STDL - A Portable Language for Transaction Processing

Philip A. Bernstein[1] Per O. Gyllstrom Tom Wimberg

Digital Equipment Corp.
151 Taylor Street
Littleton, MA 01460-1407

Abstract

Structured Transaction Definition Language (STDL) is a block-structured language specialized for distributed transaction processing, developed by the Multivendor Integration Architecture consortium. The main design goals were application portability across multiple interoperable STDL implementations, and ease of implementation on different vendors' transaction processing systems — both new and existing ones. This paper describes STDL's transaction features: demarcating transaction boundaries, transactional remote procedure call, transactional queuing, recoverable terminal I/O, and transactional exception handling.

STDL relies on standard C and COBOL for most application logic and all operations on files and SQL databases. All transactional features of STDL and new features outside standard C and COBOL are isolated in procedures written in the STDL task definition language. These features include demarcating transaction boundaries, transaction recovery, exception handling, transactional communications, access to data queues, submission of queued work requests, and invocation of presentation services. This isolation of transactional features is quite different than other persistent programming languages and has two important benefits: one can use applications written in standard C or COBOL; and to implement STDL, it is possible to map clauses of the task definition language onto operations of most any distributed TP monitor.

[1] Address: Digital Equipment Corp., One Kendall Square, Building 700, Cambridge, MA 02139.
Internet: pbernstein@crl.dec.com.

Proceedings of the 19th VLDB Conference
Dublin, Ireland, 1993

1 Introduction to MIA

Structured Transaction Definition Language (STDL) is a language for transaction processing *(TP)*, developed by the Multivendor Integration Architecture *(MIA)* consortium. The consortium produced architectural specifications (including STDL) for general-purpose computing with the goals of application portability, interoperability, and a common end-user environment [6]. The specifications are intended to be a basis for procurement — that is, customers would buy systems conforming to the MIA specification. Such a procurement standard is needed because many existing standards are incomplete (e.g. do not include standard error codes, thereby limiting application portability), insufficiently international (e.g. do not handle the Japanese language well), or insufficiently interoperable (e.g. use incompatible calling standards or different options in communications standards).

The MIA consortium is sponsored and run by Nippon Telegraph and Telephone *(NTT)* of Japan. The members of this consortium during the creation of the MIA Version 1 Specifications were Digital Equipment Corp., Fujitsu Ltd., Hitachi, IBM, NEC and NTT Data, which is NTT's system integration subsidiary.

MIA specifications are primarily based on existing standards, both *de facto* and *de jure*, and cover three major areas: application programming interfaces (APIs), protocols, and human (end-user) interfaces. The API specifications define the set of languages used for application programming and include STDL, C, COBOL, FORTRAN, and SQL. (MIA modifies C, COBOL, FORTRAN and SQL slightly, to handle the Japanese language well.) Protocols for both OSI and Internet are in the specification, including transactional remote procedure calls, file transfer, electronic mail, and network management protocols. Human interface specifications address graphical window-based interfaces. The lack of international standards in this area prompted MIA to adopt three *de facto* standards: OSF Motif, IBM Common User Access, and AT&T Open Look. Using MIA style guides ensures the same look and feel across different presentation services.

The MIA Version 1.0 specifications were completed in the spring of 1991. Today, V1.2 is available from NTT (see [6]). The MIA specifications are the basis for procurement by NTT and are gaining interest in many other industries. Joint efforts by an expanded consortium, named SPIRIT, has started under the direction of the Network Management Forum. Members of SPIRIT include many international telecommunication companies (e.g., AT&T, British Telecom, NTT) and a growing list of vendors. It is expected that SPIRIT will adopt MIA in addition to other open technologies.

2 STDL Overview

For TP, the MIA consortium developed the STDL language. This decision was made due to the lack of any mature TP API standard. STDL is modeled on the Task Definition Language of Digital's TP monitor, ACMS [4, 9]. One reason ACMS was selected as the model is that it isolates TP-specific functionality in a separate language, so that most application logic can be written in standard C, COBOL and SQL, without TP extensions. The isolation of TP functions also simplifies implementing such an API on new and existing TP systems; the language for TP functions can be mapped to TP system functions and the implementation of standard languages can be used as is.

STDL supports access to persistent resources, so it is, in effect, a persistent programming language. STDL's use of a special language for TP functions and standard languages for most application logic makes it rather different than other persistent programming languages, which either are completely new designs or extend existing languages with new constructs for transaction functions.

Another reason for selecting ACMS as the model is its use of remote procedure call (RPC) for interprocess communication. Compared to peer-to-peer programming models (e.g. send-message, receive-message), RPC has several benefits: it allows programmers to retain a simple sequential execution model; it avoids programmers introducing certain protocol errors, such as forgetting to wait for a reply message or giving up waiting for a reply that subsequently arrives; it allows one to re-configure a calling and called program to be in the same or different processes without rewriting the programs; and it hides language differences in parameter format between caller and callee. Though the latter facility could be introduced in a peer-to-peer model, it typically isn't.

In developing STDL, the ACMS API was modified and extended in many ways, both in major features and syntactic detail. These modifications were done to make STDL portable, to ensure that STDL was implementable at acceptable cost on different TP systems, and to meet functional requirements defined for the MIA architecture.

During the MIA V1.0 design effort, each vendor checked how difficult it would be to implement STDL on their existing TP system. This was important, to ensure that vendors could deliver MIA-conformant products within 2-3 years. NTT has announced that many vendors either have delivered STDL implementations to NTT or will do so within the next year, including Digital, Fujitsu, Hitachi, HP, IBM, and NEC. Some of these STDL implementations are being used by NTT in the implementation of large application projects. The first two scheduled projects are: Listing Maintenance System, which provides off-line directory maintenance (for telephone books) and was awarded to Digital Equipment Corp.; and Directory Assistance System, which provides on-line directory assistance and was awarded to NEC.

Feedback from these implementation activities has and will be incorporated in past and future versions of the MIA specifications. In addition, NTT has also held internal, experimental demonstrations of portability of STDL applications between Digital and IBM implementations.

STDL is a portable block-structured language, specialized for TP. Procedures written in STDL are called tasks. All transactional features of STDL and new features outside standard C and COBOL are isolated in tasks. These features include:

- the ability to call COBOL and C programs, which can access SQL databases and stream, relative, indexed and indexed sequential files
- demarcating transaction boundaries
- RPC-based task-to-task transactional communication
- structured exception handling, portable across STDL implementations
- queued task submission
- a standard interface to presentation services (e.g. communication with display devices) based on the ISO Forms Interface Management System (FIMS) [3]
- recoverable presentation services, called *exchanges*
- recoverable variables, called *workspaces*
- environmental features required for full application portability (i.e. features that the programmer can count on but that are not reflected in the API)

To ensure interoperability between different implementations of STDL, MIA defined a transactional remote procedure call (RPC) protocol, called the *Remote Task Invocation* protocol *(RTI)*. RTI has recently been adopted by the X/Open consortium for transactional RPC [10]. RTI is an integration of the OSI TP standard for two-phase commitment and the Open Software Foundation (OSF) Distributed Computing Environment's RPC for task-to-task data transfer [7, 8]. RTI has also been specified for layering on TCP/IP and IBM's SNA LU6.2. Furthermore, the specification of STDL includes a detailed mapping of STDL operations onto protocol messages — an unusual feature of language and protocol specifications.

MIA also addresses interactive computing environments outside of TP. Through its *Interactive Processing Extensions*, applications in standard COBOL, C and FORTRAN that execute outside the TP system (for example, on workstations and PCs) can invoke TP applications and interact with presentation services. The interface for invoking TP applications is a stub routine generated from the TP application written in STDL. The stub routine communicates with the TP system using the RTI protocol. The presentation interface is based on FIMS.

This paper focuses on the transactional aspects of STDL. It begins with a model for STDL programs, followed by a description of STDL's transaction features: demarcating transaction boundaries, transactional remote procedure call, transactional queuing, recoverable terminal I/O, and transactional exception handling. For a full language specification, see [6]. An example STDL program is given in the Appendix.

3 The STDL Model

A *TP system* is a uniquely named entity that executes STDL applications (see Fig. 1). A node can have multiple TP systems, and a TP system can span multiple nodes. STDL applications are divided into three parts: tasks, presentation procedures, and processing procedures. A *task* is a procedure written in STDL task definition language. It controls the execution flow of the TP application, demarcates transactions, and specifies exception handlers. The procedure variables for tasks are called *workspaces*. These can be local to the task *(private workspaces)* or shared with other tasks *(shared workspaces)*.

A *processing procedure* is called by a task to perform computations and access files or relational databases. Processing procedures can be written in standard C or COBOL, and can access SQL or ISAM databases or use standard file transfer protocols (FTAM or FTP). (Lack of API standardization led to acceptance of vendor-specific file transfer APIs.) Procedures can, in turn, make nested calls to other procedures written in C or COBOL.

A task invokes a *presentation procedure* to obtain input or display output. The presentation procedure interacts with an external device, called a *display*, such as a terminal, PC, workstation, bar-code-reader, or automated teller machine. Presentation procedures can be part of a forms package or can be a set of programs written in C or COBOL. Although presentation procedures do not participate in transactions, inputs and outputs can be recovered after failures, as explained later, in Section 8.

This three-part application model follows ACMS. One goal of the original ACMS design was to control the context and programming options in the main task definition. This minimizes the resources tied up across terminal I/O and ensures that context across transactions is well-defined, to support enhanced reliability features such as checkpointing (which STDL exploits in transaction restart). Three alternatives were considered:

1. Provide a set of TP services callable from standard languages. This was rejected because it was hard to limit the context across terminal I/O and transactions.
2. Provide a definition utility for the task flow and TP functions. This alternative was tried and rejected because it was harder to use than a procedural language.
3. Support a restricted language in which to define the task flow, which calls all presentation functions, delimits transactions, and calls standard procedures for data processing. This alternative was picked because it allows one to define the task flow procedurally but it restricts the context that needs to be maintained over terminal I/O and transactions.

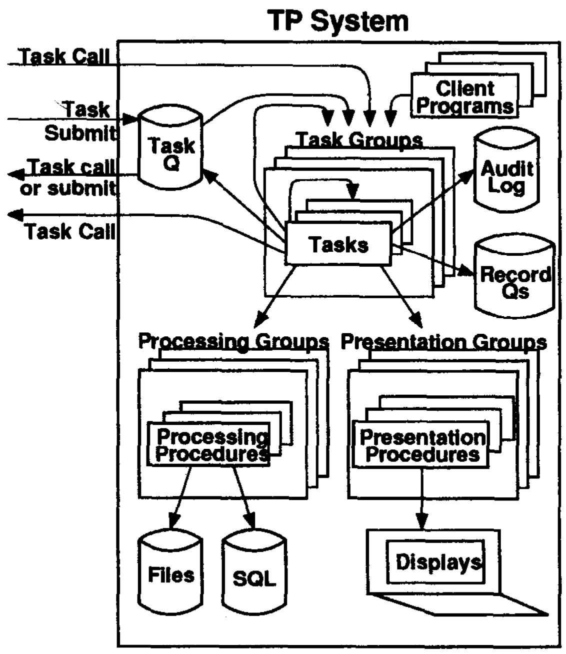

Figure 1 TP System Model

Structuring applications into three parts isolates standard 3GL code from portable STDL task definition code. The data processing part of the application is written in standard C, COBOL, and SQL. TP features are isolated in STDL tasks, which allows STDL implementors to map these features onto current TP system implementations.

These features include demarcating transaction boundaries, transaction recovery, exception handling, transactional communications, accessing data queues, submitting queued work requests, and invoking presentation services.

Due to a lack of existing standards, end-user presentation services are not portable and are therefore isolated in presentation procedures. Interoperability with different presentation services was attempted based on the FIMS work [3]. But the FIMS standard was incomplete when STDL was designed, so non-portable presentation services written in C and COBOL were allowed, to meet all of the presentation requirements.

Each TP system also has an audit log, a set of named record queues, and a task queue. The audit log is a sequential file where tasks can store interesting events. Record queues are storage areas where tasks can pass data, in the form of record structures. Task queues contain requests to run tasks and are discussed further in Section 7.

A *client program* is any program that invokes a task. It can be a task, an internal component of the TP system implementation, or an external agent.

4 Transactions

STDL supports a *flat* (non-nested) transaction model. Each transaction commits or aborts independent of the outcome of other transactions.

Since the MIA specification was intended as a basis for procurement, the designers had to balance their desire for state-of-the-art features with the practicality of what products would be available in the desired time frame. The transaction model was one area of compromise. A flat model was selected due to the lack of availability of products that support nested transactions (e.g. TP monitors, protocols and resource managers) and the lack of standardization efforts for nested transactions. We see no problems in adding nested transactions in the future, if and when that technology becomes widely available.

Recoverable operations are operations that, if executed within a transaction, are committed permanently and made visible to other transactions if and only if the transaction commits. Otherwise, they are undone. The result of a recoverable operation is stored in a *recoverable resource*, which is managed by a *recoverable resource manager*. A *durable resource* is a resource that survives system failures. STDL supports combinations of recoverable/non-recoverable and durable/non-durable resources, as summarized in Fig. 2.

Determining if an operation is recoverable depends on the type of resource. For files, it is done externally, per-file. This avoids introducing non-standard syntax into C and COBOL. For exchanges and queues, the determination is done operation-by-operation. For workspaces, it is done in the workspace declaration.

Resource	Recoverable?	Durable?
private workspace	Y & N	N
shared workspace	Y	N
SQL database (C & COBOL only)	Y	Y
indexed file (COBOL only)	Y & N	Y
relative file (COBOL only)	Y & N	Y
sequential file (COBOL only)	Y & N	Y
stream file (C only)	N	Y
recoverable send exchange	Y	Y
recoverable receive exchange	Y	N
task queue	Y & N	Y
record queue	Y	Y & N
audit log	N	Y

Figure 2 Properties of Resources

5 STDL Tasks

An STDL task definition is a named procedure consisting of *clauses*. These clauses are divided into a task name, a declarative part describing the task's attributes, and a set of executable clauses. Task attributes define the context of the task, including task arguments, task workspaces, and the "send display" (where presentation output is sent, if different from the default display).

Since all of an application's transaction-related functions are invoked in STDL tasks, understanding transactions in STDL amounts to understanding tasks.

A *task group* defines a set of tasks and is the scope of shared context. For example, a shared workspace is shared only by the tasks of the task group in which the workspace is defined. Similarly, a *processing procedure group* defines a set of processing procedures, which is also a scope for shared context. For example, processing procedures retain context (e.g. file and database cursors) across multiple calls in the same transaction.

STDL tasks use a *chained* transaction model: as soon as one transaction commits or aborts, another transaction starts. Thus, no access of recoverable resources can occur outside of a transaction. This is different than both the original ACMS model [9] and the X/Open *tx* model [11]; both ACMS and X/Open are *non-chained*, meaning they have syntax to explicitly start and end transactions and they allow operations to execute outside a transaction.

Although a non-chained model appears to be more flexible, we see no use for operations outside transactions. If an operation does not access a recoverable resource, then it makes no difference whether the operation is part of a transaction. If an operation accesses a recoverable resource, it must be part of a transaction. A chained transaction model, like STDL's, prevents programmers from accessing recoverable resources outside of a transaction. It also avoids semantic complexity around access to recoverable resources. For example, recoverable workspaces were added to STDL without needing complex rules to define where the recoverable workspaces could be used.

The use of chained transactions does not affect the architecture of transaction managers — the TP system component that implements the start, commit, and abort operations and the two-phase commit protocol. It does not imply that the underlying transaction protocol must be chained, as in IBM SNA LU6.2. And it allows a transaction manager with unchained semantics to be used, such as one implementing the X/Open model. An implementation just has to ensure that a transaction is active before a recoverable resource is accessed.

The only relationship between the chained transactions in a task is their execution order defined by the task's control flow. In particular, they commit or abort independently: the commitment of a transaction does not depend on the commitment of others (as it would in a nested transaction). The abort of a transaction does not prevent later transactions in the task from executing; the aborted transaction's exception handler decides whether to continue executing or terminate the task (see Section 9).

An implementation is free to optimize transactional aspects of STDL task execution. It can choose not to start a real transaction in certain cases, for example, if an STDL transaction contains only non-recoverable exchanges. It can also delay starting a real transaction until actually needed. For example, if an STDL transaction contains a non-recoverable exchange before calling a procedure that accesses recoverable resources, the real transaction can be started after the exchange is completed.

The STDL chained model is comparable to the transaction model in IBM's CICS, where the syncpoint operation both ends one transaction and starts the next. Originally, STDL used a similar model in which a COMMIT qualifier on some executable clauses indicated where a transaction ends. This was discarded in favor of a model in which the syntax of executable clauses defines a task's transactional structure. This allows transaction restart, exception handling, and other features to be tied to the syntax structure (see Section 9).

The syntax of STDL task definitions is organized to enforce the transactional rules of STDL. A task can be one of two types: composable or noncomposable. A *composable task* executes entirely within a transaction started by the caller of the task. A *non-composable task* executes in one or more transactions controlled by the task itself. A non-composable task uses executable clauses called *statements* to control the flow of transactions. A composable task cannot contain statements, since it cannot define a new transaction. A statement type called a transaction block delimits those recoverable actions that must occur within a single transaction. All recoverable actions must occur within transaction blocks. STDL provides other statements types (loops, if statements, case statements, and block statements) to sequence transactions. Since these statements execute outside of a transaction block, they cannot reference recoverable resources and can only reference private, non-recoverable workspaces.

Tasks use executable clauses called *steps* to perform work *within* transactions. For example, non-composable tasks use steps within transaction blocks, and composable tasks use steps as their top level executable clauses. Step types include calling tasks, submitting tasks to a queue, calling processing procedures, performing exchanges, enqueuing and dequeuing data, performing concurrent processing, operating on workspaces, loops, if steps, case steps, and step blocks.

Steps that call tasks, enqueue tasks, and perform exchanges can optionally be performed outside the transaction that invokes the step. This is useful to reduce transactional overhead or to record data before aborting or restarting a transaction.

A *concurrent-block step* allows concurrent execution of two or more steps within a transaction. It can also execute a set of transactions concurrently by including steps that call non-composable tasks, each of which executes in a different transaction.

Although STDL does not currently support nested transactions, it would be easy to add subtransaction blocks as a step type in the future.

Within a step clause, STDL provides an executable clause called an *action* to process the result of the main work of the step. Action types include operating on workspaces, raising exceptions, writing to the audit log, translating message codes into message text, transferring control to another step within the current block, if actions, and case actions.

6 Task Call

One reason for selecting ACMS as a model for MIA's TP API is its use of RPC for interprocess communication. In STDL, RPC is embodied in the CALL TASK step (see Fig. 3). A task uses CALL TASK to invoke another task synchronously. The execution model is RPC so the syntax of CALL TASK is insensitive to whether the callee is local or remote; distribution is an environmental consideration determined after the application is written.

```
PROCESSING WITH [ { INDEPENDENT } WORK ]
                  { DEPENDENT   }
    CALL TASK <task-name> IN <task-group name>
    [ AT <destination-name> ] [ USING { <workspace-name>} [,...] ]
```

(In showing STDL syntax: "[]" means use zero or one instance; "{}" means use zero or more instances; underscored words are required; non-underscored words are optional; "[,...]" means the previous clause, followed by a comma, can be repeated; "<>" denotes an STDL token which we leave undefined (see [6]).

Figure 3 Syntax for Call Task

One can execute a called task in the same transaction as the caller or in a separate transaction (see Fig. 4). This option is denoted in the call step by the phrase WITH DEPENDENT WORK (for the called task to execute as part of the caller's transaction) or WITH INDEPENDENT WORK (for the caller and callee to execute in different transactions that commit or abort independently). A task that's called WITH DEPENDENT WORK must be composable, that is, must have the keyword COMPOSABLE in its definition. A task that's called WITH INDEPENDENT WORK must not have the keyword COMPOSABLE in its definition, meaning that it's non-composable. A call WITH INDEPENDENT WORK is called a "top level transaction" in some systems (e.g. Argus [5]).

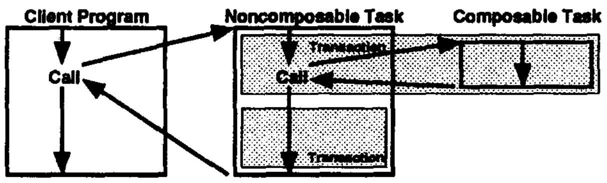

Figure 4 Composable and Non-Composable Tasks

A non-composable task is called for work that should complete whether or not the caller commits. For example, if the caller detects an illegal state based on the value returned by an earlier statement or step, then it may want to perform some work to repair that state (by calling a non-composable task), whether or not it (the caller) commits. A non-composable task is also used when the client is not executing in a transaction, for example, when the task to run is selected from a menu by an end-user.

The syntax WITH DEPENDENT (or INDEPENDENT) WORK in the call step and COMPOSABLE in the task definition is a redundant specification. In principle, it would be sufficient only to supply (or omit) the COMPOSABLE task attribute in the task interface definition of the called task. The syntax is supplied for consistency reasons. It reminds the programmer of the called task where the transaction bracket is and indicates if it is appropriate to have a transaction exception handler.

We considered allowing a call WITH DEPENDENT WORK of a non-composable task. The main problem is what to do if the caller fails before the reply is delivered. We found no useful and implementable semantics. Dropping the reply seems unsatisfactory, since the called task committed and its results may be needed. The best alternative seems to be treating the orphaned reply as a request to run a task that is an error handler for the failed calling task. But there are many messy details to make this work, complicating the language for a feature that isn't often used. In the end, this type of call was made illegal.

A <destination-name> names a TP System that knows the name of the called task and will invoke it. The USING clause lists the actual parameters to the called task

A task invokes processing procedures by a CALL PROCEDURE step. All processing procedures are composable. That is, a processing procedure always executes in the transaction of the task or processing procedure that called it. This is because STDL does all transaction demarcation in the task. No transaction demarcation is done in processing procedures, partly because there was no standard transactional demarcation API in C or COBOL when STDL was developed. SQL-specific transaction demarcation verbs are also excluded since they only apply to SQL operations.

Since the syntax for calling tasks is different from calling procedures, the caller's code must be modified if a task is rewritten as a procedure or vice versa. Clearly, this is undesirable. It was done to simplify implementation: an STDL implementation is not required to support RPC from tasks to procedures — only from task to task. Using different syntax for calling tasks and procedures makes it easier for an implementation to identify which calls can be remote.

A C or COBOL procedure can call a non-composable task. Such a procedure may be accessible as a processing procedure, but it is not required to be. The called task looks to the caller like an ordinary external procedure, accessed via a stub interface.

7 Submit Task

A task can invoke another task without waiting for the results by using a SUBMIT TASK step (see Fig. 5). Its effect is to create a request (i.e. message) to execute the task and put the request on a persistent task queue associated with the task's TP system. The TP system will dequeue the request later and invoke (i.e. call) the task.

Since the invoked task won't return to the submitting task, it cannot have return parameters.

```
PROCESSING WITH [ { INDEPENDENT } WORK ]
                  { DEPENDENT   }
  SUBMIT TASK <task-name>  IN <task-group name>
  [AT <destination-name>       ][ USING { <workspace-name>} [,...] ]
      <distribution-list-name>
  <trigger>
```

Figure 5 Syntax for Submit Task

The SUBMIT TASK step can be executed as part of the current transaction, denoted in the step using WITH DEPENDENT WORK. In this case, it is recoverable. That is, its effect (put-the-new-request-on-a-task-queue) is made permanent if and only if the transaction that executes the SUBMIT TASK commits. SUBMIT TASK can optionally execute in a separate transaction, denoted using WITH INDEPENDENT WORK.

A submitted task — the task that the SUBMIT causes to execute — is guaranteed to execute at least once and always runs in a new transaction. If the task is composable, then the system starts a transaction before calling the task. The system dequeues the request to execute the task *and* executes the task, all within one transaction. So, if the transaction aborts, the request is undone (returned to the queue) and will be processed later. Thus, composable tasks execute exactly once.

If the task is non-composable, then the system starts a transaction, dequeues the request, and calls the task; if the call returns successfully, the transaction commits, otherwise it aborts. The transactions of the non-composable task run as independent transactions. Thus, if there is an error in the called task, one or more of the transactions of the task may commit before the error is returned to the system transaction that dequeued the request. The returned error tells the system transaction to abort, so the request is returned to the queue and, depending on the type of error, may be retried later, thereby repeating the already executed transactions. So, a SUBMIT TASK request executes at-least-once, but possibly more than once.

This semantics of invoking a non-composable task from a queue is not entirely satisfactory. However, since nested transactions are not available, for reasons explained earlier, and since a non-composable task can run more than one transaction, there's little one could do in redesigning STDL to circumvent this problem fully. For example, if only the first transaction in the non-composable task ran in the same transaction as the dequeue operation, a failure in a later transaction of the task would leave the task partially completed with no queued request to perform the incomplete work. Also note that a SUBMIT TASK request isn't all-or-nothing, no matter what semantics of "dequeue" we use.

One application of SUBMIT TASK is to send messages to workstations, some of which may be temporarily unavailable. If the TP system that manages the queue is unable to invoke the target task on the workstation because the workstation is unavailable, it will retry later. If only RPC-based communication were available, then the RPC would fail after it hit its retry limit, so it would be up to the application to retry periodically.

A SUBMIT TASK clause can explicitly name the TP system that should execute the task. This is useful if a task can execute on multiple TP systems and the caller wants to control where the submitted task runs. Using a distribution list, one can send the request to multiple TP systems, thereby causing the task to execute once on each TP system. This is useful for sending messages to workstations.

One can also define a HOLD clause, which is a *trigger* that says when the SUBMIT should invoke the target task (see Fig. 6). The trigger can ask to invoke the target when an operator command is run, when a certain time has elapsed, or when a deadline has been reached. One can use a REPEATING clause to make the SUBMIT step re-execute (after the initial execution) at regular intervals defined by the trigger.

```
<trigger> =
          FOR OPERATOR
[HOLD { FOR <delta-time>                                                  } ]
          UNTIL <absolute-time>[OF { SUBMITTER } SYSTEM ]
                                   { CLIENT    }
[ REPEATING EVERY <delta-time> ]
```

Figure 6 Trigger Syntax for Submit Task

Any task with no output parameters can be either called or submitted. STDL specifies that the input parameters for the submitted task are "pickled" (the arguments marshaled as in an RPC) and stored. When the task is to be executed, it is called using the "pickled" arguments. In effect, STDL queues the RPC message. We chose this mechanism instead of a separate queued task mechanism for flexibility and simplicity. All of the task submission logic can be handled separately and can use the called task logic to actually execute the task.

A complete STDL implementation must support the ability to forward requests from one TP system to another (which can execute the tasks). For each "server" TP system (the one that will ultimately process the request), the "client" TP system can identify the TP system that holds the enqueued request; i.e. it need not be the server or client TP system. This feature can be used, for example, by a set of workstations that don't maintain persistent queues and use an intermediate "queue server" instead.

Queue forwarding is an example of an *environmental feature*, which is not reflected in STDL syntax, but is required by any MIA-conformant implementation of STDL. This allows application writers to assume that certain functionality is available in the system, so they do not have to provide those features in application code. Other environmental features are: access control for task calls; identifying which files are recoverable and non-recoverable, timeouts for transaction execution, exchanges steps, etc.; maximum transaction restart count; and task execution priority.

8 Recoverable Exchanges

An EXCHANGE step allows a task to SEND a message to an external device, RECEIVE a message from an external device, or TRANSCEIVE (which is semantically equivalent to a SEND followed immediately by a RECEIVE). (See Fig. 7 for syntax of EXCHANGE SEND. RECEIVE and TRANSCEIVE are similar.) An EXCHANGE step is used to gather the initial input to a task that's invoked by an external device and to send and receive interactive output and input after the task has started executing. An EXCHANGE SEND or EXCHANGE RECEIVE step can be declared RECOVERABLE.

```
EXCHANGE [WITH { BROADCAST LIST <broadcast-list-name> }...]
               { [NO] RECOVERABLE WORK                }

    SEND RECORD <send-record-name> IN <presentation-group-name>

      [SENDING {<workspace-name> [,...] ]
```

Figure 7 Syntax for EXCHANGE SEND Step

A goal of designing recoverable exchanges was to isolate end users from transaction restarts as much as possible. Recoverable input for a transaction should be input once. Recoverable output from a transaction should be seen by an end user if and only if the transaction commits.

In an EXCHANGE RECEIVE step, RECOVERABLE means that if the transaction that executed the exchange aborts and restarts (i.e. executes again, from the beginning), then the EXCHANGE reuses the input it received on its first execution, rather than receiving a new input from the external device. This saves the end user having to re-enter the input. An EXCHANGE RECEIVE step can be declared RECOVERABLE only if it is the first step of a transaction. It is always non-durable.

It would be inappropriate to allow an EXCHANGE RECEIVE step to be recoverable if it comes after the first step. To see why, suppose it were allowed and suppose an EXCHANGE SEND executed sometime before the EXCHANGE RECEIVE in the same transaction. Now suppose the transaction aborts and restarts. During the re-execution, the transaction may read different data from a database (via a processing procedure), and therefore send different values in the EXCHANGE SEND. Therefore, the user of the display may want to provide different input. However, since the EXCHANGE RECEIVE was recoverable, it would reuse the values it received in its first execution, which may not be what the user wants. If the EXCHANGE RECEIVE were not preceded by an EXCHANGE SEND, then it could be moved to the first step of the transaction and thereby allowed to be recoverable.

In an EXCHANGE SEND, RECOVERABLE means that the system stores the message to be sent in a durable store, and sends the message in that store *after* the transaction commits; if the transaction aborts, then the message is deleted (i.e. nothing is sent). The semantics of a recoverable EXCHANGE SEND is at-least-once; if the TP system sends the message to the display but does not get an acknowledgment, it resends the message later even if the message was displayed but its acknowledgment was lost. A non-recoverable EXCHANGE SEND is always non-durable. An EXCHANGE SEND can broadcast to more than one display. However, a broadcast EXCHANGE SEND cannot be declared RECOVERABLE. All other EXCHANGE SEND steps can be recoverable.

A TRANSCEIVE is never recoverable or durable.

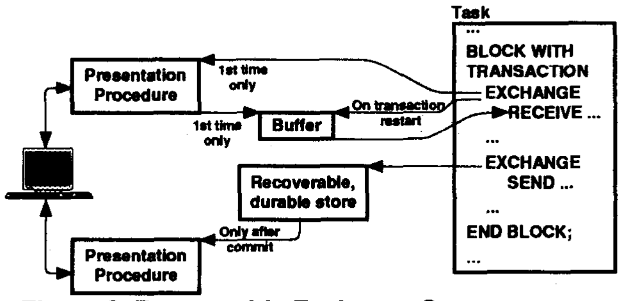

Figure 8 Recoverable Exchange Steps

An implementation of recoverable exchanges does not require implementing recoverable presentation procedures. The TP system only needs to maintain a non-durable buffer for the returned value of a recoverable EXCHANGE RECEIVE and a durable store for the values sent by recoverable EXCHANGE SENDs. See Fig. 8.

A *conversational* task is one that executes an EXCHANGE RECEIVE sometime after its first step and/or executes an EXCHANGE TRANSCEIVE. If one executes the task as a single transaction, all EXCHANGE TRANSCEIVES and the EXCHANGE RECEIVEs beyond the first one cannot be recoverable. A common way to circumvent this problem and make the entire task recoverable is to execute the task as a *pseudo-conversation* [1]. To do this, each EXCHANGE TRANSCEIVE is split into two steps, an EXCHANGE SEND followed by an EXCHANGE RECEIVE. Immediately before each EXCHANGE RECEIVE, one starts a new transaction. Now, each EXCHANGE RECEIVE and EXCHANGE SEND can be declared RECOVERABLE. The resulting task is called pseudo-conversational, since it is imitating a conversational transaction.

To be able to recover a partially executed pseudo-conversational task after a failure, the task must save its context in stable storage before committing each transaction. Since tasks normally maintain less context than C and COBOL programs, this is easier and less expensive than it would be if pseudo-conversations were implemented using TP extensions to C and COBOL — another benefit of isolating TP-specific functionality in tasks.

The advantage of a pseudo-conversational task is that it is recoverable. The disadvantage is that it no longer executes as one transaction, and therefore isn't serializable or all-or-nothing. The task isn't serializable because other transactions may be interleaved in between the transactions executing within the task. It isn't all-or-nothing because it may fail unrecoverably after the first transaction. In this case, it is too late to abort the first transaction, and there is no way to complete the execution of the task.

9 Portable Exception Handling

More than half of a typical TP application program is involved in exception handling. Therefore, to ensure that programs written in STDL are portable across different implementations, STDL provides a detailed syntax and mechanism for exceptions.

To simplify the normal processing logic of tasks, code to handle abnormal termination of clauses is segregated into *exception handlers*. An exception handler is associated with a statement, a step, or a block of statements or steps. Exception handlers can operate on workspaces and transfer control.

Upon receiving notice of an abnormal event, called an *exception*, control is passed to an associated exception handler, if one exists. Exception handlers follow the block structure of STDL: if there is no applicable exception handler on the syntactic unit where the exception was raised, then it is escalated to the next higher unit, or to the client if the next higher unit is the task itself.

An action or exception handler can be associated with a set of operations by grouping the operations in a block. If no exception handler is associated with an operation that fails, control is passed to the exception handler of the operation's surrounding block. Blocking of a set of statements allows one exception handler to handle all exceptions of the operations of the block.

Exceptions can be raised implicitly by the TP system or explicitly in tasks (by the RAISE EXCEPTION action) and processing procedures (by setting values in external variables that the TP system translates into exceptions).

The following operations can be performed by actions and exception handlers: auditing of information to an application audit log, manipulating workspace data, raising an exception, conditional operations, or transfer of control, e.g. exit the task or exit the block.

There are two types of exceptions: a *transaction exception*, which causes the current transaction to abort; and a *non-transaction exception*, which does not prevent the current transaction from committing. Transaction exceptions can only be handled by the exception handlers in statements. This type of exception handler executes in its own transaction after the transaction that caused the exception aborted. For a step, a non-transaction exception is handled in the step's exception handler, which executes in the same transaction in which the exception occurred; if the step has (or surrounding blocks have) no exception handler, then the exception escalates to become a transaction exception.

A transaction exception may be *transient*, *permanent*, or *fatal*. A transient transaction exception causes the system to retry the transaction, that is, to re-execute the transaction block. For example, a deadlock could produce a transient exception, in which case the TP system aborts one of the deadlocked transactions and retries it. More precisely, when a transient transaction exception occurs in a composable task, the task terminates and returns the exception to the task's caller. If it occurs in a non-composable task, then the TP system converts it to a permanent exception if it was raised in an exception handler, if the transaction executed a non-recoverable exchange, or if the transaction retry limit was reached. Otherwise, the retry count is incremented and the transaction (i.e. the transaction block) is re-executed.

A transaction exception that is not retryable and does not destroy the execution context of a task generates a permanent transaction exception. For a permanent transaction exception, if the task is non-composable and if there is an exception handler for the statement that executed the transaction, then the handler executes as a new transaction; in all other cases, the task is terminated and a non-transaction exception is returned to the task's caller.

A transaction exception that destroys the execution context generates a fatal transaction exception. This causes the system to abort the transaction and terminate the task in which the current transaction was started and any composable tasks called by that task as part of the current transaction. If the task is composable, then a permanent transaction exception is returned to the client. Otherwise a non-transaction exception is returned.

STDL defines various information about an exception that a TP system must put in the EXCEPTION-INFO-WORKSPACE when an exception is raised. Except where noted, this information is defined by STDL and is therefore portable. Among other things, it includes:

- an *exception class*, which classifies exception conditions based on the allowed recovery action.

- an *exception code*, which describes the exception condition in detail. Exceptions raised by an application are defined by the application and are therefore portable. For exceptions raised by the system, the exception code is not defined by STDL or the application and is therefore non-portable.

The design of STDL exception handling is inspired by that of Argus [5]. Like Argus, STDL handles named exceptions which are caught by exception handlers. However, STDL exception handling differs from Argus in several ways. First, since STDL does not support nested transactions, STDL transaction exception handlers run in top-level transactions. In Argus, they run as subtransactions. Second, STDL has a stronger emphasis on portability of exception handlers across different STDL implementations. And third, unlike Argus or Avalon [2], STDL is not designed for writing recoverable resource managers, but rather just for invoking them.

10 Implementation Strategies

During the development of the V1.0 MIA STDL specification, we and other vendors looked at possible strategies for implementing STDL on both existing and new TP systems. We summarize our view of possible implementation alternatives in this section. The descriptions are necessarily brief and assume some knowledge of the system that is the target of the implementation. Of course, each vendor is free to implement STDL in any semantically-conformant way, not necessarily using the alternatives we suggest here.

IBM CICS with conversational transactions. A compiler would translate each STDL task definition into a COBOL program that contains CICS verbs and runs as a conversational CICS transaction. Private workspaces would be mapped to the COBOL task program's working storage. Shared workspaces would be mapped to CICS workspaces. Processing procedures would be programs called by the translated COBOL task program. Presentation procedures would call CICS mapping services.

IBM CICS with pseudo-conversational transactions. This is similar to the CICS conversational transaction, but the STDL task definition would be scanned to see if the exchange steps can be moved to be immediately before or after a CICS syncpoint. If the STDL task contains recoverable exchanges, this can be done. If all non-recoverable exchanges occur in a transaction before any recoverable resources are referenced or modified, these can also be moved. If the exchanges can be moved, then the private workspace is mapped to a CICS workspace. The STDL task definition is translated into a COBOL program that uses CICS pseudo-conversations. In this model, the program takes the input, performs work, sets the next program to run, and sends the output to the terminal. The translated task COBOL program could set a marker in a workspace to indicate where in the task to continue on the next input.

OSF DCE RPC Servers. In this case, the STDL task can be translated to or interpreted in a DCE RPC server. Since DCE provides multithreading, each active task would execute as a thread. C and COBOL procedures can be run in the same multithreaded server as the STDL tasks. Or they can run in separate servers, which can be multithreaded if all resource managers are either multithreaded or execute as separate servers. For the multithreaded processing procedure case, the global variable used to return exception information must be handled either through a preprocessor or through macros in C.

11 Summary

Although most of the features of STDL appear in other languages, its combination of features and goals is unique. It is one of the few languages that embodies the *full range* of facilities needed for TP, including transactional RPC, queued requests, recoverable presentation services, and transactional exception handling. It is the only TP language we know of designed carefully for portability across underlying TP system implementations. And, via this paper, it is one of the few that are documented in the research literature, along with motivation for many of the decisions that affected its design.

12 Acknowledgments

We gratefully acknowledge the strong technical leadership of Akihiro Takagi of NTT, who directed the MIA consortium including the consortium's work on STDL. At Digital, our main collaborators on the design of STDL were Mike Gagnon, Wayne Haubner, Robert McKenzie, Hiroshi Minaguchi, Eric Newcomer, Hiroyasu Nohata, and Tom Pilitz. Many other engineers at Digital contributed, including Tony DellaFera, Bill Drury, Bob Fleming, Henry Lowe, Kiyoshi Morita, Yoshinobu Ota, Paul Ranauro, Barry Rubinson, Marc Sevigny, Al Simons, Ram Sudama, Francis Upton, Laurel Wentworth, Stephen Young, and Riaz Zolfonoon. In addition, there were dozens of contributors from other MIA consortium members, too numerous to name. We thank them all for their help. We also thank Hector Garcia-Molina and Narain Gehani for many suggestions on improving the presentation.

13 References

1. Bernstein, Philip A., Meichun Hsu, Bruce Mann "Implementing Recoverable Requests Using Queues," *1990 ACM SIGMOD Conf. on Management of Data*, ACM, NY, 112-122.

2. D.L. Detlefs, M.P. Herlihy, and J.M. Wing. "Inheritance of synchronization and recovery properties in Avalon/C++" *IEEE Computer* 21, 12 (Dec. 1988), pp. 57-69. Also in *Advanced Language Implementation Techniques*, Peter Lee (editor), MIT Press, 1990.

3. "Forms Interface Management System," ISO/IEC DIS 11730, International Organization for Standardization, Dec. 7, 1992.

4. Gray, Jim and Andreas Reuter, *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann, San Mateo, California, 1992.

5. Liskov, Barbara and Robert Scheifler, "Guardians and Actions: Linguistic Support for Robust, Distributed Programs," *ACM Trans. on Prog. Lang. and Systems 5, 3* (July 1983), 381-404.

6. *Technical Requirements, Multivendor Integration Architecture Version 1.2*, Vol. 1-11, TR 550001, Nippon Telegraph and Telephone Corporation, Jan. 29, 1993. They can be obtained from NTT at any of the following addresses:

   NTT America, Inc. 101 Part Avenue, 41st Floor New York, NY 10178 U.S.A.
   Telephone: +1 212-661-0810 FAX: +1 212-661-1078

   NTT Europe, Ltd. Level 9, City Tower 40 Basinghall Street London, EC2V5DE United Kingdom.
   Telephone: +44 71-256-7151 FAX: +44 71-256-7997

   NTT Inc. 1-6, Uchisaiwaicho 1-Chome Chiyoda-ku, Tokyo 100 Japan. Telephone: +8 1-33-509-3101 FAX: +8 1-33-580-9104

7. Rosenberry Ward, David Kenney, Gerry Fisher, *Understanding DCE*, O'Reilly & Assoc., Sebastapol, California, 1992.

8. "Information Processing Systems - Open Systems Interconnection - Distributed Transaction Processing," (Part 1, Model, ISO/IEC 10026-1:1992; Part 2, Service Definition, ISO/IEC 10026-2:1992; Part 3: Protocol Specification, ISO/IEC 10026-3:1992), International Organization for Standardization, 1992.

9. Speer, Thomas G. and Mark W. Storm, "Digital's Transaction Processing Monitors," *Digital Technical Journal 3,1* (Winter 1991), 18-33.

10. "Distributed Transaction Processing: The TxRPC Specification," X/Open Snapshot, ISB:1-872630-81-2 S218, The X/Open Company Ltd., Reading, U. K., Dec. 1992.

11. "Distributed Transaction Processing: The TX (Transaction Demarcation) Specification," ISB:1-872630-65-0 P209, X/Open Preliminary Specification, The X/Open Company Ltd., Reading, U. K, Oct. 1992.

14 Example Program

```
! This program pays a credit card bill from a demand
! deposit (checking) account.

! Define the messages that this application requires
!
MESSAGE GROUP billing-messages
  LANGUAGE IS ENGLISH;
  success-msg VALUE IS 41 CLASS IS NO-OUTPUT-ERROR
     TEXT IS "Transaction completed.";
  no-funds-msg VALUE IS 42 CLASS IS NO-OUTPUT-ERROR
     TEXT IS "Error: Insufficient funds.";
END MESSAGE GROUP;
!
! Define the records for the task
!
RECORD cc_wksp ! credit card information
 acct_num     INTEGER;
 amount_due INTEGER;
END RECORD;

RECORD dda_wksp ! demand deposit account information
 acct_num        INTEGER;
 amount_due      INTEGER;
 balance         INTEGER;
END RECORD;

RECORD ctrl_wksp
 success         TEXT SIZE 1;
 msg             TEXT SIZE 80;
END RECORD;

RECORD input_wksp
 cc_acct_num     INTEGER;
 dda_acct_num    INTEGER;
END RECORD;
```

```
TASK pay_bill
 WORKSPACES ARE cc_wksp, dda_wksp,
                ctrl_wksp, input_wksp;

txn_1:
  BLOCK WITH TRANSACTION
    !
    ! Get credit card and direct deposit account number
    ! from user
    !
    EXCHANGE WITH RECOVERABLE WORK
      RECEIVE RECORD acct_info IN input_form
        RECEIVING input_wksp;

    PROCESSING MOVE input_wksp.cc_acct_num
                TO cc_wksp.acct_num;

    PROCESSING MOVE input_wksp.dda_acct_num
                TO dda_wksp.acct_num;
    !
    ! Pay credit card debt
    PROCESSING CALL PROCEDURE pay_cc
      IN credit_proc_group USING cc_wksp;

    PROCESSING MOVE cc_wksp.amount_due
                TO dda_wksp.amount_due;
    !
    ! Debit checking account with credit card debt
    !
    PROCESSING CALL PROCEDURE debit_dda
      IN checking_proc_group USING dda_wksp, ctrl_wksp
      ACTION IS
        IF (ctrl_wksp.success = "N")
        THEN RAISE EXCEPTION CODE 42
          WITH ROLLBACK TRANSACTION;
        ELSE MOVE "Transaction completed."
              TO ctrl_wksp.msg;
        END IF;
      END ACTION;
    !
    ! Transaction was successful. Display credit card
    ! amount, new checking acount balance, and
    ! a success message
    !
    EXCHANGE WITH RECOVERABLE WORK
      SEND RECORD result_info IN billing_forms
        SENDING dda_wksp, ctrl_wksp
      ACTION IS EXIT TASK;
        ! Transaction successful.  End task.
      END ACTION;

  END BLOCK ! txn_1
  !
  ! Exception occurred - Get the message associateed with
  ! the exception. This can be due to insufficient funds or
  ! other errors. EXCEPTION-CODE and EXCEPTION-SOURCE
  ! are STDL-defined fields available to exception handers.
  !
  EXCEPTION HANDLER IS
    GET MESSAGE NUMBER EXCEPTION-CODE
      SOURCE EXCEPTION-SOURCE INTO ctrl_wksp.msg;
  END EXCEPTION HANDLER;

txn_2:
  BLOCK WITH TRANSACTION
  !
  ! Transaction unsuccessful - display credit card
  ! amount due, existing checking acount balance
  ! and a failure message
  !
  EXCHANGE WITH RECOVERABLE WORK
    SEND RECORD minus_info IN error_form
      SENDING dda_wksp, ctrl_wksp;

  END BLOCK; ! txn_2

END TASK;
```

C Program Starts Here:

```
/** Processing Procedure for PAY_CC and DEBIT_DDA **/

/* Include the workspace layouts generated from     */
/* the record definition by the STDL compiler.      */
#include "billing.h"

/***************************************************************/
/** Functional Description:                                  **/
/**     Routine pay_cc() --                                  **/
/**      retrieves amount due for credit card bill **/
/**      and sets amount due to zero                 **/
/***************************************************************/

void pay_cc (struct cc_wksp *ccwksp) {

  EXEC SQL SELECT amount
           INTO :cc_wksp->amount_due
           FROM credit_card
           WHERE acct_no = :cc_wksp->acct_num;

  EXEC SQL UPDATE credit_card
           SET amount_due = 0
           WHERE acct_no = :cc_wksp->acct_num;
  return;
}

/**********************************************************************/
/** Functional Description:                                          **/
/** Routine debit_dda() --                                           **/
/**    Retrieve balance from demand deposit account. **/
/**    If sufficient funds are available, then update    **/
/**    the demand deposit account.                       **/
/**********************************************************************/

void debit_dda (struct dda_wksp *ddawksp,
                struct ctrl_wksp *ctrlwksp) {
  EXEC SQL SELECT balance
           INTO :ddawksp->balance
           FROM accounts
           WHERE acct_no = :ddawksp->acct_num;
  if (ddawksp->balance < ddawksp->amount_due) {
             ctrlwksp->success = 'N';
             return; /* failure return without update */
             }
  EXEC SQL
         UPDATE accounts
         SET balance = balance - :ddawksp->amount_due
         WHERE acct_no = :ddawksp->acct_num;
  ctrlwksp->success = 'Y';
  return;
}
```