Interfacing Oz with the PCTE OMS: A Case Study of Integrating a Legacy System with a Standard Object Management System

Wenke Lee   Gail E. Kaiser
Computer Science Department
Columbia University
500 West 120th Street, New York, NY 10027
{wenke,kaiser}@cs.columbia.edu

Abstract

The integration of a legacy system and a standard Object Management System (OMS) is often a very challenging task. This paper details a case study, our experiment in interfacing Oz with the PCTE (Portable Common Tool Environment) Object Management System. Oz is a multi-user process-centered software development environment that has been under development in our lab since 1987, originally under the name Marvel. PCTE is a specification that defines a language-independent interface providing support mechanisms for software engineering environments (SEE). One of the premises of PCTE is that, in theory, an SEE such as Oz can be built (or extended) using the services provided by PCTE. The purpose of our experiment was to study how a legacy system such as Oz can be integrated into a new environment framework, e.g., PCTE. The architecture of the legacy system and the services of the framework are the key factors in the integration approach. Because Oz historically has included a native OMS, our experiment focused on modifying Oz to use the PCTE OMS, which has an open and standard API. This paper describes how several Oz components were changed to interface to the PCTE OMS. The resulting proof-of-concept hybrid system has process control and integration services provided by Oz, and data integration services provided in part by PCTE. We discuss in depth the solutions to the concurrency control problems that arise in such an environment, where Oz and PCTE use different approaches to transaction management (i.e., each has its own transaction manager). The PCTE implementation used in our experiment was the Emeraude PCTE V 12.5.1, and the Oz version was V 1.0.1.

keywords:

object management system (OMS), software development environment, software process, concurrency control, transaction management.

1 Introduction

The goal of a software development environment (SDE) is to provide support for software engineering activities such as system design, implementation (coding), testing, and documenting. Support is more effective if the environment is integrated – if all its components function as a single, consistent, coherent, and integral unit [16]. 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 and the workflow among 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 [13]. In fact, the invocation order of tools to perform routine development tasks, e.g., compiler (to compile) then linker (to link), 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, e.g., executables are produced before being used in test runs. It is very clear that a process, manual or automated, is required to insure the proper order for tool invocation and software artifacts production and consumption. Therefore an important goal of many integrated software engineering environments is support for the definition, enforcement and automation of a software process.

PCTE is a specification of an environment framework which provides support for environment builders to write, assemble, and customize integrated software development environments. A number of systems, for example, the Object-Oriented Tool Integration Services (OOTIS) [9] from IBM and the PCTE Workbench [5] from Vista Technologies, have been successfully built on top of PCTE. An SDE that has built-in support for process integration is called a process-centered environment (PCE). Oz [2] is a process-centered, multi-user and multi-server environment. It (with its predecessor, Marvel) is one of the oldest PCEs still in existence. Regarding PCTE as a good example of a standard framework and Oz as an example of a legacy PCE (SDE), the main goal of our experiment was to make a transition from Oz to the PCTE framework. Ultimately, we wanted

Oz to conform to the PCTE standard and be able to integrate with other software tools in a PCTE environment. We believe that such an experiment is valuable because:

First, we can test the capabilities of the PCTE framework in facilitating the migration of legacy SDEs. We feel that it is very important for a standard framework, such as PCTE, to be able to support legacy systems, such as Oz, because while adapting to a new standard the investments in the existing SDEs should be preserved. The architecture of a legacy system and the types of standard services and their APIs provided by the framework, are the key factors in determining the approaches to the transition. Oz has several major components: the Process Engine, the Transaction Manager, the Object Management System, and the Activity Manager. They were loosely coupled through the access functions of each component. In particular, since the services provided by the Object Management System are conventional object storage and manipulation operations, we felt that we should be able to replace this component with another objectbase without much changes to other Oz components. The PCTE OMS is open and standard, its services and the API address many data integration problems. Thus, a natural pathway in moving Oz to PCTE is for Oz to use the data integration services provided by PCTE, in other words, for Oz to interface with the PCTE OMS.

Second, we can also study whether it is feasible for a standard framework and a legacy SDE to complement each other in providing support for integration. A commonly recognized limitation of PCTE is the lack for the sort of high-level control integration capabilities (it only has low-level controls similar to the basic Unix mechanisms) on which much of process integration support is built [12]. It is typically the case that given any single framework there is strong support for some forms of integration but only weak support for other forms. Thus to remedy weak support for a certain form of integration, it is often necessary to add additional support - in our case from another environment (or framework). Since Oz has strong support for process integration, we wanted to determine whether Oz can provide process integration support in a PCTE environment.

The challenge of moving a legacy SDE into a standard framework (vs. developing a new SDE using the framework) lies in the potentially substantial changes required in the SDE (since we can not assume that the standard framework can be changed to suit the legacy system - in fact, this would defeat the purpose of the “standard”). The exercise of making the changes can reveal the adaptability of the legacy SDE as well as the limitations of the framework. As a major side effect, our experiment tested the functionalities of the PCTE OMS and the ease of replacing an Oz component, its native OMS, with an outside component, the PCTE OMS.

In a truly componentized system, the OMS component can be replaced by another objectbase management system without changing the internals of other components, since ideally the OMS provides an API from which other components access the objectbase. Thus ideally only a translation layer that maps the Oz OMS API to the PCTE OMS API would need to be implemented for Oz to use the PCTE OMS. However, the implementation of Oz had incorporated a large amount of legacy code from Marvel, the predecessor of Oz. As a result, the native OMS had a weak and incomplete API and other modules made fixed syntactic format (e.g., using pointer operations to access objects) and semantic assumptions (e.g., files are stored in the host file system rather than as first class objects in the objectbase) on the underlying data model and data manipulation primitives. Since the PCTE OMS is different from the Oz native OMS in both the data model and the set of primitives, we needed to deal with these fixed syntactic and semantic assumptions. We made changes mainly to two sets of Oz functions in our experiment. The first set of functions was the object storage and retrieval functions, which were changed to use the PCTE OMS API [4]. We tried to preserve the internal (run-time) format of Oz objects by reading PCTE objects into the Oz data structures so that code that used the fixed syntax (e.g., pointers) did not need to be changed. The second set of functions was the concurrency control functions, which were expanded to incorporate PCTE activities [7] (sequences of object operations) into Oz transactions.

This paper is organized as follows: Section 2 describes briefly the services provided by PCTE, with the PCTE OMS covered in detail. Section 3 provides an overview of the functionalities of Oz components. Section 4 compares the Oz OMS model and the PCTE OMS model, and discusses how to define an 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 arise when the PCTE OMS is in use. Two different solutions are discussed in detail. Section 7 describes an example Oz environment that uses the PCTE OMS as the data repository. Section 8 compares our experiment to other related work. Section 9 concludes this paper with a summary of our experience 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 facilitate CASE tool integration is to use an Open Software Integration Platform, known as an Environment Framework. PCTE is a widely supported framework based on the Reference Model for Software Engineering Frameworks developed by ECMA and NIST [14]. 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-into" a software development environment. Instead, the environment should provide facilities for users to define and tailor a particular process suitable for the particular software development policies and activities.

For presentation integration, PCTE supports Motif-compliant tools. In the area 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 SDE. It typically consists of a network of hosts, which are workstations that allow users to access the Emeraude PCTE environment. It provides an operating system-like environment where users can log into the PCTE shell and execute PCTE commands (e.g., objectbase navigation commands) or invoke integrated tools. It also provides APIs (in various programming language bindings) that implement the PCTE specifications. These APIs enable developers to implement or re-engineer software tools to integrate them into the PCTE environment.

2.1 The PCTE OMS Model

The main focus of our case study is on 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 (both the product and control data of an SDE). 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 (partitions of of the objectbase) corresponding to the physical distribution of storage devices over the network. A volume is mounted on a Unix file system or a device of a particular host so that the data held in the volume is available transparently throughout the environment.

2.1.1 Object Model

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

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** (paired mutually dependent links) 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.

Direct reference to a particular object or link in the objectbase is through navigation. A pathname that walks the composition hierarchy top-down can be used to access an object or a link. For example, assuming *book* is a top-level object, then the pathname to access object *chapter_1* of *book* is *book/chapter_1*. 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 a 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.

2.2 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 the OMS directly through a set of OMS commands.

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 (at run-time) are managed as PCTE "client" processes. These processes are not stored 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 [15]. On the other hand, Unix tools that are interfaced to a PCTE environment need to be linked with the PCTE Unix libraries, and their executions (which are standard Unix processes) are "alien" processes to PCTE. In our case study, we modified Oz and linked it with the PCTE libraries instead of integrating it into PCTE.

3 Oz Overview

Oz is a process-centered software development environment that supports cooperation among engineers who follow a project-specific software development process (workflow) [2]. It facilitates both data sharing and process coordination. The architecture of an Oz environment is based on the client/server model. An Oz server manages the objectbase that stores both product and control data of a software development project. It also automates and enforces the project-specific process. An

Oz server can support multiple Oz clients sharing the same process and accessing the same objectbase. A user interfaces directly with an Oz client, which in turn communicates with an Oz server. An Oz client provides a user interface from which a user can view the objectbase hierarchy, query the objectbase, and can perform development tasks (process steps), e.g., edit, compile, and build. In general, a task (process step) execution cycle in Oz is the following:

- The Oz client sends a task execution request (from the user) to the Oz server.
- The Oz Server checks the prerequisites of the task according to the process definition, and sends the necessary data (e.g., a C file object) back to the client to perform the task. In case that the prerequisites are not satisfied, Oz server may invoke other process steps whose results may satisfy the prerequisites of the current process step.
- The Oz client then invokes the appropriate tool (e.g., an editor) on the product data.
- After the user finishes the task (exits the tool), the Oz client sends back the data (e.g., a changed C file object) to the Oz server.
- The Oz server propagates appropriate changes to the objectbase, and according to the process definition, may invoke other process steps if their prerequisites are now satisfied because of the changes made by the current process step.

3.1 Major Oz Components

As stated earlier, the architecture of Oz is one of the most important factors in our approach of moving Oz to PCTE. We now briefly describes the major components of the Oz.

The Process Engine. Tasks are the key elements of a process definition. Each task definition 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 task to take place; an optional (physical) activity that specifies a tool envelope (see Section 3.3) 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. Tasks are implicitly related to each other through matches between a predicate in the condition of one task and an assertion in the effect of another task. Process enactment is done through the dynamic construction of task segment. Backward chaining of a sequence of tasks is attempted to satisfy the condition of a task. Forward chaining through "atomicity chains" is performed 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 [10], 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 approach of associating tasks with transactions. A task segment consists of all tasks executed during backward chaining, followed by the user-invoked task (which caused the backward chain), followed by all tasks executed during forward chaining. Pern supports a nested transaction model in which a task segment corresponds to a series of top-level transactions. Each atomicity chain is a subtransaction of the triggering transaction and each task 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 inheritance lattice, an 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 Communicating with the Oz OMS

All the major Oz components need to communicate with the OMS. The Process Engine queries the objectbase to obtain the actual parameters of a task and evaluates its condition. The task's activity accesses and modifies data in the objectbase. The Process Engine also modifies data in the objectbase when it asserts an effect of a task. 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 was assumed that the whole objectbase is loaded into Oz buffers when the Oz server is started and access (read) to objects is then done by directly addressing (using memory pointers) into these buffers instead of function calls to the OMS interface. Object modification (updates) is done by a set of 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 hidden file system, and *composite* attributes, which refer to one or more component objects.

3.3 Oz Tool Envelopes

A typical Oz process may utilize several external tools to carry out software development activities. Oz provides one or more tool envelopes for each external tool so that the external tool can be used as a "black box". In our case study, we developed a special envelope to wrap around some PCTE related object operations, see Section 6.3. A tool envelope represents the implementation of an activity and is executed by Oz client. Tool envelopes are written in SEL, an extended Unix shell language [8]. SEL allows the tool integrator to write shell-like 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 the 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, which may be assigned to object attributes in a task's effects. A tool envelope written in SEL is translated using the Oz *make_envelope* utility into a standard Unix shell script (currently, *sh*, *csh* or *ksh*).

4 Implementation – Data Model Mapping

In an Oz and PCTE OMS hybrid system, Oz objects (all control and product data of an Oz environment) are to be stored in the PCTE objectbase. Thus we need a general solution (not ad hoc for each Oz environment) of using the PCTE object model to define the data schema used by an Oz environment. Since an Oz data schema is defined according to the Oz object model, we need to first compare the Oz object model and the PCTE object model, then devise a mapping between the two.

4.1 Comparison of Oz and PCTE OMS Models

4.1.1 Similarities

Since both the Oz object model and the PCTE object model are intended to be general-purpose, there are many similarities between the two.

- Both Oz and PCTE support the concept of an object, that 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 (time stamp), real, and user-defined enumerated types. In PCTE, only string, integer, boolean, and date are allowed.

- Both use links to model binary 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 main differences between the two object models are:

- 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 scoped within each class where the attributes and links are defined and applied. In PCTE, all objects, links and attributes in a schema share a single name space.

- Oz objects may have file attributes whose values are the pathnames of files in a "hidden" file system (with directory hierarchy and file names not comprehensible outside of the Oz 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 Using the PCTE Object Model to Define Oz Schemas

The following are general guidelines for defining an Oz data schema in the PCTE OMS:

- Each link and attribute name in PCTE is prefixed with the object type (class) name. For example, attribute *title* of class *MANUAL* is defined (named) as *MANUAL__title* (see Figure 2). This guarantees that links and attributes names are unique in a schema.

- Oz link attributes are modeled as relationships in PCTE and each composite attribute (for component object) in Oz is mapped to a composition link.

- 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 *Oz_FILE* named *contents*.

- *Oz_OID* and *Oz_NAME* are Oz system-attributes for all Oz objects in PCTE. 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 pathname of an object can then be regarded as the internal (navigational) 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. Therefore we omitted the validation of enumerated attributes for the sake of simplicity.

We limited our case study to those Oz data models that do not use multiple inheritance in their class hierarchies. Although one can always devise an algorithm that 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 in the objectbase. Suppose we have an Oz class definition as shown in Figure 1, which defines the class *MANUAL* as having base attributes *title*, *reformat* and *format_status*, consisting a set of objects (*submanuals*) of class *SUBMANUAL* as its composite attributes (component objects), a file attribute (*first_page*), and a link (*first_sub*) to an object of class *SUBMANUAL*. Then the relevant information in the PCTE schema definition file will be as shown in Figure 2. Here, we can see that the expansion factor (in terms of lines of definition) is about two because each attribute (base attribute, link, component) has to be first defined individually, and then be referenced in the class definition.

5 Implementation – Interface to the PCTE OMS

Figure 3 depicts the simplified (run-time) system architecture of Oz interfacing with the PCTE OMS. The arrows represent data flows. Here we see that Oz modules make direct reference to

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

Figure 1: An Oz Class Definition

(read from) the object buffers instead of always using the OMS API functions[1]. This means that 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 interested in investigating how the process engine and transaction manager need to be changed when an external OMS is used. We wanted to minimize our efforts while achieving our goals. Therefore we tolerate this fixed assumption by loading the PCTE objectbase into the in-memory buffers in the Oz server. The Oz *read_objectbase* function was rewritten to read the PCTE objectbase.

Once the objectbase is loaded into Oz buffers, objectbase navigation can be done in memory using the existing Oz functions without changes. On the other hand, the effects of object update operations are written immediately to the PCTE objectbase. This is done to facilitate data sharing with other tools running in the PCTE environment. The PCTE-specific code (that implement the updates) 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 (in the Storage Manager) handle the details. The PCTE objectbase has effectively replaced the storage manager.

5.1 OMS Primitives in PCTE

Before we go into the details of reading and updating the PCTE objectbase, we first briefly describe the APIs of the PCTE OMS.

The PCTE OMS C language binding contains a rich set of OMS primitives, which are essential

[1]this is no longer the case in Oz Version 1.2.1

```
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

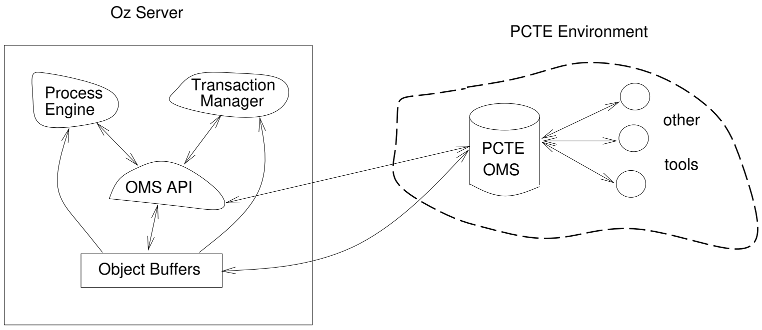

Figure 3: System Architecture of Oz Interfacing with PCTE OMS

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 and browsing.
- Data Manipulation Primitives: object, link, and attribute manipulation.
- Establishing the Working Schema – establish a working schema consisting of a set of schema definitions. Tools can access only the objects, links, and attributes defined in the working schema.

5.2 Reading the Class and Object Hierarchy into the Oz Server

When the Oz server starts up, it first reads the class hierarchy of the data model in use and then reads the whole objectbase. We now describe our re-implementation of these two functions.

Every Oz environment is associated with a file system directory, called the environment directory, which contains environment-specific process definition files, tool envelope files, an objectbase file, and a subdirectory containing the "hidden file system", which stores the file attributes of objects. The *strategy* file contains the class hierarchy in an internal format, which is loaded into the Oz server when it starts. Thus Oz effectively employs a single (working) schema for each Oz server. In PCTE classes are instead 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 schema definitions 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 (of the PCTE OMS) are then used to get all the type definitions in the PCTE objectbase.

For each object type definition, 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 (recall from Section 4.2 that we use PCTE relationships to represent Oz links and PCTE composite links to represent Oz composite attributes).

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 is thus the only needed contents of the *objectbase* file in the Oz environment directory. To read the object hierarchy from these top-level objects, 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 the reference link and the 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 (a component object) of A. The consequence would be that the reference links of A can not be set up when A is being processed, thus a second pass would be needed to set up the reference links between objects. It is thus clear that our modified DFS is more efficient.

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 to update the objectbase had to be rewritten using PCTE OMS primitives. These changes were in general very straightforward.

The exception was the *move* operation, which 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*.

5.4 Accessing PCTE Objects in an Oz Task

As discussed in earlier sections, the Oz process engine needs to access objects during all phases of process enactment, namely, bindings of task inputs, checking of prerequisites, execution of task activity, and assertion of task effects. Since the Process Engine only calls the high level built-in operations to modify the objectbase, there should not be any changes required after the built-in functions have been changed to interface the PCTE OMS. However, in our experiment, we have found some fixed semantic assumptions about how files are modeled, that are beyond the scope of the built-in object operations and need to be dealt with separately.

In the Oz OMS, a file contained within an object is represented by 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, only the process engine code that handles *file* attributes had 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 task 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 are known and fixed once the object instance is created. In the original Oz implementation, when an object is read into the Oz server (at 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 because the file path for a file attribute is already known even if the file is not physically created yet, so long as the object instance exists. When a task activity that requires access to a file attribute is executed, Oz just passes the file attribute (the file path) to the task; it then relies on the tool envelope (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 task, say *edit*, is applied to *foo.c* the first time, it requires access to its *C_contents*. Oz will pass to the editor the *C_contents* attribute, which is just a file name (for a nonexistent file) in the host file system. After *edit* completes, 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*. Therefore, creating a pointer (in memory) to a nonexistent component object is obviously not acceptable because it will seriously compromise 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 task that requires a file attribute parameter is applied to this object. This action of creating the component is logged automatically by the governing transaction; therefore it can be undone in the case of a 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. 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 flexible, powerful and elaborate system. In contrast, 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 (task) level transactions and PCTE handles the lower (object operations) 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 normally abort.

There are certainly other more desirable approaches that we wish we could have used to integrate 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 task). 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 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. The effect of a lock on a resource is the following:

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

- **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.

PCTE has a different implementation of nested transactions than many other DBMS's 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. 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

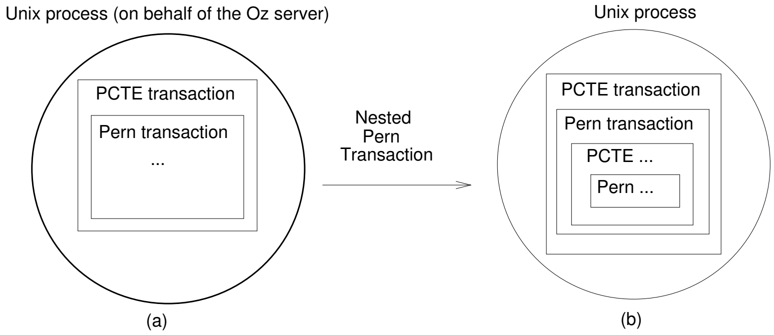

Figure 4: Nesting a Pern Transaction within a PCTE Transaction (not possible)

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 task often requires nested transactions, for example, because of an atomicity task segment. 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, see part (a) of Figure 4. If the Pern transaction happens to have nested subtransactions, a PCTE transaction should also be started before each Pern subtransaction is started. But this second PCTE transaction would then be nested within the first PCTE transaction, as seen in part (b) of Figure 4, and they are both started by the same Unix process for the Pern (actually the Oz server) executable. 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 task segment) 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. The semantics of these operations are well-defined and have no dependencies on other operations. Therefore they can be implemented either as protected activities or as transactions because either can provide the same protection against other concurrent activities. 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 task'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 task'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 task. This requires 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 task 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 source 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 our Oz/PCTE integration, 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 receives 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 task activity) from the Oz client. It then starts a PCTE transaction, tries to lock the required objects, and then calls a **tool envelope**. Figure 5 (the arrows represent control and data flows) 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 and returns the result values to its caller. Normally the return code from a tool envelope is used to choose one of the task'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.

When an external tool is called as a task's activity, the Process Engine of Oz marshals arguments (objects or attributes) required by this tool in a command line arguments buffer and sends it to an external program, the tool 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, however, 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 Process Engine translates it to

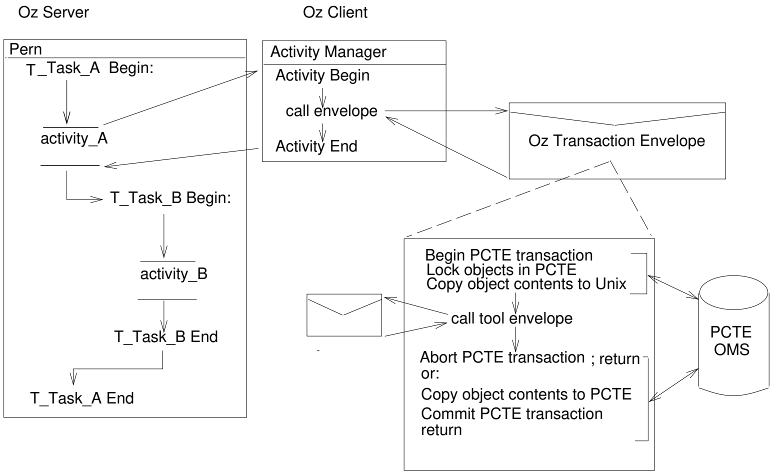

Figure 5: Transaction Envelope

"*lock_mode* $*$ *PCTE_file* $>$ *Unix_file* $>$ *backup_file*", where:

- *lock_mode* is the kind of lock required on this object.

- *PCTE_file* is the full path of the object contents in the PCTE objectbase.

- *Unix_file* is the file (in the Unix file system) that initially contains the PCTE object contents. It is the file that the Unix tool (called by the tool envelope) will operate on.

- *backup_file* is the Unix file that maintains the original object contents.

The *lock_mode* indicates the lock mode that Pern requires on this object. For our experiment, 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, which would generally require the "promotion" of the Oz lock mode. For example, if a *SX* (for shared exclusive) was requested in Pern, it would be promoted to X, which would then lead to a write-transaction lock in PCTE.

The object contents in PCTE, specified by *PCTE_file*, are copied to *Unix_file* before the tool envelope and thus an external tool (a Unix tool such as Emacs) is invoked. After the tool envelope execution, the contents of the *Unix_file* are 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 prior to 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 creates a named pipe and passes 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. When the transaction envelope tries to lock an object in the PCTE objectbase, it may encounter a lock conflict because another PCTE tool is already holding the lock. In such a case, the transaction envelope aborts its PCTE transaction, then the upper level Pern transaction also aborts. The recovery is also handled by Pern.

Because file (PCTE object) copying is required for most of the Unix tool invocations, the performance of task activities when the PCTE OMS is in use must be worse than the case when the native 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. When recovery is needed, 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 both subtransactions of the upper level Pern transaction. Therefore their results need to be undone if the Pern transaction aborts.

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 Dealing with Concurrent PCTE Tools

Notice that the order of data lockings and the fact that PCTE transactions are subtransactions of 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 tasks *task_A* and *task_B*, respectively. Further, the activities (or effects) of these two tasks try to gain exclusive access to a PCTE object *bar* at the same time. Since the execution of each Oz task 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 *task_A* or *task_B*. Actually, Oz (and hence, the Pern instance in use) requires that all the objects that a task intends to update be locked before the task's activity and effect take place. When, say, *task_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 *task_A* starts accessing *bar*. This mechanism (a strict two phase locking scheme) guarantees serializability between *task_A* and *task_B*, and is deadlock-free because: 1) *task_A* and *task_B* have to obtain all the locks at once from Pern; and 2) when *task_A* or *task_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 task (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 the same Oz server, and thus the same 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 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,

```
task_A:          task_B:          external activity C:

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

Figure 6: Concurrent Activities

unless all PCTE activities follow a transaction model such as two-phase locking in accessing the PCTE objectbase, 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") [3]. To see how this can happen, suppose that we have two tasks *task_A* and *task_B* and they are chained together (from *task_A* to *task_B* by an atomicity chain). The activities of *task_A* and *task_B* are shown in Figure 6. The two tasks are chained because of, for example, a consistency constraint: "whenever X is updated, update Y accordingly". In Oz, *task_B* will be in a Pern subtransaction of *task_A*. X is locked in *task_A*'s transaction, and the lock is passed to *task_B*'s transaction. But when the transaction envelope is used, the activities of *task_A* and *task_B* will be in different PCTE transactions, and passing locks between the two envelopes, which are two different Unix processes, is not possible because a PCTE object can be locked by only one activity (in this case, on behalf of a transaction envelope, a Unix process) in any single moment, and the lock is released when the activity (and hence the PCTE transaction) is ended.

Imagine that after the transaction envelope for *task_A* commits the corresponding PCTE transaction which updates X (at which point the lock on X is released), but before the transaction envelope of *task_B* can lock X and Y, another external activity C can start. The activity of C is also shown in Figure 6. C can finish quickly enough so that the transaction envelope (of *task_B*) 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 *task_B* uses it, this results in a "incorrect summary" on X.

Figure 7 shows the inadequate protection against external activities when transaction envelopes

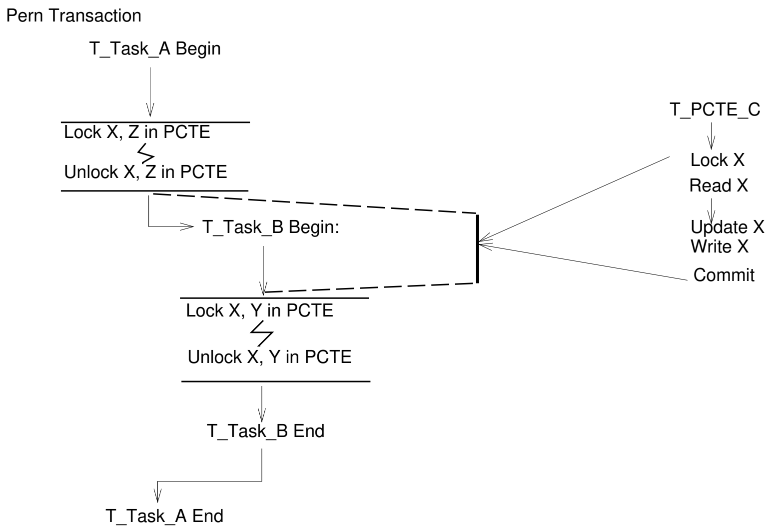

Figure 7: Incorrect Summary Problem

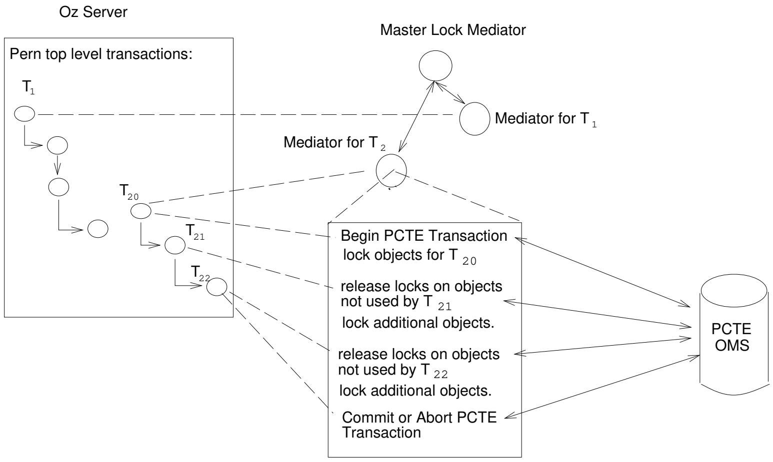

Figure 8: Lock Mediator

are used in a nested Pern transaction (for a task segment). 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.

A solution to this problem is to have a **lock mediator** process to lock all the objects needed for a Pern (nested) transaction. As shown in Figure 8 (the dashed lines represent correspondence between Pern transactions and lock mediators), one instance of such a mediator process is needed for each top level Pern transaction. Our implementation of the lock mediator is the following: when the Oz server is started, also start the master lock mediator process and set up a pipe (Unix FIFO) between it and the Oz server process. Whenever a top level Pern transaction is started, Oz (on behalf of Pern) notifies the master mediator process to fork and spawn a child mediator process for this new transaction. The master mediator 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 pipe between the Oz server and the master mediator process. The latter then sends the lock/unlock request to the corresponding child mediator process using the named pipe between these two processes. The child mediator process does the actual lock/unlock for 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. Copying object contents from/to PCTE should also be done by the child mediator when it locks/unlocks an object. Any failed lock request to PCTE will trigger the child mediator to abort the PCTE transaction. It then notifies Pern, through the master mediator, to abort the top level Pern transaction. Using this scheme, each top level Pern transaction, along with its possible subtransactions, correspond to a single PCTE transaction. Locks can be transferred between Pern transactions that belong to the same top-level transaction because all the locks within a top-level transaction are held by a single process.

Comparing to the 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 (the Emeraude PCTE has commands for setting and changing user/group permissions), 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.

We did not fully implement the lock mediator approach due to time and resources constraints.

7 An Example: Oz/Doc with PCTE OMS

We now demonstrate an example of using Oz and the PCTE OMS together. We ported an existing Oz environment, Oz/Doc, to test the modified Oz server (which interfaces with the PCTE OMS), and to conduct some performance analysis.

7.1 The Oz/Doc Environment

Oz/Doc is a document production environment implemented on top of 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 an Oz object-oriented data model. A set of Oz task definitions was 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 relatively simple and thus it can be mapped into the PCTE OMS easily. Another motivation is that there are readily available PCTE tools, for example, the "Edit Object" command in the OMS Browser, that can be used as concurrent PCTE tools (with regard to the Oz tasks in Oz/Doc, e.g., *edit*) to test our concurrency control mechanisms.

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 Oz Client GUI, which is responsible for sending object and task 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. Tasks and Oz built-in commands can be selected from the menus. Clicking an object identifies it as a parameter for a selected task. External tools (for example, *emacs*) are encapsulated as the activities of Oz tasks. 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 a Oz/Doc data model onto the PCTE OMS model using the approaches discussed in section 4. Most of the data definition mappings are straightfor-

```
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 9: Definition of the FILE class (with multiple inheritance) in Oz/Doc

ward. Figure 1 and Figure 2 give an example on how to define an Oz object class in the PCTE OMS. Oz/Doc has only one class definition, *FILE*, that is derived from multiple super classes, namely, *TIMESTAMPED*, *VERSIONABLE*, *HISTORY*, *ENTITY*, as shown in Figure 9. The multiple inheritance is only one level. Therefore, as shown in Figure 10, 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 11 shows the contents of these two files. For *objectbase*, it specifies 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. The *strategy* file usually contains the compiled Oz data model and task definitions. In our Oz/Doc environment, it first specifies the names of the schemas that make up the working schema set. Then it lists the name of the schema, oz_doc, that defines the Oz/Doc data model (Figure 10 is actually part of the output of the PCTE command "sds_decompile oz_doc", which prints out the

```
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 Oz_FILE;
FILE__version : composition link to Oz_FILE;
FILE__contents : composition link to Oz_FILE;

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

```
objectbase:          strategy:

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

Figure 11: Objectbase and Strategy Files

definitions of a 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 can start the *edit* task on a *FILE* object named *introduction* (for example, by selecting the *edit* command off the menu and identifying *introduction* by clicking it on the objectbase display). Oz copies the contents of *introduction* from PCTE to a Unix copy and creates a backup copy for it, before invoking the Unix edit program on the Unix copy. It also copies the Unix copy back to PCTE if the task succeeds (the governing Pern transaction commits), or the backup copy back to PCTE if the Pern transaction rollbacks. Both the Unix copy and the backup copy of *introduction* are removed after the *edit* task (and the appropriate commit or rollback actions) completes.

To the users, all these file I/O operations happen "behind the scene" in the sense that the users are not aware at all that the PCTE OMS (instead of the native Oz OMS) is in use.

7.3 Performance Analysis

To get some idea of how file I/O operations has affected the performance of task activities, we defined a *file_io* task with an activity that concatenates "hello world" to the end of a file. We then used an Oz tty client, which can execute tasks in batch mode, to fire this task on the same file (a PCTE object) 100 times. The average total time it took to finish all 100 task invocations was 316 seconds. Using the same task in an identical Oz/Doc environment where the Oz OMS is in use, it took an average 115 seconds to finish all 100 task invocations. From the discussion in the previous section, we see that the following additional file I/O operations were performed when the PCTE OMS was in use: file creation in Unix, file copied from PCTE to Unix, file copied from Unix to PCTE, and file deletion in Unix. The two copy operations were responsible for the bulk of the difference in total elapse time.

It is evident from this result that when the PCTE OMS is in use, file-related Oz tasks will take longer to finish (from a user's perspective) because of the additional file I/O operations. This performance hit is inevitable.

8 Related Work

HyperWeb [6] is a framework that supports the construction of hypermedia-based software development environments. In Hyperweb, software artifacts is stored in the PCTE OMS. Moreover, the hypermedia linking of these software artifacts are also modeled and stored in the PCTE OMS – each hypermedia 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 is that the server and clients run outside of the PCTE environment. There was no discussion of concurrency control or process integration.

The CORBA, ATIS and PCTE integration [1] was an interesting experiment to test the possibility of coupling CORBA, PCTE and CASE tools. It layered CORBA on top of PCTE so that CORBA provides the interfaces for external access to the underlying object model implemented in the PCTE OMS. To guarantee the interoperability between CORBA and PCTE, the CORBA distributed and secure execution services were implemented to conform with the PCTE Execution and IPC specification. Some CASE tools, i.e., the ATIS (A Tool Integration Standard) Version and Configuration Services, were added to this hybrid environment to demonstrate the advantages of the distributed object (CORBA) interface on top of PCTE in terms of data and control integration.

9 Conclusion

Our approach might be considered as a light integration with PCTE because the Oz server, Oz client and external tools remain as alien processes to PCTE. Theoretically we could have put all the Oz code into the PCTE objectbase so that Oz runs within the PCTE environment, but given the sheer amount of source code and the heavy Unix dependencies, we did not believe a complete porting was practical. We believe that our situation is typical for a legacy SDE that is adapted to a new framework. Because the Oz components were loosely coupled and the PCTE OMS provides a standard API, we were able to interface Oz with the PCTE OMS, thus accomplished the integration between the two.

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 showed that Oz and PCTE are complementary and add value to one another. For tools running in an Oz environment, using the PCTE OMS enables them to also share data with other PCTE tools (that are not in the Oz environment). For the PCTE environment, using Oz allows process integration among Unix tools that access the PCTE OMS. This can be very useful since not all software tools will be "tightly" integrated into PCTE (moved into PCTE); rather, they will still run on Unix while using the PCTE OMS and other services.

This experiment shows that the Oz server can be modified to work with an external OMS with a somewhat different data model. 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 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

Since Oz already had a fully functional process engine and a certain degree of abstraction in terms of object access, we were able to focus on the making an interface to the PCTE OMS in order to build an environment that has both process integration and (open and standard) data integration services.

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. Obviously if PCTE allows nested transactions within a single Unix process, mapping (starting) a PCTE transaction for each Pern transaction becomes an easy solution to this problem. Alternatively, Pern can be re-engineered into a separate program; and in run-time, a Pern process is created (spawned) for each top-level Pern transaction and locks (PCTE) objects in a scheme similar to the lock mediator approach. This later solution is much more complicated.

Finally, as mentioned throughout this paper, there were fixed assumptions about the OMS in the Oz code. These fixed assumptions have been eliminated since the work reported here was completed. All access to class, object and attribute are now done through function calls to an OMS API. Had this interface been in place when we started our experiment, we could have just implemented a set of functions that can be used by the API to access the PCTE objectbase. However the major data model mismatch (for example, files are component objects instead of object attributes) still requires changes outside of Oz's OMS interface (for example, in the process engine if the code assumes only *file* attributes, it needs to be changed to also consider *composite* attributes of *Oz_FILE* if the PCTE OMS is in use).

9.2 Future Work

Pern and and more recently, a new Process Engine called Amber, have been developed as independent 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 into 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 [11]. Similarly, putting Amber into PCTE would add process integration not discussed in this paper [17]. Amber has a mediator interface that enables environment developers to tailor it to support environment-specific process enforcement and assistance models. Since our experiment is a light integration, it is difficult for Oz to enforce controls on tools running within PCTE other than using data locking, which provides only passive control. However if the Oz components are put into PCTE, they can take advantage of the control integration services provided by PCTE, e.g., the message queue, support for user roles, and security control, to provide active control and integration for the PCTE tools.

Acknowledgments

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.

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

References

[1] A. Argento, C. Bonferini, F. Dematte, and S. Manca. ECMA PCTE, CORBA and ATIS. In *PCTE Newsletter*, pages 20–27, Nanterre Cedex, France, June 1992. Jean-Claude Rault.

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

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

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

[5] James C. Ferrans. The PCTE workbench. In *PCTE Newsletter*, pages 12–15, Nanterre Cedex, France, October 1993. Jean-Claude Rault.

[6] 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.

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

[8] 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.

[9] William Harrison, Harold Ossher, and Mansour Kavianpour. OOTIS: extending PCTE with fine-grained tool composition. In *PCTE Newsletter*, pages 11–19, Nanterre Cedex, France, December 1992. Jean-Claude Rault.

[10] George T. Heineman. *A Transaction Manager Component Supporting Extended Transaction Models*. PhD thesis, Columbia University, May 1996.

[11] George T. Heineman and Gail E. Kaiser. The cord approach to extensible concurrency control. In *13th International Conference on Data Engineering*, April 1997.

[12] 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.

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

[14] 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.

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

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

[17] Andrew Z. Tong, Gail E. Kaiser, and Steven S. Popovich. A flexible rule-chaining engine for process-based software engineering. In *9th Knowledge-Based Software Engineering Conference*, pages 79–88, September 1994.