STDL
A Portable Language for Transaction Processing

Philip A. Bernstein[1], Per O. Gyllstrom[2], Tom Wimberg

Digital Equipment Corporation
Cambridge Research Lab

CRL 93/5 March 8, 1993

Structured Transaction Definition Language (STDL) is a language for distributed transaction processing, developed by the Multivendor Integration Architecture consortium. STDL is a block-structured language, specialized for transaction processing. It is designed for application portability across multiple STDL implementations on different vendors' transaction processing systems. This paper describes STDL's transaction features: transaction bracketing, 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 SQL databases and files. All transactional features of STDL and new features outside standard C and COBOL are isolated in procedures written in the STDL language. These procedures are called tasks. This isolation of transactional features is quite different than other persistent programming languages: one can use applications written in standard C or COBOL; and to implement STDL, it is possible to map clauses of task language onto operations of most any distributed TP monitor.

Keywords: transactions, transaction processing, TP monitor, remote procedure call, queuing, exception handling

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

[2]Address for Per Gyllstrom and Tom Wimberg: Digital Equipment Corporation, 151 Taylor Street, Littleton, MA 01460-1407

INTRODUCTION

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 goal to achieve application portability, interoperability, and a common user operation environment. The MIA consortium is sponsored and run by Nippon Telegraph and Telephone (NTT) of Japan. Digital Equipment Corp., Fujitsu Ltd., Hitachi, IBM, NEC and NTT's systems integration subsidiary, NTT Data, were the members of this consortium during the creation of MIA Version 1 [MIA].

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 include STDL, C, COBOL, FORTRAN, and SQL. Protocols for both OSI and Internet are in the specification, including transactional remote-procedure calls (OSI only), file transfer, electronic mail, and network management protocols. Human interface specifications address GUI window-based interfaces. The lack of international standards in this area prompted MIA to adopt three *de facto* standards: OSF Motif, IBM CUA, and AT&T Open Look. The use of MIA style guides ensure the same look and feel across different presentation services.

The MIA specifications provide an integration of existing *de facto* and *de jure* standards along with the provision of a new API for TP applications. The first version of MIA was completed in the spring of 1991. Today, MIA Version 1.1 is available from NTT (the address appears with the references). The MIA specifications are the basis for procurement by NTT and are gaining interest in many other industries. It is expected that an expanded MIA consortium will be created during 1993 to work on Version 2 of the architecture.

For transaction processing, the MIA consortium developed STDL. This decision was made due to the lack of any mature TP API standard. STDL is based on the Task Definition Language, TDL, of Digital's TP monitor, ACMS [Speer and Storm] [Gray and Reuter]. The selection of ACMS as the model for developing the API for MIA was based on the ability to isolate TP-specific functionality, to make use of standard C, COBOL and SQL, and to layer such an API on top of other TP monitors. The completeness criteria for STDL included a thorough implementability study by each member of the MIA consortium to verify that STDL could be implemented on each of their platforms, including IBM's CICS. Although STDL was modeled on TDL, it extends and differs from TDL in many ways, both in major features and syntactic detail.

STDL is a portable block-structured language, specialized for TP. Procedures written in STDL are called tasks. Key features of STDL include:

- ability to call COBOL and C programs, which can access relational databases (using embedded SQL) and stream, relative, indexed and indexed sequential files
- transaction bracketing
- RPC-based transactional communication
- structured exception handling, portable across STDL implementations

- queued task submission
- standard interface to presentation services based on ISO Forms Interface Management System (FIMS)
- recoverable presentation service data exchanges
- recoverable workspaces
- specification of environmental information required for full application portability (i.e., features that the programmer can count on but that are not reflected in the API)

STDL relies on standard C and COBOL for most application logic and all operations on SQL databases and files[1]. All transactional features of STDL and new features outside standard C and COBOL are isolated in tasks written in the STDL language. This isolation of transactional features is quite different than other persistent programming languages. It allows one to use applications written in standard C or COBOL. It also simplifies the implementation of STDL; one just maps clauses of task language onto operations of most any TP monitor.

The MIA transactional remote procedure call (RPC) protocol, *Remote Task Invocation* protocol *(RTI),* has been adopted by X/Open [X/Open1]. RTI is an integration of OSF RPC [WKF] as the task-to-task data transfer protocol and the OSI TP standard for two-phase commitment [OSI]. RTI has also been specified for layering on TCP/IP and IBM's SNA LU6.2. The specification of STDL includes a detailed mapping of STDL semantics 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*. In this environment, applications in standard COBOL, C and FORTRAN run on small workstations and/or PCs. The extensions add interfaces to these 3GLs to invoke TP applications, using STDL syntax and semantics and the RTI protocol, and to interact with presentation services, using syntax and semantics based on FIMS.

This paper focuses on the transactional aspects of STDL. It begins with a short summary of the non-transactional parts of STDL. This is followed by a description of STDL's transaction features: transaction bracketing, transactional remote procedure call, transactional queuing, recoverable terminal I/O, and transactional exception handling. For a full language specification, see [MIA].

STDL Overview

A *TP system* is a uniquely named entity that executes STDL applications (see fig. 1). Each TP system has a task queue, an audit log and a set of named record queues. These entities are described later in the paper.

STDL applications are divided into three parts: tasks, presentation procedures, and processing procedures. A *task* is a procedure written in STDL. 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

[1] The MIA specification includes modifications to C, COBOL, and SQL, primarily to handle Japanese well.

with other tasks *(shared workspaces)*. A *client program* is any program that invokes a task; it can be a task, a TP system component, or an external agent.

A *processing procedure* is called by a task to perform computations and access files or relational databases. Processing procedures, written in standard C or COBOL, 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.) Nested calls to other procedures written in C or COBOL is allowed.

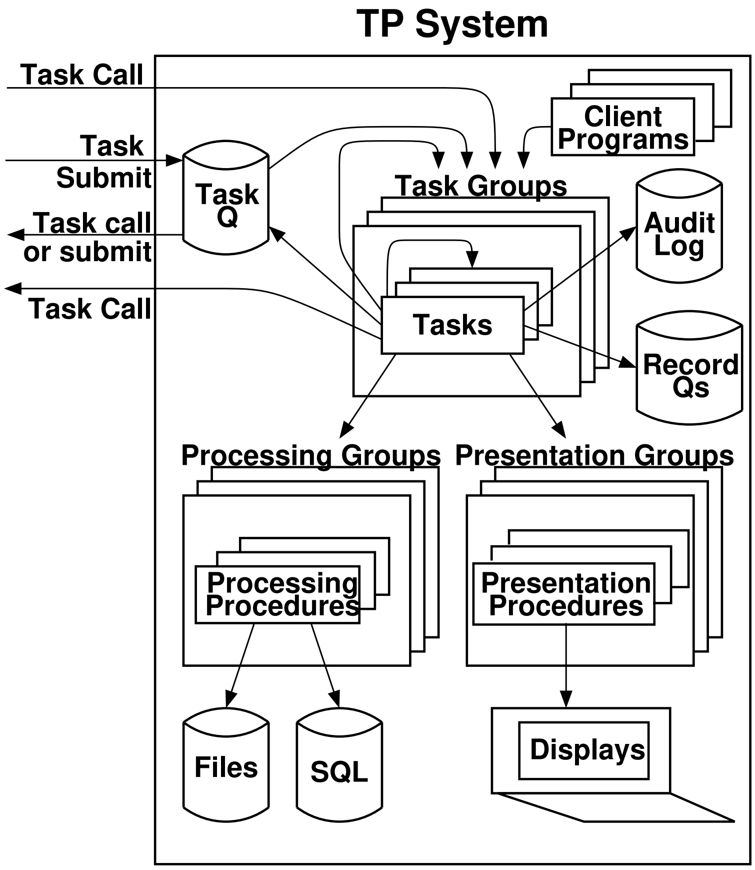

Figure 1 TP System Model

A task invokes a *presentation procedure* to interact with an external device, called a *display,* such as a terminal, PC, workstation, bar-code-reader, or automated teller machine. Tasks interact with presentation procedures to obtain input from or provide output to displays. Presentation procedures can be part of a forms package or can be a set of programs written in standard C or COBOL. Presentation procedures do not participate in transactions.

Structuring applications into three parts is designed to isolate standard code and portable code, and to make the maximum usage of current standards and thereby minimize the amount of new language specification. The data processing part of the application is written in standard C, COBOL, and SQL. The TP concerns are isolated in the STDL task, which allows STDL implementors to map the higher level TP concepts onto their current TP system implementations: transaction bracketing and recovery, exception handling, communications, access to data queues, submission of queued work requests, and invocation of presentation services. Due to a lack of existing standards, end-user presentation services are not portable and are therefore isolated into presentation procedures. Some amount of interoperability with different presentation services was attempted based on the FIMS work. 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.

Transactions

STDL supports a *flat* (non-nested) transaction model. In this transaction model, two transactions cannot be nested. Each transaction commits and 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 to incorporate state-of-the-art features with the practicality of what products were likely to 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. However, we see no serious 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.

STDL supports the following resources (accessible from tasks, except as noted):

- task workspace (recoverable and non-recoverable, non-durable)
- shared workspace (recoverable, non-durable)
- SQL database (recoverable, durable) - C and COBOL access only
- indexed file (recoverable and non-recoverable, durable) - COBOL access only
- relative file (recoverable and non-recoverable, durable) - COBOL access only
- sequential file (recoverable or non-recoverable, durable) - COBOL access only
- stream file (non-recoverable, durable) - C access only
- recoverable send exchange (recoverable, durable)
- recoverable receive exchange (recoverable, non-durable)
- task queue (recoverable, durable)
- record queue (recoverable, durable and non-durable)
- audit log (non-recoverable, durable)

An exchange is an operation that calls a presentation procedure. A task queue stores messages (for task invocation) and a record queue stores data. An audit log is a sequential file for storing interesting events. Exchanges and queues are described in detail later in the paper.

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

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 include task arguments, task workspaces, and the ‘‘send display’’ (if different from the default display). A task can

- commit or abort a transaction
- call processing procedures, presentation procedures, and other tasks
- enqueue and dequeue data on persistent and volatile queues
- raise and handle exceptions
- evaluate Boolean conditions and execute loops
- write to an audit log
- translate a message code into message text

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. A task group 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 within the same transaction.

Recall that *workspaces* are variables. *Private workspaces* are local to the task that declares them. *Shared* workspaces are global, that is, they can be read from or written to by a set of tasks. Private workspaces can be recoverable or non-recoverable. Shared workspaces must be recoverable. However, workspaces are not durable.

There are three types of executable clauses: statements, steps and actions. *Statements* control transactions and the sequencing between transactions. *Steps* perform major operations within transactions: calling and queuing tasks, calling procedures, performing exchanges, enqueuing and dequeuing data, performing concurrent steps, executing blocks, and operating on workspaces. *Actions* perform work within steps after the major operation has been done: operations on workspaces, raising exceptions, and transferring control.

Statements and steps can be grouped together in blocks. There are three types of blocks: statement blocks, step blocks and transaction blocks. A *statement block* is a set of statements (transactions); a *step block* is a set of steps; a *transaction block* is a set of steps that executes as one transaction. Each transaction block starts and ends in the same task. Blocks define the scope of actions and of exception handlers.

STDL has four kinds of conditional operations: if, while, select-first, and control-field. The latter two are types of case statements. Select-first uses Boolean expressions to determine the next executable operation; control-field uses the value of a workspace field to determine the next executable operation. Conditionals are allowed as statements, as steps, as actions or as operations of exception handlers (excluding the while conditional).

A *concurrent-block step* allows concurrent execution of two or more steps. The use of concurrent-block steps allows multiple synchronous calls of other tasks and processing procedures to take place as part of the same transaction.

Transaction Bracketing

STDL tasks use a *chained* transaction model: as soon as one transaction commits or aborts, another transaction is started. No accesses of recoverable resources occur outside of a transaction. This is guaranteed by the STDL syntax: recoverable resources can only be accessed by steps or procedures called by steps; steps can only be specified in transaction blocks or composable tasks (which are defined below); transaction blocks can be contained in other statements (such as an IF statement); however, statements can only read non-recoverable workspaces, so the transaction in which they execute is unimportant. This chained model is comparable to the transaction model in IBM's CICS, where the syncpoint operation both ends one transaction and starts the next.

In contrast to a chained transaction model, a non-chained model allows operations to execute outside a transaction. Each transaction is explicitly bracketed, e.g. by start-transaction and either commit or abort. Any operation outside these brackets does not execute as part of a non-chained model. For example, the X/Open transaction bracketing interface, called *tx,* uses a non-chained model [X/Open2]. Although a non-chained model appears to be more flexible, we see no use for operations outside transactions. If an operation accesses a recoverable resource, it must be part of a transaction, possibly a one-step transaction in STDL. If an operation does not access a recoverable resource, then it makes no difference whether the operation is or is not part of a transaction. So, a non-chained model simply gives the programmer an opportunity to make a mistake by inadvertently executing an operation that accesses recoverable resources outside a transaction. For this reason, we believe the chained model is superior.

Task Call

A task can invoke other tasks synchronously by using a CALL TASK step (see Fig. 2). The execution model is that of remote procedure call. That is, the syntax of CALL TASK is insensitive to whether the callee is local or remote. So distribution is an environmental consideration determined after the application is written.

One can execute a called task in the same transaction as the caller or in a separate transaction (see fig. 3). This option is denoted in the call step by the phrase WITH DEPENDENT WORK (to be part of the same transaction) or WITH INDEPENDENT WORK (to be part of a different transaction). A task that’s called WITH DEPENDENT WORK must have the keyword COMPOSABLE in its definition and consists only of steps and actions. It does not include statements, because a composable task may not start a new transaction. By contrast, a non-composable task must be a sequence of statements.

<u>PROCESSING</u> WITH [{ <u>INDEPENDENT</u> / <u>DEPENDENT</u> } WORK]

<u>CALL TASK</u> <task-name> <u>IN</u> <task-group name>

[AT <destination-name> / <distribution-list-name>] [<u>USING</u> { <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 [MIA]).

Figure 2 Syntax for Call Task

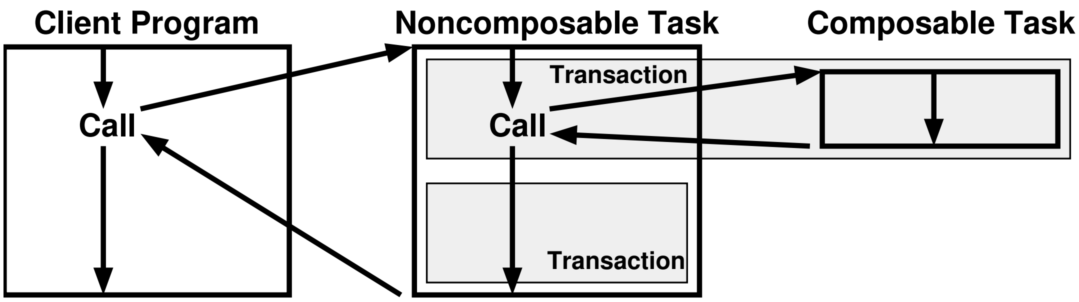

Figure 3 Composable and Non-Composable Tasks

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, whether or not 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 to supply only the COMPOSABLE (or NON-COMPOSABLE) task attribute in the task interface definition of the called task. The syntax is supplied for consistency reasons.

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 can be delivered. We found no useful and implementable semantics. We believe dropping the reply is unsatisfactory, since the called task committed and its results may be needed. The best alternative seems to be treating the orphaned reply message 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, which would complicate the language for a feature that isn't often used. In the end, this type of call was made illegal.

A task can also invoke 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. The main reason for this is that STDL does all transaction bracketing in the task. No transaction bracketing is done in processing procedures, partly because there was no standard general transactional bracketing API available to processing procedures (when STDL was developed). SQL specific transaction bracketing verbs are also excluded since they only apply to SQL operations.

Note that if a processing procedure accesses no recoverable resource, an implementation is free to optimize STDL task execution by not actually starting a transaction. Whether or not an implementation can detect this is problematic. A simpler related case that an implementation *can* detect is a transaction block that only calls non-recoverable exchange steps; such a block need not run as a transaction.

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.

Submit Task

A task can invoke another task without waiting for the results by using a SUBMIT TASK step (see fig. 4). This creates a request (i.e., message) to execute the task and puts 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 will not return to the submitting task, it must have no return parameters.

The effect of the SUBMIT TASK step, i.e. creating and putting the request on a task queue, is recoverable. So its effect is made permanent if and only if the transaction that executes it commits. STDL allows the SUBMIT TASK step to be executed as part of the current transaction, denoted in the step using WITH DEPENDENT WORK, or in a separate transaction, denoted using WITH INDEPENDENT WORK.

A submitted task is guaranteed to execute at least once. A task invoked by SUBMIT TASK 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 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.

```
PROCESSING WITH [ { INDEPENDENT } WORK ]
                    { DEPENDENT   }
   SUBMIT TASK <task-name> IN <task-group name> [ AT <destination-name>
                                                     <distribution-list-name> ]
         [ HOLD { FOR OPERATOR
                  FOR <delta-time>
                  UNTIL <absolute-time> [ OF { SUBMITTER } SYSTEM ] } ]
                                             { CLIENT    }
         [ REPEATING EVERY <delta-time> ]
         [ USING { <workspace-name>} [,...]
```

Figure 4 Syntax for Submit Task

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 part of 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.

In a SUBMIT step, one can define a HOLD clause, which is a *trigger* that says when the SUBMIT should invoke the target task. 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 REPEAT clause to make the SUBMIT step re-execute (after the initial execution) at regular intervals defined by the trigger.

A SUBMIT TASK statement 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 also send the request to multiple TP systems, thereby causing the task to execute once on each TP system.

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 should enqueue the 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 in the MIA specification. The feature is not reflected in STDL syntax, but the functionality 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. There are many other environmental features, such as: 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.

Recoverable Exchanges

An EXCHANGE step allows one to SEND a message to an external device, RECEIVE a message from an external device, or both (TRANSCEIVE). (See fig. 5 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> WORK }]
               { [NO] RECOVERABLE WORK                     }
     SEND RECORD <send-record-name> IN <presentation-group-name>
         [SENDING {<workspace-name> [,...] ]
         [WITH { RECEIVE CONTROL <receive-control-text> } ]
               { SEND CONTROL <send-control-text>       }
```

Figure 5 Syntax for EXCHANGE SEND Step

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. 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 display had already processed the message 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 always non-recoverable and non-durable.

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. 6.

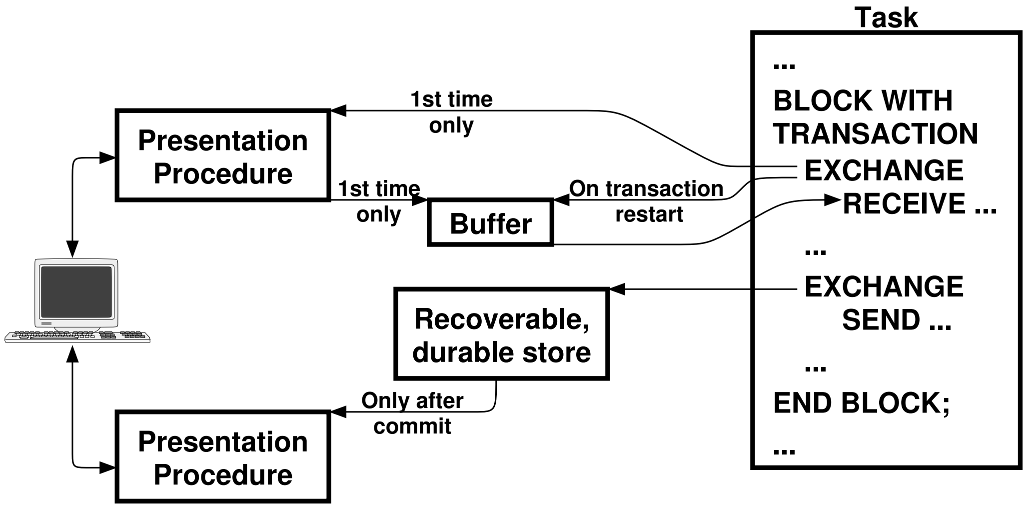

Figure 6 Recoverable Exchange Steps

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, the EXCHANGE RECEIVE steps beyond the first one cannot be recoverable. EXCHANGE TRANSCEIVE steps are never recoverable. A classical way to circumvent this problem and make the entire task recoverable is to execute the task as a pseudoconversation [BHM]. To do this, each EXCHANGE TRANSCEIVE is split into two steps, an EXCHANGE SEND followed by an EXCHANGE RECEIVE. Then, 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 said to be *pseudo-conversational*, since it is imitating a conversational transaction.

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. The task isn't all-or-nothing because the task 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.

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.

Actions perform housekeeping work upon normal completion of the associated statement or step. Operations performed by actions can also be performed by *exception handlers*. An exception handler is associated with a statement, a step, or a block of statements or steps. Upon receiving notice of abnormal termination of processing, called an *exception,* control is passed to an associated exception handler, if one exists. Exception handlers follow the block structure of STDL. So, if there is no applicable exception handler on the 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 syntactic unit is the task itself.

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

By creating a block, an action or exception handler can be associated with a set of operations. If no exception handler is associated with an operation that fails, control is passed to the exception handler associated with 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.

The following operations can be performed by actions and exception handlers: auditing of information to an application audit log, manipulation of 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. Non-transaction exceptions can be handled in exception handlers in steps which execute in the same transaction in which the exception occurred. If no exception handler is defined for the step that experienced the non-transaction exception, then the exception escalates to become a transaction exception. In this section, we will focus on transaction exception handling. Non-transaction exception handling is similar.

A transaction exception may be *transient*, *permanent*, or *fatal*. A transient transaction exception causes the system to retry the current transaction, that is, to re-execute the block that demarcates the transaction. 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 block defining the transaction) 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. A fatal transaction exception causes the system to abort the transaction and to 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 a variety of information about an exception, which the TP system puts in the EXCEPTION-INFO-WORKSPACE when an exception is raised. Except where noted, this information is defined by STDL and is therefore portable. It includes:

- an *exception class*, which classifies exception conditions into 18 classes based on the allowed recovery action. It specifies whether
  - the exception may be fixed by retrying the operation (has ''ERROR'' in the class name) or only by changes to source code (has ''FAULT'' in the class name).
  - the error caused by a problem in the environment or in the application (''ENV'' vs. ''AP'' in the class name). This does not cover all exceptions.
  - the error in the invocation or the execution of a task (''INVOCATION'' vs. ''EXECUTION'' in the class name). This does not cover all exceptions.
  - the error a type of timeout (''TIMEOUT'' in the class name)

  The exception class is intended to be used by the exception handler to determine the general nature of the actions it should take.
- an *exception source*, which specifies whether the exception was raised by the application or the TP system.
- an *exception code*, which describes the exception condition in detail. Exception codes that have an application exception source are defined by the application and are therefore portable.

  For system exception sources, the exception code is not defined by STDL or the application, and is therefore non-portable. It is accompanied by an error code group, which is a Universal Unique IDentifier (UUID). Each type of system has a different UUID. The specific interpretation of these error codes is dependent on the error code group, which is different for each TP system implementation (and each TP system implementation may use more than one error code group).

- an *exception group,* for exceptions with a system source, which tells the system implementation type, so the exception handler can determine whether it can interpret the system exception code. This is not defined by STDL and is therefore non-portable.
- an *exception level*, which tells whether the exception arose in the current task or in a called task or procedure.
- an *exception location*, which gives a text description of where the exception occurred.

The design of STDL exception handling is inspired by that of Argus [Liskov and Scheifler]. 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 [DHW], STDL is not designed for writing recoverable resource managers, but rather just for invoking them.

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 one of the few TP languages (perhaps the only one) 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.

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 to the design, 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.

References

[BHM] Bernstein, Philip A., Meichun Hsu, Bruce Mann ‘‘Implemented Recoverable Requests Using Queues,’’ *1990 ACM SIGMOD Conf. on Management of Data*, ACM, NY, 112-122.

[DHW] 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.

[FIMS] ‘‘Forms Interface Management System,’’ ISO/IEC DIS 11730, International Organization for Standardization, Dec. 7, 1992.

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

[Liskov and Scheifler] 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), pp. 381-404.

[MIA] *Technical Requirements, Multivendor Integration Architecture Version 1.1*, Vol 1-11, TR 550001, Nippon Telegraph and Telephone Corporation, March 31, 1992.

[WKF] Rosenberry Ward, David Kenney, Gerry Fisher, *Understanding DCE*, O’Reilly & Assoc., Sebastapol, California, 1992.

[OSI] ‘‘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.

[Speer and Storm] Speer, Thomas G. and Mark W. Storm, ‘‘Digital’s Transaction Processing Monitors,’’ *Digital Technical Journal 3,1* (Winter 1991), 18-33.

[X/Open1] ‘‘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.

[X/Open1] ‘‘Distributed Transaction Processing: The TX (Transaction Demarcation) Specification,’’ ISB:1-872630-65-0 P209, X/Open Preliminary Specification, The X/Open Company Ltd., Reading, United Kingdom, Oct. 1992.

Note: MIA Specifications, 11 volumes, can be obtained by contacting 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

A subset of the MIA Specifications, including MIA Overview and STDL, RTI, and Interactive Processing Environment specifications can be obtained directly from Digital Equipment Corporation. The Digital part number for the subset is EK-TPSMI-KT-001. Orders can also be placed through DECdirect in U.S. (1-800-DIGITAL).