_X=1
VARIABLE_Y=0
CURR_DIR=`pwd`

# look if service file Useful_Info has already been created
FOUND=`find . -name Useful_Info -print`

#if environment dir. is not found
if [ "x$FOUND" = "x" ]  #Useful_Info not present
         ...
#invoke tool MTP_Tool with certain parameters
         MTP_Tool VARIABLE_X CURR_DIR
else                   #Useful_Info already present
         ...
#invoke tool MTP_Tool with other parameters
         MTP_Tool VARIABLE_Y CURR_DIR
fi
```

Figure 19: Excerpt of a Tool Customization Script: Preliminary Operations

```
#!/bin/sh
...
# prompt the user with a request to provide a path for the data directory
         echo "type data directory path:" >> 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 within
# a file named ``reply''
# look out for that answer
        FOUND=`find . -name reply -print`
        while [ "x$FOUND" = "x" ]
        do
              sleep 5   #wait for 5 seconds
              FOUND=`find . -name reply -print`
        done
# acquire content of the reply file
        read DATA_DIR < reply
...
# invocation of the program ``MTP_tool'' on the directory DATA_DIR
        MTP_Tool DATA_DIR
...
```

Figure 20: Excerpt of a Tool Customization Script: Interactive Request of An Argument for the Tool to The User

The shell commands issued within the scripts follow a fixed sequence and are conceptually divided in 3 phases, in which each envelope does the following:

1. It copies the **file arguments** provided by the rule into the persistent tool's dedicated directory (the path to this directory is also passed to the envelope as an initial parameter).

   NOTE: Since the parameter passing mechanism has not been chaned from SEL, there are currently two serious limitations that will be removed in the future: it is the Administrator's responsibility to keep track of the position of the file parameters in the sequence of arguments and therefore to provide the correct arguments to the *cp* statement; moreover arguments including a set of objects are not correctly handled. Both of these shortcomings could be easily overcome if MTP envelopes were compiled, thus augmenting them with a dynamic argument passing and with an argument type declaration, similar to the ones implemented for SEL.

2. It creates a service file named **filetable** in that same directory, to the benefit of the watcher utility, that will use it throughout the duration of the activity. The statements accomplishing this are repetitive steps that could be easily be transformed in a single command, were MTP envelopes compiled in the same fashion SEL ones are.

3. It performs a series of *echo* commands, whose arguments are textual strings all beginning with the prefix #***# and contains the instructions for the tool's user, in order to guide the loading of the arguments of the activity in the tool's memory. The content of each *echo* statement is shown to the user in a special pop-up window, in sequence. The prefix is essential in order to differentiate these special messages from other output, that would be sent instead to the window dedicated to the activity (its *rframe*) and displayed there. This part of the activity envelope is actually the only that is really dependent on the nature of each single tool; nevertheless, the echo commands with prefixed argument could be easily transformed in a new primitive of compiled MTP envelopes.

A commented and complete example of an activity envelope can be seen in Figure 21 and can be used as a template.

```
#!/bin/sh
#input parameters:
#                       $1 tool dir. <------ MTP additional parameter
#                       $2 file
#                       $3 status
#                       $4 file
#                       $5 rule identifier <------ MTP additional parameter
#                       $6 client identifier <------ MTP additional parameter

####### 1st part: copy of file arguments into the tool directory ######
cp $2 $4 $1  #copy all FILE parameters in the tool dir.

####### 2nd part: preparation and creation of ``filetable'' ######
FileName1=`basename $2`         # for all the FILE parameters
FilePath1=`echo $1/$FileName1`  # for all the FILE parameters
FileName2=`basename $4`         # for all the FILE parameters
FilePath2=`echo $1/$FileName2`  # for all the FILE parameters
F_LIST_DUMMY=$1/filelist_tmp    # always
F_LIST=$1/filetable             # always
touch $F_LIST_DUMMY             # always
echo $6 $5 $FileName1 $2 >> $F_LIST_DUMMY  # for all the FILE parameters
echo $6 $5 $FileName2 $4 >> $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

####### 3rd part: customized on the single tool #######
echo \#***\#TYPE: CTRL-O                # tool-dependent
echo \#***\#SELECT $FileName1           # tool-dependent
echo \#***\#CLICK on "Open" button      # tool-dependent
#if the status argument of this envelope has a certain value, take action
if [ $3 = "Status_foo" ]
then
        echo \#***\#TYPE: CTRL-X $FileName2    # tool-dependent
fi
```

Figure 21: Template of an Activity Envelope

10.2 Rivendell User Manual

10.2.1 Sessions: Handling Instances of MTP Tools

In order to operate on MTP tools persistently [20], the user must in general issue session-handling primitives, represented by the OPEN-TOOL and CLOSE-TOOL commands. These commands are made available on Xview and Motif clients through the *Session* menu. No support for tty clients has been implemented to date.

To initiate a session, the user must click on the corresponding button and select the OPEN-TOOL menu entry. A submenu is displayed, showing the identifiers of all the tools defined as MTP by the Administrator in the toolbase of the process model. Selection of the needed tool is accomplished by choosing its identifier.

At this point a new submenu appears, listing the options of the OPEN-TOOL command for that tool. These include an entry marked **New** (the default), that must be selected when asking for a new instance of the application (an entirely new session), plus entries showing identifiers for each currently active session employing that program the user is allowed to join. [21] These identifiers are generated by the system at the moment of session initiation and include the user ID of the agent who created them.

Following the OPEN-TOOL command, the system executes whatever is necessary to comply with the request; this may include some requests for initial parametrization of the application, which are performed by providing the user with a message and a prompt, asking him/her to indicate his/her choices with textual input. If the invocation is successful, Oz finally returns a message in the client's Message Window, reporting the type of operation accomplished. The nature and the current state of the tool are then checked by the system, and, if it is the case, its User Interface is dispatched to the user's monitor for immediate use [22]. In case the requested instance of a tool is not available, due to failures either by the tool itself or by the system, the user is notified with a pop-up message. When the access to

[20] See Section 5 for the meaning of this term in the context of MTP.
[21] This applies only to collaborative tools of the MULTI_QUEUE and MULTI_NO_QUEUE classes.
[22] In general, this applies to tools of the UNI_QUEUE, UNI_NO_QUEUE and MULTI_NO_QUEUE classes that sport GUIs, while for MULTI_QUEUE ones it may or may not happen, depending if the instance is currently executing activities on behalf of other participants in the session.

an instance is precluded by limitations specified in the toolbase (i.e. the maximum number of instances the system can support) the user is notified and asked if he/she wants to rollback his/her request altogether or to place it in a queue; in the latter case it will be automatically serviced whenever enough tool resources become available for it.

To terminate a session, the user must select within the *Session* menu the CLOSE-TOOL option, which again lists entries for all the MTP tools; each of those shows the identifiers for the currently open sessions, plus the label **Queued**. The latter is used when asking to delete requests which are still unserviced and waiting in the queue, while selection of one of the former labels is interpreted as a request to leave an active tool session. The systems acts accordingly, displays a message with the results of its operation in the client's Message Window, or, in case of failure or of illegal requests (i.e. issued on sessions in which the user has not taken any part), with a pop-up message. The actions performed may or may not lead to killing the UNIX process made previously available to the by the OPEN-TOOL command, depending if the user is the last active participant in that tool session or not.

Important: the current version of RIVENDELL does not support "implicit" termination of a session (that is, to close the copy of a tool via its own internal command(s)), but cannot disallow it. It is crucial that the users train themselves to finish up their sessions <u>only</u> with the provided session-handling primitive and <u>not</u> with the tool's means.

Note: If the user intends to employ a tool in a non-persistent fashion, similar to the working model supported by the SEL integration protocol, he/she is free not to employ the session-handling primitives described above. MTP supports *atomic sessions* (See Section 6), consisting of one MTP activity only (See Section 10.2.2), automatically and transparently instantiated by Oz at the rule invocation, and closed at the end of the corresponding task (that is, the chain generated by that rule).

10.2.2 MTP Activities: Interacting with the Tools

Once the user has obtained access to a copy of an MTP tool, via the session mechanism, as explained in Section 10.2.1, he/she is free to invoke rules that involve that application and manipulate pieces of data in the objectbase of the environment, following the usual Oz model.

The selection of a rule and its parametrization with arguments taken from the objectbase are accomplished in the usual way, based on selection of an item within the *Rules* menu and mouse pointing on the objects displayed in the main client window. A difference resides in that MTP supports overlapping requests for multiple rules employing the same pre-existing tool instance. The user is free to fire rules at will, which are all directed to his/her controlled application. The consequences of such actions are different, depending on the class of the tool; they might be immediately executed (for UNI_NO_QUEUE ones) [23], or sequentially queued. In the latter case, a notification message is immediately displayed in the Message Window of the client and the execution of that rule is delayed until the end of the currently active task.

The successful firing of a rule has two consequences:

1. The creation of an Activity Window (or Rframe), managed by the Graphical User Interface of the Oz client. This window is quite similar to the ones used in SEL activities, with two main differences: the string **MTP Activity:** is displayed in the header of the window, immediately before the signature of the rule, and two supplementary buttons, labeled *Good* and *Bad*, are present in its top-right corner. This are used to terminate the activity and to assign its return code, as explained below.

2. If the tool's GUI is not yet present on the user's monitor it is dispatched there by the system, for immediate use.

The role assumed by the Rframe is separated by the one played by the tool's GUI; the latter accommodates direct and full interaction of the user with the features of the application, while the former is intended, as it is in SEL, to show the progress of the task initiated by the rule, in all its components (that is, during the span of its rule chain(s)) in the top half, and to support I/O flow between the user and the envelope executing the activity in the bottom half.

[23]With this property , MTP intends to support *incremental* loading of various sets of data into memory during the tool's life cycle: this is useful for those programs that can handle multi-tasking or multi-buffering and that is not provided by the SEL approach (See Section 6.3.3 for an example of a UNI_NO_QUEUE session, showing overlapping activities.

Another very important difference from SEL is the way the activity and its corresponding envelopes are carried out: MTP currently accounts for only *semi-automated* execution of its activities. User intervention is usually necessary in the phase of loading data into the memory of the external program: this is accomplished by the system displaying appropriate messages in a pop-up window, with the purpose of guiding the user in such operation. The messages accurately indicate, according to the tool's nature, what actions must be taken to successfully load the parameters of the activity (e.g. menu selections, mouse actions, typing of textual commands or of shortcuts).

After this initialization phase, which must be correctly followed by the user, manipulation of the data with the means offered by the program is completely free for an indefinite amount of time. Saving of intermediate processing results via the tool's means is captured and supported by the protocol in a transparent way.

More user interaction with the protocol is requested by MTP in order to recognize the end of an activity: for this purpose, we have implemented a typical dual *commit vs. rollback* choice, that is given to the user through the *Good* and *Bad* buttons mentioned above. When *Good* is selected, the system recognizes the successful end of the MTP activity, records into the objectbase all the modifications operated on the various pieces of data involved and executes a corresponding set of effects. If *Bad* has been chosen, all the changes are discarded and do not influence the state of the environment and of its objectbase, and an alternative set of effects takes place. Clicking on *Good* without having made any modification to the arguments of an activity defaults to the set of effects, in the current version of RIVENDELL. This behavior is different from that of SEL envelopes, which always save the results of the activity, regardless of its return code.

The control of the MTP tool is held by the user until the end of the chaining resulting from the termination of the activity and dependent on the chosen set of effects, then is released. In the case of an atomic session (See Section 10.2.1), this event also causes the killing of the tool's instance.

References

[1] 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.

[2] Noureddine Belkhatir, Jacky Estublier, and Walcelio L. Melo. Adele 2: A support to large software development process. In Mark Dowson, editor, *1st International Conference on the Software Process: Manufacturing Complex Systems*, pages 159–170, Redondo Beach CA, October 1991. IEEE Computer Society Press.

[3] Israel Z. Ben-Shaul. Oz: A Decentralized Process Centered Environment. Technical Report CUCS-024-94, Columbia University Department of Computer Science, December 1994. PhD Thesis.

[4] 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.

[5] 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.

[6] Christian Bremeau. The PCTE Contribution to Ada Programming Support Environments (APSE). In Fred Long, editor, *Software Engineering Environments International Workshop on Environments*, volume 467 of *Lecture Notes in Computer Science*, pages 151–166, Chinon, France, September 1989. Springer-Verlag.

[7] Geoffrey Clemm and Leon Osterweil. A mechanism for environment integration. *ACM Transactions on Programming Languages and Systems*, 12(1):1–25, January 1990.

[8] Michael DeBellis, Kanth Miriyala, Sudin Bhat William, C. Sasso, and Owen Rambow. KBSA Concept Demo. Technical Report RL-TR-93-38, Rome Laboratory, April 1993.

[9] Prasun Dewan and Rajiv Choudary. A High-level and Flexible Framework for Implementing Multiuser User Interfaces. *ACM Transactions on Information Systems*, 10(4):345–380, October 1992.

[10] Prasun Dewan and John Riedl. Toward Computer-Supported Concurrent Software Engineering. *Computer*, 26(1):17–27, January 1993.

[11] Klaus R. Dittrich, Willi Gotthard, and Peter C. Lockemann. DAMOKLES — a database system for software engineering environments. In Reidar Conradi, Tor M. Didriksen, and Dag H. Wanvik, editors, *Advanced Programming Environments*, volume 244 of *Lecture Notes in Computer Science*, pages 353–371. Springer-Verlag, Berlin, 1986.

[12] Mark Dowson. ISTAR — an integrated project support environment. In *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 27–33, Palo Alto, CA, December 1986. Special issue of *SIGPLAN Notices*, 22(1), January 1987.

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

[14] 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.

[15] J. Estublier, S. Ghoul, and S. Krakowiak. Preliminary experience with a configuration control system for modular programs. In Peter Henderson, editor, *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 149–156, Pittsburgh PA, April 1984. Special issue of *SIGPLAN Notices*, 19(5), May 1984.

[16] Christer Fernstrom. 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.

[17] F. Gallo, G. Boudier, and I. Thomas. Overview of PCTE and PCTE+. *ACM SIGPLAN Notices*, 24(2), February 1989.

[18] 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.

[19] Mari Georges and Claude Koemmer. Use and Extension of PCTE: The SPMMS Information System. In Fred Long, editor, *Software Engineering Environments International Workshop on Environments*, volume 467 of *Lecture Notes in Computer Science*, pages 271–282, Chinon, France, September 1989. Springer-Verlag.

[20] Mark A. Gisi and Gail E. Kaiser. Extending a Tool Integration Language. In Mark Dowson, editor, *1st International Conference on the Software Process: Manifacturing Complex Systems*, pages 218–227, Redondo Beach CA, October 1991. IEEE Computer Society Press.

[21] Adele Goldberg and David Robson. *Smalltalk-80 The Language and its Implementation.* Addison-Wesley, Reading MA, 1983.

[22] A.N. Habermann and D. Notkin. Gandalf: Software Development Environments. *IEEE Transactions on Software Engineering*, SE-12(12):1117–1127, December 1986.

[23] William Harrison. RPDE^3: A Framework for Integrating dtool fragments. *IEEE Software*, 4(6):46–56, November 1987.

[24] 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.

[25] 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.

[26] G. E. Kaiser, N. S. Barghouti, and M. H. Sokolsky. Preliminary Experience with Process Modeling in the Marvel Software Development Environment Kernel. In *23rd Annual Hawaii International Conference on System Sciences*, volume II, pages 131–140, Kona HI, January 1990.

[27] Gail E. Kaiser, Peter H. Feiler, and Steven S. Popovich. Intelligent assistance for software development and maintenance. *IEEE Software*, 5(3):40–49, May 1988.

[28] Alain Karsenty, Cristophe Tronche, and Michel Beaudouin-Lafon. GroupDesign: Shared Editing in a Heterogeneous Environment. *Computing Systems*, 6(2):167–195, 1993.

[29] Brian W. Kernighan and John R. Mashey. The UNIX programming environment. *Computer*, 12(4):25–34, April 1981.

[30] N. Kiesel, A. Schurr, and B. Westfechtel. GRAS, a graph-oriented database system for software engineering applications. In Hing-Yang Lee, Thomas F. Reid, and Stan Jarzabek, editors, *6th International Workshop on Computer-Aided Software Engineering*, pages 272–286, Singapore, July 1993.

[31] Michael J. Knister and Atul Prakash. DistEdit: A Distributed Toolkit for Supporting Multiple Group Editors. In *CSCW90: Conference on Computer-Suppported Cooperative Work*, pages 342–355, Los Angeles, California, October 1990.

[32] S. G. Kochan and P. H. Wood, editors. *UNIX Shell Programming*. Hayden Books, Indianapolis, 1988.

[33] 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, Garmisch-Partenkirchen, Germany, September 1993. Springer-Verlag.

[34] Programming Systems Laboratory. Marvel 3.0 administrator's manual. Technical Report CUCS-032-91, Columbia University Department of Computer Science, October 1991.

[35] 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.

[36] Robert Munckand Patricia Oberndorf, Erhard Ploedereder, and Richard Thall. An Overview of the DOD-STD-1838A (proposed), The Common APSE Interface Set, Revision A. In Peter Henderson, editor, *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 235–247, Boston MA, November 1988. ACM Press. Special issues of *Software Engineering Notes*, 13(5), November 1988 and *SIGPLAN Notices*, 24(2), February 1989.

[37] Harold Ossher and William Harrison. Support for change in RPDE[3]. In Richard N. Taylor, editor, *4th ACM SIGSOFT Symposium on Software Development Environments*, pages 218–228, Irvine CA, December 1990. Special issue of *Software Engineering Notes*, 15(6), December 1990.

[38] Steven S. Popovich and Gail E. Kaiser. An architectural survey of object management systems. *International Journal of Intelligent & Cooperative Information Systems*, 1(3&4):515–577, December 1992.

[39] James M. Purtilo and Pankaj Jalote. An Environment for Developing Fault-Tolerant Software. *IEEE Transactions on Software Engineering*, 17(2):153–159, February 1991.

[40] Reasoning Systems, Palo Alto CA. *Refine Software Development Tool*, 1986.

[41] Steven P. Reiss. Connecting Tools Using Message Passing in the Field Program Development Environment. *IEEE Software*, 7(4):57–66, July 1990.

[42] 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.

[43] Wilhelm Schafer, editor. *8th International Software Process Workshop: State of the Practice in Process Technology*, Wadern, Germany, March 1993. IEEE Computer Society Press.

[44] *2nd International Conference on the Software Process: Continuous Software Process Improvement*, Berlin, Germany, February 1993. IEEE Computer Society Press.

[45] 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.

[46] Peter D. Skopp and Gail E. Kaiser. Disconnected operation in a multi-user software development environment. In Bharat Bhargava, editor, *IEEE Workshop on Advances in Parallel and Distributed Systems*, pages 146–151, Princeton NJ, October 1993.

[47] Richard Snodgrass and Karen Shannon. Supporting flexible and efficient tool integration. In Reidar Conradi, Tor M. Didriksen, and Dag H. Wanvik, editors, *Advanced Programming Environments*, volume 244 of *Lecture Notes in Computer Science*, pages 290–313. Springer-Verlag, Trondheim, Norway, 1986.

[48] Richard Snodgrass and Karen Shannon. Fine grained data management to achieve evolution resilience in a software development environment. In Richard N. Taylor, editor, *4th ACM SIGSOFT Symposium on Software Development Environments*, pages 144–156, Irvine CA, December 1990. Special issue of *Software Engineering Notes*, 15(6), December 1990.

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

[50] Ian Thomas. PCTE Interfaces: Supporting Tools in Software-Engineering Environments. *IEEE Software*, 6(6):15–23, November 1989.

[51] Ian Thomas. Tool Integration in the PACT Environment. In *11th International Conference on Software Engineering*, pages 13–22, Pittsburgh PA, May 1989. IEEE Computer Society Press.

[52] Ian Thomas and Brian A. Nejmeh. Definitions of Tool Integration for Environments. *IEEE Software*, 9(2):29–35, March 1992.

[53] 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.

[54] A. I. Wasserman. Tool Integration in Software Engineering Environments. In Fred Long, editor, *Software Engineering Environments: International Workshop on Environments*, volume 467 of *Lecture Notes in Computer Science*, pages 137–149, Chinon, France, September 1989. Springer-Verlag.