An Architecture for Integrating Concurrency Control into Environment Frameworks

George T. Heineman   Gail E. Kaiser

Department of Computer Science, Columbia University
500 West 120th Street, New York, NY 10027
(212)-939-7085 Fax: (212)-666-0140
CU-021-094

Abstract

Research in layered and componentized systems shows the benefit of dividing the responsibility of services into separate components. It is still an unresolved issue, however, how a system can be created from a set of existing (independently developed) components. This issue of integration is of immense concern to software architects since a proper solution would reduce duplicate implementation efforts and promote component reuse. In this paper we take a step towards this goal within the domain of software development environments (SDEs) by showing how to integrate an external concurrency control component, called Pern, with environment frameworks. We discuss two experiments where we integrated Pern with Oz, a multi-site, decentralized process centered environment, and ProcessWEAVER, a commercial process server. We introduce an architecture for retrofitting an external concurrency control component into an environment.

keywords: Componentization, Transactions, Software Architecture, Collaborative Work

[1]This paper is based on work sponsored in part by Advanced Research Project Agency under Contract F30602-94-C-0197, in part by National Science Foundation Grant CCR-9301092, and in part by grants from Bull HN Information Systems and IBM Canada Ltd. Heineman is supported in part by an AT&T Fellowship and in part by IBM Canada. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the US Government, AT&T, Bull or IBM.

1 Introduction

Multi-user software development environment frameworks (henceforth, just "SDEs") need some form of concurrency control mechanism to detect and resolve conflicts since more than one user task may attempt to access the same data in incompatible ways: generally, a read (or write) overlapping with at least one (other) write. Many SDEs provide some variant of the file checkout paradigm, typically coupled with versioning, while others incorporate a database system with conventional transactions. But some SDEs do not provide any concurrency control at all, either because the system was initially envisioned as supporting only one-user/one-task (at a time) and later extended to multiple concurrent tasks per user and/or to multiple users with essentially manual synchronization (e.g., passing the "floor" in multi-user editors [8]), or because the designers assumed that some external facility would provide concurrency control (such as for the Cap Gemini Innovation ProcessWEAVER [11]). In this paper, we present an external concurrency control (ECC) architecture with example applications drawn from the latter category, where concurrency control was left for another component. We believe our approach could be adapted to checkout and conventional transaction systems to enhance their concurrency control capabilities, and possibly assist in extending single-user systems to support multiple users.

The concept of componentized systems has obvious potential benefits from dividing the technical and economic responsibilities for providing services; in the case of SDEs, we can refer to the "Toaster" model [9] to find user interface, task management, data integration, data repository and communication components, as well as individual tools. It is still unclear, however, how one can and should integrate SDE components (as opposed to tools) together to produce a coherent, useful, and usable system. Previous work on componentized SDEs, such as various systems constructed on top of PCTE [23], has generally adopted ad hoc solutions suitable for their particular integrated system rather than introducing a general *architecture* for a class of component integrations. We do not attempt to address the entirety of this very large problem here: this paper is concerned primarily with architectures for integrating the task management services (TMS) component of an SDE with ECC, although we also discuss integration of ECC with data repository services.

From the viewpoint of software architecture, this paper explores the integration of independently developed, pre-existing components, in contrast to work on (1) construction of systems based on one (or a small number of) pre-existing component(s), with the rest of a system implemented more-or-less from scratch to take advantage of these components (e.g., Mach [20], Camelot [10], and many database applications); (2) building-block kits, where sets of components that can be mixed and matched are designed and implemented together (e.g., Genesis and Avoca [4]); and (3) componentization, where a pre-existing system is reengineered into individual components, perhaps in preparation for (1) or (2). We also distinguish our work from tool integration (such as via a message bus), which is concerned with how tools fit into a common framework rather than how that framework is constructed; program generators (notably compiler generators), where a standard system template is filled in by parameterization; toolkits (there are numerous X windows and other user interface toolkits), which raise the level of abstraction for programming but usually do not provide a prefabricated component; and module interconnection languages (including "mega-programming"), where the main concern is specifying connections between existing interfaces. The most significant difference compared to all of the above technologies is that we deal with pre-existing, independently developed components *whose interfaces do not match* and *whose interfaces cannot be modified to match* – much like trying to fit a square peg in a round hole. To make integration possible under such circumstances, we argue that external *mediators* must be implemented to overcome the mismatch between components (see [22], for a supporting view).

Our challenge is that TMS components (as described in the literature) do not specify any particular model of what they require for concurrency control, nor do the (known) implementations provide any pre-defined interface to an ECC utility. While the omission allows for flexibility, it puts an extra burden on the system designer/builder. Further, it is not clear whether either the checkout paradigm or conventional transactions, the main models explicitly incorporated into some existing SDEs, are really adequate. The database community reports [18, 1, 21] that short, pre-programmed, atomic and serializable transactions, developed for data processing (payroll, inventory, and the like), are not appropriate for the long-duration, open-ended (e.g., interactive), and potentially collaborative applications prevalent in engineering design applications – and, we believe, in SDEs. The checkout paradigm addresses long-duration and open-ended activities, but is weak on collaboration since user tasks are performed in a private workspace; version merging, notification, group workspaces, and other work-arounds that attempt to extend checkout to collaborative work are discussed elsewhere [17].

SDE applications probably require what are sometimes termed "cooperative transactions". A variety of collaborative concurrency control mechanisms have been proposed in the literature; see [3] for a survey. In general, cooperative transactions make it <u>possible</u> to guarantee atomicity (rollback of an entire atomic unit if it cannot be completed) and <u>possible</u> to enforce serializability (isolation that makes it appear as if only one user/task is accessing the data repository). Cooperative transactions can use application-specific semantics-based policies to resolve failures and concurrency conflicts, often to enable collaborative work. The purpose of this paper is neither to examine the collaborative concurrency control problem nor to present yet another possible solution. Instead we assume one or more suitable concurrency control mechanisms are available (briefly describing our design and implementation of one such component), and direct our attention to interfacing to TMS components constructed without any specific means to exploit such facilities.

We first present the requirements we have identified, both for the internal concurrency control model of TMS (perhaps brought out through a mediator) and for interfacing

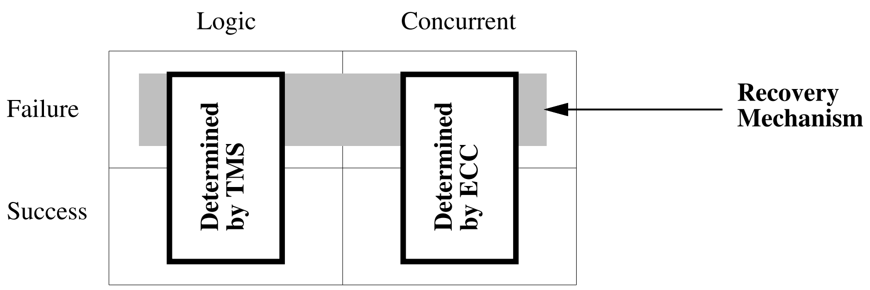

Figure 1: Division between logic and concurrency

to an ECC component. Then we discuss the design and implementation of a prototype ECC component, PERN. We describe two experiments where we successfully applied our approach to a TMS component without its own concurrency control mechanism. We conclude with lessons learned and suggestions for future work.

Since our focus in this paper is on *interfacing* task management services and external concurrency control mechanisms, we do not discuss the cooperative transaction *functionality* supported by our component (for details, see [2, 14]). Instead, for simplicity in presentation, we employ conventional transactions in all of our examples.

2 Requirements

The basic requirements imposed on TMS by concurrency control mechanisms, whether internal or external, include the following:

- Logical units that could potentially be mapped to transactions. Ideally, multiple granularities would be represented: at least individual activities and full sessions as well as tasks, and possibly intermediate granules such as hierarchical tasks. This would permit different concurrency control properties to be ascribed to different levels, such as isolation during activities (or individual data accesses within activities) while still allowing collaboration within a circumscribed group during tasks (that is, group members' tasks might interleave at activity boundaries).
- Capability for recovering logical units when failures occur, typically via rollback (undo to a point before the logical unit began) or compensation (restoration to a semantically consistency state, not necessarily the same state the system was in before the logical unit began). As illustrated in Figure 1, TMS is responsible for determining when tasks logically succeed (e.g., goals are achieved) or fail (e.g., logical prerequisites cannot be satisfied or implications cannot be fulfilled). ECC detects when a (cooperative) transaction fails (e.g., unresolvable conflicting accesses among transactions so that one or more must be disallowed) or succeeds

(e.g., the absence of such conflicts). Ideally, the mapping from transactions to tasks would permit the same recovery mechanism to be used for both.

When the concurrency control mechanism is external to TMS, several additional requirements must be fulfilled:

- The ECC component must not require source code modifications or recompilation of the TMS component (this is impractical for most vendor offerings, typically provided as binary executables or as object code libraries). Ideally, the TMS component should not require code changes to ECC, although an application programming interface (API) or other parameterization mechanism should be available.
- The special-purpose mediator, or "glue", code should be relatively small, compared to the code size of either TMS or ECC. A competent system builder should be able to construct the mediators from the interface specification of TMS and ECC; that is, no special knowledge of the interior workings of TMS or ECC should be required.
- The performance level should be appropriate for the interfacing style of TMS. In particular, if TMS provides a library for direct linking, higher performance would be expected than one where interaction with ECC occurs over a message bus. This implies that a practical ECC should provide multiple interconnection vehicles.
- The choice among ECCs (assuming a range is available), or the decision to employ an ECC at all, should not unduly restrict the selection of other system components, notably the database management system. However, there must be some means for ECC to interact with the other components, such as through additional mediators or an extended interface provided by TMS.

3 ECC Architecture

To fix the terminology for this paper, we turn to the "Toaster" model by Earl [9]. This reference model targets particular services of an SDE and groups them by functionality; the various layers are shown in Figure 2. Data repository services are responsible for storing the data in any combination of file system and database(s). Data integration services manage access to the data, possibly using a transaction service. Task management services (TMS) insulate the users from the actual details of the tools operating on the data. Generally, a TMS layer organizes user activities into goal-oriented tasks and provides a high-level abstraction for managing tasks. However, some existing SDEs provide no particular notion of tasks except (degeneratively) as

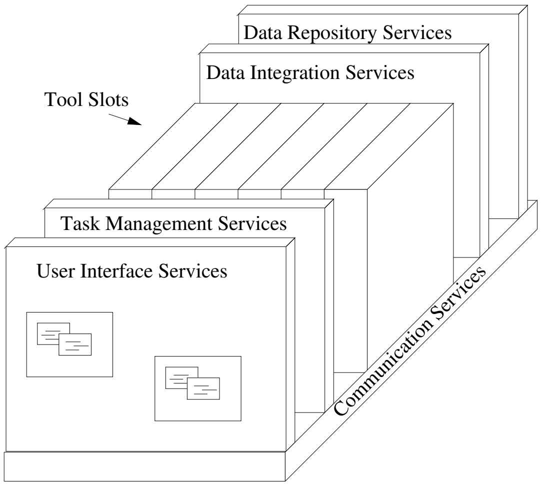

Figure 2: "Toaster" Reference Model

individual tool invocations. Finally, user interface services provide the appropriate abstractions for users.

SDE architectures do not necessarily match this reference model, since it is not a blueprint or a design specification. Individual SDEs may or may not be constructed from modules or components that map in any direct way to the various Toaster model services. From the viewpoint of concurrency control, there are two possibilities: Either a special concurrency control component is built in (i.e., internal) for each SDE, or the community provides a variety of ECC components to choose from and system-specific mediators are constructed to compensate for any interface mismatch with regard to the selected ECC. In theory, the latter should involve less total work, at least if the mediators tend to be small compared to TMS and ECC. In addition, an ECC component would be more robust after being employed across a range of SDEs.

Figure 3 presents an abstract representation of the ECC component interface as it fits in to a larger architecture. Through ECC's interface, a TMS (including, for the purposes of this discussion, any ECC/TMS mediator) can *Begin* transactions, *Lock* data items, and *Commit* or *Abort* transactions. Since data repository services are external to ECC there should be no restriction on the information stored. However, ECC does assume that each data item can be uniquely referenced by an identifier. This assumption is reasonable considering the increasing popularity of object-oriented databases; it also holds true for relational databases with unique key fields, and file systems with unique file pathnames.

ECC manages all transactions created by TMS and allows TMS to define the composition and dependencies of transactions. Each transaction is either a "top level" transaction or has at least one parent. Through composition, TMS can create nested transaction hierarchies of arbitrary depth and breadth. Transactions may have dependencies on other transactions, such as commit and abort dependencies. If transaction

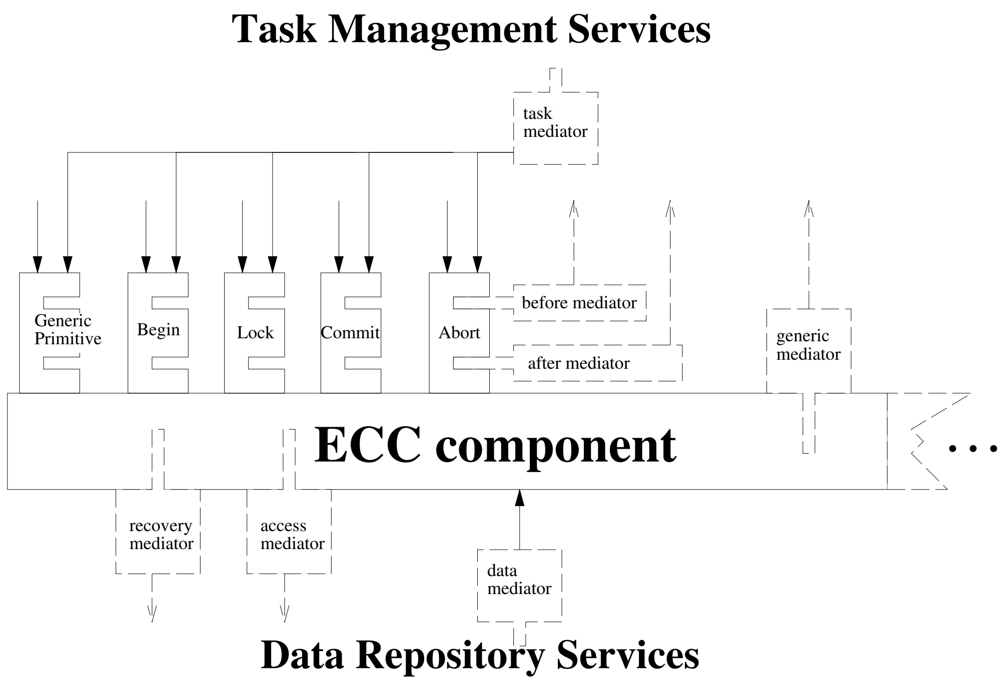

Figure 3: ECC component architecture

T1 has a *commit dependency* upon transaction T2, then T1 can't commit until T2 does. If T1 has an *abort dependency* upon T2, then T1 must abort if T2 aborts. TMS sets commit dependencies to group individual transactions into atomic units.

ECC defines its interface using *mediators*, fragments of special code needed to integrate ECC with a particular TMS. Each ECC primitive should have two mediators associated with it, namely *before* and *after*. When the primitive is requested by TMS, these special code fragments are executed before ECC services the request, and after. Mediators can override a primitive request, in effect denying the operation to take place. Other mediators, such as the `access mediator`, allow ECC to interact more closely with data repository services. For example, if ECC is requested to acquire a lock on a sub-item of a larger set of data items, the mediator can request intention locks on all ancestors (as in the Orion system [12]). During recovery, the `recovery mediator` communicates with data repository services to restore the data appropriately, thus abstracting ECC from any one particular data representation.

The architecture in Figure 3 is open-ended and flexible, allowing for several different types of mediators. The closer the ECC and TMS interfaces match, the less need there is for mediators; an exact match would require none. New primitives can be added for a specific TMS, each with appropriate mediators. Task mediators can work closely with TMS, translating actions at the task level into the required ECC operations; this is most appropriate for a TMS that has no notion of transactions, where transactional units must be retrofitted to the existing facilities.

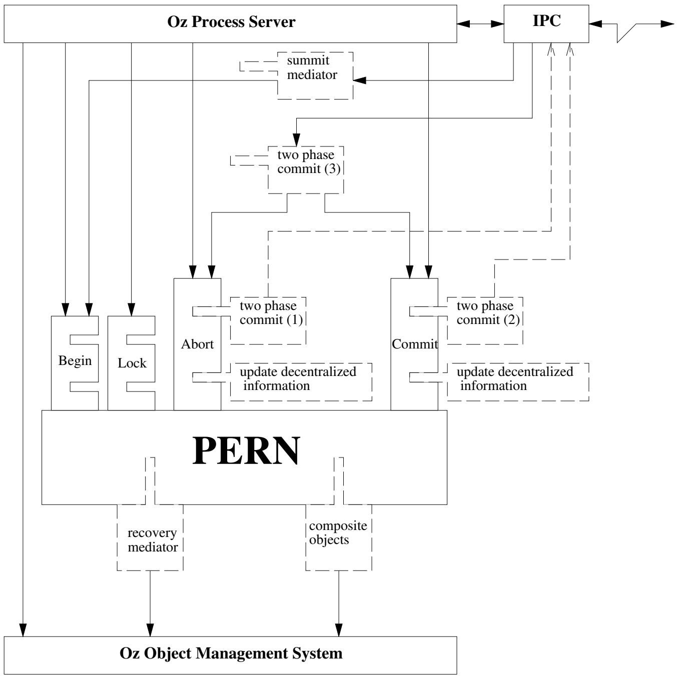

Figure 4: Interfacing of PERN with OZ

An example ECC component

PERN was created as part of a general strategy to componentize the existing MARVEL 3.1 system. Process-centered environment support for such componentization is described briefly in [15]. The transaction manager from MARVEL was isolated and reengineered as a separate component. First, all references to MARVEL rules and rule chaining was eliminated from within the transaction manager; this effectively severed the link with MARVEL's TMS. Second, external lock tables (i.e., separate from the data) and a generic recovery mechanism were implemented, severing ties with MARVEL's data repository services. Finally, the mediator architecture was established, allowing special purpose code to be separate from PERN and a TMS.

The two architectures in Figures 4 and 7 show how we integrated PERN with OZ and ProcessWEAVER. It is important to note that the basic architecture is the same. In both cases, special purpose mediators (drawn in dashed boxes) were written to attach PERN to the specific SDEs. In the following sections we discuss the details of these experiments. The OZ experiment builds PERN into the OZ runtime executables through direct linking. The ProcessWEAVER experiment utilizes remote procedure calls (RPC) – here, the mediator is the boilerplate RPC code that provides the standard client/server stub interface.

4 Experiment 1: Oz

Oz [5] is a multi-site, *decentralized* process centered environment. The Oz process server is the TMS component of the Oz architecture and defines a three-level hierarchy of nested contexts. The lowest level, the *activity* level, is where Oz interfaces to actual tools (e.g., through envelopes [13]). The *process-step* level encapsulates activities with prerequisites and immediate consequences (if any) of tool invocations as determined by a process. The *task* level is a set of logically related process steps with the combined set of their prerequisites and consequences. An Oz environment is a collection of multiple Oz *sites*, each containing its own process server that enacts the local process at that site. Each site communicates with other sites through an interprocess communication layer (IPC). In Oz, each process-step corresponds to a transaction.

The decentralized process modeling aspects of Oz, discussed in [6], allow *treaties* to be formed between sites (in pairwise fashion) to define the specific collaboration that may occur between those sites. A treaty between SiteA and SiteB defines a common sub-process and sub-schema (a “unit of commonality”) that becomes part of each site's local process. This “unit” can be on any of the levels discussed above – activity, process-step, or task – and represents those process fragments that involve both local processes. The enaction of such a shared sub-process by multiple sites is called the *summit enaction protocol* or summit, for short. The summit protocol emphasizes site autonomy and the use of treaties to relax autonomy to the degree (and only the degree) that each local site wishes to collaborate with other sites.

We now present a summit example to clarify the concurrency control issues we needed to address for Oz. Consider the set of tasks in Figure 5. Three development groups using Oz sites **SE1**, **SE2**, and **SE3** are responsible for three disjoint modules, **M1**, **M2**, and **M3**. Each Oz site has a process server that enacts its local process and each site has already agreed to the multi-site treaty depicted here. When a change request for **L** is made at **SE2**, each of the sites must pre-approve the change. At site **SE1**, the manager *must* be notified of the proposed change. At site **SE3**, the change request should be analyzed with respect to **M3**. Once the change request is accepted at each site and propagated with respect to each site's local process, the actual modifications are made to the respective modules and an inspection phase is performed. Once the inspection is complete, a unit test of each module is performed at its home site, ultimately resulting in a final integration test of all modules.

The success and failure of a particular summit can be organized into the following four cases:

	Process	Concurrency
Failure	Pre-requisites not met	Conflicting data access
Success	Goals achieved	All updates can be safely committed

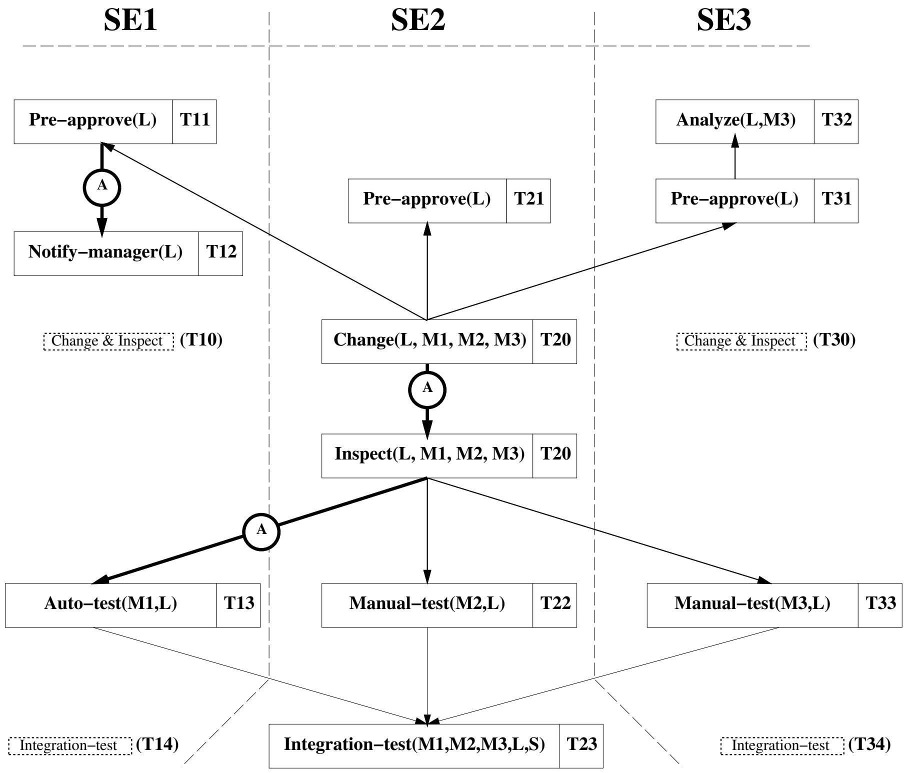

Figure 5: Summit Example

OZ is responsible for determining the status of a summit based upon its process knowledge (the left column), but the concurrency control mechanism is responsible for the right column. A concurrency control failure can occur at any time during a summit, since multiple users can be executing multiple tasks in OZ, independent of any simultaneously executing summits. To prevent the system from being left in an "inconsistent state" when a concurrency failure interrupts a summit, the concurrency control mechanism restores the system to a "consistent state". Since this mechanism is unaware of the semantics of the data it manages, it must rely on OZ to define what it means to be consistent.

The OZ process server defines consistency based on atomic units, which must be enacted in their entirety or not at all. The Process Modeling Language (PML) of OZ allows one to define atomic units by composing individual process steps. In our example in Figure 5, the process at site **SE2** has specified that the sequence of three process steps from **Change** to **Auto-test** be atomic. This atomic unit is spread across all three sites since it acts on data from all sites. A second atomic unit is local to **SE1**, from **Pre-approve** to **Notify-manager**. Here, one "inconsistent state" is the situation whereby the change is incorporated into **SE1** and **SE2**, but not **SE3**. Since each process-step corresponds to a transaction, OZ creates the atomic units by setting appropriate dependencies between transactions.

The Marvel system [7] (from which Oz evolved) already provided support for enforcing local atomic units within individual sites (Marvel supported only single site environments). We considered two possible approaches to providing transactional semantics for summits. The first required building a special purpose transaction manager for Oz. The better alternative – and the one carried out in this experiment – attached a copy of the Pern component to each Oz process server to fulfill the requirements of executing a summit. The mediators for each Pern component were important since we didn't want to modify either Oz or Pern during their integration. There are non-trivial differences in orientation between Pern and Oz; in particular, Oz is a decentralized system and Pern is, in effect, a centralized transaction manager. Pern's mediators maintain the necessary decentralized information that allows a collection of Pern instances working together to fulfill the requirements of executing a summit.

The problem of providing transactional semantics for summits can be divided into two parts, synchronizing the updates by a process step across several sites (horizontal) and guaranteeing atomic units across process steps (vertical). Any solution for summit transactions must not greatly affect the processing of normal, independent transactions at each site. We now describe how Pern, as an example of an ECC component, was able to provide the appropriate concurrency control mechanism for enforcing the atomicity of decentralized units.

Summit Transaction Protocol

In Oz, the default is for each site to have complete autonomy with respect to all other sites, and any relaxation of autonomy must be agreed to *a priori* by all parties (in a treaty). Each Pern instance agrees to this separation. For example, the Pern instance at site **SE1**, **PERN1**, is fully responsible for the data at that site. **PERN1** can determine whether to permit or deny data access from local process steps. When a site is involved in a summit, however, it must necessarily give up some autonomy in order to collaborate with other sites. The site which starts a summit is called the *coordinator;* the other participating sites are called *participant* sites. The summit transaction protocol follows these rules:

1. The coordinator cannot commit its transaction for a summit process step until all atomicity requirements have been fulfilled at all sites. If a summit step has an atomicity requirement with respect to another summit process step, then the transaction is simply continued across both steps. This is the only time when one transaction is "owned" by more than one process step.
2. A summit process step must access data from multiple sites, therefore one local transaction is needed at each site for each summit process step. These are the transactions in parentheses in Figure 5.

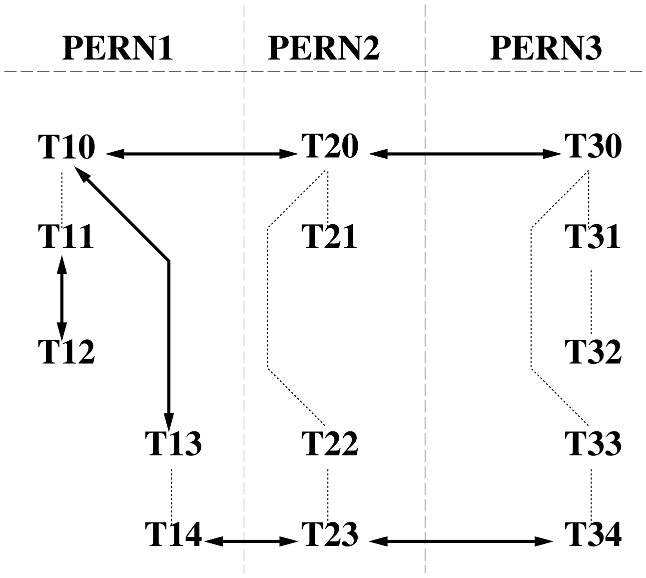

Figure 6: Transaction dependencies in summit example

3. A participant site cannot commit the local transaction it created for the summit until the coordinator directs it to.

The mediators in Figure 4, shown for a Pern instance at a single site, maintain the appropriate inter-site transaction dependencies. Figure 6 shows the dependencies between the transactions for our summit example of Figure 5. The solid arrow lines show commit dependencies. *Inter*-site dependencies (for example, from T20 to T10) are maintained at each affected site, while *intra*-site dependencies (for example, from T10 to T13) are local to one site.

When the summit process step **Change(L, M1, M2, M3)** is requested at site **SE2**, the coordinating site begins transaction T20 and notifies sites **SE1** and **SE3**. These sites receive this request, through their IPC layer, and execute the `summit mediator` to create transactions T10 and T30 that act on behalf of T20. Since T10 and T30 are transactions at participant sites, they will not be able to commit until **SE2** directs them to. As defined in this summit example, T20 will be able to commit once site **SE1** has successfully executed the **Auto-test** process step. As **SE2** attempts to commit T20, the `two phase commit` mediator (number 2) is invoked (since it is the *before* mediator for commit). This mediator initiates a two phase commit protocol with all sites upon which T20 has a dependency. It sends messages to the remote sites through the IPC layer, and the `two phase commit` mediators (number 3) at sites **SE1** and **SE3** receive the request and commit the appropriate transactions, T10 and T30.

Since Oz is a research prototype being developed in our lab, it is easy to argue that the Oz/Pern integration was not an entirely fair experiment. So as a second test

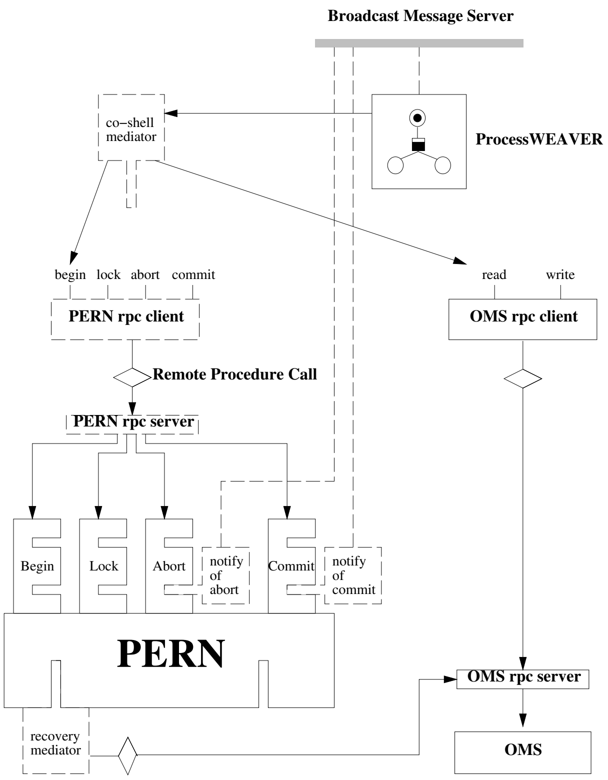

Figure 7: Interfacing of PERN with ProcessWEAVER

case, we chose a commercial product – ProcessWEAVER – where we had no special knowledge of the system nor access to the source code.

5 Experiment 2: ProcessWEAVER

ProcessWEAVER [11] is a set of utilities that adds process support capability to Unix-based toolkits. ProcessWEAVER utilities communicate with each other via a Broadcast Message Server (BMS). These utilities support modeling and enactment of process models. A process model in ProcessWEAVER has two levels, the topmost being an *activity* hierarchy, which recursively refines activities into sub-activities. The second level contains a set of *cooperative procedures* (CPs); each activity and sub-activity is implemented by a CP. A CP is a transition net (a form of Petri net) consisting of places, tokens, and transitions. A CP maintains its state by marking some of its places by tokens. A transition has a set of input and output places associated with it. When all the input places for a transition are marked by tokens, the transition fires, and the tokens are moved to the output places. Each transition can have *co-shell* (ProcessWEAVER's shell-like language, analogous to OZ's envelopes) code that is executed (like a subroutine) when the transition fires. To map the

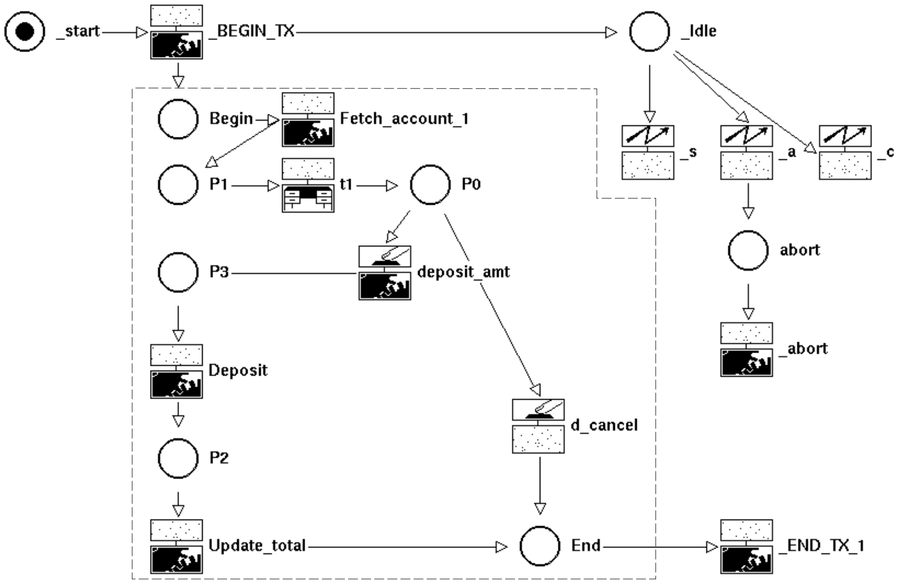

Figure 8: Augmented Deposit CP

ProcessWEAVER terminology onto the terms from Section 2, a cooperative procedure is a task, the transitions are activities, a transition's input places are an activity's prerequisite, and the output places are an activity's consequences. ProcessWEAVER doesn't assume any particular data repository services; the desired repositories – file system or database – are accessed through co-shell code.

Petri nets are excellent for explicitly modeling the synchronization of concurrent activities of cooperating agents but there is no underlying mechanism for treating conflicting actions of concurrent, independent agents that incidentally access the same data. This distinction between concurrency through synchronization and concurrency control reveals a need for transactions in ProcessWEAVER. Once again, we needed to decide which component is responsible for concurrency control. Since ProcessWEAVER was a commercial product, however, it had to be used "as is" without any source code modifications.

Instead, we implemented a 200 line *co-shell* library (see [16] for details) that defined a mediator for ProcessWEAVER to communicate with PERN. ProcessWEAVER was thus parameterized so that only individual CPs had knowledge of transactions, not the rest of the process modeling facilities and framework. This mediator is typical of what would be needed to integrate PERN into an SDE that provides an extension language, in this case co-shell, that makes it possible to directly augment its task management services. In our experiment, ProcessWEAVER is the TMS component and PERN becomes a physically separate utility working in conjunction with the other ProcessWEAVER components to provide the necessary transaction services.

In [16], we discussed several alternatives for integrating transactions with Process-

WEAVER, ultimately deciding upon *augmenting* a cooperative procedure with additional places and transitions. The CP in Figure 8, for example, is the result after manually augmenting the original transition net contained within the dotted lines; this requires only three additional places and six transitions. We envision that a preprocessor could be built to generate these augmented CPs with some user guidance, for example, in determining the appropriate starting and ending places for a CP. These starting and ending places determine the logical unit for which the transaction is responsible.

This particular CP retrieves the balance for **account_1** and prompts the user for an amount to deposit into the account. We now step through the execution of the augmented CP. A transaction is started at transition **_BEGIN_TX** through its co-shell action:

```
{ Begin Transaction }
$tid = Begin (NOCOMMIT, NOABORT, TOP, ROLLBACK);
```

This issues a remote procedure call to PERN that creates a top-level transaction that can be rolled-back and has no commit or abort dependencies on any other transaction. This transition activates the original starting place for the CP, **Begin**, and also moves into the **_Idle** place. Note that the original CP (within the dashed lines) continues its normal actions. The **_Idle** place monitors the newly created transaction and has three separate transitions that are activated if the transaction commits (**_c**), aborts (**_a**) or suspends (**_s**). The condition for **_a**, for example, fires if a message appears on the BMS of type "`pern_abort $tid`". If this transaction ever aborts, the CP moves into the **abort** place, which exists to allow other CPs to take action in response to this transaction abort.

For this experiment, we constructed a miniature OMS that allowed a CP to read and write the attributes of an object. This OMS became the data repository for this particular ProcessWEAVER application. Transition **Fetch_account_1**, for example, executes the following co-shell code:

```
{ Access account-1 (through an object identifier $account_1).}
{ First we request access from PERN, then we get the data from the OMS.}
$action = Access ($tid, $account_1, X);
$v1 = Read_Attribute($account_1, balance);
```

The `Access` function issues a remote procedure call to PERN, attempting to set a lock on the given object with the given lock mode (`X` means Exclusive). Once it succeeds, the balance information is retrieved from the OMS by means of another remote procedure call that returns the value of the "balance" attribute for the appropriate object.

At transition **Deposit_amt**, the user determines the deposit amount to be added to the balance `$v1` (in **Deposit**) and in transition **Update_total**, the new balance is written back to the OMS and the CP moves into the final **End** place. At this point, the new transition **_END_TX_1** commits the transaction by issuing a remote procedure call to PERN. The *after* mediator for `Commit` sends a message to the BMS of type "`pern_commit $tid`" The transition **_c** will retrieve this message, clearing out the token at **_Idle**, and the CP will complete successfully. Note that if the `Access` function had failed, the *after* mediator for `Abort` would have sent out a PERN_ABORT message, thus moving the CP into the **abort** place.

6 Lessons Learned

We were pleasantly surprised at how easy it was to support decentralized transaction management from multiple instances of a centralized transaction manager; the mediator architecture made this possible. For example, there was no need to implement a deadlock prevention scheme since the mediators were able to reuse the deadlock logic already inherent in the OZ process servers. Not all problems have been solved, however, since the transaction mechanism has not yet been fully integrated with OZ's Cache Manager, which caches remote objects and/or files locally during, and perhaps across, summits. Completing this integration will be interesting since both PERN and the Cache Manager are external to TMS. The OZ system is roughly composed of 200,000 lines of C code; the PERN component has 13,000 lines of C code while the PERN/OZ mediators required 1,800 lines of code. Since OZ already had "hooks" for transactions, it was relatively simple to integrate PERN.

The experiment with ProcessWEAVER was, admittedly, a preliminary one, as can be seen by the trivial example from Section 5. This was a necessary first step, however, since ProcessWEAVER had no notion of concurrency control at all. Now that we have simple notions of transactions implemented in ProcessWEAVER, we are ready to move on to realistic software development processes. Discussions with the developers of ProcessWEAVER have revealed areas where the performance of the mediators could be increased through gateways between RPC and ProcessWEAVER's BMS.

It is interesting to note that if OZ had no notion of transactions, the integration of PERN with OZ and ProcessWEAVER would probably have been more similar. For example, we could have incorporated PERN into OZ's enveloping mechanism so that when OZ executes an activity through a envelope, PERN would intervene and first lock all objects used during the activity. When the activity completes, PERN would release the locks. This implementation would map each activity to one transaction. Atomic units could be constructed by adding "dummy" process-steps, having appropriate prerequisites and consequences to begin and end an atomic unit. The mediators would have to change, but the underlying architecture would remain valid.

7 Contributions and Future Work

Integrating pre-existing, independent components is not simple. The architecture presented in this paper shows how mediators can aid this effort. In the absence of standard interfaces for components, mediators can provide the necessary veneer to allow components to work together.

The main contributions of this paper are:

- Requirements for a concurrency control component separate from task management services in SDEs.
- A general architecture for integrating task management services and an external concurrency control component.
- A sample external concurrency control component, Pern.
- Two successful experiments in integrating an external concurrency control component into existing systems.

There are many avenues for future work. The most pressing problem is dealing with environment frameworks and object management systems (in the guise of data repository services) that already have their own concurrency control mechanisms. It is not clear how an ECC component can provide cooperative transaction capabilities if the underlying data repository has hardwired support for conventional transactions, or the SDE builds in the checkout paradigm. More than likely, a sophisticated mediator would have to override at least part of this functionality, perhaps while exploiting the existing code where this functionality is invoked. In this paper we have omitted most references to the crash recovery (as opposed to concurrency control recovery) aspects of ECC, but there are serious issues regarding recovery in tandem with cooperative transaction management.

Acknowledgements

Christer Fernström of Cap Gemini Innovation provided assistance in obtaining a ProcessWEAVER license and useful feedback on our preliminary experiment. Israel Ben-Shaul collaborated with the authors on devising the transactional semantics for Oz summits, and along with Naser Barghouti was instrumental in developing the original cooperative transaction management facilities of Marvel (the distinctions between Pern's and Marvel's transaction models are outlined in [14]). Several members of the Programming Systems Laboratory are working on developing other environment framework components, including a multi-lingual, multi-modal process server [19, 24].

References

[1] Malcolm Atkinson, Francois Bancilhon, David DeWitt, Klaus Dittrich, David Maier, and Stanley Zdonik. The object-oriented database system manifesto. In Won Kim, Jean-Marie Nicolas, and Shojiro Nishio, editors, *1st International Conference on Deductive and Object-Oriented Databases*, pages 40–57, Kyoto, Japan, December 1989. Elsevier Science.

[2] Naser S. Barghouti. *Concurrency Control in Rule-Based Software Development Environments*. PhD thesis, Columbia University, February 1992. CUCS-001-92.

[3] Naser S. Barghouti and Gail E. Kaiser. Concurrency control in advanced database applications. *ACM Computing Surveys*, 23(3):269–317, September 1991.

[4] Don Batory and Sean O'Malley. The design and implementation of hierarchical software systems with reusable components. *ACM Transactions on Software Engineering and Methodology*, October 1992.

[5] Israel Z. Ben-Shaul. Oz: A decentralized process centered environment. CUCS-011-93, Columbia University Department of Computer Science, April 1993. PhD Thesis Proposal.

[6] Israel Z. Ben-Shaul and Gail E. Kaiser. A paradigm for decentralized process modeling and its realization in the OZ environment. In *16th International Conference on Software Engineering*, pages 179–188, Sorrento, Italy, May 1994. IEEE Computer Society Press.

[7] Israel Z. Ben-Shaul, Gail E. Kaiser, and George T. Heineman. An architecture for multi-user software development environments. *Computing Systems, The Journal of the USENIX Association*, 6(2):65–103, Spring 1993.

[8] Prasun Dewan, editor. *Special Issue on Collaborative Software*, volume 6:2 of *Computing Systems, The Journal of the USENIX Association*. University of California Press, Spring 1993.

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

[10] Jeffrey L. Eppinger, Lily B. Mummert, and Alfred Z. Spector, editors. *Camelot and Avalon A Distributed Transaction Facility*. Morgan Kaufman, San Mateo CA, 1991.

[11] Christer Fernström. PROCESS WEAVER: Adding process support to UNIX. In *2nd International Conference on the Software Process: Continuous Software Process Improvement*, pages 12–26, Berlin, Germany, February 1993. IEEE Computer Society Press.

[12] Jorge F. Garza and Won Kim. Transaction management in an object-oriented database system. In *SIGMOD International Conference on Data Management*, pages 37–45, Chicago IL, June 1988. Special issue of *SIGMOD Record*, 17(3), September 1988.

[13] 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.

[14] George T. Heineman. A transaction manager component for cooperative transaction models. CUCS-017-93, Columbia University Department of Computer Science, July 1993. PhD Thesis Proposal.

[15] George T. Heineman and Gail E. Kaiser. Incremental process support for code reengineering (Experience Report). In *International Conference on Software Maintenance*, Victoria BC, Canada, September 1994. In press.

[16] George T. Heineman and Gail E. Kaiser. Integrating a transaction manager component with ProcessWEAVER. Technical Report CUCS-012-94, Columbia University Department of Computer Science, May 1994.

[17] Gail E. Kaiser. Cooperative transactions for multi-user environments. In Won Kim, editor, *Modern Database Systems: The Object Model, Interoperability, and Beyond*. ACM Press, New York NY, 1994. In press.

[18] Erich Neuhold and Michael Stonebraker (editors). Future directions in DBMS research. *SIGMOD Record*, 18(1):17–26, March 1989.

[19] Steven S. Popovich. Rule-based process servers for software development environments. In *1992 Centre for Advanced Studies Conference*, volume I, pages 477–497, Toronto ON, Canada, November 1992. IBM Canada Ltd. Laboratory.

[20] Richard Rashid, Avadis Tevanian, Michael Young, David Golub, Robert Baron, David Black, William Bolosky, and Jonathan Chew. Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures. In *2nd International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 31–39, Palo Alto CA, October 1987. Special issue of *SIGPLAN Notices*, 22(10), October 1987.

[21] Lawrence A. Rowe. Report on the 1989 Software CAD Databases Workshop. In Gerhard Ritter, editor, *11th World Computer Conference IFIP Congress '89*, pages 719–725, San Francisco CA, August 1989. Elsevier Science.

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

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

[24] 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*, Monterey CA, September 1994. In press.