Expanding the Repertoire of Process-based Tool Integration

— MS Thesis Proposal —

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

CUCS-004-94

February 9, 1994

Abstract

The purpose of this thesis is to design and implement a new protocol for tool enveloping, in the context of the Oz Process Centered Environment. This new part of the system would be complementary to the already existing Black Box protocol for Oz and would deal with additional families of tools, whose character would be better serviced by a different approach, providing enhanced flexibility and a greater amount of interaction between the human operator, the tools and the environment during the execution of the wrapped activities. To achieve this, the concepts of persistent tool platforms, tool sessions and transaction-like activities will be introduced as the main innovative features of the protocol. We plan to be able to encapsulate and service conveniently classes of tools such as interpretive systems, databases, medium and large size applications that allow for incremental binding of parameters and partial retrieving of results, and possibly multi-user tools. Marginal modification and upgrading of the Oz general architecture and components will necessarily be performed.

1 Introduction

The issue of integrating a set of tools in a Software Development Environment (SDE) [29] is of great relevance to the degree of functionality that the SDE can provide, to its flexibility and power, to its ability to model a variety of operations and to assist users in many ways and in different situations. The more generic is the set of software engineering activities to be supported, the more diverse are the necessary tools and the more flexible must be the integration principle. Therefore, in the field of Process Centered Environments (PCEs) [23] [24], which are a subclass of SDEs designed to define, enforce and support a variety of software processes using a built-in process modeling formalism, the need for such generality becomes clearly of utmost importance.

The approaches to tool integration can be very different. They vary from the definition of a tool family dedicated to the SDE, according to the specifications and the structure of the environment (an option that can be very costly without allowing a high degree of generality, although achieving the best results in terms of efficiency and simple design), to the use of interfaces supporting a dialogue between existing external tools and the SDE, that can be usually implemented with relatively little and repetitive modifications to the tools' structure and code, to the use of commercial off-the-shelf tools (COTS) without modifications. In the latter case the SDE must provide a generic mechanism for interaction with the tool, which allows for parameter passing and retrieving results, and for some deal of control over the activity performed by the tool. Such an approach is commonly referred to in the SDE community as *tool enveloping.* Conceptually, envelopes, beside executing activities, perform the task to extract data from the internal representation in the SDE, to present them to their "wrapped" applications in the correct format and, in general, to provide mapping between the system's data repository and the tool's own one.

Oz Overview

Oz [2] [3] is a multi-user PCE that realizes the process description and support with a rule-based approach, and stores all the data, software components and their mutual relations in an object oriented repository called the objectbase. It is based on the experience gained in developing the Marvel 3.1 PCE [16], to which it is intended to be the successor, and while it inherited from Marvel most of its main features, it differs from it in several ways, noticeably in the fact that it is more oriented towards distributed use. The implementation of this thesis will be carried on as a part of the Oz project.

Marvel 3.1 represented one of the few examples of PCEs employing already existing tools to carry on the activities defined in the process on its objects. No dedicated tools, nor code changes, nor recompilation were necessary, since Marvel fully exploited the principle of *tool enveloping*; Oz follows the same approach.

The current mechanism to achieve this kind of integration is called the Shell Envelope Language (SEL) [14]. SEL realizes augmented versions of shell scripts that handle the passing of parameters to the envelope from outside, execute the tool inside the script using the parameters to customize the execution and to provide the tool with arguments, and return to the external caller a status code and other relevant output data. Each SEL envelope is invoked during the execution of an Oz rule (defined in a specific language called MSL [1]), which consists of several different parts:

- A name and a list of typed parameters that it accepts. This is what is called the *signature* of the rule;
- The condition section: first, additional objects, related to the parameters, are gathered from the objectbase; then, the system verifies for each of the objects in this set if some specified properties hold. The objects which don't comply with these conditions are then discarded from the set;
- The activity section in which a rule-specific operation is performed on the objects collected by the condition. In this context, the SEL script is the *envelope* or *wrapper* that represents the activity's implementation. The activity executes in a very straightforward fashion: an input — execution — output sequence;
- One or more mutually exclusive sets of effects, to be chosen in accordance to the return code from the activity: the effects are assertions in which data returned by the envelope

are recorded into some of the objects bound in the condition section, therefore modifying the state of the objectbase and of the process.

One of the peculiar properties of the Oz and Marvel systems is how *process enaction* is carried on: following from the modifications incurred because of the effects, automatic firing of additional rules, whose conditions match the new state of the process, is performed. This is what we call *forward chaining*. *Backward chaining* can also automatically occur in Oz when a rule is invoked and its conditions are not entirely satisfied by the bound objects; rules whose effects could fulfill those requirements are then fired. Both backward and forward chaining are recursive and represent a method to provide the PCE with automation facilities and to enforce the process' policies and its desired behavior.

We call the protocol provided by SEL for handling tool enveloping and activities' execution a *Black Box* protocol, since it is not concerned at all with the internal structure and nature of the wrapped tool, but only with passing input data and retrieving outputs to the tool in a convenient way. The execution mechanism, as seen above, is very simple and therefore it is also quite useful. Actually, it can adequately support a rather wide range of conventional tools: most Unix utilities, for example, accept all their arguments from the command line at invocation time and return simple status information at the end of execution.

However there are numerous tools which don't fit this description and may be convenient to integrate into Oz processes. To do this it would be necessary to augment the current tool enveloping facility and its underlying Black Box concept.

2 Motivation

The main concern of this work is to design and implement a new tool enveloping protocol for the Oz PCE. It should be general enough to apply to several classes of tools, with different characters and needs. This should greatly extend the ability of Oz in describing and supporting an increasing number of different processes.

Of course, it is highly unlikely to be able to define a general-purpose mechanism that can encapsulate in our rule-based process description any chosen tool. It is therefore important to focus on some classes that we consider more important or urgent to integrate in our system.

Some interesting test cases would be:

- Tools that allow incremental request of parameters and/or return partial results during their execution; also, tools that support heavily interactive work sessions with the user. These tools are qualified here as *medium size*, referring to the duration of the work session and the amount of resources allocated to them. Good examples are provided by multi-buffer text editors such as GNU Emacs. We have already in the past conducted experiments on this class, aimed to provide a way to exploit at a greater degree its functionality, with some modifications to the Black Box approach of SEL. The testbench for these experiments has been the GNU Emacs text editor, for which an ad hoc protocol had been written. We classified it as *Grey Box*, since we needed to use the GNU Emacs extension language E-Lisp to implement it. (In principle, tool enveloping using a similar Grey Box approach can be done for all those tools having their own extension language. We classify an approach as *White Box* if the internal code of the tool itself needed to be manipulated.)

- Tools based on interpretive query systems: KBSA tools written in Lisp (for example, FUF or OPS5) or databases are classic examples. In this class we could gather all those systems that allow the users to run, after their initial invocation, series of functions, each having their own parameters. Many of these tools have an intrinsic interpretive nature and keep track of the activities performed and of the state of the data manipulated by querying the interpreter. The interaction between such systems and the user is heavy and long and the amount of computing power necessary to handle the queries and keep track of their results can be huge. Therefore, they are referred to here as *large size* tools. Even if, in principle, such tools could be handled by a Black Box protocol, with each query mapping to a different envelope, this is highly impractical, sometimes because of the start-up overhead, but mainly because it would not be possible to keep track of the state of the program and the data across different invocations, unless the work session and the state were saved and loaded each time (clearly an inefficient solution).

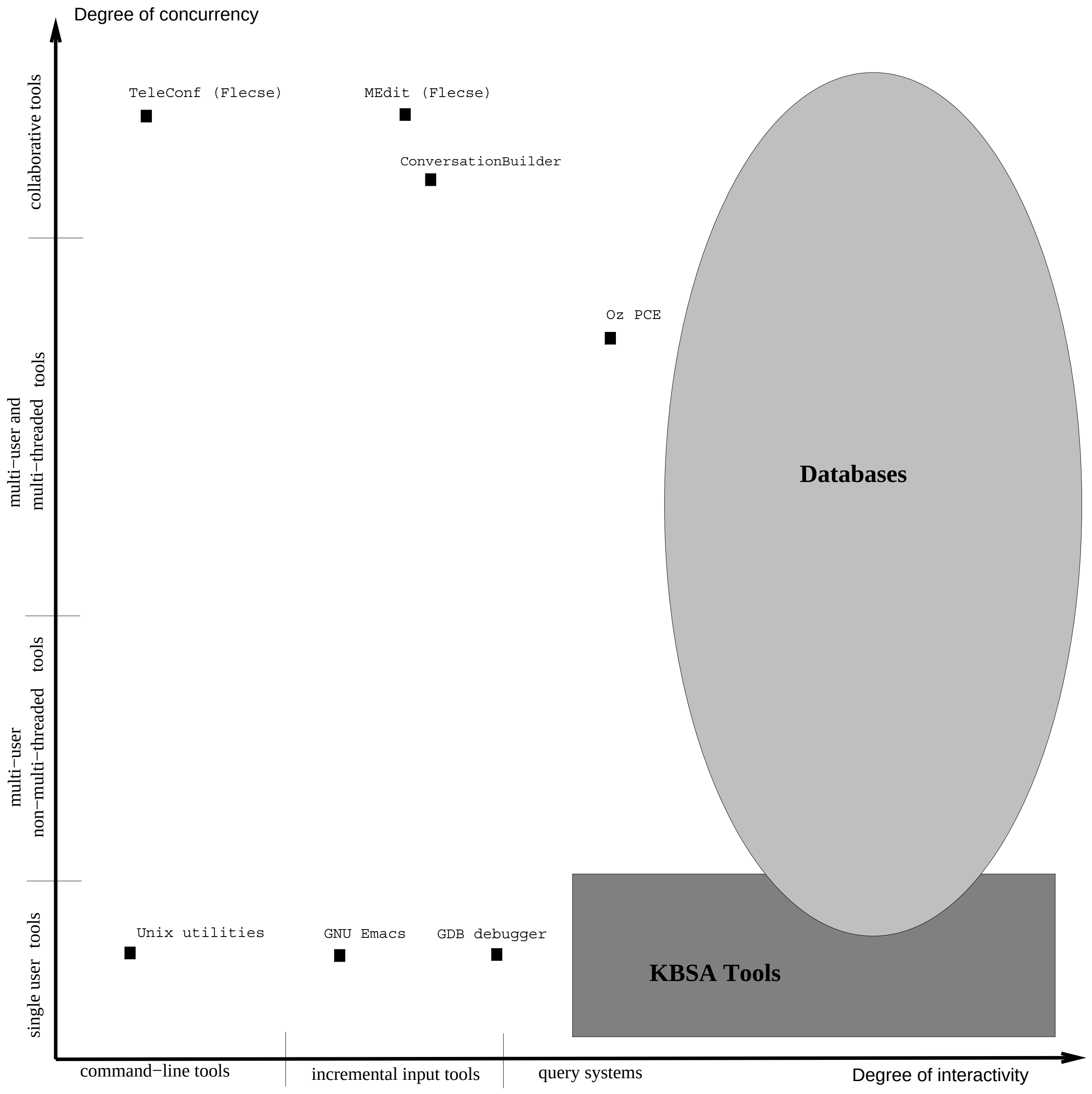

Figure 1: Dimensions of Interest for a New Approach to Tool Integration

To deal with such tools, the wrapping mechanism of the environment should be able to conduct a highly interactive transaction and should provide means to support the user during it: for example, dealing with automation of the most trivial tasks, coping with the changes made to the data (and hence the objectbase) during different phases of the transaction, suggesting to the user what operations are more likely to be executed at a given moment, and so on.

- Multi-user tools, including those allowing collaboration among the users (addressing issues like concurrent debugging, collaborative code inspection, or teleconferencing) , the ones which provide isolated service for multiple individuals at the same time (as for example some database systems), which we refer to as *multi-threaded* tools, or the ones accepting to service many users, queueing their requests and executing them sequentially, sometimes called *multi-user but non-multi-threaded* programs. Steps to investigate and experiment with collaborative SDEs have been conducted recently. It is clear that the use for such a tool family is endless in the context of a multi-user PCE such as Oz. Their integration and use would carry along a number of questions and problems not addressed yet, mainly related to other main components of our system, including the process modeling language and the concurrency model; however it is the job of the enveloping facility to handle their invocation and execution, as necessary parts of the integration.

These examples should be fairly representative of the reasons why we need a new tool enveloping protocol.

Some important characteristics we anticipate as desirable would be a greater degree of interaction between the human users, the wrapping facilities and the tools, as well as the ability to run several activities on the same persistent instance of a program.

We believe that such features would help to integrate a number of new tool families, proportionally enhancing the flexibility of Oz and its power to model a larger set of processes.

3 Related Work

Tool integration is a topic of central importance to every effort to build efficient and practical SDEs; several studies have concentrated on defining and exploring the meaning and the different dimensions of the term *integration* as applied to SDEs. Wasserman [30] for example identified five kinds of integration (platform, presentation, data, control, process). Moreover, Earl [10] proposed a well known reference model for Computer Aided Software Engineering Environments (CASEEs, another term for addressing SDEs), in which a lot of emphasis is on the issues of portability and interoperability of tools.

In the attempt to fulfill these requirements for integration in SDEs, a large number of different approaches to the problem have been investigated.

One of the most popular is equipping common tools with standard interface modules. These modified applications are then able to communicate with a centralized message server, in order to coordinate their operations. Here, the message server and the interfaces constitute the subsystem in charge of tool integration. This component is often referred to as a *message bus*. Such an approach is used, among others, by Field [22], Conversation Builder [17] [7], and SoftBench [6] [13].

PCTE [27] [11] is one of the most representative examples of efforts directed to define a widely recognized and accepted public standard for building tools with better portability. The aim 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. The result would be a generation of homogeneous tools, widely reusable under the PCTE specifications. Many SDE prototypes and projects [28] [5] [12] in Europe as well as in the USA have already adopted the PCTE standard. However, such a standardized approach is only useful if the SDE developers can or choose to abide to the standard conventions and their unavoidable limitations.

Another widely explored approach, and probably the most flexible and general one, is *tool wrapping* or *tool enveloping*, in which the aim is the encapsulation in the environment of external tools with no changes to their code. The envelope idea was first introduced by the

ISTAR [26] system.

Marvel envelopes are augmented Unix shell scripts [20], invoking external tools and able to achieve tool integration without modification in a Black Box [15] fashion. The Marvel project (on which Oz is based) also explored a different enveloping mechanism with the experimental implementation of a Grey Box style protocol, that allows feeding of parameters in an incremental fashion to medium size tools during their execution. The chosen test case was the GNU Emacs text editor. This protocol needs to add further functionality to the tools (the Emacs extension language was used in our test case) in order to equip them with some means to carry on a simple dialog with Marvel. The system then accomplishes incremental requests as if the same tool had been invoked multiple times with different arguments. However, all the invocations map to the same instance of the program. One of the limitations of our implementation is that all invocations must complete at the same time and cannot return separate status codes. This does not allow the separate handling of each object during the effect part of the rule: for example, if any of the files incrementally loaded in the multiple buffers of GNU Emacs is written at some point and then saved, all of them, regardless if they underwent changes or not, will be treated in the same way. Among the issues that were not explored thoroughly enough is also the potential interaction between multiple different rule chains generated by different argument requests. Such automatic rule invocations could at any point conflict with each other, for example by binding overlapping sets of objects, or by undoing changes to the objectbase just performed by another independent chain.

The Grey Box experiment was conducted on the Emacs case with an ad hoc implementation, which is neither robust nor reusable with different examples.

The implementation of our new protocol for Oz will maintain the extreme flexibility typical of the "wrapper" concept, trying to address most of the shortcomings of the simplistic Black Box mechanism. We call it the Multi Tool Protocol (MTP), where *Multi* refers to the fact that its enveloping facilities will accept to perform *multiple* activities with their separate data sets and *multiple* commands issued during each activity, as well as *multiple* users operating on the same instance of the tool. Once such features are available, the generality of our tool enveloping method for integration would be greatly improved.

The ability to integrate multi-user collaborative tools in SDEs would allow existing multi-user environments to support not only team work, but also concurrent software engineering activities in a very flexible manner. Currently, a few extensive tool-kits are available, that allow for cooperation: Patel and Kalter realized a Unix tool-kit called COeX [21], providing primitives for building diverse collaborative applications in a high-level fashion, abstracting basic multi-user issues and implementation details. Another effort, specifically directed towards group editing and using a similar approach, is DistEdit [19], which exploits the communication mechanism available in another tool-kit, called ISIS [4]. GroupDesign [18] allows group sessions and it is oriented towards drawing in structured graphics. It supports this kind of activity providing features as "Tele-Conference" and means of recognizing actions performed on the current project by each member of the group. Dewan and Riedl presented FLECSE [9], a Software Engineering environment using dedicated collaborative tools, all built on the common framework provided by multi-user Suite [8].

While all of these these tool-kits maintain a certain amount of integration, none of them can fully enforce rules and desirable behaviors in collaborative software development. We hope that the ability to use some of these tools inside a PCE such as Oz, intrinsically able to define and support such policies in its process, would benefit team work on software projects as much as the availability of collaborative applications and primitives already does.

4 Goals and Scope of the Research

In extending the Oz tool enveloping facilities to classes of applications that are not currently properly handled by the Black Box protocol, we focus on specific classes of tools, which we think expecially interesting and suitable.

The first and minimal goal would be to open Oz to integrate large size tools and query systems, such as Lisp-based AI tools or databases. We will use as a testbench for this phase an NLP tool named FUF, concerned with generation of natural language sentences, given a grammar and a well-formed data structure as input. The need, during a typical working session, to incrementally modify the grammar and the input and to run complex interactive debugging sections within Lisp and FUF makes this a perfect example to test a new model of activity protocol, that would go beyond the simple input — execute — output schema and would require more interaction to take place. Some other similar tools would be considered and tested for generality.

Moreover, to efficiently run multiple activities on FUF or other AI applications, the system should provide a way to maintain the underlying Lisp environment persistently through all related rule invocations. This would constitute a *platform* that can also be used by multiple tools to perform different operations.

This part of the implementation would also account for dealing with those medium size programs that allow for a relatively limited form of interaction, as incremental binding of new arguments and releasing of partial results.

An additional goal that could be achieved would be providing a way to integrate multi-threaded and non-multi-threaded multi-user applications. We believe that the platform concept expressed above could be helpful in this additional phase, the platform being the multi-user tool itself, to which the envelopes rely to fulfill requests. The basic concept seems to be the same, with a persistent process acting as basic support to the activities invoked via multiple concurrent rules.

We feel we would find increasing levels of complexity in integrating the following in Oz.

- Large size tools,
- Non-multi-threaded but multi-user tools,
- Multi-threaded but non-collaborative tools,
- Collaborative tools.

For the latter category, it would definitely be necessary to reuse other parts of our system, but the integration protocol could possibly vary just slightly from the one necessary for the previous ones. We perceive the implementation of the facilities to allow and control multi-user collaborative work from inside Oz mainly as an advanced feature of MTP, that could be the theme for a future extension or upgrade to the protocol's functionalities, if it cannot be directly addressed in this thesis.

5 Design Requirements and Highlights

The new protocol should either be fully compatible with or completely disjoint from SEL. The approach we foresee is that the process architect (also called the *Administrator*) would choose at the process definition phase which activities need the new protocol and which not, on the basis of the specific tools employed for that activity. In the Oz process definition language, MSL, activities are expressed as follows:

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

The envelope contains the actual tool invocation (the locks information is concerned with the desired concurrency and coordination policy for the parameters and will be briefly mentioned below). We could extend the MSL definition of TOOL, in order to include some other parameters expressing additional useful properties, one of which would be the chosen protocol:

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

The tool integration should be as transparent as possible to both the tool and the Oz system (that is, as little change as possible should be made to the code and structure of both); it is possible that some deeper insight of the structure of the tools is necessary to implement the new protocol, partially dropping the fact that COTS tools can be used without modifications (i.e., we might have to abandon for the MTP protocol the Black Box approach to tool integration we use in SEL).

We should fully support incremental binding of objects to be used in the tool operations, given the nature of the tools we want to integrate. In the current Black Box protocol, all the data needed by a rule and its activity are gathered from the objectbase at the beginning, in the parameters specification and in the condition section, which bind objects to the rule. In the activity section, the envelope receives sets of data extracted from those bound objects as its only input. The envelope and its tool then process them; once the activity section is entered there is no way to feed the tool with more pieces of useful information, since no further interaction is supported. The Grey Box experiment tried to address exactly this limitation and MTP would overcome it.

We need to support long-duration transactions involving the human user, the tool and Oz with its objectbase: attached to this fact are considerations about the locks and concurrency mechanisms to be adopted for objects involved in the transaction. It would be important to cause minimal hindrance at all to other users who would access those same objects from outside the transaction. For example, given an interpretive tool that allows for incremental binding of data and multiple incremental queries, we could have USER1 executing a rule, whose activity part (`<activityA> <arg-setA>`) would map to a transaction. In its context, USER1 would issue a number of different commands, sometimes introducing new data, which would be presented to the application in the same fashion as the initial parameters (e.g., `<arg-setB>` etc.); possibly, the results of each single command in the transaction context might be relevant to the process; they should then be sent to the environment to be recorded.

The duration of such an activity can be long and numerous objects may be involved in it as part of arg-setA (initially bound) and arg-setB (incrementally bound). If USER2 needed some of those objects for other tasks, we would run into concurrency control problems that could hinder USER2's work for a while, unless the concurrency policy is appropriately planned and finely tuned to the needs of the users. The option to define different types of locks on the objects passed to the envelopes, as we saw above, together with other means (not discussed here) allow the Administrator of an Oz environment to define flexible policies. We therefore believe this kind of situation can be properly handled, but the implementation of the protocol should not be too demanding, nor rely exclusively on the Administrator;

We need to introduce the concept of a platform process, that stays persistent across multiple invocations of rules, providing a basic environment on which to employ different tools in the same family, or to address multiple requests for operation to a single tool. This brings along some non-trivial issues, as for example how the system can recognize when an activity is over (expecially if it is an interpretive one), or how to deal with multiple simultaneous activities requested by different users. We can think of the platform process as having the same role that Unix itself implicitly holds now, when we run Unix utilities as tools: as this particular family lives on Unix, we could say, for instance, that FUF or other similar KBSA applications live on Lisp. We therefore need to provide a persistent Lisp instance on which to run those tools, when we support a process in which Lisp-based applications are widely used.

The protocol should be able to assist users with as much automatization as possible during operations related to the transaction in the light of the process designed for the given environment. In the context of a transaction, every time some intermediate results are sent back to the environment, there is potential for a change in the state of the process. This could lead to firing chains of rules, according to the Oz automation mechanism, as soon as the activity part of the rule is over and we get to the effect part.

6 Design Insights

6.1 Modification of the System Architecture

Oz is currently organized in a Client/Server fashion, with multiple clients attached to each server (see Figure 2).

The client processes usually represent users and their means to interact with the environment and to address requests to it. The server processes are persistent, with respect to the clients, and communication between clients and servers is handled via sockets. While the clients run under their operators' User ID, the server runs under the Oz User ID. Activities' processes are currently forked by the clients.

Since one of the basic points for our tool enveloping approach is to provide persistent

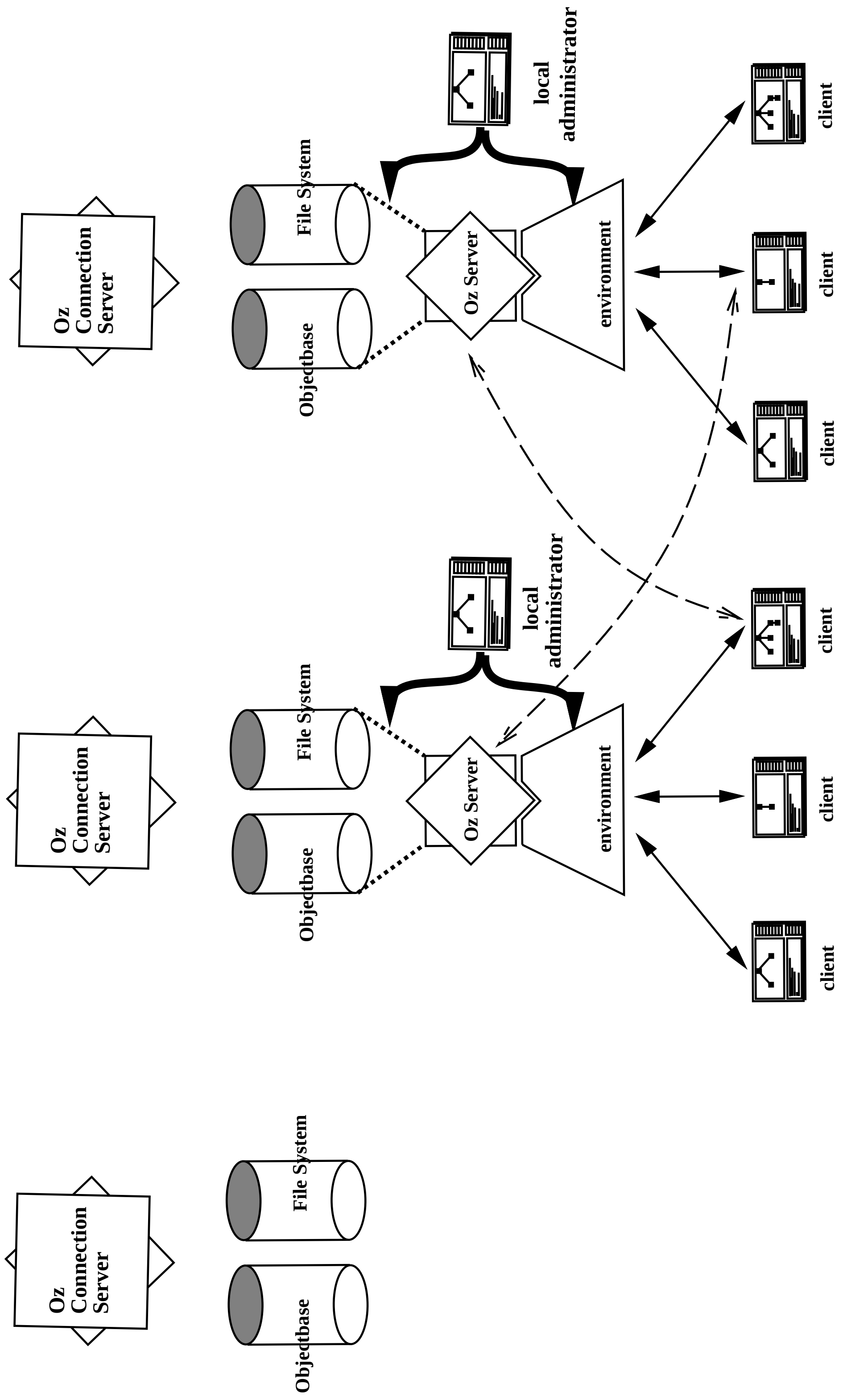

Figure 2: The Client/Server structure of Oz

processes, called platforms, on which to run certain families of applications, the system must have some means to deal with them. The servers are a persistent part of the model and it is therefore natural to think that the platform processes should be hierarchically dependent on them. In order to keep the greatest amount of modularity and to change as little as possible the structure of the system, we decided to introduce the concept of Special Purpose Clients (SPCs), which are created and handled by the servers, with the main purpose of forking and maintaining the platforms (see Figure 3).

SPCs differ from the usual ones (that we'll call from now on General Purpose Clients - GPCs) in a few ways, primarily in the fact that they are as persistent as their parent server and not used or controlled directly by any human operator. They are mainly a convenient mechanism to handle communication with the persistent processes and the tools and to run MTP-activities.

The ability of the servers to work with multiple clients on different hosts is exploited by our design and it is used to run tools that are available only on certain machines, or on specific architectures.

As usual, the request for an MTP-tool is sent from a GPC to the connected server; this recognizes the need of employing the MTP protocol and assigns the execution of the activity to the most appropriate SPC, instead of to the calling GPC; the SPC sends the commands to the tool and retrieves the results for the GPC; communication between the GPC (the user) and the tool passes therefore through the server and the SPC, exploiting the client/server dialogue facility already provided by Oz. Note that both the SPC and its tool would run under the common Oz User ID, since they are forked by the server.

6.2 Modifications to the Process Language

The Administrator must specify in the process definition phase which tools are going to be used under the MTP protocol. For MTP-tools some additional data must be stored in the tool description, to account for:

- path into the file system to retrieve the binaries of the tool;

- definition of the architecture on which the tool must be run;
- alternatively, definition of a specific host on which the tool must be run;
- maximum number of instances of the tool that we want our SPC to handle at the same time (this is strictly dependent, among all other considerations, on the size of the tool): a value of 0 for this parameter means that a possibly infinite number is legal;
- the multi-user character of the tool: for example, to describe a multi-user and multi-threaded program, we could assign 1 to the instance number parameter and set a flag to MULTI.

The MSL definition of a tool would then change as follows:

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

All the new data, enclosed in the square brackets, would be optional. When all of these are missing in a tool definition, the parser for MSL would assume that the activities specified should be executed according to the default Black Box protocol. In the case that only the architecture, but not the host machine is specified, the servers should have a way to choose a default machine with the chosen architecture on which to run the tool. We foresee that these additional specification are useful extentions to MSL not just for the MTP-tools, but also in the general case.

6.3 Modifications to the Server

In the schema described above, the servers have the additional task of dealing with their SPCs and with requests from GPCs involving MTP-tools.

First of all, a server must be able to recognize when the the activity requested by a GPC involves an MTP protocol and, if this is the case, it must execute the activity on the right SPC; currently, since we don't support the concept of an SPC, all the activities are started on the calling GPC, instead. This would be still appropriate for the usual SEL tools.

Moreover, if none of the currently existing SPCs are suitable for the task, the server should be able to create a new SPC, on the right machine and architecture, according to those specified in the tool definition. To be able to choose which machines supporting certain architectures are to be chosen by default, we need to keep such information in a service file that the server can read and the Administrator modify. Further pieces of data could be stored there, such as if some SPC needs to be created at server-startup (for example, for a tool requiring a long initialization, we would not like to do it in real time, when it is requested for the first time by a GPC), or if other process-specific customizations and set-ups are needed. This would result in an even greater flexibility of the processes and the whole system.

We foresee that to achieve these functionalities, an acceptably small amount of changes has to be performed on the current Oz server.

6.4 The Protocol

The new MTP protocol, that is the central point of this work, deals mainly with the way the activity part of a rule (the one involving tool invocation) is performed. Currently, according to the Black Box approach of SEL, each single activity maps to something in the form:

```
Activity1 begins
     activity invocation: <system tool1> <arg-set1>
```

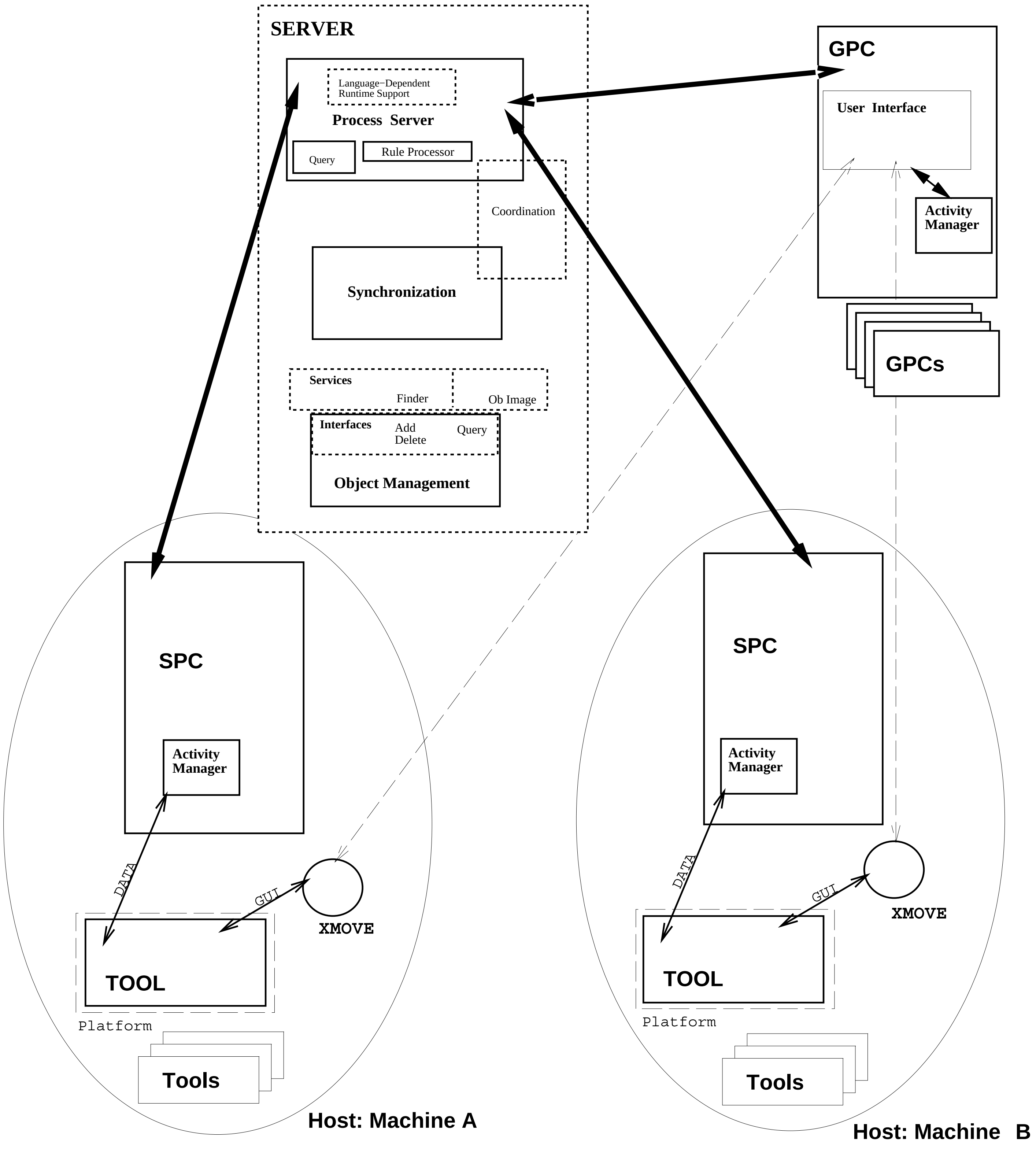

Figure 3: SPCs and persistent tools for MTP

```
    command execution
    return of <return-arg-set1>
Activity1 ends
```

The pair `<system tool>` represents a program, which is loaded with `<arg-set>`, executes and returns results in `<ret-arg-set>` via the wrapping mechanism. (Actually, the `system` specification is at the moment implicit and unrequired, since the common underlying platform on which the program is run is the Unix operating system.) The program is instantiated and terminated by the SEL envelope at each single invocation.

Therefore, beside the modifications caused by the activities and stored into the internal data repository of the system, there is no memory of what happened in Activity1 when we perform Activity2, even if both invoke the same program, since we run two completely separated instances of it.

For interpretive systems, which would hold the whole history of the supplied data and operations on the program state, and for other applications which can be incrementally queried by new commands and fed with more data, this is a very serious limitation. Our approach should rather allow multiple activities to refer to the same program instance, as long as this is useful. The program is not terminated at the end of an activity, but is persistent, until a user decides to close it.

6.4.1 Tool Sessions

To do this, we decided to allow users to specify the duration of each MTP-tool's persistency. We introduce the concept of *session* for a tool: a session begins with an **OPEN-TOOL** command and its body is made up of multiple activities, with their own sets of parameters; it is closed by a **CLOSE-TOOL** command.

A session has therefore the following form:

```
OPEN-TOOL <system tool1>
    <tool1> <MTP-activityA> <arg-setA> (this maps to a single rule)
    <tool1> <MTP-activityB> <arg-setB>
              .
              .
              .
CLOSE-TOOL <tool1>
```

The `system` here could be any of the various persistent platform programs handled by the SPCs, to provide support to diverse families of applications; currently the only available platform is Unix itself and it never needs to be specified, but in the more general MTP case it must be.

According to our design, it would also be possible to use an application without being compelled to issue the OPEN-TOOL and CLOSE-TOOL commands every single time; in this case, if an instance of the application is already running on some SPC, the activity is assigned to it, while if none is available, an implicit atomic session (with just one MTP-activity in its body), is executed. Also multiple copies of the same tool could be present simultaneously, as long as they don't exceed the limit fixed in the instances field of the tool definition, as seen above.

Our idea of sessions would certainly allow several classes of applications to be integrated, but it would also open a number of questions on how different users could access the same instance of the persistent tool. The answers are largely dependent on whether the tools are multi-user.

Imagine that USER1 opened TOOL1 and is executing MTP-ruleA on it; now USER2 wants to execute MTP-ruleB on TOOL1:

- If TOOL1 is not multi-user, or multi-user but not multi-threaded, then USER2's request should be held in a queue until U1 closes the transaction for MTP-ruleA; for each tool session, therefore, the SPC must have the ability to properly handle what we call an **Activity Queue**. It is important to notice that, given the nature of the Oz system, USER2 would not be stuck, waiting for its request to be processed, but could still execute different operations (as long as they don't interfere with the objects chosen as parameters of MTP-ruleB), or even decide to abort the rule and try later;

- If TOOL1 is multi-threaded, but not collaborative, then USER2's request is handled by the multi-threaded nature of TOOL1, and USER1 and USER2 work in isolation. Possible conflicts, because of overlapping object sets bound by the users, would have been dealt with by the Oz concurrency control mechanism already in the condition section of the rules;

- If TOOL1 is multi-user and allows for collaborative work, then, even if it is likely that most of the machinery to deal with it is offered by the tool itself, MTP must still provide the means for communicating with the tool and an appropriate concurrency control policy to account for the full sharing of the data involved.

6.4.2 Transaction-like activites

Besides the concept of session that we just introduced, we also plan to modify the way each single activity in a session is handled. For several of the tool categories we want to address, we found that a transaction-like structure of the activities is desirable. Therefore we would like to add one more level of nesting and complexity beyond the one represented by tool sessions: every single

```
<tool> <MTP-activity> <arg-set>
```

triplet could lead to a transaction or interactive work session, so that a more complete view of operating a tool can be the following:

```
OPEN-TOOL <system tool1>
   <tool1> <MTP-activityA> <arg-setA>
         transaction A begins
               .
               .
               .
         transaction A ends
   <tool1> <MTP-activityB> <arg-setB>
         transaction B begins
               .
               .
               .
         transaction B ends
   <tool1> <MTP-activityC> <arg-setC>
         transaction C begins
               .
               .
               .
         transaction C ends
      .
      .
      .
CLOSE-TOOL <tool1>
```

A single transaction would involve heavy communication between the GPC, the SPC and the Oz objectbase and could resemble something like this:

```
 TOOL (handled by SPC)          USER (GPC)                      OBJECTBASE

                                1 invoke rule

           <--------------------2 issue command

3 exec command

       .

       .

4 send results------------------>------------------------------->

                                5 rec. results                  5 rec. results

                                                                6 CHANGE ENV.

                                7 ask for data------------------>

                                             <------------------8 acknowledge data

           <--------------------9 issue command

           <----------------------------------------------------10 send data

11 exec command

        .

        .

        .

12 send results----------------->------------------------------->

                                13 rec. results                 13 rec. results

                                                                14 CHANGE ENV.

                                15 close transaction

16 send <ret-code>---------------------------------------------->

   and <ret-arg-set>
```

The activity part of the current rule is exhausted (and the effect part is evaluated) only after the transaction is explicitly closed by the user. It is easy to see how this model enhances the interaction between the user and the tool, a property that can and needs to be exploited with some applications. However, it is more complex and demands for some problems to be solved. Inter-process communication during the transaction is conducted via the normal means, the sockets connecting the SPC and GPC with the server. It is important to see that we need some mechanism at the OBJECTBASE end, to screen the result data arriving during the transaction and to enact modifications to the state of the environment accordingly. Moreover, the way we deal with the effect part of the rules, because of possible modifications to the state of the process occurring in the middle of the transaction, must be different. When we return from the activity part, we have four sets of relevant data to deal with:

- **A** — a return code;
- **B** — a set of return arguments (optional);
- **C** — a set of objects which were bound to the activity at rule invocation and in the condition section and are released at the end of the transaction;
- **D** — possibly, a set of objects which have been modified during the transaction.

The return code would tell if the activity was successful or not and, if it was, which effect to be executed. The assertions in the effect would usually modify a subset of **C** (let us call it **E**); according to the changes occurred to objects in **E**, chaining would be then instantiated.

Under MTP, modifications could have occurred also to the set:

$$\mathbf{F} = (\mathbf{D} \cup \mathbf{C}) - \mathbf{E}$$

during different phases of the transaction; it would be therefore necessary for the process enaction mechanism to take those elements in account, too, and to evaluate if the changes to them could fire additional chains of rules.

Another point of interest is how the rollback mechanism needs to be modified: beside the trivial case of a single rule with no chains attached, in Oz roll back is only necessary during a *consistency chain*. This is a set of rules fired one after the other, in a forward chain that must (according to the process definition) succeed or fail in an atomic fashion: if one of these rules fails, then all the work performed by previous rules in the chain is rolled back. Under MTP, we would need to record those changes which occurred in the activity part of the rules (i.e., during the transaction) as well as in the effect part, to be able to roll back also those ones, if necessary.

6.5 The Special Purpose Client

Even if the general structure of the already existing Oz clients will remain the same, SPCs will have a number of peculiar features, given their different role in the system:

- SPCs don't need to interact directly with any human operator, therefore no user interface is needed; however, they will run the tool applications and need to manage input to and output from them. This task involves also making the tools' user interfaces available to the various GPCs executing activities in the context of tool sessions. This is expecially relevant to applications that are not multi-user or are multi-user but not multi-threaded, while real multi-threaded tools are usually already able to dispatch different copies of their user interface to their users. To accomplish this task, we plan to exploit an utility written for another project, called **xmove** [25], which allows the GUI of a tool to be transfered across hosts and terminals. It would be the SPC's responsibility to properly use xmove to dispatch the user interface of its children tool processes, accordingly to the GPC's requests;

- SPCs must also be able to handle multiple children tools at the same time, according to our design; this feature has already been implemented for Oz GPCs and should therefore not be too hard to replicate;

- Every SPC must deal with the concept of sessions and keep track of the number of different instances of a tool active at any moment in the whole Oz system. This may not exceed at any time the value specified in the tool's MSL definition (see 6.2). As a consequence, when the boundary has been hit, the users requesting new instances should be notified of the fact that the resource is not available at the moment and should have the choice to withdraw their requests or to leave them waiting in what we call the **Session Queue**. OPEN-TOOL commands in this queue will be sequentially serviced by the SPC each time a currently running session is over (after a CLOSE-TOOL command is issued).

- We already examined that additional queueing may be necessary internally to each session, when dealing with non-multi-threaded applications (see 6.4.1). Because of the new concept of transactions, though, it may not be always clear when an activity is finished and a new one can be extracted from the Activity Queue and executed. The human user must therefore explicitly specify when each transaction is to be closed; following this, the effect part of the rule is performed, together with all the involved chaining. After this phase, a new activity can be performed by the non-multi-threaded tool. Moreover, when the current activity is a CLOSE-TOOL command and the Activity Queue is not empty, we must choose a policy to handle the remaining requests. We foresee two main approaches: either to close that instance of the tool and to execute each activity in the queue inside an atomic session (this may be very costly, for the amount of overhead due to atomic sessions for large size tools), or to delay the end of the current session until all the requests held on the queue are serviced. This approach should deal also with the fact that in the meanwhile new requests might arrive. These should be queued separately, in a new session context, which would be instantiated right away or after the previous one is finally closed, depending on the boundary on the number of contemporary active sessions.

7 Contributions

We hope to expand the spectrum of tools that can be integrated efficiently in the Oz system and, as a consequence, the domain of processes to which it can be applied and its conceptual generality.

In order to achieve this, we introduce:

- The concept of a new enveloping protocol that goes beyond the current Black Box approach, thanks to the two basic ideas of tool sessions and transaction-like activities. This would account for increased interaction between the users and the wrapped tools;

- Persistent platform processes on which to conveniently run large tools or multi-user tools, fully exploiting their nature and without paying the price of invoking and shutting them down at each single activity execution, as under the current enveloping approach; this includes providing the means to manage such processes and to allow communication between them and the clients, a task carried out by the new Special Purpose Clients;

- The ability of dispatching activities' execution on the most appropriate host machine, a nice feature for a multi-user and multi-site environment such as Oz and that can be available for any tool with marginal modifications to the process definition language (MSL) and to the server structure.

8 Schedule

The following is a rough schedule of the milestones of this project, that we plan to follow; we will try to use as much as possible an incremental approach towards the implementation of the core of the protocol, as towards support of different classes of tools:

- **November 1993 :** Implementation of persistent clients, platform processes and communication between the components of the system
- **December 1993 :** Modifications to MSL language and the server
- **February 1994 :** Prototype implementation of tool sessions and transactions; experiments in supporting large-size interpretive tools
- **March 1994 :** Refinements to the protocol features as sessions and transactions; experiments with multi-user tools
- **April 1994 :** Refinements to the wrapping mechanism and concurrency control policy, in order to fully support multi-user tools
- **May 1994 :** Writing and defending of the dissertation

References

[1] 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.

[2] Israel Z. Ben-Shaul. Oz: A Decentralized Process Centered Environment. Technical Report CUCS-011-93, Columbia University Department of Computer Science, April 1993. PhD Thesis Proposal.

[3] 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*, Sorrento, Italy, May 1994. In press.

[4] Kenneth Birman. ISIS: A System for Fault-tolerant Distributed Computing. Technical Report TR-86-744, Cornell University, Department of Computer Science, Ithaca, NY, November 1986.

[5] 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.

[6] M. R. Cagan. The HP SoftBench Environment: An Architecture for a New Generation of Software Tools. *Hewlett-Packard Journal*, 41(3):36–47, June 1990.

[7] Alan M. Carroll. The ConversationBuilder Kernel and Applications. Technical report, University of Illinois, 1992. PhD thesis.

[8] 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.

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

[10] 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.

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

[12] 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.

[13] C. Gerety. A New Generation of Software Development Tools. *Hewlett-Packard Journal*, 41(3):36–47, June 1990.

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

[15] 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.

[16] 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.

[17] Simon M. Kaplan. Conversationbuilder: An open architecture for collaborative work. In D. Diaper, D. Gilmore, G. Cockton, and B. Shackel, editors, *IFIP TC 13 3rd International Conference on Human-Computer Interaction — INTERACT '90*, pages 917–922, Cambridge, United Kingdom, August 1990. North-Holland.

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

[19] 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.

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

[21] Dorab Patel and Scott D. Kalter. A UNIX Toolkit for Distributed Synchronous Collaborative Applications. *Computing Systems*, 6(2):105–133, spring 1993.

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

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

[24] *2nd International Conference on the Software Process: Continuous Software Process Improvement*, Berlin, Germany, February 1993. IEEE Computer Society Press.

[25] Ethan Solomita, James Kempf, and Dan Duchamp. xmove: A Pseudoserver for X Window Movement. Technical report, SMLI, 1993.

[26] Vic Stenning. An introduction to ISTAR. In Ian Somerville, editor, *Software Engineering Environments*, volume 7 of *IEEE Computing Series*, pages 1–22. Peter Peregrinus Ltd., London, 1986.

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

[28] 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.

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

[30] 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.