

An Architecture for Integrating Concurrency Control into Environment Frameworks

George T. Heineman Gail E. Kaiser
Department of Computer Science
Columbia University
New York, NY 10027

Abstract

Layered and componentized systems promise substantial benefits from dividing responsibilities, but it is still unresolved how to construct a system from *pre-existing, independently developed* pieces. Good solutions to this problem, in general or for specific classes of components, should reduce duplicate implementation efforts and promote reuse of large scale subsystems. We tackle the domain of software development environments and present an architecture for retrofitting external concurrency control components onto existing environment frameworks. We describe a sample ECC component, explain how we added concurrency control to a commercial product that had none, and briefly sketch how we replaced the concurrency control mechanism of a research system.

1 Introduction

Multi-user software development environment frameworks (SDEs) need a concurrency control mechanism to detect and resolve conflicts since more than one user task may attempt to access the same data in incompatible ways. Many SDEs provide some variant of the file checkout paradigm, typically coupled with versioning, while others incorporate a database system with conventional transactions. SDE applications sometimes require “cooperative transactions” (see [2] for a survey of collaborative concurrency control mechanisms). Cooperative transactions make it possible to guarantee atomicity (rollback of an entire atomic unit if it cannot be completed) and possible to enforce serializability (the ap-

pearance that only one user task is accessing the data at a time). In addition, cooperative transactions can exploit application-specific semantics to resolve concurrency conflicts, often to enhance concurrency and enable collaborative work.

Some SDEs, however, 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 with manual synchronization (e.g., passing the “floor” in multi-user editors [7]), or because the designers assumed that some external facility would provide concurrency control (as for ProcessWEAVER [10]). In this paper we present an external concurrency control (ECC) architecture with our main example drawn from the latter category, where concurrency control was left for another component.

Componentized systems offer potential benefits by dividing the technical and economic responsibilities for providing services; in the case of SDEs, the ECMA/NIST Reference Model [17] specifies user interface, task management, object management, and communication components, as well as individual tools. It is still unclear, however, how one can and should integrate SDE components together to produce a coherent, usable system. We do not attempt to address the entirety of this very large problem: we are concerned primarily with architectures for integrating the task management services (TMS) component of an SDE with ECC; we also discuss integration of ECC with object management services.

In general, 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. Thus we must supply an interface to the ECC from TMS (i.e., entry points) that would cover the plausible range of ECC functionality, along with an interface from the ECC to TMS (i.e., callbacks) that provides a means for ECC to obtain the semantic information required to support

that functionality. In particular, we took the abstract concept of mediators and made them concrete. We devised a generic ECC component with specific entry and callback points that could be directly invoked, or responded to, by a new system constructed around the component. More significantly, they are designed to be exploited by a mediator (or set of mediators) between ECC and a pre-existing TMS (above) and between ECC and a pre-existing OMS (below).

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 components, with the rest of the system implemented mostly from scratch to take advantage of these components (the approach taken for PCTE [19] and many systems in other domains, such as Mach [9], Camelot [8], and database applications); or (2) building-block kits, where sets of components that can be mixed and matched are designed and implemented together (e.g., Genesis and Avoca [3]). The most significant difference compared to the above technologies is that we deal with 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. Under such circumstances, we argue, integration is possible by external mediators that overcome the mismatch between components (see [18] for a related approach that assumes components with extension languages).

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 to an ECC component. Then we sketch our prototype ECC component, PERN [14]. We demonstrate how we applied our approach to a TMS component without its own concurrency control mechanism. This is followed by a sketch of a later study where we replaced a system’s existing concurrency control mechanism, and a discussion of lessons learned. We conclude by summarizing our contributions and directions for future work. Since our focus in this paper is on *interfacing* TMS and ECC mechanisms, we do not discuss the cooperative transaction *functionality* supported by our component (for details, see [14]); for simplicity, we assume conventional atomic and serializable transactions.

2 Requirements

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

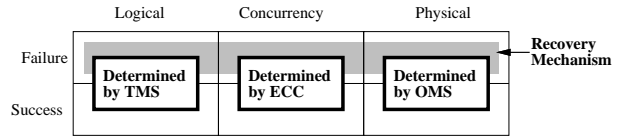


Figure 1: Division between logic and concurrency

- *Logical units* that could 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).
- *Recovery* of 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 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). Object management services (OMS) is responsible for detecting any failure to store data. Ideally, the mapping from transactions to tasks would permit the same recovery mechanism to be used for all three components.

When the concurrency control mechanism is external to TMS, there are several additional requirements:

- 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 run-time 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 code, or “glue”, should be relatively small, compared to the code

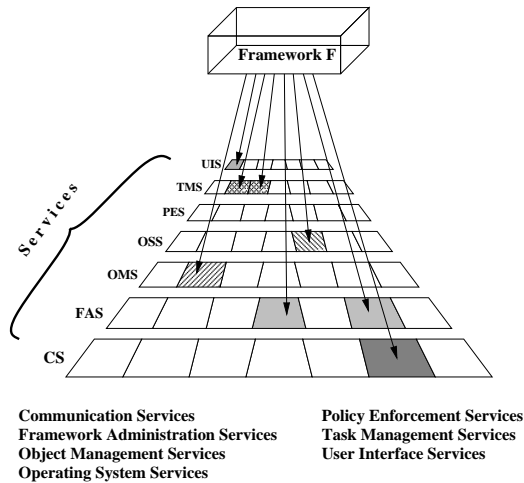


Figure 2: ECMA/NIST Reference Model [17]

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, most notably the object management system. There may need to be some means, however, 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 ECMA/NIST Reference Model [17]. This model targets particular services of an SDE and groups them by functionality. Figure 2 shows the seven major service groupings and how a framework F provides a particular set of services from these groups. Object management services store data in any combination of file system and database(s) and manage access to the data, possibly using a transaction service. Task

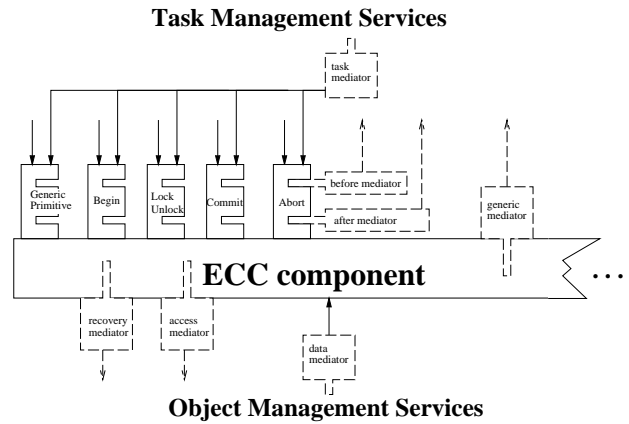


Figure 3: ECC component architecture

management services¹ (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. Unfortunately, some existing SDEs provide no particular notion of tasks except (degeneratively) as 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 to the various Reference 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 Software Development Environment as part of its OMS, 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 into a larger architecture. Through ECC's interface, TMS (including, for the purposes of this discussion, any ECC/TMS mediator) can *Begin* transactions, *Lock/Unlock* data items, and *Commit* or *Abort* transactions. Since object management services are external to ECC there should be no restriction on the information stored. However, ECC does assume that each data item can

¹The newest version of the Reference Model uses the term process management, whereas previously this was called task management.

be uniquely referenced. 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 (or TMS/ECC mediators) 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 [6]. If transaction T_1 has a commit dependency upon T_2 , then T_1 can’t commit until T_2 does. If T_1 has an abort dependency upon T_2 , then T_1 must abort if T_2 aborts. TMS use dependencies to group individual transactions into units.

ECC defines its interface using *mediators*, fragments of special code needed to integrate ECC with a particular TMS. Each ECC primitive has two mediators associated with it, namely *before* and *after*. When the primitive is requested by TMS, these special code fragments are executed before (and after) ECC services the request. 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 object management 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 Orion [11]). During recovery, the **recovery mediator** communicates with object management 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; this may require modifying the ECC or switching to an existing, more sophisticated ECC. **task mediators** can work closely with TMS, translating actions at the task level into the required ECC operations; this is most appropriate for retrofitting transactional units onto a TMS that has no notion of transactions.

An example ECC component

We created the PERN component as part of a general strategy to componentize the existing MARVEL system [5]. The transaction manager from MARVEL was

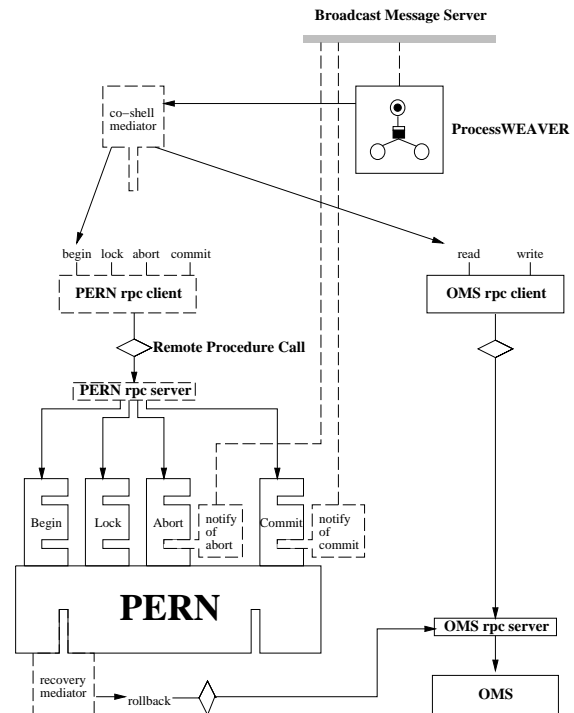


Figure 4: Interfacing of PERN with ProcessWEAVER

isolated and reengineered as a separate component. First, references to MARVEL’s TMS were eliminated from within the transaction manager, so PERN no longer made assumptions about the structure of user tasks. PERN was no longer consulted during the instantiation of tasks or their scheduling and execution. This enables PERN to manage transactions for a wide range of SDEs.

Second, external lock tables and a generic recovery mechanism were implemented in PERN, severing ties with MARVEL’s OMS. Separating the data items from the locks being held was a big step towards allowing PERN to be used with arbitrary data repositories. In addition, the generic recovery mechanism removes assumptions about the operations that can be performed within a transaction.

Finally, the mediator architecture was established, allowing interface code to be written separately from PERN. These mediators, drawn in dashed boxes in Figure 3, make the necessary connections to integrate PERN with TMS and OMS. The two architectures in Figures 4 and 6 show how we integrated PERN with Oz and ProcessWEAVER. It is important to note that the basic architectures are the same: in both cases mediators were written to attach PERN to the specific SDEs.

4 ProcessWEAVER

ProcessWEAVER [10] is a set of utilities that adds process support capability to Unix-based toolkits, including definition and execution of process models [12, 16]. ProcessWEAVER executables communicate with each other through a Broadcast Message Server (BMS).

A process model in ProcessWEAVER has two levels, the topmost being an *activity* hierarchy that recursively refines activities into sub-activities. The second level contains a set of *cooperative procedures* (CPs) each of which implements an activity or sub-activity. A CP is a transition net (a form of Petri net) consisting of places, tokens, and transitions; its state is determined by the assignment of tokens to places. Each transition has a set of input and output places associated with it. When all the input places for a transition contain tokens, the transition fires and the tokens are moved to the output places. Each transition can have *co-shell* (ProcessWEAVER’s shell-like language) code that is executed (as a subroutine) when the transition fires. ProcessWEAVER doesn’t have an object management service; 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. Since ProcessWEAVER is a commercial product without source code availability, it could not be modified – so we provided transaction support by integrating PERN, externally.

We implemented a 200 line co-shell library (see [15] 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 execution 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 demonstration, ProcessWEAVER provides the TMS component and PERN becomes a physically separate utility working in conjunction with the other ProcessWEAVER utilities.

In [15] we discussed several alternatives for adding transactions to ProcessWEAVER, ultimately deciding upon *augmenting* a cooperative procedure with

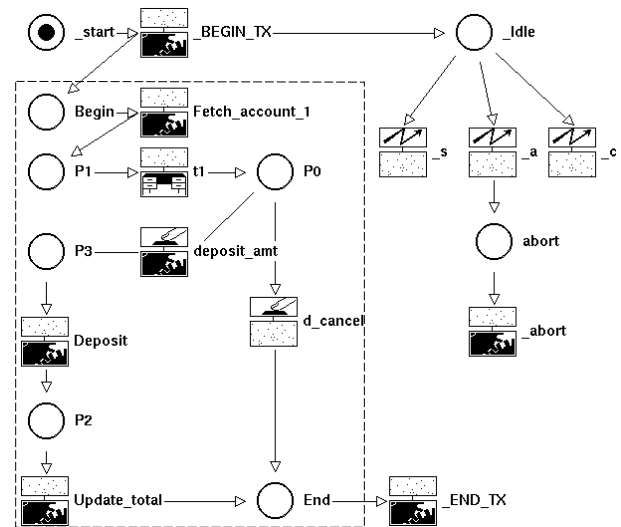


Figure 5: Augmented Deposit CP

additional places and transitions. The CP in Figure 5, 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 pre-processor 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 that 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 – a conventional transaction processing application. We now step through the execution of the augmented CP. A transaction is started at transition **_BEGIN_TX** through its co-shell action²:

```
{ 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

²Co-shell variables start with a \$ character. Arbitrary string values, such as **TOP**, don't need to be quoted. The **Begin()**, **Access()**, and **Read_attribute()** functions are described in [15].

(**_a**) or suspends (**_s**). The condition for **_a**, for example, is satisfied if a message of type “**pern_abort \$tid**” appears on the BMS. If transaction **\$tid** 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 demonstration, we constructed a miniature OMS that allowed a CP to read and write the attributes of an object. Transition **Fetch_account_1**, for example, executes the following co-shell code:

```
{ Access account-1 (through an object identifier $account_1.)
{ Request access from PERN, then get 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 account object in **eXclusive** mode. If this request succeeds, **Read_attribute** issues a remote procedure call to the OMS to get the balance for **\$account_1**.

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** 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.

5 Oz

Oz [4] 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 (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 executes the local process at that site. Each site communicates with other sites through an interprocess communication layer (IPC).

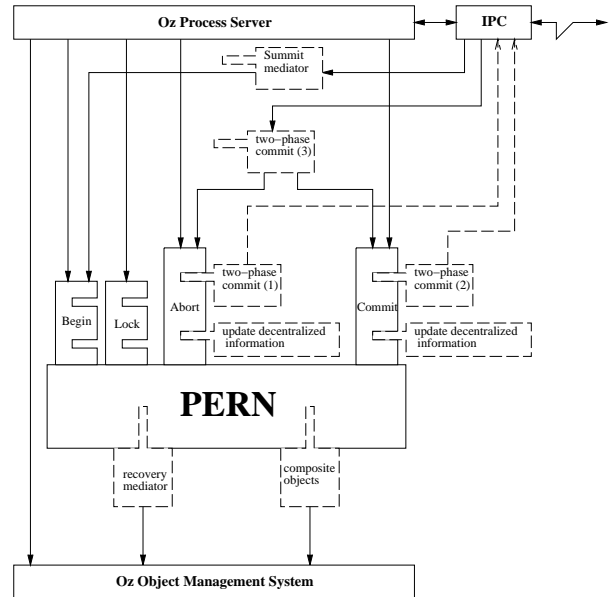


Figure 6: Interfacing PERN with Oz

The decentralized process modeling aspects of Oz allow *Treaties* to be formed between sites to define the specific collaboration that may occur among those sites. A Treaty between sites SiteA, SiteB, and SiteC, for example, defines a common sub-process, one or more process-steps, that becomes part of each site’s local process and represents those fragments that involve all three local processes. The execution of such a shared sub-process by multiple sites is called the *Summit execution protocol* or Summit, for short. This international alliance metaphor emphasizes site autonomy, using Treaties to relax autonomy to the degree (and only the degree) that each local site wishes to collaborate with other sites.

Multiple users can execute tasks in each of the local process servers, independent of any simultaneously executing Summits. To prevent an environment from being left in an “inconsistent state” when a concurrency failure interrupts a Summit, the concurrency control mechanism restores the environment to a “consistent state”. Oz defines consistency by grouping individual process-steps together into atomic units that must be executed in their entirety or not at all. Since each process-step corresponds to a transaction, Oz creates these units by setting dependencies between transactions.

The MARVEL system, from which Oz evolved, already provided support for guaranteeing consistency within a site, but MARVEL supported only single-site environments and had no notion of Summits. We considered two possible approaches to providing transaction semantics for Summits. The first required build-

ing a special purpose transaction manager for OZ. The better alternative – and the one implemented – attached a copy of the PERN component to each OZ process server. PERN’s mediators maintain the necessary decentralized information that allows a collection of PERN instances to work together to fulfill the requirements of executing a Summit.

Transaction semantics for Summits can be divided into two parts: synchronizing the updates by a process-step across several sites (horizontal) and guaranteeing consistency across process-steps (vertical). The approach to Summit transactions must not greatly affect the processing of local, independent transactions at each site. Recall that each site, by default, has complete autonomy with respect to all other sites, and any relaxation of autonomy must be agreed to *a priori* by all parties. Each PERN instance, therefore, determines 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 *participants*. The Summit transaction protocol (not to be confused with the execution protocol) follows these rules:

1. A Summit process-step, by definition, accesses data from multiple sites; therefore a local transaction is needed at each site for each Summit process-step.
2. A participant site cannot commit the local transaction it created for a Summit until the coordinator directs it to.
3. The coordinator cannot commit its transaction for a Summit process-step until all atomicity requirements have been fulfilled at all sites.

The mediators in Figure 6, shown here for a PERN instance at a single site, maintain the appropriate inter-site transaction dependencies. When the coordinating site wishes to complete a Summit, the **two-phase commit** mediator (number (2) in Figure 6) is invoked (since it is the *before* mediator for commit). This mediator initiates a two-phase commit protocol with all sites involved in the Summit by sending messages to the remote sites through the IPC layer. The remote **two-phase commit** mediators (number (3) in Figure 6)) receive the request and commit their transactions, thus guaranteeing consistency at all sites involved in the Summit.

6 Lessons Learned

The ProcessWEAVER demonstration covered only very simple cases, such as the example in Section 4.

This was a necessary first step, however, since ProcessWEAVER itself has no notion of concurrency control at all. Now that we have a structure that employs ECC to support conventional transactions, we are ready to move on to cooperative transactions for software development processes.

After completing this pilot study with ProcessWEAVER we removed the existing transaction manager from OZ and substituted PERN in its place. Since PERN is descended from MARVEL’s transaction manager module, we expected an easy integration. We were pleasantly surprised at how easy it was to support decentralized transaction management using multiple instances of a centralized transaction manager; the mediator architecture made this possible and reduced the need to develop duplicate functionality. For example, there was no need to implement distributed deadlock prevention since the mediators were able to reuse the communication deadlock logic from the OZ process servers. The OZ system is composed of roughly 200,000 lines of C code; the PERN component is 13,000 lines of C while the PERN/OZ mediators are 1,800 lines.

The basic architecture in Figures 4 and 6 is the same. If OZ had not had “hooks” for transactions, such as the **consistency** annotations inherited from MARVEL [1], the details of integrating PERN with OZ and with ProcessWEAVER would also have been very similar. For example, we could have incorporated PERN via OZ’s enveloping mechanism [13] so that when OZ invoked an activity, PERN would intervene and first lock all parameters to the activity. When the activity completed, PERN would release the locks. Such an implementation would map each activity to one transaction. Multiple process-steps could be grouped into transactions by adding “dummy” process-steps, with appropriate prerequisites and consequences, to begin and end the transactions. The mediators would be different, of course, but the underlying architecture would remain the same.

7 Contributions and Future Work

Integrating pre-existing, independently developed components is far from straightforward; the architecture presented in this paper shows how mediators can aid this effort. In the absence of standard interfaces for components, particularly legacy systems, mediators can provide the necessary veneer to allow components to work together and/or be added onto “complete” systems.

Our main contributions are:

- Requirements for a concurrency control component that exists separately 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.
- Successful application of our architecture to add an external concurrency control component onto a commercial product, ProcessWEAVER, with no prior notion of concurrency control.

There are many avenues for future work. The most pressing problem is dealing with environment frameworks with their own built-in concurrency control policy that is nonetheless deemed unsuitable for some potential framework applications, such as computer-supported cooperative work. A sophisticated mediator that introduces concurrency control or overrides an existing mechanism (like conventional transactions or the checkout model) cannot operate if there is no way to insert effective entry points and callbacks to/from the mediator. Fortunately environment frameworks already provide some facility for invoking external tools, making a standard architecture with application-specific mediators plausible. In this paper we have minimized crash recovery (as opposed to rollback for concurrency control purposes), but there are complex issues regarding recovery in tandem with cooperative transactions [14].

Acknowledgements

Christer Fernström of Cap Gemini Innovation provided assistance in obtaining a ProcessWEAVER license and useful feedback on our case study. Israel Ben-Shaul collaborated with the authors on devising the transaction semantics for Oz. Several members of the Programming Systems Laboratory are working on developing other environment components, including a multi-lingual, multi-modal process engine [20], following a similar mediator-based integration architecture.

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 was 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.

References

- [1] Naser S. Barghouti. Supporting cooperation in the MARVEL process-centered SDE. In *5th ACM SIGSOFT Symposium on Software Development Environments*, pages 21–31, Tyson’s Corner VA, December 1992.
- [2] Naser S. Barghouti and Gail E. Kaiser. Concurrency control in advanced database applications. *ACM Computing Surveys*, 23(3):269–317, September 1991.
- [3] Don Batory and Sean O’Malley. The design and implementation of hierarchical software systems with reusable components. *ACM Transactions on Software Engineering and Methodology*, 1(4):355–398, October 1992.
- [4] Israel Z. Ben-Shaul and Gail E. Kaiser. A paradigm for decentralized process modeling and its realization in the OZ environment. In *16th International Conference on Software Engineering*, pages 179–188, Sorrento, Italy, May 1994.
- [5] Israel Z. Ben-Shaul, Gail E. Kaiser, and George T. Heineman. An architecture for multi-user software development environments. *Computing Systems, The Journal of the USENIX Association*, 6(2):65–103, Spring 1993.
- [6] Panos K. Chrysanthis and Krithi Ramamritham. Synthesis of Extended Transaction Models using ACTA. *ACM Transactions on Database Systems*, 19(3):450–491, September 1994.
- [7] 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.
- [8] Jeffrey L. Eppinger, Lily B. Mummert, and Alfred Z. Spector, editors. *Camelot and Avalon A Distributed Transaction Facility*. Morgan Kaufman, San Mateo CA, 1991.
- [9] Richard Rashid *et al.* 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.
- [10] Christer Fernström. ProcessWEAVER: Adding process support to UNIX. In *2nd International Conference on the Software Process: Continuous Software Process Improvement*, pages 12–26, Berlin, Germany, February 1993.

- [11] 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.
- [12] Carlo Ghezzi, editor. *9th International Software Process Workshop*, Airlie VA, October 1994. In press.
- [13] Mark A. Gisi and Gail E. Kaiser. Extending a tool integration language. In *1st International Conference on the Software Process: Manufacturing Complex Systems*, pages 218–227, Redondo Beach CA, October 1991.
- [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. Integrating a transaction manager component with ProcessWEAVER. CUCS-012-94, Columbia University Department of Computer Science, May 1994.
- [16] *2nd International Conference on the Software Process: Continuous Software Process Improvement*, Berlin, Germany, February 1993.
- [17] 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`.
- [18] Kevin J. Sullivan and David Notkin. Reconciling environment integration and component independence. In *4th ACM SIGSOFT Symposium on Software Development Environments*, pages 22–33, Irvine CA, December 1990.
- [19] Ian Thomas. PCTE interfaces: Supporting tools in software-engineering environments. *IEEE Software*, 6(6):15–23, November 1989.
- [20] Andrew Z. Tong, Gail E. Kaiser, and Steven S. Popovich. A flexible rule-chaining engine for process-based software engineering. In *9th Knowledge-Based Software Engineering Conference*, pages 79–88, Monterey CA, September 1994.