Interfacing Oz with the PCTE OMS

Wenke Lee   Gail E. Kaiser
{wenke, kaiser}@cs.columbia.edu

Columbia University
Department of Computer Science
500 West 120th Street
New York, NY 10027
(212)939-7086 Fax: (212)666-0140

TR CUCS-012-95

January 23, 1996

Abstract

This paper details our experiment interfacing Oz with the Object Management System (OMS) of PCTE. Oz is a process-centered multi-user software development environment. PCTE is a specification which defines a language independent interface providing support mechanisms for software engineering environments (SEE) populated with CASE tools. Oz is, in theory, a SEE that can be built (or extended) using the services provided by PCTE. Oz historically has had a native OMS component whereas the PCTE OMS is an open data repository with an API for external software tools. Our experiment focused on changing Oz to use the PCTE OMS. This paper describes how several Oz components were changed in order to make the Oz server interface with the PCTE OMS. The resulting system of our experiment is an environment that has process control and integration services provided by Oz, data integration services provided by PCTE, and tool integration services provided by both. We discusses in depth the concurrency control problems that arise in such an environment and their solutions. The PCTE implementation used in our experiment is the Emeraude PCTE V 12.5.1 supplied by Transtar Software Incorporation.

keywords:

object management system (OMS), software engineering environment, environment framework, software process, process-centered environment, concurrency control

This paper is based on work sponsored in part by Advanced Research Project Agency under ARPA Order B128 monitored by Air Force Rome Lab F30602-94-C-0197, in part by National Science Foundation CCR-9301092, and in part by the New York State Science and Technology Foundation Center for Advanced Technology in High Performance Computing and Communications in Healthcare NYSSTF-CAT-95013.

1 Introduction

The goal of a software engineering environment is to provide support for software engineering activities. Support is more effective if the environment is integrated – if all its components function as a single, consistent, coherent and integral unit [15]. There are several key aspects of integration: tool, data and process. Tool integration provides a development tool set and an invocation mechanism to control its use within an environment. Data integration normally is based on an object model of software artifacts, it uses object management technologies to handle the repository, duplication, sharing and consistency of the artifacts. Process integration normally uses a software process model to explicitly represent the software development activities, it guides and coordinates development activities and integrates tools and data in the environment. Process integration is at a higher level than tool and data integration [11]. In fact, the invocation chain of tools to perform routine development tasks in an environment implicitly defines the conditions and orders of tools. Similarly, the production and consumption of software artifacts normally has a partial order during a software life cycle. Therefore an important goal of many integrated software engineering environments is the support for the definition, enforcement and automation of a software process.

A SEE that has built-in support for process integration is called a process-centered environment. Oz is a process-centered, multi-user and multi-server environment. It supports process coordinations among Oz subenvironments. The architecture of Oz [1] is founded on componentization, with the emphasis that each Oz component provides distinct services (some of the major components are the Rule Processor, the Transaction Manager, OMS and the Activity Manager), and with particular concern for the capability to replace a component with minimal effects on other components while still supporting integration. In other words, Oz should treat process integration as an orthogonal issue to data management and integration, it should be able to use many different data repositories. One of the motivation of this experiment was therefore to find out how difficult it is to replace the OMS component of Oz with an "external" OMS, in this case, the PCTE OMS.

PCTE is a specification of an environment framework which provides support for environment builders to write, assemble and customize integrated software development environments. The PCTE OMS is open and standard, its services address many data integration problems. Thus, Oz should be able to use the PCTE OMS to provide the necessary data integration services. On the other hand, a commonly recognized limitation of PCTE is the lack for the sort of hight level control integration capabilities (it only has low level ones similar to the basic Unix mechanisms) on which much of process integration support is built [10]. It is very common that given any single framework there is strong support for some forms of integration but also weak support for other forms. Thus to remedy weak support for a certain form of integration, it is often necessary to add additional support from another framework. Since Oz has strong support for process integration, it should be a strong candidate to "fill in the gap" for PCTE. Therefore, the more important motivation of our experiment was to build an Oz with PCTE hybrid system that provides strong support for tool, data and process integration.

Our experiment tested both the easiness of replacing an Oz component and the feasibility of developing additional framework support, process integration, to PCTE. The rest of the papers details our experience in both of these areas.

In a truly componentized system, the OMS component can be replaced by an arbitrary objectbase management system without changes in other components of the system. However, the implementation of Oz had incorporated a large amount of legacy code from Marvel, the predecessor of Oz, that made fixed syntactic and semantic assumptions on the underlying data model and data manipulation primitives of the native Oz OMS. The native OMS has evolved into a component called Darkover. References to objects in Oz have been changed to function calls to the Darkover API [9]. This change is intended to eliminate all fixed syntactic assumptions. Replacing the native OMS (Darkover) with an arbitrary external OMS can in principle be done by implementing a translation layer that maps the external OMS API to the Darkover API, and modifying the Darkover functions that handle ad hoc queries and cache management. Identifying and removing the fixed semantic assumptions are more subtle and difficult. This can be accomplished by interfacing the Oz server with an external OMS (e.g., the PCTE OMS) and find out what can go wrong and what are the causes.

In this experiment we did not try to replace the Oz OMS completely with the PCTE OMS. Instead, we replaced the Storage Manager of Oz with the PCTE objectbase. We reused as many Oz OMS functions as possible in order to minimize the code changes in Oz components. For example, the data model validating functions and the cache management functions remain unchanged. There were two main sets of Oz functions that were rewritten in our experiment. The first set of functions was the object storage and retrieval functions, which were changed to use the PCTE OMS API [3]. The second set of functions was the concurrency control functions, which were expanded to incorporate PCTE activities [5] into Pern transactions. Since these OMS functions were not isolated in a single component, several Oz components [1], namely OMS, Rule Processor and Pern were modified.

It is our experience that the fixed semantic assumptions about the OMS are somewhat independent of the fixed syntactic assumptions in that even if we have eliminated the fixed syntactic assumptions in Oz, we may still find fixed semantic assumptions. On the other hand, we can identify the fixed semantic assumptions while ignoring and tolerating the fixed syntactic assumptions by dynamically translating the fixed native object format to the appropriate format suitable to the external OMS in use. This is the approach we used in this experiment. Ideally, we should first eliminate the fixed syntactic assumptions and then identify and remove the fixed semantic assumptions. We could not follow this order at the time of this experiment because the Darkover API was not completely specified and implemented, in other words, the Oz components still had all the fixed syntactic assumptions about the OMS.

A better part of our experiment was spent on the concurrency control problems. These problems arise due to, in part, some serious limitations of the current PCTE implementations (see section 6).

The rest of this paper is organized as the following: Section 2 describes briefly the services provided by PCTE, with the PCTE OMS covered in details. Section 3 provides an overview of the Oz system architecture, its concurrency control, data repository, and tool integration support mechanisms. Section 4 compares the Oz OMS model and the PCTE OMS model, and discusses how to define the Oz data model using the PCTE OMS. Section 5 describes the implementation of the Oz interface to the PCTE OMS. This interface consists of object operations implemented using the PCTE OMS primitives. Section 6 examines the concurrency control problems that arises when the PCTE OMS is in use. Two different solutions are discussed in detailed. Section 7 describes an example Oz environment which uses the PCTE OMS as the data repository. Section 8 compares our experiment with other related work in this research area. Section 9 concludes this paper with a summary of our experiences and a discussion of possible future work.

2 PCTE Overview

PCTE, for Portable Common Tool Environment, is a framework for CASE tool integration. Tool integration not only means that all of the tools exhibit some measure of uniformity, but also that they should be able to share the same data and communicate with each other. The commonly accepted solution for the development of environments that facilitates CASE tools integration is to use an Open Software Integration Platform, known as an Environment Framework. A widely supported framework is based on the Reference Model for Software Engineering Frameworks developed by ECMA and NIST [12]. This model suggests that a framework should support at least three main sets of services, namely, User Interface Services, Communications Services and Object Management Services, which correspond to these three dimensions of tool integration: presentation, control, and data. The fourth important set of services of the framework, the Process Management Services, is to support the development of an environment enforcing a particular software development process. Note that a process should not be “built-in” for an software development environment. Instead, the environment should provides facilities for users to define and tailor a particular process suitable for the particular software development policies and activities.

PCTE is a framework that conforms to the ECMA/NIST Reference Model. For presentation integration, PCTE supports MOTIF-compliant tools. In the are of data integration, PCTE provides a single, common data repository, open to all components of the environment. This data repository provides all the basic services necessary in an Object Management System (as described in the ECMA/NIST Reference Model) for a high level of data integration between tools. For control integration, PCTE provides message and notification facilities.

The Emeraude PCTE environment is a distributed software engineering development environment [5]. It typically consists of a network of hosts, which are workstations that allow users to access the Emeraude PCTE environment.

2.1 The PCTE OMS Model

The main interest of this experiment is the OMS services provided by PCTE. These services include data typing and data storage along with concurrency control. This subsection briefly explains the key concepts and features of the PCTE OMS model.

The OMS model defines the static information about the data stored in the objectbase, which is the repository of all persistent PCTE data. The objectbase is managed by the PCTE OMS, and (in the Emeraude PCTE environment) is distributed over the environment on a number of logical volumes corresponding to the physical distribution of storage devices over the network. A volume is mounted on a Unix file system or device of a particular host so that the data held in the volume is available transparently throughout the environment.

2.1.1 Objects, Links, Relationships, Attributes, and Types

The objectbase represents discrete items of data as typed objects connected in a logical network by typed links and relationships.

- **Objects** represent the basic entities upon which operations are to be performed, and one object exists for every distinct entity that is to be operated on by a tool. Examples of objects are text files, documents, source and compiled modules.
- **Links** represent relations between objects. A link is directional, emanating from one source object to the other destination object; it is always paired with a reverse link going in the opposite direction, either as part of a relationship or created automatically as a system reverse link. An example is a link *writes* from object *user* to object *document*, representing the fact that *document* is created by *user*. The link in this example is called a **reference link**, which models dependencies between objects. Another type of link is the **composition link**, which models composite objects. A composition link is automatically created by PCTE when an object is created. It links an existing object (the parent object) to the newly created object (the child object). An example is the composition link *contains* from object *book* to *chapter_1*. A link also has a **cardinality**, either one or many. This defines whether more than one link of this link type can start from the same source object.
- **Relationships** are paired mutually dependent links that represent a two-way relationship between pairs of objects, each half of the pair representing one direction of the relationship. For example, a link *is_written_by* from object *document* to object *user* can be paired with the link *writes* from *user* to *document* to form a relationship between theses two objects. Depending on the cardinality of its links, a relationship can represent one-to-one, one-to-many and many-to-many relations between (instances of) two object types.

- Objects and links can be qualified by OMS **attributes**, which have a particular value type (for example, integer, string, boolean or date) and give more information about the object or link. For example, *document* can have *status* and *release_date* attributes, of value type string and date, respectively. A **key attribute** is associated with a cardinality many link so that its value can be used to distinguish the possibly multiple links of the same type from the same source object.

- Objects, links and their attributes in the objectbase are all typed, that is, they are created in accordance with a specific **type definition**. A type definition defines the properties that all instances of the type in the objectbase will have. When an object is created, it is created as an instance of a given type. Object types, which can also be called object classes, are created as specializations of existing types, and they inherit the properties of their ancestor types. Thus an object also has all the properties of its ancestor types in addition to the properties that are specifically appropriate to its type. The PCTE OMS model supports single inheritance only.

Which objects and links can be seen and created at any given time depends on the **working schema** of the currently running PCTE process (on behalf of an external tool). A working schema contains all the type information needed by the tools being executed. The use of working schema allows different views of the objectbase to be presented to different tools so that only the appropriate types are visible to a particular tool.

2.1.2 Objectbase Composition and Navigation

A composition link is automatically created when an object is created. It links an existing object, the parent, to this new object, a component. An object has only one composition link leading to it, and when this composition link is deleted, the object is no longer accessible and is implicitly deleted.

Direct reference to a particular object or link in the objectbase is through navigation. A pathname needs to be specified in order to access an object or link. The pathname starts from the **reference object**, and contains the names, in order, of all the links that must be traversed to get to the object or link to be accessed. A reference object is like a pre-evaluated path, which is convenient to use and allows short pathnames.

2.2 Communicating with the PCTE OMS

Tools communicate with the OMS through a number of tool interfaces provided by PCTE. These interfaces include the C language interface, which is a set of C libraries that can be linked with the object code of tools programmed in C; and the PCTE shell interface, which allows users to communicate with OMS directly through a set of OMS commands. In addition, Emeraude PCTE provides a tool called the OMS Browser. It has a graphical user interface that gives users a view of the objectbase, and menus that contain the major OMS commands. It allows users to navigate the objectbase by following links from object to object; see information about objects, attributes and links; create, delete and modify objects, attributes and links; use the Emacs-like Emeraude PCTE object editing tool to edit object contents; change the working schema; and start, abort or end a transaction.

Tools that are integrated into or developed within a PCTE environment are stored in the PCTE objectbase as objects (of type *Sctx*) and their executions are managed as PCTE "client" processes. These processes are not objects because they are purely dynamic. PCTE provides facilities that are similar to Unix to manage the start, stop, suspension, communication and synchronization of PCTE processes [14]. Unix tools that are interfaced to a PCTE environment need to be linked with the PCTE Unix libraries and their executions (standard Unix processes) are "alien" processes to PCTE.

Within an Emeraude PCTE environment, there is one object server process for each host that has objectbase volumes (a physical partition of the objectbase). This object server process, which is started by the PCTE *host_start* command, provides the PCTE OMS functions to other user processes – alien processes or PCTE client processes. (A host that does not have any physical volume of the PCTE objectbase has no object server. The user processes on that host actually communicate with the object server on another host.)

3 Oz Overview

3.1 The Architecture of Oz

Oz is a process-centered software development environment that supports cooperation among groups of engineers at different sites who follow a range of software development specific processes (workflows) [1]. It facilitates both data sharing and process coordination. The architecture of an Oz environment is based on the client/server model. An Oz server can support multiple Oz clients sharing the same process and accessing the same objectbase. It also communicates with other Oz servers to share (distributed) data and coordinate different processes. A user interfaces directly with an Oz client, which in turn communicates with one or more Oz servers.

The major components of the Oz server are

- The rule processing engine. Rules are the key elements of a process definition. Each rule consists of a name; a list of typed parameters; a condition consisting of bindings of local variables and a complex property clause that must hold on the actual parameters (instances of control and product data) and bound variables for the rule to fire; an optional activity that specifies a tool envelope and its arguments; and a set of mutually exclusive effects, each consisting of assertions to the objectbase that reflect one of the possible results of executing the activity. Rules are implicitly related to each other through matches between a predicate in the condition of one rule and an assertion in the effect of another rule. Process enaction is done through rule chaining. Backward chaining is attempted to satisfy the condition of a rule. Forward chaining through "atomicity chains" is performed mandatorily to propagate data changes to preserve the atomic nature of a transaction. Forward chaining through "automation chains" is for the purpose of automating sequences of activities, not necessary atomically.
- The transaction manager is a separate component called Pern [8], which interfaces to the rest of Oz (or another environment framework) through application-specific mediator code. Because multiple clients can access the same objectbase, conflicts may arise among user activities and concurrency control is thus in order. Oz follows the traditional solution of associating tasks with transactions. A task consists of all rules executed during backward chaining, followed by the user-invoked rule (which caused the backward chain), followed by all rules executed during forward chaining. Pern supports a nested transaction model in which a task corresponds to a series of top-level transactions. Each atomicity chain is a subtransaction of the triggering transaction and each rule in an automation chain starts an independent top-level transaction.
- The object management system (OMS). In an Oz environment, all data, both process control data and product data, are stored in and managed by the OMS. The Oz OMS contains an Object Manager, a Storage Manager, and a File Manager. The Object Manager implements the data model, provides persistence and performs all requests for access and modification of both control and product data. It assumes an object-oriented data model, with a class lattice, object composition hierarchy and arbitrary relationships between objects in the same objectbase. The Storage Manager is responsible for low-level disk and buffer management for control data. Since Oz integrates file-based external tools and maintains its product data in ordinary files, the File Manager is responsible for accessing files requested by other Oz

components. Usually, the data model of a process encapsulates the product data within control data and abstracts the file system as objects by providing typing and relationship information. In this case, the File Manager is just a mapping function between objects' file attributes and their file contents. The Oz OMS also provides an ad hoc query processor and a set of object manipulation functions.

3.2 Pern Transaction Model and Its Architecture

Pern, which is currently used as the transaction manager for Oz, supports a cooperative transaction model [7]. It uses a project-specific lock compatibility matrix to check for lock conflicts. In the case of a lock conflict, Pern checks the project-specific coordination model to take the appropriate actions (for example, suspend or abort).

Pern may act as an external concurrency control (ECC) component in a variety of software engineering environments (SDEs) [8]. Pern has no fixed assumptions about how a task is structured (e.g., a task corresponds to a rule chain in Oz). It also uses external lock tables and a generic recovery mechanism so that it does not require the knowledge of the OMS in use. And most importantly, Pern uses mediator architecture where mediators establish the connections necessary to integrate Pern with the task management services (TMS) component and the OMS of an SDE.

3.3 Communicating with the Oz OMS

All the major Oz components need to communicate with the OMS. The rule processor queries the objectbase to obtain the actual parameters of a rule and evaluates its condition. The rule's activity accesses and modifies data in the objectbase. The rule processor also modifies data in the objectbase when it asserts an effect of a rule. Pern locks objects accessed within a transaction and directs the OMS to undo changes in the objectbase when a transaction is aborted.

At the time of this experiment, the Oz components were tightly coupled with the OMS. For example, it is assumed that the whole objectbase is loaded into Oz buffers when the Oz server is started and access to objects is then done by directly addressing into these buffers instead of function calls to the OMS interface. Object modification is done by calling OMS functions that handle both updating the in-memory buffer and the on-disk persistent data. But these functions only work with an assumed data model where objects can have *file* attributes, which are files stored in the Unix file system, and *composite* attributes, which are component objects. In the following sections, we will see more detailed examples of these fixed assumptions and how they should be changed.

3.4 Oz Tool Envelopes

A typical Oz process may utilize several external tools to carry out activities. Oz provides one or more tool envelopes for each external tool so that the external tool can be used as a "black box". A tool envelope represents the implementation of an activity. Since the activity of a rule is always executed by Oz client, tool envelopes are invoked by Oz clients. Tool envelopes are written in SEL, an extended Unix shell language [6]. SEL allows the tool integrator to use a standard Unix shell language to write code to wrap around the call to the tool. It also requires the explicit declaration of all object attributes, along with the corresponding types, that are input or output variables of an envelope. This requirement hides the details of object data model from the envelope writer. SEL also handles multiple output (return) values so that it can not only report the status of tool execution back to Oz, but also return the execution results. A tool envelope written in SEL is compiled using the Oz *make_envelope* utility into a standard Unix shell script.

4 Implementation – Data Model Mapping

4.1 Comparison of Oz and PCTE OMS Models

Needless to say, most of the complications of using the PCTE OMS for the Oz server arise from the differences in the PCTE OMS model and the Oz OMS model. However, since both of them are intended to be generic object-based models, there are many similarities between the two.

4.1.1 Similarities

- Both Oz and PCTE support the concept of object, which can be uniquely identified in the objectbase – in Oz via the unique object identifier (OID) and in PCTE via the navigational pathname to the object.
- Both associate a type (class) with an object. Both support a type hierarchy and inheritance. However, PCTE only allows single inheritance while Oz supports multiple inheritance.
- Both have a pre-defined root in the type hierarchy – in Oz the *ENTITY* class; in PCTE the *object* type.
- Both apply attributes to objects. In Oz, primitive attributes can be of string, integer, boolean, date, real, and user-defined enumerated types. In PCTE, only string, integer, boolean, and date are allowed.
- Both use links to model relations between objects. Both have implicit reverse links to facilitate objectbase navigation.
- Both support an object hierarchy through composite objects. The definition (and construction) of the object hierarchy is identical. In Oz, an object can have *composite* attributes each of which can be a component object or a set of component objects. In PCTE, a parent object can have composition links to one or a set of component objects.

4.1.2 Differences

The following are the main differences between the two systems which we had to resolve.

- In PCTE, type definitions (classes) are stored in the objectbase as first class objects; in Oz, class definitions are stored separately from the objectbase in a file and loaded into the Oz server at startup.
- In Oz, the name space of attributes and links is within the class where the attributes and links are defined and applied. In PCTE, all objects, links and attributes in a Schema Definition Set (SDS) share a single name space.
- Oz objects may have file attributes whose values are the pathnames of files in a “hidden” file system. It is therefore possible to associate an Oz object with one or many files. In PCTE, objects that can have file contents are of type, or subtype, of the *file* object type predefined in the PCTE OMS. A PCTE object can be associated with at most one file.

4.2 Data Modeling Using the PCTE OMS Model

In order for the Oz server to interface with the PCTE OMS, we need to first define the Oz data model, which defines the schema of process data and product data in an Oz environment, in the

PCTE OMS. Since the two OMS models are not the same, we use the following approach to overcome the differences.

- Each link and attribute name in PCTE is prefixed with the object type (class) name. This guarantees that links and attributes names are unique in a schema.
- Oz link attributes are modeled as relationships in PCTE. Although links are unidirectional in Oz persistent storage, at run time the Oz OMS creates an in-memory reverse link for each link to speed up in-memory object navigation and to support inverse queries. The original motivation of using relationships is that Oz components may take advantage of the persistent reverse links in PCTE for object navigation in the objectbase. However we found out later that while a reverse link can be used to access an object, it can not be added or deleted directly because it is always **implicit** in PCTE – its existence depends wholly on its paired link. This implies that if we make the reverse implicit links visible to Oz components, Oz has to distinguish whether a link is implicit or not when it tries to operate on it. Since we don't consider this requirement a general one, we decide not to read the reverse implicit links into Oz at run time. In other words, we are currently not taking advantage of the persistence reverse links in PCTE.
- A composite attribute for a component object in Oz is mapped to a composition link in PCTE.
- A file attribute for a file in Oz is mapped to a composition link to an object of type Oz_FILE, which is a subtype of $file$ in PCTE. For example, if an Oz (object) class $cfile$ has a $file$ attribute named $contents$, then in PCTE, $cfile$ will have a composition link named $contents$ to Oz_FILE.
- Oz_OID and Oz_NAME become Oz system-attributes for all objects in PCTE. Since every class in Oz is assumed to be a subclass of $ENTITY$, we extend the object type definition $ENTITY$, which is a subtype of the PCTE root object type $object$, in PCTE with these two attributes so that all (derived) object types automatically have the two as attributes. The only exception to this is that Oz_FILE is not a subtype of $ENTITY$ (this is because Oz_FILE is a subtype of $file$, which is in turn a subtype of $object$) so we also add these two attributes to it. Each Oz_OID has a unique value in an objectbase; it is added to facilitate Oz components in referencing objects. Oz_NAME is needed mainly for the purpose of objectbase display in Oz clients. Oz_OID and Oz_NAME can therefore be regarded as the external identifiers of an object, with the first used in the programming interface and the second used in the user interface. The reference pathname of an object can then be regarded as the internal identifier.
- An attribute of a user-defined enumerated type is mapped to an attribute of type string in PCTE. While the default value can be set, the list of the possible valid (enumerated) values is not stored.

We limited our experiment to those Oz data models that do not use multiple inheritance in their class hierarchies. Although one can always devise an algorithm which translates a multiple inheritance class hierarchy into a single inheritance class hierarchy, it is neither the goal nor the interest of this experiment to actually implement such an algorithm. We have found few significant fixed assumptions about multiple inheritance in Oz components.

PCTE also provides a schema compiler that validates and then stores the type definitions (classes) in the objectbase. Suppose we have an Oz class definition as shown in figure 1. Then the relevant information in the PCTE schema definition file will be as shown in figure 2.

5 Implementation – Interface to the PCTE OMS

The legacy code in Oz components assume that the entire objectbase is loaded into memory when the Oz server is started. While this assumption could potentially be changed, our experiment is mostly

```
MANUAL :: superclass ENTITY
  title : string;
  reformat : boolean = false;
  format_status : (Initialized, Working, Done) = Initialized;
  /* enumerated-type attribute */
  submanuals : set of SUBMANUAL; /* composite attribute */
  first_page: text /* file attribute */
  first_sub : link SUBMANUAL;
```

Figure 1: An Oz Class Definition

```
OZ_OID : string;
OZ_NAME : string;
MANUAL__title : string;
MANUAL__reformat : boolean := false;
MANUAL__format_status : string := ``Initialized'';
/* OZ_OID is the key attribute of this cardinality many link */
MANUAL__submanuals : composition link (OZ_OID) to SUBMANUAL;
/* file attribute is mapped to a composition link to a file object */
MANUAL__first_page: composition link to Oz_FILE;

relationship (/* link mapped to relationship */
  first_sub : reference link to SUBMANUAL;
  first_sub_of : implicit link to MANUAL );

ENTITY : subtype of object;
MANUAL : subtype of ENTITY;
Oz_FILE: subtype of file;

extend ENTITY
  with
  attribute OZ_OID;
            OZ_NAME;
end ENTITY;

extend Oz_FILE
  with
  attribute OZ_OID;
            OZ_NAME;
end Oz_FILE;

extend MANUAL
  with
  attribute MANUAL__title;
            MANUAL__reformat;
            MANUAL__format_status;
  link MANUAL__submanuals;
       MANUAL__first_page;
end MANUAL;
```

Figure 2: Corresponding PCTE Schema Definition

interested in finding out how Rule Processor and Pern will need to be changed when an external OMS is used. We wanted to minimize our efforts while achieving our goals. Therefore we tolerate as many unrelated fixed assumptions as possible by loading the PCTE objectbase into the in-memory buffers in the Oz server. To match the formats of the in-memory buffers of classes, objects, and attributes, some run-time translations are required when the PCTE objectbase is loaded into Oz. Since the Oz in-memory formats match the Oz OMS data model, translations are needed exactly at the places where the PCTE OMS model mismatches the Oz OMS model. From the discussion in the last section, it is clear that this kind of translation is straightforward and isolated.

Once the objectbase is loaded in Oz, objectbase navigation can be done in memory. On the other hand, the effects of object update operations are written immediately in the PCTE objectbase. We will again see that the PCTE-specific code is isolated to a handful of low level functions because there is already some degree of abstraction in Oz – the components do not have knowledge of how the objects are physically stored: low level functions (the Storage Manager) handle the details.

5.1 OMS Primitives in PCTE

The PCTE OMS C language interface contains a rich set of OMS primitives which are essential for developing an access interface for an external system (such as the Oz server) to communicate with the PCTE OMS:

- Data Definition Primitives
  - Schema Definition Set Level Primitives – initialize, validate, or delete a schema definition set.
  - Definition Level Primitives – create, import, extend, or apply an object, attribute, link or relationship type definition.
  - Browsing Primitives – get all the type definitions in a schema definition set; get information about a type definition.
- Data Manipulation Primitives
  - Object Manipulation Primitives – create an object; get its type information; determine its accessibility;
  - Link and Relationship Manipulation Primitives – create or delete a link; get the name of a link or its reverse link; get links starting from an object.
  - Attribute Manipulation Primitives – get or set the value of an attribute.
- Establishing the Working Schema – establish a schema definition set as the working schema, or get the list of schema definitions that constitute the current working schema.

5.2 Reading the Class and Object Hierarchy into the Oz Server

When the Oz server is started, it first reads the class hierarchy of the data model in use and then reads the whole objectbase. We now describe our implementation of these two functions for the PCTE interface.

Every Oz environment has an environment directory which contains environment specific process definition files, tool envelope files, transaction coordination files, an objectbase file, and a subdirectory of "hidden file system" which stores the file attributes of objects. Within the environment directory, the *strategy* file contains the class hierarchy in an internal format. When an Oz server is started for a specified Oz environment, it reads the class definition from the strategy file in the environment directory. Thus Oz effectively employs a single (working) schema for each Oz server. In

PCTE classes are first class objects that are actually stored in the objectbase, and type definitions are visible within a specific working schema. We put the keyword *PCTE*, followed by the names of the schemas that constitute the working schema, in the Oz *strategy* file. This information is used by the Oz server to first set the working schema. Data definition browsing primitives are then used to get all the type definitions in the PCTE objectbase. For each object type definition (class), its parent type definition is used to build the class hierarchy. Also for each class, the definitions of its attributes of type integer, string, boolean or date are read. The definitions of its composition links are mapped to Oz composite attributes definitions; the definitions of its relationships are mapped to Oz link attributes definitions.

Reading the objectbase from PCTE is achieved through navigation starting from a set of top level objects. A list of absolute pathnames of PCTE top level objects are stored in the *objectbase* file in the Oz environment directory. To read the object hierarchy from these top level objects, either depth first search (DFS) or bread first search (BFS) can be used.

We use a modified DFS in our implementation. The modification is the following: when an object A is read, if A has a reference link (as part of a relationship) to object B, then B is read (recursively using DFS) if B has not been read already. This is done to facilitate the set up of reference link and reverse link between A and B. If we used the standard DFS, B would not be read at this point since B is not a child of A. The consequence is that the reference links of A can not be set up when A is being processed, thus a second pass is needed to set up the reference links between objects. It is thus clear that our modified DFS is more efficient.

For each object being read, its attributes are loaded into its in-memory copy; additionally, its pathname composed strictly of composition links is stored in its *ref_path* field. The format of the internal object representation in Oz, the *instance* structure for an Oz object, has been modified to have this additional field. *ref_path* is used for direct access to an object in the PCTE objectbase. If an object B is read the first time because of a reference link, the pathname that is used to access B is not stored in *ref_path* because it contains a reference link, i.e., from A to B. Only when B is visited later because of a composition link, is the pathname then stored in its *ref_path*. We make this deliberate effort for two reasons: first, we want the pathname to an object to represent its object hierarchy path, thus every sub-path within the pathname must represent object composition (vs. object reference); second, the composition links are more stable than the reference links in the sense that deleting a reference link does not affect the existence of the object being linked to, but removing the composition link deletes the linked object.

5.3 Oz Built-in Operations

There are a number of built-in object operations in Oz, namely *add*, *delete*, *copy*, *move*, *link*, *unlink*, and *rename*. These operations first update the in-memory copy of the objectbase and then write the updates to the persistence storage. The functions for in-memory updates required few changes but the functions to update had to be rewritten using PCTE OMS primitives.

- *add*: function *do_add_operation* handles the creation of an object. It passes the pathname of the parent object, the name of the composition link, and the in-memory *instance* of the object to be added to function *OMS_write_object*, which creates the persistent object in the PCTE objectbase and links the parent object to it. *OMS_write_object* also sets attribute values for this object in the objectbase. Its *Oz_NAME* is set using the user-entered name, and *Oz_OID* is set using the Oz system-generated id. After an object is actually added, its pathname, which is the parent object pathname plus the composition link name, is stored as *ref_path* in its *instance*.

- *delete*: function *do_delete_operation* handles the deletion of an object. It passes the in-memory *instance* of the object to be deleted to function *oms_delete_object*. The later function first recursively calls *oms_delete_object* to delete all the descendant objects. Then it deletes all the links to/from this object. The actual deletion of this object is done by deleting the composition link from its parent object. The pathname of this object is used to parse the pathname of its parent object and the name of the composition link.

- *copy*: function *do_copy_operation* copies object $from$ to object to so that to has an exact copy of $from$ as its child object. The copy operation copies the whole object composition hierarchy starting at $from$, as well as all the links in this object hierarchy. If an object A within $from$ links to object B (B could be anywhere in the object hierarchy), then A's copy within to also links to B.

- *move*: function *do_move_operation* moves object $from$ to object to so that to has an exact copy of $from$ as its child object and the original parent object of $from$ no longer has it as a child object. PCTE only provides primitives to move an object across physical storage volumes (in PCTE, the objectbase can be partitioned into multiple volumes). Since we do not assume to and $from$ are in different physical volumes, the *move* operation has to be done in a number of steps. In the first step, the whole $from$ object hierarchy is copied and placed under to. The links are not copied; instead, for each link, the object ids of the source object and the destination object, along with the link name, are stored in a table called *link_table*, which has entries of the form $(from\_id, to\_id, link\_name)$. For each object being copied in this step, ids of the original object and the copied object are stored as (old_id, new_id) in table *object_table*. In the second step, the original $from$ object hierarchy is deleted. The links are also deleted as a result. In the third step, *object_table* and *link_table* are used to create links. For example, a link recorded as (A, B, l) is now created as link l from A' to B if (A, A') is found in *object_table*.

- *link*: function *do_link* creates a link as the *link_attribute* from *source* to *destination* using the corresponding PCTE primitive.

- *unlink*: function *do_unlink* deletes the *link_attribute* link from *source* to *destination* using the corresponding PCTE primitive.

- *rename*: function *do_rename* change the name of *instance* to *new_name*. It calls OMS_write_object to reset the Oz_NAME of *instance* to *new_name* in the PCTE objectbase. Note that the name used by Oz, the Oz_NAME, is irrelevant to the object pathname in the PCTE objectbase. Thus, renaming does not require moving the object in objectbase.

5.4 Accessing PCTE Objects in an Oz Rule

As discussed in earlier sections, the Oz rule processor needs to access objects during all phases of rule processing, namely, bindings, condition, activity, and effects. Since the rule processor only calls the high level built-in operations to modify the objectbase, there would be little to change here if these built-in operations are interfaced to an external objectbase. However, in our experiment, we have found some fixed semantic assumptions about how files are modeled.

In the Oz OMS, a file of an object is one of its *file* attributes, while in PCTE it is a component object of subtype *file*. When reading the class hierarchy and objectbase from PCTE into Oz, file objects are treated the same as other component objects, i.e., they are translated into Oz *composite* attributes of their parent objects. As a result, the rule processor code that handles only the *file* attribute have to be changed to also consider *composite* attributes. The attribute type definition, in this case class Oz_FILE, is used to determine whether the *composite* attribute is a file object.

A more serious problem is when a rule is applied to an object that has a file attribute in its object type definition but has no such physical attribute in the object instance yet. In the Oz OMS, the file attributes stored in the objectbase are just file paths to the host file system, which is known and fixed once the object instance is created. In the original Oz implementation, when an object is read into the Oz server (in server startup time) or when it is created in Oz, a file attribute pointer is created for each of its file attributes, regardless of whether the file exists or not. This works fine because the file path for a file attribute is already known even if file is not physically created yet, so long as the object instance exists. When a rule which requires access to a file attribute is fired on an object, Oz just passes the file attribute (the file path) to the rule; it then relies on the rule activity (e.g., invocation of the editor on the file path) to create/modify the file. For example, suppose we have class C_FILE with file attribute $C\_contents$; an instance of C_FILE, $foo.c$ can be created without having physical $C\_contents$ due to, for example, that $foo.c$ has never been edited. When a rule, say *edit*, is applied to $foo.c$ the first time, it requires access to its $C\_contents$. Oz will pass the $C\_contents$ attribute, which is just a file name (for an nonexisting file) in the host file system, to the editor. After *edit* is completed, the $C\_contents$ is now the (unchanged) file path to an existing file. When the PCTE objectbase is in use, however, files are stored as first class objects in the objectbase, and a file attribute of an object is modeled as a component object of object type $file$ of the object. Therefore, creating a pointer (in memory) to a nonexisting component object is obviously not acceptable because it will seriously comprise the data integrity of the objectbase in memory. Our solution to this problem is the following: the file component object is created only when a rule which requires a file attribute parameter is applied to this object. This action of creating the component is logged automatically by the governing Pern transaction; therefore it can be undone in the case of a Pern transaction abort (see next section). Since most external tools that access files require that files be in the host file system instead of in the objectbase, file copying in both directions is also required. More details are discussed in section 6.3

6 Concurrency Control and Recovery

Since Oz is a multi-user system with multiple Oz clients sharing the same objectbase, concurrency control is needed to prevent chaos. For the same reason, PCTE also provides concurrency control mechanisms for concurrent foreign and PCTE tools to cooperate with each other. Note the Oz server may be only one of these (foreign) tools, and multiple foreign and PCTE tools may be accessing the PCTE objectbase at the same time as Oz.

Pern, the transaction manager used by Oz, is a more flexible, powerful and elaborate system. PCTE provides only the basic primitives for concurrency control. In this experiment, we try to make Pern and the PCTE OMS work together in such a way that Pern manages the high level transactions and PCTE handles the lower level subtransactions. Data locking is done first in Pern (in memory) and then in PCTE (in the objectbase). If a lock can not be acquired in PCTE, the current Pern transaction will abort.

There are certainly other more desirable approaches that we wish we could have used to cooperate Pern and PCTE. However, as the following detailed discussion will show, the current implementation of PCTE has some serious limitations that make other alternatives very difficult to implement.

6.1 Concurrency and Integrity Control in PCTE

Concurrency and integrity controls in PCTE are enforced by means of **activities** (not to be confused with the activity of an Oz rule). An activity in PCTE is a defined framework within which a set of related operations takes place. These operations involve accesses to the objectbase. Each operation is always carried out on behalf of a single activity.

An activity is characterized by its class. There are three classes of activity in PCTE:

- An **unprotected activity** does not require that its accesses to the objectbase be protected from other concurrent activities.
- A **protected activity** requires that its accesses to the objectbase be protected from the effects of other concurrent activities. The data it reads remain stable and the data it writes is not

compatible modes	read-unprotected	write-unprotected	read-protected	write-protected	write-transaction
read-unprotected	*yes*	*yes*	*yes*	*yes*	*yes*
write-unprotected	*yes*	*yes*	*no*	*no*	*no*
read-protected	*yes*	*no*	*yes*	*no*	*no*
write-protected	*yes*	*no*	*no*	*no*	*no*
write-transaction	*yes*	*no*	*no*	*no*	*no*

Table 1: Compatibilities of PCTE Lock Modes

overwritten by other concurrent activities. Changes to the objectbase made by the activity are permanent even if the activity terminates prematurely.

- A **transaction** requires the same protection as a protected activity, but in addition it must be atomic; that is, all or none of its results should be applied to the objectbase. This corresponds to traditional database transactions, with full rollback of the objectbase updates when a transaction fails for any reason.

An activity can acquire **locks** on **resources**. Here a resource refers to: either an object with its contents and its set of attributes, but not the links originating from the object; or a link and its attributes. In other words, if an object is locked, all its attributes and contents are locked except its outgoing links. A read mode lock is required for any access that does not intend to modify the accessed resource. A write mode lock is required for any access that intends to modify the objectbase. A write access implies a read access. The effect of a lock on a resource is the following:

- **read-unprotected** mode allows other activities to write to this resource concurrently.
- **write-unprotected** mode allows other activities to write to this resource concurrently. Any modifications, from whatever source, are immediately applied.
- **read-protected** mode prevents other activities from writing to this resource concurrently. This mode is applicable to both protected activities and transactions.
- **write-protected** mode prevents other activities from writing to this resource concurrently. Any modifications, by the activity which holds the lock only, are applied immediately.
- **write-transaction** mode prevents other activities from writing to this resource concurrently. Any modifications, by the activity which holds the lock only, are either applied or discarded once the lock is released. This mode is applicable to transactions only.

Compatibilities of lock modes are shown Table 1.

A lock in PCTE also has a lock duration. A **short lock** can be released before the termination of the associated activity, once the resource is no longer required. A **long lock** can only be released when the activity finishes. Long locks are held by transactions, whereas short locks are held by either unprotected or protected activities.

PCTE has a different implementation of nested transactions than many other DBMSs' because of the association of process and activity. A process (an executing program), which carries out operations that access the objectbase, is always initiated in the context of an activity. A process can start and control new activities, which are treated as **nested** (internal) to the one in which the process was started. The outer-most activity (normally an unprotected one) is implicitly set up by the system so that when a user program (process) is first started, it is within the context of this activity. In the current implementation of PCTE in the Unix environment, the only way for a process to start a new activity is for this process to spawn a child process and then let this child process start an activity. Thus, within each Unix process, there can be only one (flat) PCTE activity.

In the following sections, we will see the consequences and limitation of this implementation. Basically, since in the Oz environment there is a single Oz server per objectbase and Pern is part of the Oz server, which is implemented as a single Unix process, there is only a single Unix process on behalf of Pern with respect to a given objectbase at any given moment. A Pern transaction corresponding to a rule often requires nested transactions, for example, because of an atomicity rule chain. If we try to put a Pern transaction within a PCTE transaction, then the implementation will require that a PCTE transaction is started before a Pern transaction is started. If the Pern transaction happens to have nested subtransactions, then a PCTE transaction will also be started before each Pern subtransaction is started. But this second PCTE transaction would be nested within the first PCTE transaction, and they are both started by the same Unix process (Pern). This is not allowed by the current implementation of PCTE. Therefore there is currently no simple and direct way for Pern to take full advantage of the PCTE nested transactions facility. Instead, we have devised mechanisms that enable low-level PCTE object operations to be controlled by PCTE protected activities and transactions, while high level object manipulations (as in an Oz rule chain) to be guarded by Oz's Pern.

6.2 Built-in Operations as PCTE Protected Activities

The built-in object operations in Oz, namely *add*, *delete*, *copy*, *move*, *link*, *unlink*, and *rename*, require write access to the PCTE objectbase and thus require protection from other concurrent PCTE activities. These operations are normally done in isolation (i.e., have no dependencies on other operations). Further, once such an operation succeeds in changing the objectbase, there is no reason (within the same operation) to undo the effects. Therefore they can be implemented either as protected activities or as transactions because either can provide the same protection against other concurrent activities. Implementing them as PCTE transactions would probably leave the full rollback functionalities unused. We decided to implement these operations as PCTE protected activities in order to gain more experience with PCTE, since we implemented some other object operations (discussed in the next section) as PCTE transactions.

Each of the built-in operations eventually calls a low level function that uses PCTE OMS primitives to carry out the object operations in the PCTE objectbase. These low level functions can be placed within PCTE protected activities. For example, *do_add_operation* calls the function *OMS_write_object* to write objects in the PCTE objectbase. Before calling *OMS_write_object*, a PCTE protected activity is started so that all object operations following this call are protected. When *OMS_write_object* returns, this activity is ended. Within *OMS_write_object*, before writing an object (changing its attributes or creating a child object), an attempt is made to lock this object in write-protected mode. If the lock is obtained, then this object is written to PCTE's OMS and the lock is released. This sequence of actions ensures that no other concurrent PCTE activities can write to this object if an Oz built-in operation is accessing it. These built-in object operations are normally called upon by a rule's effects and are therefore part of a Pern transaction. If the PCTE protected activities on behalf of these Oz built-in operations have to be aborted, for example, if an exclusive lock can not be obtained, the upper level Pern transaction also aborts. Say, a rule's effect has two update operations: $add(a,b)$ and $link(c,d)$. These two operations both have to succeed. Suppose that the *add* succeeds but the *link* fails; Pern would then abort the transaction of this rule. This requires that the result of the *add* to be undone in the PCTE objectbase. Pern maintains an entry for each invocation of such operations in its log file so that the effects of these operations can be undone when a Pern transaction aborts. The details of putting entries in the log file and performing recovery for these PCTE object operations are discussed in section 6.4.

6.3 External Tools Wrapped in PCTE Transactions

The activity part of an Oz rule can invoke an external tool such as the default editor. These external tools normally require access to the contents of objects (for example, the content of a C

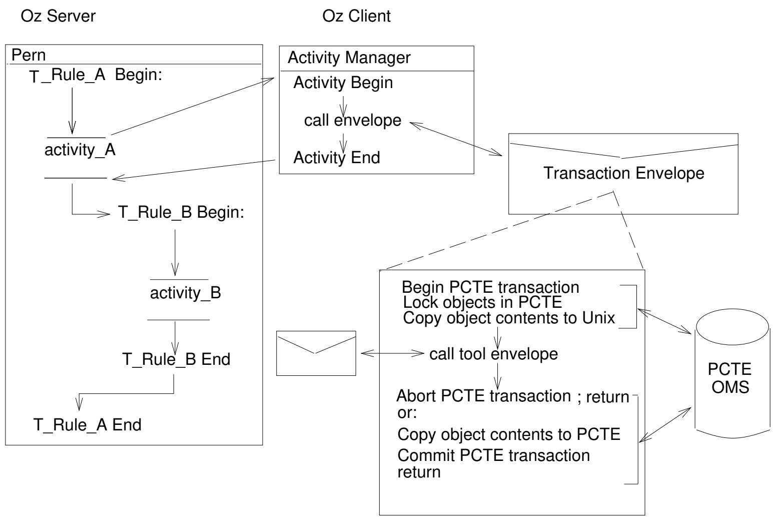

Figure 3: Transaction Envelope

file). Object contents, i.e. files, are of utmost importance in a (coarse-grained) software development environment and thus demand vigorous protection. For this reason we decided to wrap the tools that access PCTE object contents in PCTE transactions. This implementation ensures that the updates of object contents are all-or-nothing, and no intermediate changes can be seen by other concurrent PCTE activities (this is a given, since in Oz, a PCTE file object is always copied to the host file system before an external tool can use it). Recall the PCTE limitation that a Unix process can only have one PCTE transaction. If it did not have this limitation, we could easily implement the top level Pern transaction as a PCTE transaction and the tool invocation as a nested PCTE transaction.

We overcame this limitation by implementing a wrapper Unix program, a **transaction envelope**, that runs as a separate Unix process. This transaction envelope takes the message regarding what tool operates on what objects, along with some Oz environment-specific information (e.g., the id of the Oz client that runs the rule activity), from the Oz client, starts a PCTE transaction, tries to lock the required objects, and then calls a **tool envelope**. Figure 3 shows how the transaction envelope interacts with the Oz client and the PCTE OMS. It also shows that the PCTE transaction carried out by the transaction envelope is part of a Pern transaction. The tool envelope is a shell script that invokes the actual tool in the system and reports the status to its caller. Normally the return code from a tool envelope is used to choose one of the rule's several effects. We have designated a return code to indicate an aborted request. The transaction envelope then decides whether to commit or to abort this PCTE transaction according to the return status from the tool envelope. The return status of the transaction envelope is the same as that of the tool envelope so that in the case of a PCTE abort, the abort request will be propagated to the Oz server and Pern; otherwise, this return code is used to choose the effect as usual.

Some implementation details are worth discussing. When an external tool is called as a rule's activity, the rule processor of Oz marshals arguments (objects or attributes) required by this tool in a command line arguments buffer and sends it to an external program, an envelope, normally via the Oz client. The Oz client also creates a named pipe, a file in the host file system, and sends the pipe name to the envelope as a command line argument. This named pipe is used by the envelope to write the return values so that the Oz client can read them after the envelope terminates. When a transaction envelope is called, the command line arguments are marshaled differently than for a tool envelope. First, the name of the tool envelope (which the transaction envelope will call) is passed in as an argument; second, for each external tool argument that is of a file type, the rule processor translates it to "$lock\_mode * PCTE\_file > Unix\_file > backup\_file$". If such an argument is a set rather than a singleton, then a set of such strings (one for each instance), separated by a |, is used for this argument.

The *lock_mode* indicates the lock mode that Pern requires on this object. It is S for shared or X for exclusive. At the point when the transaction envelope receives this message from the Oz server, the in-memory copy (within the Oz server) of the object has already been locked in an appropriate lock mode by Pern. The transaction envelope then tries to lock the persistent copy of the object in PCTE accordingly: X as write-transaction and S as read-protected (if a lock mode other than shared and exclusive is used in Oz, it also needs to be mapped to one of these two PCTE lock modes, this would generally require the "promotion" of the Oz lock mode).

Pern has the notion of **primary** locks for an object and **intention** locks for all ancestors of the object. PCTE does not support such flexible lock modes; instead, it has the notion of a **concerned domain**. The concerned domain for an object is the object, the set of links leading to the object, and the set of links originating from the object. The concerned domain for a link is the link and the object from which it starts. If the link is the composition link leading to an object, the link's concerned domain also includes the concerned domain of the destination object. A lock can be placed on an object (or link) only if this lock is compatible with all the locks held by other activities on the objects and links in the concerned domain. Therefore if an object is locked exclusively (in write-protected or write-transaction) by an activity, none of its ancestors can be locked exclusively (for updates) by other activities. On the other hand, its sibling and descendant objects may still be locked exclusively. To see this, suppose that an object *module* is locked exclusively by an activity *make*. Implicitly, the composition link from *module*'s parent object *system* to *module* is also locked by *make*. Now if another activity *delete* tries to lock *system* exclusively, it would fail since one link (the composition link to *module*) in *system*'s concerned domain is locked exclusively by *make*. Now that *system* can not be locked by other activities, it is implicitly locked by *make*, and as a result *system*'s parent can not be locked by other activities either. It follows that none of *module*'s parent can be locked by other activities. However, if an activity *edit* tries to lock *module*'s child *cfile*, it will succeed since *module* is not in *cfile*'s concerned domain. It follows that all of *module*'s descendants can be locked. Therefore we can see that in PCTE the use of concerned domain parallels Pern's intention lock strategy.

The *PCTE_file* is the full path of the object contents in PCTE. The object contents are copied to *Unix_file* before the tool envelope and thus an external tool (a Unix such as Emacs) is invoked. After the tool envelope execution, the contents of the *Unix_file* is copied back to *PCTE_file* before the transaction is committed. The *backup_file* is for the purpose of recovery; it is a Unix file that contains a copy of the object contents from before the tool execution. The reason why we need this backup file is the following: although the tool execution is performed within a PCTE transaction, it is also part of a higher level Pern transaction, which may abort. But when this occurs, the changes to the object contents in PCTE have been committed and need to be undone. Our solution here is to put an entry in Pern's log file to indicate that the object contents have been updated and there is a backup file that contains the original contents. In the event of a Pern abort, the recovery function will use this log entry to copy the original contents back to PCTE, thus undoing the committed changes in PCTE.

Since the transaction envelope calls the tool envelope to actually execute the external tool, it needs to create a named pipe and pass the pipe name to the tool envelope so that the tool envelope can write the return status and output values to the pipe. The transaction envelope can then read the output after the external tool envelope terminates. After a PCTE commit or abort based on the return status, the transaction envelope just writes the same output to the named pipe that was passed from the Oz client, thus enabling the client to get the return status and output values.

Just as in the case of Oz built-in operations, if the PCTE transaction on behalf of an external tool invocation aborts, for example, because of lock conflict or error in executing the external tool, the upper level Pern transaction also aborts. Pern handles the recovery.

Since file (PCTE object) copying is required for most of the Unix tool invocations, the performance of rule activities when the PCTE OMS is in use must be worse than the case when the Oz OMS is in use. Section 7.3 gives an example in tool performance comparison.

6.4 Recovery Using Pern Log and PCTE Transactions

In Oz, the Pern log file is used for the purpose of recovery. Each object operation has an entry in the log file. In the event of recovery, the entries are used to undo the effects of these object operations. Although Oz built-in operations are implemented as PCTE protected-activities and external tool invocations as PCTE transactions, they are subtransactions of the upper level Pern transaction. Therefore their results need to be undone if the Pern transaction aborts. We use the Pern log file for recovery. This can be easily achieved since Pern puts entries in the log for all operations of a transaction and its subtransactions automatically.

There are some complications here, however. For the delete object operation, the original Pern implementation puts only one entry for the deletion of the top level object. It also stores in the log file the ASCII representation of the subtree objects (the descendents) that will be deleted. This works fine with Oz's OMS because all objects are stored in the Unix file system, so an undo of a delete operation only requires rewriting in the file system using the stored ASCII representation. But for PCTE, the actual deletion of an object involves a sequence of delete operations, from the bottom up, one for each descendent of the object. If there is only one entry (for the top level object) in the log file, it is impossible to recover from a partial delete. Storing the ASCII description of the subtree objects is not practical in terms of efficiency because we need to traverse the objectbase in PCTE to get this description. We have implemented a better solution to put an entry for each actual PCTE object delete operation in the log file. Ideally PCTE should act as a lower level transaction manager and provide an interface for Pern to use its log file to initiate recovery for PCTE object operations. But the current implementation of PCTE does not provide such an interface.

To prevent a partial undo, we implemented each undo operation in a PCTE transaction. This ensures that undo can be done idempotently in case of failures during a previous undo.

6.5 Discussion

Notice that the order of the data lockings and the fact that PCTE transactions are subtransactions of the Pern transactions only guarantees serializability between Oz activities within the same Oz environment, but not between an Oz activity and a PCTE activity that accesses the same PCTE OMS through its own interface. To see this, suppose that we have two Oz clients A and B that connect to the same Oz server, and they are firing rules $rule\_A$ and $rule\_B$, respectively. Further, the activities (or effects) of these two rules try to gain exclusive access to a PCTE object bar at the same time. Since the execution of each Oz rule is done within a Pern transaction, and there is a single Pern per Oz server, Pern can grant the exclusive lock on bar (locking done in memory) to either $rule\_A$ or $rule\_B$. Actually, Pern requires that all the objects that a rule intends to update be locked before the rule's activity and effect take place. When, say, $rule\_A$ gets the exclusive lock from Pern and actually accesses bar in the PCTE objectbase, it first does the actual locking of bar in the PCTE objectbase. This is done to prevent other external (outside the Oz environment) activities from accessing bar after $rule\_A$ starts accessing bar. This mechanism (a strict two phase locking scheme) guarantees serializability between $rule\_A$ and $rule\_B$,and is deadlock-free because: 1) $rule\_A$ and $rule\_B$ have to obtain all the locks at once from Pern; 2) when $rule\_A$ or $rule\_B$ locks and accesses an object in the PCTE objectbase for which it has obtained the Pern lock, there can not be any other Oz rule (within the same Oz server environment) that can access the same object.

The story is quite different if an external activity, say C is accessing the PCTE objectbase through the PCTE OMS interface directly (without going through Oz, and thus Pern). Unless C locks all the objects it intends to update at once before it actually writes to the PCTE objectbase, there is no guarantee of serializability between C and A (or C and B). For example, C can have an

```
rule_A:         rule_B:         external activity C:

read(X);        read(Y);        lock(X);
X=X-N;          Y=Y+X;          read(X);
write(X);       write(Y);       X=X+N;
                                write(X);
                                unlock(X);
```

Figure 4: Concurrent Activities

old copy of *bar* (done by reading without first locking *bar*) before *A* locks it, and then after *A* has committed the changes to *bar* and released the lock, *C* writes its own update of *bar*, resulting in a "lost update" problem (*A*'s update is gone). The point here is that, unless all PCTE activities follow a transaction model in accessing the PCTE objectbase, for example, two phase locking, there will be unpredictable results in the face of concurrent activities.

However, even if we are in a "perfect world" where all PCTE activities use the transaction model in accessing the PCTE objectbase, there can still be a "Incorrect Summary Problem" (also known as the "Inconsistent Analysis Problem") [2]. To see how this can happen, suppose that we have two rules *rule_A* and *rule_B* and they are chained together (from *rule_A* to *rule_B* by an atomicity chain). The activities of *rule_A* and *rule_B* are shown in figure 4. The two rules are chained because, for example, a consistency condition that "whenever X is updated, update Y accordingly". In Oz, *rule_B* will be in a Pern subtransaction of *rule_A*'s. *X* is locked in *rule_A*'s transaction, and the lock is passed to *rule_B*'s transaction. But when the transaction envelope is used, the activities of *rule_A* and *rule_B* will be in different PCTE transactions, and passing locks between the two envelopes, which are two different Unix processes, is not possible under the current implementation of transaction envelopes.

Imagine that after the transaction envelope for *rule_A* commits the corresponding PCTE transaction which updates *X* (at which point the lock on *X* is released), but before the transaction envelope of *rule_B* can lock *X* and *Y*, another external activity *C* can start. The activity of *C* is also shown in figure 4. *C* can finish quick enough so that the transaction envelope can successfully lock *X* and *Y*, and then go on to do the update *X*. Since *X* has actually been changed by *C* already when *rule_B* uses it, this results in a "incorrect summary" on *Y*.

Figure 5 shows the inadequate protection against external activities when transaction envelopes are used in a nested Pern transaction (for a rule chain). Here, a PCTE transaction (on behalf of an external activity) that starts and finishes between the time of two Oz activities could potentially cause the incorrect summary problem.

At the time of this writing, the authors are implementing a solution to this problem. The basic ideas is to have a **lock mediator** process to lock all the objects needed for a Pern (nested) transaction. As shown in figure 6, one instance of such a mediator process is needed for each top level Pern transaction. One possible implementation of the lock mediator is the following: when the Oz server is started, start the master lock mediator process and set up a predefined named pipe between it and the Oz server process. Whenever a top level Pern transaction is started, Pern notifies the master mediator process to fork and spawn a child mediator process for this new transaction. It also sets up a named pipe, identified by the top level transaction id, between it and the child mediator process. Whenever a Pern transaction needs to lock or unlock an object, it notifies the master mediator process of its top level transaction id, the object id and the action (lock or unlock), using the named pipe between the Oz server and the master mediator process. The latter then turns the lock/unlock request into the corresponding child mediator process using the named pipe between theses two process. The child mediator process does the actual lock/unlock objects in the PCTE objectbase. It also starts a PCTE transaction when the top level Pern transaction starts, and commits the PCTE transaction when the top level Pern transaction is to commit. Any failed lock request to PCTE will

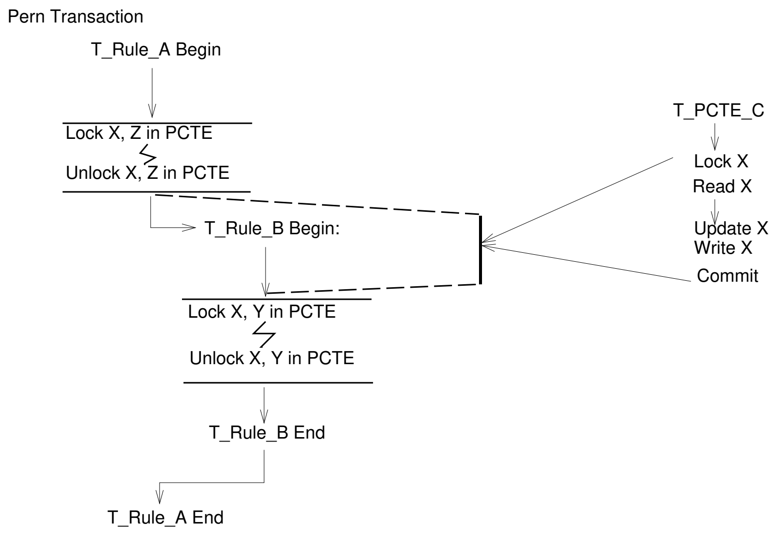

Figure 5: Incorrect Summary Problem

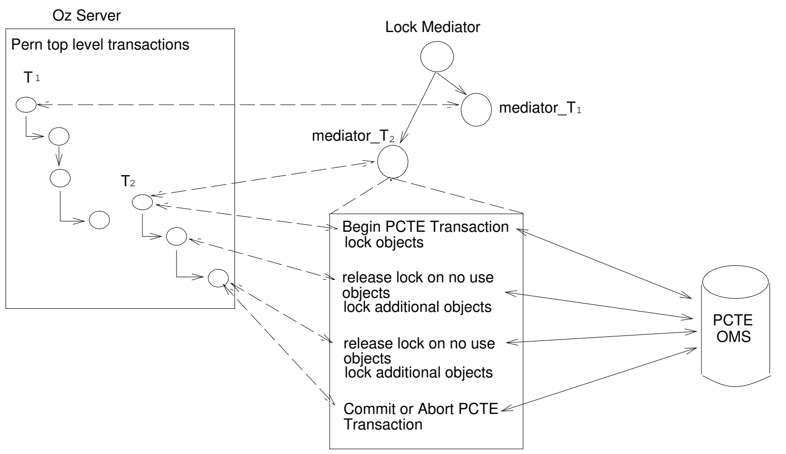

Figure 6: Lock Mediator

trigger the child mediator to abort the PCTE transaction. It then notifies Pern, through the master mediator, to abort the top level Pern transaction.

Copying object contents from/to PCTE should also done by the child mediator when it locks/unlocks an object. Otherwise, when an envelope, which is a Unix process other than the child mediator, tries to copy the object contents from PCTE, it would fail because the object is locked by a different process (the lock mediator). Note that when the lock mediator is in use, there is no longer a need for the transaction envelope because by the time the tool envelope is executed, the object is already locked in PCTE, with its contents copied to the local file system. Comparing to our transaction envelope approach, the advantage of using a lock mediator is that it actually maps each top level Pern transaction to a PCTE transaction and can therefore guarantee the integrity of the Pern transactions. The disadvantage is that it may be relatively heavy weight since one child mediator process is created for each top level Pern transaction. The choice between the two solutions may depend on the actual application environment and the degree of protection that is required. If it can be assumed that most of the objects used by Oz would not be accessible to other PCTE activities, for example, by setting the proper group permissions on the objects, then using transaction envelopes is probably adequate. Otherwise, if Oz intends to use the objects that are shared by other PCTE activities, the lock mediator is in order.

Since the actual data locking is still required in the PCTE objectbase even if Pern has locked the object, one might ask "Is locking in Pern really necessary?" The Pern transaction that governs a rule would have to abort anyway if it fails to obtain the lock in the PCTE objectbase, for example, because an external activity has already placed an exclusive lock on this object. We can see that the locking done by Pern is just like a front door guard to the PCTE objectbase. It is actually redundant since we can always skip the Pern locking and let the transaction envelope (or the lock mediator) try to lock an object in the PCTE objectbase, and abort an transaction if the lock can not be acquired. We keep the Pern locking for PCTE objects for several reasons: 1) In order to support the Pern cooperative transaction mechanism, Pern utilizes its own flexible lock modes and makes decisions as to whether a split or join transaction is possible when conflicts occur. 2) By performing the Pern locking first, concurrency conflicts (among Oz activities) can be detected earlier since all the objects are in memory. This will show a more obvious advantage in terms of performance if the physical objectbase is on a remote host in the network (for example, due to geographical distribution of the objectbase or multi-process interoperability of Oz subenvironments) and therefore requires significantly more time in accessing and locking objects in the objectbase.

7 An Example: Oz/Doc with PCTE OMS

We now present an example of using Oz and the PCTE OMS together.

7.1 The Oz/Doc Environment

Oz/Doc is a document production environment implemented using Oz. The document composition structure (for example, a manual is composed of a set of submanuals each of which consists of a header, a set of chapters and figures) is represented using the Oz object-oriented data model. A set of Oz rules were employed to implement the process of document production (for example, first *reserve*, then *edit*, *format*, *deposit* and *print*).

We chose this environment as a test case in our experiment because the data model of Oz/Doc is relative simple and thus it can be mapped into the PCTE OMS easily. Another motivation is that there may be many document production tools already implemented in PCTE, our plan was to use Oz/Doc to experiment the process control and tool integration using the Oz with PCTE hybrid system (for example, the activity of an Oz rule can be a PCTE tool instead of a Unix tool). However, due to the resource limitation, we could not find any useful PCTE document development

```
HISTORY  :: superclass ENTITY;
       history : text;
end

TIMESTAMPED :: superclass ENTITY;
  time_stamp : time;
end

VERSIONABLE :: superclass ENTITY;
          version_num : integer = 0;
          state       : integer = 0;
          locker      : user;
          reservation_status : (CheckedOut,Available,None) = None;
          version     : text    = ",v";
end

FILE :: superclass TIMESTAMPED, VERSIONABLE, HISTORY, ENTITY;
       owner    : user;
       contents : text;
end
```

Figure 7: Definition of the FILE class (with multiple inheritance) in Oz/Doc

tools to experiment with.

In our experiment, both the control data (Oz process data) and the product data are stored in the PCTE OMS. However, end-users need not be aware that the PCTE OMS is in use. They normally work in an Oz environment using the GUI of Oz clients, which are responsible for sending object and rule requests to the Oz server. The Oz client interface includes an objectbase display which shows the entire objectbase hierarchy. Users can "navigate" the objectbase (and thus "zoom in" to any part of the objectbase) by clicking the objects in the display. Rules and Oz built-in commands can be selected from the pulldown menus. Clicking an object identifies it as a parameter for a selected rule. External tools (for example, *emacs*) are encapsulated as the activities of Oz rules. They are invoked by the Oz clients and thus normally run on the same workstation where the Oz client was started.

7.2 Implementing the Oz/Doc Environment with Oz and the PCTE OMS

To implement the Oz/Doc process, we first map the Oz/Doc data model onto the PCTE OMS model using the approaches discussed in section 4. Most of the data definition mappings are straightforward. figure 1 and figure 2 gives an example on how to define an Oz object class in the PCTE OMS. There is only one class definition, $FILE$, that is derived from multiple super classes, namely, $TIMESTAMPED$, $VERSIONABLE$, $HISTORY$, $ENTITY$, as shown in figure 7. The multiple inheritance only goes one level. Therefore, as shown in figure 8, we simply duplicated the attribute definitions of the first three super classes in the class definition for $FILE$ and make it a subclass of $ENTITY$.

In the Oz/Doc environment directory, there are two files that need to be set up specifically when the PCTE objectbase is in use. Figure 9 shows the contents of these two files. For *objectbase*, the first line gives the absolute path of the root of all objects in the PCTE objectbase; the second line gives the root (top level) object for all Oz objects (of Oz/Doc) stored in the PCTE objectbase. It is from this root Oz object that the *read_objectbase* function in Oz server starts the objectbase traversal. For example, for the Oz/Doc environment, there are several $MANUAL$ objects $manual1$, $manual2$,

```
FILE__time_stamp : integer := 0 ;
FILE__version_num : integer := 0 ;
FILE__state : integer := 0 ;
FILE__user : integer := 0 ;
FILE__locker : integer := 0 ;
FILE__reservation_status : string := "None" ;
FILE__owner : integer := 0 ;

ENTITY : subtype of object ;
FILE : subtype of ENTITY ;

FILE__history : composition link to MARVEL_TEXT;
FILE__version : composition link to MARVEL_TEXT;
FILE__contents : composition link to MARVEL_TEXT;

extend FILE
  with
  attribute FILE__version_num ;
    FILE__time_stamp ;
    FILE__state ;
    FILE__locker ;
    FILE__reservation_status ;
    FILE__owner ;
  link  FILE__history ;
        FILE__version ;
        FILE__contents ;
end FILE ;
```

Figure 8: Corresponding Definition of FILE in PCTE (using single inheritance)

```
objectbase:          strategy:

_/                   PCTE_BEGIN
_/siteA.sys          sys|oz_doc|env
                     oz_doc
                     PCTE_END
```

Figure 9: Objectbase and Strategy Files

etc. that are the component objects of *siteA.sys*. Their path names are "_/*siteA.sys/manual*1", "_/*siteA.sys/manual*2", etc. These *MANUAL* objects and their descendants can be accessed once the path for *siteA.sys* is known.

The *strategy* file first gives the names of the schemas that make up the working schema set. Then it gives the name of the schema, oz_doc, that defines the Oz/Doc data model (figure 8 is actually part of the output of the PCTE command "sds_decompile oz_doc", which lists the object, link and attribute definitions in an PCTE schema). *sys* and *env* need to be in the working schema set because *oz_doc* references object types defined in these two schemas. The *read_strategy* function in the Oz server uses these schema names to set the working schema set, and then reads the class hierarchy from the PCTE OMS according to *oz_doc*.

Once the data model for Oz/Doc and the object instances are loaded into the Oz server, users can start working in this environment via the Oz clients. For example, say a user invokes the *edit* rule on object *chapter*1 (for example, by selecting the *edit* rule off the menu and identifying *chapter*1 by clicking it on the objectbase display). Assume that *chapter*1's reference path is "_/*siteA.sys/manaul*1/*chapter*1" and its object id is 59. The Oz server first constructs the Unix file path for *chapter*1, which in this case would just be "59" within the Oz/Doc environment directory. If the Pern transaction id for *edit* is 4, then a backup file path is constructed as "4_59" (also in the Oz/Doc environment directory). The Oz client passes these file paths to the transaction envelope which does the necessary file locking and copying before invoking, say *emacs* on "59". After the editing is ended, the file contents of "59" is copied back to "_/*siteA.sys/manaul*1/*chapter*1". In the case of a later Pern rollback, the contents of "4_59" is copied to *chapter*1 in PCTE. If the top Pern transaction commits, then both "59" and "4_59" are removed because they are no longer needed. Note that everything described here happens "behind the scene" in the sense that the users are not aware at all that the PCTE OMS is in use.

7.3 Performance Analysis

To get some idea on how file copying has affected the performance of rule activities, we installed a *file_io* rule with an activity that concatenates "hello world" to the end of a file. We then used an Oz tty client, which can execute rules in batch mode, to fire this rule on the same file (a PCTE object) for 100 times. The average time it took to finish all 100 rule invocations was 316 seconds. Using the same rule in an identical Oz/Doc environment where the Oz OMS is in use, it took an average 115 seconds to finish all 100 rule invocations. It is evident from this result that file I/O is the bulk of tool execution time and this performance hit is inevitable when Unix tools operate on PCTE objects.

8 Related Work

HyperWeb [4] is a framework that supports the construction of hypermedia-based software development environments. In Hyperweb, software artifacts are stored in the PCTE OMS. Moreover, the hypermedia linking of these software artifacts are also modeled and stored in the PCTE OMS – each hypermdeia node is an object and each hyperlink is a relationship between two objects. The (PCTE) working schema set is used to constrain the types of links that can leave or enter a node. The architecture of HyperWeb is client/server. The HyperWeb server communicates with the PCTE OMS which acts as a software object base, providing data integration. The server coordinates tool activities among clients through message passing. HyperWeb is very similar to Oz in using the PCTE OMS for data integration. Both have the client/server architecture where the server is responsible for communicating with the PCTE OMS. Also inferred from the HyperWeb paper [4] is that the server and clients run outside of the PCTE environment. There was no discussion on concurrency control and process integration.

The SoftBench with PCTE experiment [13] attempted to add the Broadcast Message Services (BMS) on top of PCTE, thus providing a higher level of control integration than what was provided by PCTE. SoftBench is an environment framework for integrated tools. The SoftBench environment consists of a set of integration services and an extensible set of tools that communicate by sending and receiving messages. The key components of SoftBench are the Broadcast Message Server (BMS) and the Execution Manager (EM). An executing SoftBench tool can send a message to BMS when it requests a service, completes a service that are relevant to others, or has a failure to report. BMS then forwards the message to other SoftBench tools that have registered to this message. The EM keeps track of the tools that are executing within a SoftBench environment. It coordinates with BMS to determine whether a service request can be satisfied by an already running tool or that a new tool needs to be started. Compared with the integration services provided by PCTE, which are essentially low level, Unix-like mechanisms, BMS provides a high level control integration among SoftBench tools. The SoftBench with PCTE experiment re-implemented the BMS on top of PCTE, replacing the socket connections in the SoftBench BMS with the PCTE inter-process communication mechanism of message queues. Tools are grouped into classes and each class has a class manager which has the similar functionalities as the SoftBench EM. The BMS, the class managers and each tool all run as PCTE processes. They each have an associated message queue to facilitate the communication and coordination among them. This strategy of re-implementing the major parts of environment in PCTE is very different from our approach of interfacing an environment with PCTE. It best served the assumption that SoftBench tools can be modified to run within PCTE (as PCTE processes). The SoftBench experiment had not used the PCTE support for concurrency and integrity control and activities [13].

9 Conclusion

Our experiment has successfully enabled Oz to interface with the PCTE OMS. The resulting system is an environment where software tools can be integrated by sharing the same Oz process and the same PCTE objectbase. From the point of view of integration technology, we prove that Oz and PCTE are complementary and add value to one another.

This is a light integration with PCTE because the Oz server, Oz client and external tools remain as alien processes to PCTE. Although we could have put all the Oz code into the PCTE objectbase so that Oz runs within the PCTE environment, we did not believe that we could import every potential external tool used in an Oz process into PCTE because doing so requires code level knowledge of these tools to make modifications. But if we just import Oz into PCTE but not the external tools, it is still not clear how a PCTE process (say, Oz within PCTE) can invoke an alien (Unix) process, say a tool envelope, and communicate with it to copy object contents in both directions.

This experiment not only shows that the Oz server can be modified to work with an external OMS with a different data model. It also demonstrates that a process management service component, in this case, Oz, can be added on top of PCTE. Therefore, our system can be used to enforce policies (workflows) for tools that access the PCTE objectbase. A better part of this experiment was devoted to the concurrency control problem, where we wanted to devise a mechanism to incorporate PCTE transactions into the more flexible, cooperative Pern transactions. This is necessary to prevent the tools written to use the PCTE interface directly from avoiding our policy enforcement mechanism. The transaction envelope is a light-weight, clean but a bit simple-minded solution. The lock mediator solution is more elaborate and sophisticated but heavy-weight.

9.1 Lessons Learned

The PCTE OMS provides only primitive object operations but does not support encapsulation of methods with data. This could have made it very difficult to attach a software process model to

PCTE. But in our experiment, this is not a big problem because Oz already had a fully functional process engine and a certain degree of abstraction in terms of object access. Mapping the data model and re-implementing the low level object access functions using the PCTE OMS were not conceptually challenging at all.

The limitation that one Unix process can have only one PCTE transaction, coupled with the current Oz architecture (one Pern process per Oz server), certainly made it difficult to elect a simple and yet complete solution to implement Pern nested transactions. However, since Pern uses a mediator architecture, we can use the lock mediator solution to integrate Pern and PCTE transactions.

Finally, as mentioned throughout this paper, there were fixed assumptions about the OMS in the Oz code. Work is under way to eliminate these fixed assumption in a separate Darkover project. At the time of this writing, the Darkover project is just completed, all access to class, object and attribute are now done through function calls to Darkover API. Had this interface in place when we started our experiment, we could have just implemented a set of functions, within Darkover, that access the PCTE objectbase. However major data model mismatch (for example, files are component objects instead of object attributes) may still require changes outside of Darkover (for example, in the rule processor if the code assumes only $file$ attributes, it needs to be changed to also considers *composite* attributes of Oz_FILE if the PCTE OMS is in use).

9.2 Future Work

Pern and the Oz Rule Processor are being separated from Oz to become independent service components. Although re-implementing the entire Oz system in PCTE is not practical, it is now feasible and would be interesting to experiment putting (re-implementing) these components into PCTE. Putting Pern on top of PCTE would add support for flexible transaction models to PCTE. Further, its mediator interface would enable PCTE environment builders to tailor Pern to suit their environment-specific concurrency control policies. Similarly, putting the Oz Rule Processor on top of PCTE would add process integration to PCTE. This rule processor component will also have a mediator interface that enables environment developers to tailor it to support environment-specific process enforcement and assistant models. Since our experiment is a light integration, it is difficult for Oz to enforce controls to tools running in PCTE other than using data locking, which provides only passive control. However if the Oz components are put into PCTE, they can take advantages of the control integration services provided by PCTE, e.g. the message queue, the support for user roles and security control, to provide active control and integration for the PCTE tools.

Acknowledgements

We are very grateful that Bull HN Information Systems supplied us a free copy of Emeraude PCTE. We would like to thank members of the Programming Systems Laboratory at Columbia for helping us in this experiment. Particularly, George Heineman had in-depth discussions with the authors on the Pern transaction model and the possible ways of interfacing Pern with PCTE. Jack Yang explained the design of the Darkover interface. Peter Skopp helped out on numerous occasions. We also wish to thank John Hinsdale for his earlier work in trying to interface Marvel with PCTE.

References

[1] Israel Ben-Shaul and Gail E. Kaiser. *A Paradigm for Decentralized Process Modeling.* Kluwer Academic Publishers, Boston, 1995.

[2] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems, 2nd Edition*. The Benjamin/Cummings Publishing Company, Inc., Redwood City, CA, 1994.

[3] European Computer Manufacturers Association. *ECMA PCTE C Programming Language Binding*, 1991.

[4] James C. Ferrans, David W. Hurst, Michael A. Sennett, Burton M. Covnot, Wenguang Ji, Peter Kajka, and Wei Ouyang. Hyperweb: A framework for hypermedia-based environments. In Herbert Weber, editor, *5th ACM SIGSOFT Symposium on Software Development Environments*, pages 149–158, Tyson's Corner VA, December 1992. Special issue of *Software Engineering Notes*, 17(5), December 1992.

[5] GIE Emeraude. *Emeraude PCTE Environment Guide*, 1994.

[6] 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.

[7] George T. Heineman. A transaction manager component for cooperative transaction models. In Ann Gawman, W. Morven Gentleman, Evelyn Kidd, Perke Larson, and Jacob Slonim, editors, *1993 Center for Advanced Studies Conference (CASCON)*, volume II, pages 910–918, Toronto ON, Canada, October 1993. IBM Canada Ltd. Laboratory and National Research Council Canada.

[8] George T. Heineman and Gail E. Kaiser. An architecture for integrating concurrency control into environment frameworks. In *17th International Conference on Software Engineering*, pages 305–313, Seattle WA, April 1995. ACM Press.

[9] Programming Systems Lab. Darkover 1.0 manual. Technical Report CUCS-023-95e, Columbia University Department of Computer Science, 1995.

[10] Fred Long and Ed Morris. An overview of PCTE: A basis for a portable common tool environment. Technical Report CMU/SEI-93-TR-1, ESC-TR-93-175, Software Engineering Institute, Carnegie Mellon University, March 1993.

[11] Peiwei Mi and Walt Scacchi. Process integration in case environments. *IEEE Software*, 9(2):45–53, March 1992.

[12] Reference Model for Frameworks of Software Engineering Environments: Edition 3 of Technical Report ECMA TR/55, August 1993. NIST Special Publication 500-211. Available as `/pub/isee/sp.500-211.ps` via anonymous ftp from nemo.ncsl.nist.gov.

[13] H. Oliver. Adding control integration to PCTE. In *Software Development Environments and CASE Technology, European Symposium*, pages 69–80, Berlin, Germany, June 1991. Springer-Verlag.

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

[15] Ian Thomas and Brian A. Nejmeh. Definitions of tool integration for environments. *IEEE Software*, 9(2):29–35, March 1992.