# Integrating a Transaction manager component with Process Weaver

George T. Heineman
Gail E. Kaiser

Columbia University
Department of Computer Science
500 West 120th Street
New York, NY 10027
heineman@cs.columbia.edu
TR CUCS-012-94
May 10, 1994

## Abstract

This paper details our experience integrating a transaction manager component, called PERN with Process Weaver. Process Weaver's Petri-net based approach is excellent for explicitly modeling concurrent activities of cooperating agents, but there is no underlying mechanism for treating conflicting actions of concurrent, independent agents. In addition, there is a need for advanced transaction support if we are to extend petri nets to use object management systems to store and access data. This paper shows several experiments we performed and our resulting implementation.

## 1 Introduction

Process Weaver [2] is a set of tools that adds process support capability to UNIX-based environments. It consists of tools for modeling and enactment of activity-centered process models. We are currently constructing a transaction manager component, called PERN [3], for the Oz [1] decentralized process centered environment. It seemed natural to test PERN with a foreign system. This paper shows the results of this experiment.

## 2 Transactions

There is a definite distinction between concurrency through synchronization and concurrency control. Petri nets are excellent for explicitly modeling concurrent activities

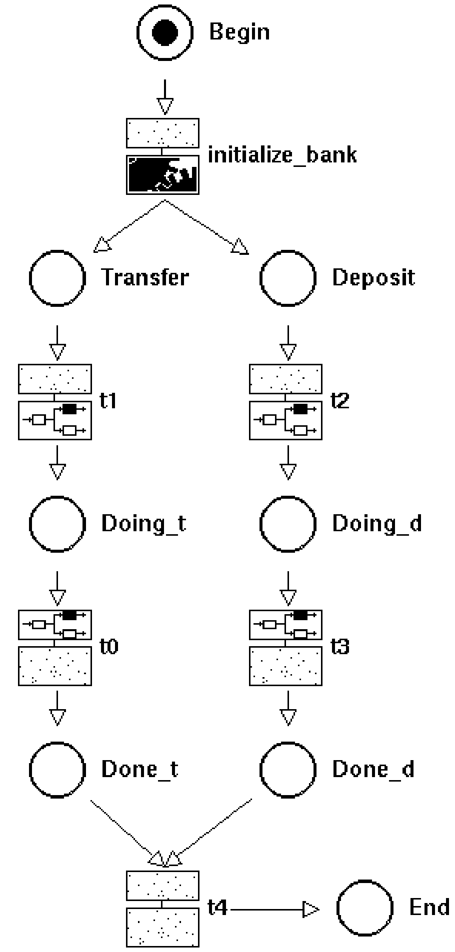


Figure 1: Banking example

of cooperating agents but there is no underlying mechanism for treating conflicting actions of concurrent, independent agents. In this section, we discuss alternatives of how transactions can be integrated with Process Weaver.

## 2.1 Concurrency conflicts

Consider the small fragment of a cooperative procedure (CP) shown in Figure 1[1]. A small "bank" is initialized with two accounts containing \$100 each. Two sub-activities then occur; the first allows the user to transfer \$10, \$20, or \$40 from account 1 to account 2; the other allows the user to deposit \$10, \$20, or \$40 to account 1. The bank is stored using Process Weaver's Universal Storage Mechanism (USM) that stores data in ASCII files. These USM files are structured into sections, each headed by `[section-name]` that may contain several *attributes* denoted by `attribute-name=`.

[1]Throughout this paper, we assume the reader has a working knowledge of Process Weaver.

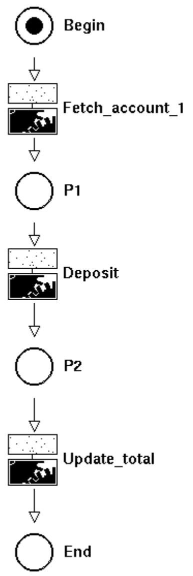


Figure 2: Simplified Deposit CP

All attributes related to one section are located under the appropriate section heading. If, for example, the user decides to deposit \$20 and transfer \$40, the valid result should be:

```
[account-1]
balance=80
[account-2]
balance=140
```

However, when executing the CP, Process Weaver instead produces

```
[account-1]
balance=60
[account-2]
balance=140
```

Transitions **t1** and **t2** from Figure 1 instantiate sub-procedures at run-time. Figure 2 shows a simplified Deposit cooperative procedure started by **t2**; the Transfer CP is similar. Since there is no synchronization between these two CPs, data is corrupted as they each attempt to perform their actions.

This brief example shows that Process Weaver has no concurrency control mechanism for its USM. This complicates the writing of cooperative procedures since the writer must be aware of the global set of cooperative procedures and explicitly serialize all access to a particular USM file. This example reveals a larger problem with Process Weaver, namely its inability to provide fault tolerant computing. Since we are currently building PERN, a transaction manager component, we experimented integrating it with Process Weaver and have come up with some preliminary results.

### 2.2 Transaction mechanism

One standard way to provide fault tolerance is to use the *transaction* concept. All database operations performed by a program are grouped into sequences called transactions. This is done for three distinct purposes; as summarized by Lynch [4], a transaction is a:

1. Logical Unit – Grouping together operations comprising a "complete" task.
2. Atomicity Unit – Giving appearance that all operations are consecutively carried out.
3. Recovery Unit – Ensuring that either all the steps, or none, are executed on the database.

We do not consider using transactions with the Process Weaver's USM, since it is too simplistic to be considered a database. Rather, we propose an architecture that uses PERN as a component that manages access to a separate object management system. This architecture is described in Section 2.3. First, we need to consider different ways of applying transactions to Process Weaver.

Since we could not rewrite Process Weaver, we only considered options that allowed us to add transaction support to Process Weaver using its existing functionality. We chose to embed transactions within cooperative procedures using the *co-shell* interface. The second design decision is concerned with locking conflicts. There are several alternatives, including:

- Blocking – A transaction requesting a lock L that conflicts with an existing lock M must wait until lock M is released.
- Immediate – A transaction requesting a conflicting lock automatically abort.
- Optimistic – When a transaction commits, it verifies that it did not corrupt data, otherwise it rolls back.

PERN provides an interface that allows us to program the desired concurrency control policy; we will show our experiments with each of these alternatives. We now discuss several design alternatives for embedding transactions into cooperative procedures. Our goal is to find the least-intrusive method of integrating transactions.

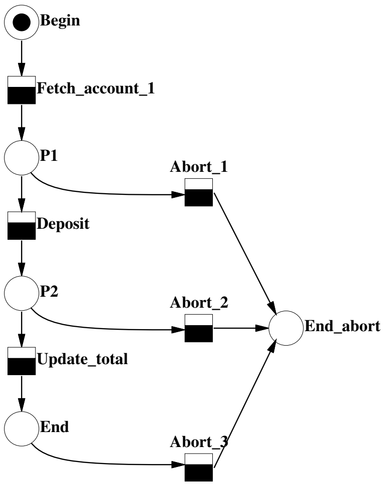


Figure 3: Case I: Rewriting

### 2.2.1 Rewriting

The first attempt to integrate transactions with a CP results in a modified CP that adds extra transitions from every *non-initial* state in Figure 2. All states in a CP not involved in the initial marking are *non-initial* states. When the CP is in one of these states, data could have been changed by the CP. Therefore, if the underlying transaction for the CP aborts (and it could abort at any non-internal state), the CP needs to undo its modifications. An additional state needs to be added signifying an abort-end. Figure 3 shows how a simplified version of the Deposit CP of Figure 2 is rewritten.

This method of integration is not desirable for several reasons. First, it complicates the original CP with additional transitions that confuse the original logic of the CP. We would like to design a method that—as much as possible—leaves the original CP unchanged. Second, it focuses solely on the abort of the underlying transaction; there are many other important events of transactions, such as commit and suspend. The CP would have to include separate transitions from each internal state for all important events. Third, this method cannot be used in Process Weaver because of an implementation-specific design decision. In Process Weaver, if a state has multiple transitions from it that are true, one is enabled in non-deterministic fashion; these semantics make it impossible to apply rewriting to Process Weaver.

### 2.2.2 Augmenting

Instead of rewriting each CP, the second attempt augments each CP with a new *header* and *trailer*. The augmented CP in Figure 4, for example, shows how extra states and transitions are added to bracket the CP with transitions that *begin* and *commit*

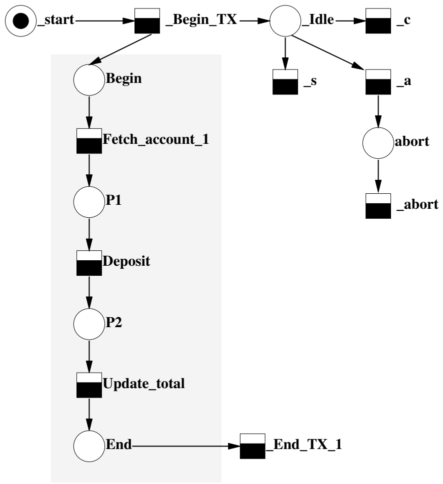


Figure 4: Case II: Augmenting

transactions for the CP. The actual co-shell details that implement the transactions are discussed in Section 2.3.

It seems natural to associate transactions one-to-one with cooperative procedures. Each CP can be viewed as a refinement of a higher level activity. Process Weaver manages the complexity of having multiple cooperative procedures through methods. A method is an organized set of cooperative procedures in hierarchical fashion. This organization dovetails nicely with the concept of nested transactions [5]. For our purposes, therefore, one logical unit equals one cooperative procedure.

#### 2.2.3 Global Transaction CP

The final alternative, shown in Figure 5, has one cooperative procedure for the transaction processing of the entire global set of CPs. Having one transaction CP responsible for so many CPs, however, is equivalent to having one single transaction for the entire system. Any attempt to have multiple transactions is impeded by the inability to model them within the global CPs. This approach is of too coarse a granularity to consider.

### 2.3 Architecture

Process Weaver tools communicate internally with each other by means of message passing using a Broadcast Message Server (BMS). In a BMS environment, agents emit requests and notifications to a central BMS, and subscribe to messages by registering message templates with the central BMS which later selectively broadcasts received messages according to the registered subscriptions. In such an open system, we

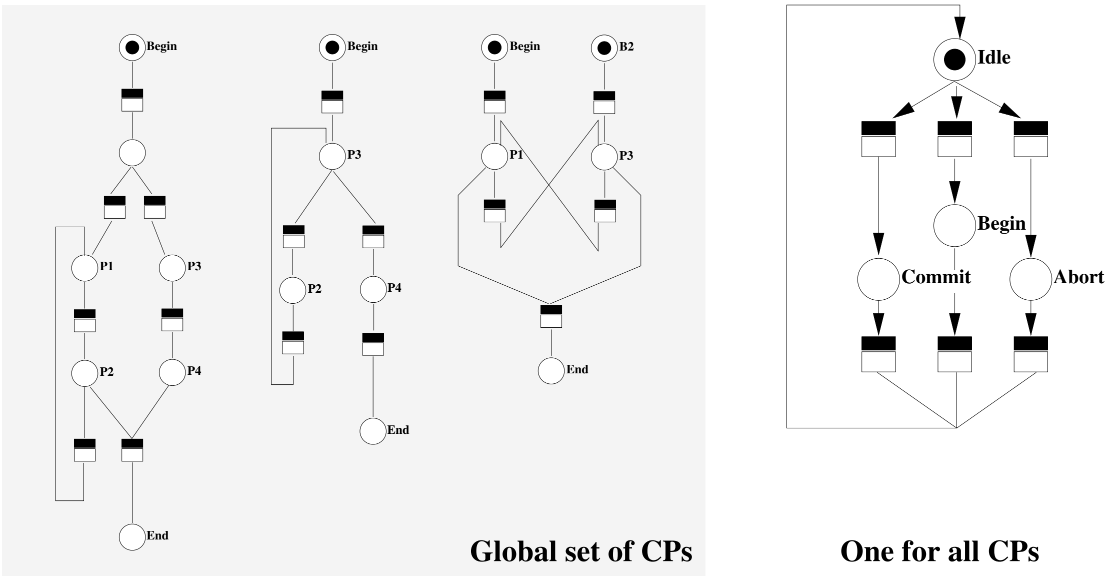


Figure 5: Case III: Global

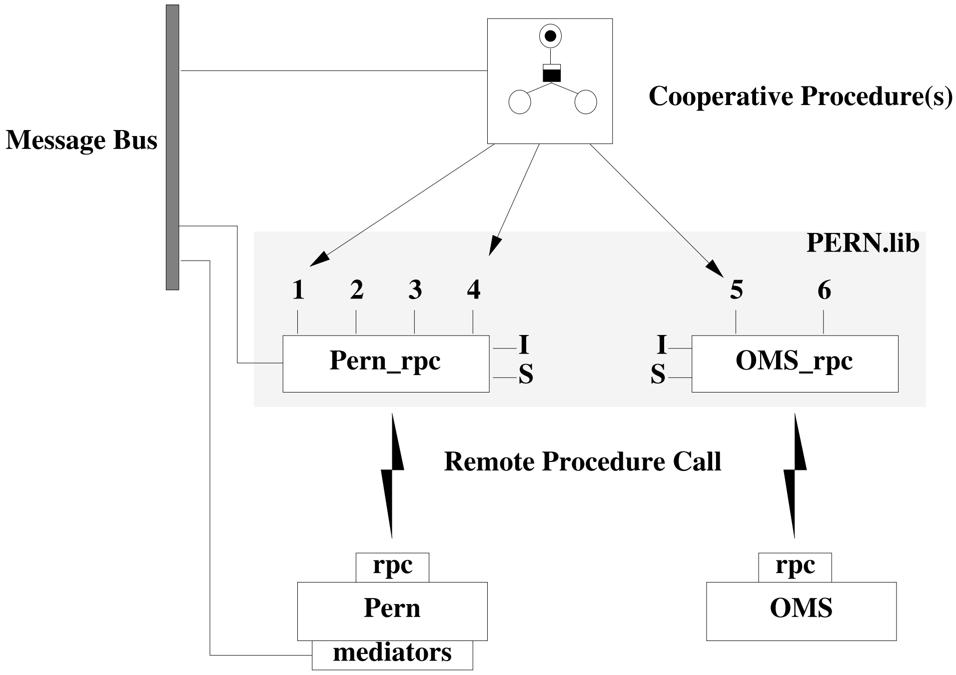


Figure 6: Architecture

anticipated that we could easily incorporate the PERN component; in Section 4 we assess how successful our experiments were.

Process Weaver provides a *co-shell* language that extends the power of cooperative procedures. We have implemented a co-shell library, called `PERN.lib`, that integrates PERN and an Object Management System (OMS) into Process Weaver. `PERN.lib` uses Remote Procedure Calls (RPC) to communicate with PERN and the OMS. The full `PERN.lib` is reproduced in Appendix A. Our basic approach is to embed calls to PERN and the OMS within the transitions of the cooperative procedures.

The PERN and OMS remote procedure interface each have an (I) initialization and (S) shutdown function call. The other six procedure calls are as follows (details in Appendix A):

1. `BEGIN` a transaction
2. `COMMIT` a transaction
3. `ABORT` a transaction
4. `LOCK` a particular object in a given lock mode

5. `GET_ATTRIBUTE_A` for a particular object.
6. `WRITE_ATTRIBUTE_A` for a particular object.

PERN allows mediators to be written, so that special actions can be taken when a transaction begins, commits, aborts, or locks an object. In our experiments, we have provided two mediators which place a message on the BMS of class PERN, of operation PERN_COMMIT (PERN_ABORT), and with the transaction id as the message id. We now work through an extended example to show how transactions can be integrated into cooperative procedures.

# 3 Extended Example

We return to the banking example from Section 2.1 and describe the actual Deposit and Transfer cooperative procedures, shown in Figures 7 and 9. In these CPs, a work context is sent to the user, requesting a dollar amount (or cancel) to deposit or transfer between accounts. Upon receipt of the user-action, the corresponding bank accounts are updated.

The first step is to augment the Deposit and Transfer CPs with the transaction constructs discussed in Section 2.2.2. This is done by hand, for now, but we envision a translator capable of automatically generating these augmented CPs based upon user guidance. For space reasons, the original CPs have been "folded" up. A transaction is started at **_BEGIN_TX** by the following co-shell action:

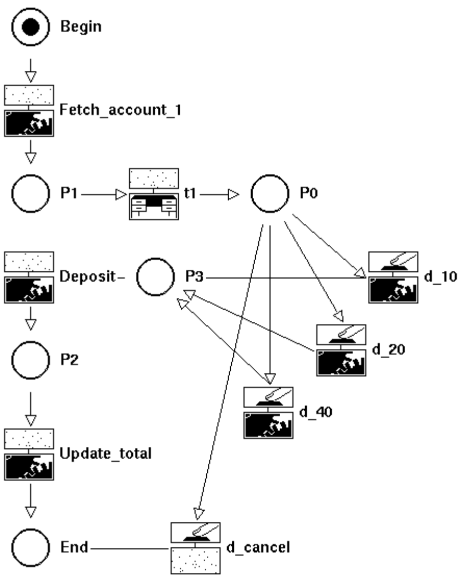


Figure 7: Full Deposit CP

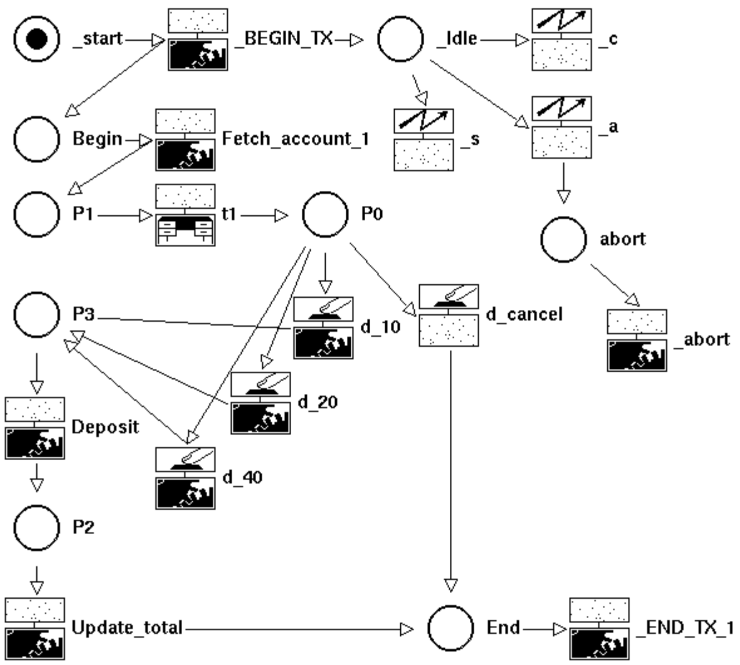


Figure 8: Deposit CP with Pern

```
{ Begin Transaction }
$tid = tx_BEGIN ('-1', NOCOMMIT, NOABORT, TOP, ROLLBACK);
```

This issues a remote procedure call to PERN that creates a top-level transaction (with parent -1) that can be rolled-back and has no commit or abort dependencies on any other transaction. This transition activates the original starting state for the CP, **Begin**, and also moves into the **_Idle** state. The **_Idle** state 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, waits for a broadcast message of the PERN tool class, of operation PERN_ABORT, and of message id `$tid`. This filters out all PERN messages not dealing with this particular transaction. If the transaction aborts, the CP moves into the **abort** state. This state exists to allow the parent CP, the one which instantiated this CP, to take action in response to this transaction abort. Note that the original CP has its own thread of control and continues its normal actions.

The CP uses an OMS to store and retrieve data. Transition **Fetch_account_1**, for example does the following:

```
{ Access account-1 (through an object identifier $account_1). Note: $account_1 }
{ is a parameter to this CP, and set by the parent. First we request access from }
{ PERN, then we get the information from the OMS. }
action = Access ($tid, $account_1, X);
$v1 = Read_Attribute($account_1, balance);
```

The `Access` function, as described in Appendix A, issues a remote procedure call to PERN, attempting to place 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.

The user is involved next, since a work context (not shown here) is sent to the user's agenda, requesting the amount to be deposited, either $10, $20, or $40. The user clicks on the appropriate button, enabling one of **d_10, d_20, d_40**, or **d_cancel**. Finally, the deposit amount is added to the balance `$v1` and in transition **Update_total**, the new balance is written back to the OMS:

```
Write_Attribute($account_1, "balance", $v1);
```

and the CP moves into the final **End** state.

At this point, the new transition **_END_TX_1** commits the transaction with:

```
{ Commit the Transaction }
tx_COMMIT($tid);
```

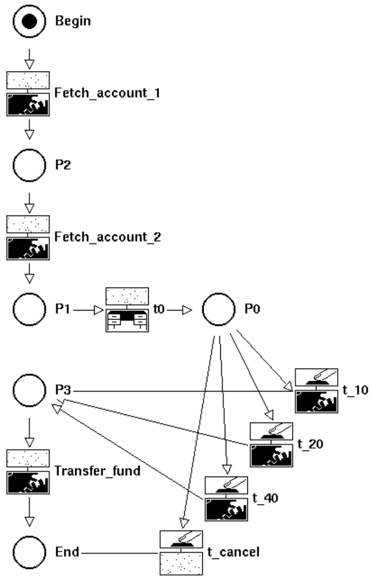


Figure 9: Full Transfer CP

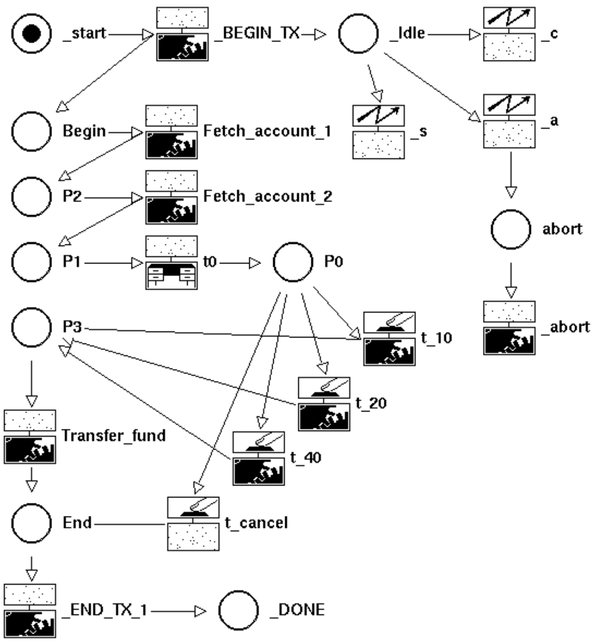


Figure 10: Transfer CP with Pern

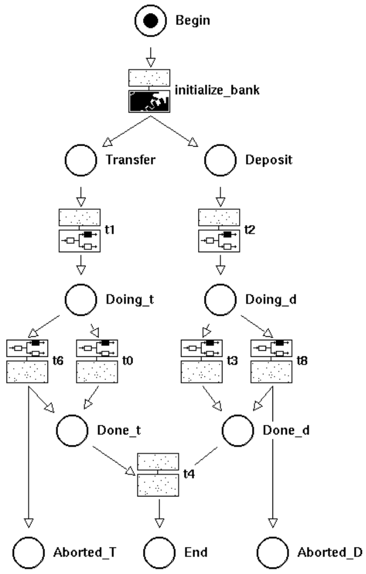


Figure 11: Modified bank CP

tx_COMMIT issues a remote procedure call to PERN to commit the given transaction. This will eventually[2] result in a message broadcast of tool class PERN, operation PERN_COMMIT, and message id $tid. The transition **_c** will retrieve this message, clearing out the token at **_Idle**, and the CP will complete successfully. The Transfer CP (Figure 9) is modified in similar fashion as shown in Figure 10.

The final step is to modify the bank CP so that it can take action in response to a transaction abort. Two additional transitions, **t6** and **t8** are added which are activated when the corresponding Transfer or Deposit sub-procedure reaches the **abort** state. If a sub-procedure aborts, the appropriate state (either **Aborted_T** or **Aborted_D**) is marked.

## 3.1 Execution Trace

We are now ready to walk through the execution of the bank CP. The full message trace generated during this example is reproduced in Appendix B. Messages that are in response to a previous message requested are numbered differently. For example, the second message is labeled `1R` since it is a notification message in response to the request message numbered `1`. The Bank CP is launched by the user `[1]` and starts executing `[2-8]`, at which point it is in states { **Doing_t, Doing_d**}. The two sub-procedures, Transfer and Deposit, are launched at `[6-7]`. The Transfer CP is the first one instantiated `[9-10]` and it starts executing. Between `[11]` and `[12]`, the Transfer CP begins a transaction. Proceeding on to `[13-14]`, we see that the Transfer

[2]Through the use of a mediator, this instantiation of PERN sends a message to the BMS.

CP has acquired the necessary locks and a new working context, 00001001 is created and sent to the user's agenda [15-16, 18-20]. The Transfer CP then moves into state **P0** [17].

The Deposit CP is then instantiated at [21-22]. Between [23] and [24], the Deposit CP begins a transaction. When Deposit tries to access the appropriate objects, however, between [24] and [26], a locking conflict occurs, and a message [25] is generated by the PERN mediator. The particular transaction id, 5, is part of the message so only this Deposit CP is registered for it. Deposit moves from {**P1, _Idle**} into {**P0, abort**}. This transition generates a working context, 00001002, at [27-28], but also prepares for the abort. Messages [29-30] shows Deposit moving into the abort state, and message [31] is generated by the co-shell action for transition **_abort**. This message is the same one that would be generated had the user manually decided to "kill" the cooperative procedure. [32] shows the CP moving out of the **abort** state, and [31r] shows the receipt of the END_PROC message by the system. [33-36, 38-41] shows the sequence of messages which destroys the work context 00001002 at the user's agenda. This ability to destroy the appropriate work contexts is part of Process Weaver, and made it easier to incorporate transactions. The parent Bank CP makes a transition (**t8**) to **Aborted_d** and **Done_d** at [37] since it was monitoring for a transaction abort.

At this point, the user's agenda has one valid work context, Transfer, and a "ghost" work context for Deposit. The user then opens the Transfer work context and selects $20 by clicking on the appropriate button [43-44]. The Transfer CP then moves into the **End** state at [45-46]. At this point, the original CP is completed, and the augmented transition, **_END_TX_1** makes a call to tx_COMMIT that commits transaction 4 and generates message [47]. This commit broadcast is recognized between [48, 51] as the system moves out of the **_Idle** state through transition **_c**. The Transfer CP finally completes at [51-53] and the parent Bank CP moves into the appropriate states. Note that before Bank completes [54-57] it had moved into the **End** state, while registering the fact that **Aborted_D** had occurred.

The final bank balance, needless to say, only registers the transition of $20; the Deposit CP has no effect since its transaction was aborted.

# 4 Conclusion

The success of these experiments shows the flexibility of PERN. The overhead induced through the architecture in Section 2.3 is limited. For each transaction, only one message (PERN_ABORT or PERN_COMMIT) is broadcast on the BMS, so this scheme doesn't produce excessive messages. The augmented CPs need only add a fixed number of states (4) and transitions (6) to work properly, thus incurring only a small space overhead.

There are many other experiments we are planning:

- Semantic Based Concurrency Control – PERN has a coordination rule language by which one can tailor the concurrency control based upon available semantics. In Process Weaver, for example, an administrator could program certain conflict scenarios based upon states, transitions, and work contexts. Barghouti [?] has already shown a language for a rule-based system; we would like to show how PERN's language can apply to petri-net based systems.
- Transactions for software development environments – the example in this paper is a very small one. We would like to integrate transactions into a much broader, realistic example.
- Cooperative Transactions – Process Weaver provides exceptional modeling capabilities for cooperation. Since PERN is attempting to provide support for cooperative transaction models, Process Weaver is a very useful testbed for ongoing experimentation.

# Acknowledgments

We would like to thank Cap Gemini for providing the Process Weaver tool at a substantial university discount; none of this research would have been possible without their generosity.

# References

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

[2] 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.

[3] 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.

[4] Nancy A. Lynch. Multilevel atomicity – a new correctness criterion for database concurrency control. *ACM Transactions on Database Systems*, 8(4):484–502, December 1983.

[5] J. Eliot B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing.* Information Systems. The MIT Press, Cambridge MA, 1985. PhD Thesis, MIT LCS TR-260, April 1981.

# A PERN.lib co-shell implementation

```
{------------------------------------------------------------}
{  Provides necessary interface with PERN and OMS components   }
{                                                             }
{  $rc = Access ($tid, $oid, $mode)                           }
{                                                             }
{  $value = Read_Attribute($obj_handle, $att_name)            }
{  $rc = Write_Attribute($obj_handle, $att_name, $NewValue)   }
{                                                             }
{ PERN.lib assumes that the PERN and OMS server have already  }
{ been started.                                               }
{------------------------------------------------------------}

{ -------------------- assoc() }

{    assoc(c, 'a 1 b 2 c 3 d 4') --> 1                        }
{    assoc(2, 'a 1 b 2 c 3 d 4') --> ''                       }

define assoc($elt, $list)
  WHILE (not(null($list)) and not(equal(first ($list), $elt)))
  DO
    { skip over pair }
    $list = rest($list); $list = rest($list);
  DONE;
  first(rest($list));
end;

{ -------------------- abort_cp() }

define abort_cp()
  { kills the current CP.  Useful for aborting a TX.  }

  SetContext ($CPHost, $CPDir, $CPFile);
  SendEvent (R, PROCEDURE, END_PROC, '*', $CPUser);
end;


{ -------------------- Define PERN global values }

define pern_globals()
  $pern_HOST       = 'americas';
  $pern_TASK       = '1';

  { RPC interface with PERN }
  $pern_INITIALIZE = 999;
  $pern_SHUTDOWN   = 998;
  $pern_BEGIN      = 1;
  $pern_COMMIT     = 2;
  $pern_ABORT      = 3;
  $pern_LOCK       = 4;

  { pairwise lock list for PERN }
  $pern_lock_list  = 'IS 4 IX 8 S 1 X 2 SX 16 SS 32';

  $pern_server = "/proj/dorothy/heineman/rpc/IBM.rpc/pern_rpc/pern_rpc_svc";
  $pern_client = "/proj/dorothy/heineman/rpc/IBM.rpc/pern_rpc/pern_rpc";

  $pern_directory = "/proj/dorothy/heineman/rpc/IBM.rpc/pern_rpc";
end;

{ -------------------- Define OMS global values }
define oms_globals()
  $oms_HOST                          = 'americas';
```

```
  { RPC interface with OMS }
  { Note: For this experiment, only #5 and #2 are used }

  $oms_INITIALIZE                   = 999;
  $oms_SHUTDOWN                     = 998;
  $oms_GET_TOP_LEVEL                = 1;
  $oms_GET_ATTRIBUTE_A              = 2;
  $oms_GET_ALL_ATTRIBUTES           = 3;
  $oms_GET_ALL_ANCESTORS            = 4;
  $oms_WRITE_ATTRIBUTE_A            = 5;
  $oms_DELETE_OBJECT_X_FROM_A_OF_Z  = 6;
  $oms_ADD_NAME_TO_A_OF_Z_SUBCLASS_S = 7;
  $oms_DELETE_TOP_LEVEL             = 8;
  $oms_ADD_TOP_LEVEL                = 9;
  $oms_LINK_OBJECT_X_TO_A_OF_Z      = 10;
  $oms_COPY_OBJECT_X_TO_A_OF_Z      = 11;
  $oms_MOVE_OBJECT_X_TO_A_OF_Z      = 12;
  $oms_RENAME_OBJECT_X_TO_NAME      = 13;

  $oms_server    = "/proj/dorothy/heineman/rpc/IBM.rpc/oms_rpc/oms_svc";
  $oms_client    = "/proj/dorothy/heineman/rpc/IBM.rpc/oms_rpc/oms";

  $oms_objectbase = "/proj/dorothy/heineman/rpc/IBM.rpc/mini_test";
end;

{ -------------------- send rpc call to OMS Server }
define oms_rpc_call($data)
  $t = format($oms_client, " ", $oms_HOST, " ", $data);

  { return the output }
  system($t);
end;

{ -------------------- send rpc call to PERN Server }
define pern_rpc_call($data)
  $t = format($pern_client, " ", $pern_HOST, " ", $pern_TASK, " ", $data);

  { return the output }
  system($t);
end;


{ -------------------- Start OMS server }
define START_OMS()

  { Assumes that OMS server is already started in background }

  { initialize appropriately }
  oms_rpc_call(format($oms_INITIALIZE, " ", $oms_objectbase));
end;

{ -------------------- Start PERN server }
define START_PERN()

  { Assumes that PERN server is already started in background }

  { initialize appropriately }
  pern_rpc_call(format($pern_INITIALIZE, " ", $pern_directory));
end;


{ -------------------- Initialize PERN and OMS }
define INITIALIZE_PERN()
  pern_globals();
  oms_globals();
```

```
  { -- initialize servers -- }
  START_OMS();
  START_PERN();
end;


{ -------------------- Shutdown PERN and OMS }
define SHUTDOWN_PERN()
  { Shutdown OMS and PERN server }
  oms_rpc_call($oms_SHUTDOWN);
  pern_rpc_call($pern_SHUTDOWN);
  Exit(0);
end;


{ -------------------- Access Object }

define interpret_lock($name)
  assoc($name, $pern_lock_list);
end;

define Access($tid, $oid, $lock_name)
  pern_globals();
  $mode = interpret_lock($lock_name);
  $rc = pern_rpc_call(format($pern_LOCK, " ", $tid, " ", $oid, " ", $mode));

  $rc;   { return status }
end;


{ -------------------- Read Attribute Value }

define Read_Attribute($oid, $att_name)
  oms_globals();

  $att_value = oms_rpc_call(format($oms_GET_ATTRIBUTE_A, " ", $oid, " ", $att_name));

  { $att_value is of form <header> <att_name> <value> }
  nth(3, $att_value);   { return value }
end;


{ -------------------- Write Attribute Value }

define Write_Attribute($oid, $att_name, $NewValue)
  oms_globals();

  { <attribute-name>, <object-id>, <string-value> }
  $rc = oms_rpc_call(format($oms_WRITE_ATTRIBUTE_A, " ", $att_name, " ", $oid, " ", $NewValue));

  { $rc is of form <head> <att_name> <value> }
  nth (1, $rc);      { return code }
end;


{ -------------------- tx_BEGIN() }
define tx_BEGIN($parent_tid, $commit_t, $abort_t, $nesting_t, $rollback_t)

  pern_globals();
  $data = pern_rpc_call(format($pern_BEGIN, " ", $parent_tid, " ", $commit_t, " ",
                               $abort_t, " ", $nesting_t, " ", $rollback_t));

  { $data is <STATUS tid> }
  $tx_id = nth(2, $data);
  if (null($tx_id)) then $tx_id = '-1'; endif;
```

```
  { return the <tid> }
  $tx_id;
end;

{ -------------------- tx_COMMIT() }
define tx_COMMIT($tid)
  pern_globals();
  $rc = pern_rpc_call(format ($pern_COMMIT, " ", $tid));

  $rc;
end;


{ -------------------- tx_ABORT() }
define tx_ABORT($tid)
  pern_globals();
  $rc = pern_rpc_call(format ($pern_ABORT, " ", $tid));

  $rc;
end;
```

## B Message trace

| # | T | Tool Class | Operation | [Agent] | [EventData] |
|---|---|---|---|---|---|
| 1 | R | PROCEDURE | START_PROC | [HOST] | [weaver default PROC/Bank.CP] |
| 1R | N | PROCEDURE | BEGIN_PROC | [Bank.CP] | [weaver 20646] |
| 2 | R | AGENDA | OPEN_PROC_SESSION | [Bank.CP] | [Bank.CP HOST 1] |
| 3 | N | PROCEDURE | STATE_PROC | [Bank.CP] | [weaver Begin] |
| 4 | N | PROCEDURE | START_PROC | [Bank.CP] | [weaver] |
| 2R | N | AGENDA | OPEN_PROC_SESSION | [Bank.CP] | [Bank.CP HOST HOST] |
| 5 | N | PROCEDURE | STATE_PROC | [Bank.CP] | [weaver Deposit Transfer] |
| 6 | R | PROCEDURE | START_ACTIVITY | [Bank.CP] | [weaver HOST Transfer.CP] |
| 7 | R | PROCEDURE | START_ACTIVITY | [Bank.CP] | [weaver HOST Deposit.CP] |
| 8 | N | PROCEDURE | STATE_PROC | [Bank.CP] | [weaver Doing_t Doing_d] |
| 9 | N | PROCEDURE | BEGIN_PROC | [Transfer.CP] | [weaver 20651] |
| 10 | R | AGENDA | OPEN_PROC_SESSION | [Transfer.CP] | [Transfer.CP HOST 1] |
| 11 | N | PROCEDURE | STATE_PROC | [Transfer.CP] | [weaver _start] |
| 10R | N | AGENDA | OPEN_PROC_SESSION | [Transfer.CP] | [Transfer.CP HOST HOST] |
| 12 | N | PROCEDURE | STATE_PROC | [Transfer.CP] | [weaver _Idle Begin] |
| 13 | N | PROCEDURE | STATE_PROC | [Transfer.CP] | [weaver _Idle P2] |
| 14 | N | PROCEDURE | STATE_PROC | [Transfer.CP] | [weaver _Idle P1] |
| 15 | F | PROCEDURE | ERROR | [Transfer.CP] | [E3071 - Error: unable to send work-context instance to] |
| 16 | R | AGENDA | ADD_WCTX | [00001001] | [WEAVER Transfer.CP] |
| 17 | N | PROCEDURE | STATE_PROC | [Transfer.CP] | [weaver _Idle P0] |
| 18 | R | USER_AGENDA | ADD_WCTX | [00001001] | [WEAVER Transfer.CP] |
| 19 | N | USER_AGENDA | UPDATE_WCTX | [00001001] | [Transfer.CP WEAVER Agenda Owner WEAVER] |
| 20 | N | AGENDA | UPDATE_WCTX | [00001001] | [Transfer.CP WEAVER Agenda Owner WEAVER] |
| 18R | N | USER_AGENDA | ADD_WCTX | [00001001] | [Transfer.CP WEAVER] |
| 6R | N | PROCEDURE | START_ACTIVITY | [Transfer.CP] | [weaver] |
| 16R | N | AGENDA | ADD_WCTX | [00001001] | [Transfer.CP WEAVER] |
| 21 | N | PROCEDURE | BEGIN_PROC | [Deposit.CP] | [weaver 20661] |
| 22 | R | AGENDA | OPEN_PROC_SESSION | [Deposit.CP] | [Deposit.CP HOST 1] |
| 23 | N | PROCEDURE | STATE_PROC | [Deposit.CP] | [weaver _start] |
| 24 | N | PROCEDURE | STATE_PROC | [Deposit.CP] | [weaver _Idle Begin] |
| 7R | N | PROCEDURE | START_ACTIVITY | [Deposit.CP] | [weaver] |
| 22R | N | AGENDA | OPEN_PROC_SESSION | [Deposit.CP] | [Deposit.CP HOST HOST] |
| 25 | N | PERN | PERN_ABORT | [ ] | [5] |
| 26 | N | PROCEDURE | STATE_PROC | [Deposit.CP] | [weaver _Idle P1] |
| 27 | F | PROCEDURE | ERROR | [Deposit.CP] | [E3071 - Error: unable to send work-context instance to] |
| 28 | R | AGENDA | ADD_WCTX | [00001002] | [WEAVER Deposit.CP] |
| 29 | N | PROCEDURE | STATE_PROC | [Deposit.CP] | [weaver _Idle P0] |
| 30 | N | PROCEDURE | STATE_PROC | [Deposit.CP] | [weaver abort P0] |
| 31 | R | PROCEDURE | END_PROC | [Deposit.CP] | [weaver] |
| 32 | N | PROCEDURE | STATE_PROC | [Deposit.CP] | [weaver P0] |
| 31R | N | PROCEDURE | END_PROC | [Deposit.CP] | [weaver] |
| 33 | R | PROCEDURE | CLEAN_PROC | [Deposit.CP] | [00001002] |
| 34 | R | AGENDA | REMOVE_WCTX | [00001002] | [weaver_proc Deposit.CP] |
| 35 | R | AGENDA | CLOSE_SESSION | [00001002] | [Deposit.CP] |
| 36 | R | USER_AGENDA | ADD_WCTX | [00001002] | [WEAVER Deposit.CP] |
| 37 | N | PROCEDURE | STATE_PROC | [Bank.CP] | [weaver Aborted_D Done_d Doing_t] |
| 38 | N | USER_AGENDA | UPDATE_WCTX | [00001002] | [Deposit.CP WEAVER Agenda Owner WEAVER] |
| 39 | R | USER_AGENDA | REMOVE_WCTX | [00001002] | [WEAVER Deposit.CP] |
| 36R | N | USER_AGENDA | ADD_WCTX | [00001002] | [Deposit.CP WEAVER] |
| 34R | N | AGENDA | REMOVE_WCTX | [00001002] | [Deposit.CP AGENDA] |
| 35R | N | AGENDA | CLOSE_SESSION | [00001002] | [Deposit.CP] |
| 40 | N | AGENDA | UPDATE_WCTX | [00001002] | [Deposit.CP WEAVER Agenda Owner WEAVER] |
| 39R | N | USER_AGENDA | REMOVE_WCTX | [00001002] | [Deposit.CP WEAVER] |
| 28R | N | AGENDA | ADD_WCTX | [00001002] | [Deposit.CP WEAVER] |
| 41 | N | AGENDA | REMOVE_WCTX | [00001002] | [Deposit.CP WEAVER] |
| 42 | R | PROCEDURE | STATE_PROC | [Bank.CP] | [weaver] |
| 42R | N | PROCEDURE | STATE_PROC | [Bank.CP] | [weaver Aborted_D Done_d Doing_t] |
| 43 | R | AGENDA | CLOSE_WCTX | [00001001] | [Transfer.CP WEAVER T20] |
| 44 | R | USER_AGENDA | CLOSE_WCTX | [00001001] | [Transfer.CP WEAVER T20] |
| 45 | N | PROCEDURE | STATE_PROC | [Transfer.CP] | [weaver _Idle P3] |
| 44R | N | USER_AGENDA | CLOSE_WCTX | [Transfer.CP] | [WEAVER Transfer.CP WEAVER T20] |

| | | | | |
|---|---|---|---|---|
| 46 | N PROCEDURE | STATE_PROC | [Transfer.CP] | [weaver _Idle End] |
| 43R | N AGENDA | CLOSE_WCTX | [Transfer.CP] | [WEAVER Transfer.CP WEAVER T20] |
| 47 | N PERN | PERN_COMMIT | [ ] | [4] |
| 48 | N PROCEDURE | STATE_PROC | [Transfer.CP] | [weaver _DONE _Idle] |
| 49 | N USER_AGENDA | CLOSE_WCTX | [Transfer.CP] | [WEAVER Transfer.CP WEAVER T20] |
| 50 | N AGENDA | CLOSE_WCTX | [Transfer.CP] | [WEAVER Transfer.CP WEAVER T20] |
| 51 | N PROCEDURE | STATE_PROC | [Transfer.CP] | [weaver _DONE] |
| 52 | N PROCEDURE | END_PROC | [Transfer.CP] | [weaver] |
| 53 | R AGENDA | CLOSE_SESSION | [Transfer.CP] | [Transfer.CP] |
| 53R | N AGENDA | CLOSE_SESSION | [Transfer.CP] | [Transfer.CP] |
| 54 | N PROCEDURE | STATE_PROC | [Bank.CP] | [weaver Aborted_D Done_d Done_t] |
| 55 | N PROCEDURE | STATE_PROC | [Bank.CP] | [weaver Aborted_D End] |
| 56 | N PROCEDURE | END_PROC | [Bank.CP] | [weaver] |
| 57 | R AGENDA | CLOSE_SESSION | [Bank.CP] | [Bank.CP] |
| 57R | N AGENDA | CLOSE_SESSION | [Bank.CP] | [Bank.CP] |