Extensions to Standard ML to Support Transactions

Jeannette M. Wing, Manuel Faehndrich, J. Gregory Morrisett, and Scott Nettles*
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

A transaction is a control abstraction that lets programmers treat a sequence of operations as an atomic ("all-or-nothing") unit. This paper describes our progress on on-going work to extend SML with transactions. What is novel about our work on transactions is support for multi-threaded concurrent transactions. We use SML's modules facility to reflect explicitly orthogonal concepts heretofore inseparable in other transaction-based programming languages.

1 Revisiting Transactions

1.1 Separation of concerns

Transactions are a well-known and fundamental control abstraction that arose out of the database community. They have three properties that distinguish them from normal sequential processes: (1) A transaction is a sequence of operations that is performed *atomically* ("all-or-nothing"). If it completes successfully, it *commits*; otherwise, it *aborts*; (2) concurrent transactions are *serializable* (appear to occur one-at-a-time), supporting the principle of isolation; and (3) effects of committed transactions are *persistent* (survive failures). Systems like Tabs [8] and Camelot [3] demonstrate the viability of layering a general-purpose transactional facility on top of an operating system. Languages such as Argus [4] and Avalon/C++ [2] go one step further by providing linguistic support for transactions in the context of a general-purpose programming language. In principle programmers can now use transactions as a unit of encapsulation to structure an application program without regard for how they are implemented at the operating system level.

In practice, however, transactions have yet to be shown useful in general-purpose applications programming. One problem is that state-of-the-art transactional facilities are so tightly integrated that application builders must buy into a facility *in toto*, even if they need only one of its services. For example, the Coda file system [7] was originally built on top of Camelot, which supports distributed,

*This research was sponsored by the Avionics Lab, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U. S. Air Force, Wright-Patterson AFB, OH 45433-6543 under Contract F33615-90-C-1465, Arpa Order No. 7597.

concurrent, nested transactions. Coda needs transactions for storing "metadata" (e.g., inodes) about files and directories. Coda is structured such that updates to metadata are guaranteed to occur by only one thread executing at a single-site within a single top-level transaction. Hence Coda needs only single-site, single-threaded, non-nested transactions, but by using Camelot was forced to pay the performance overhead for Camelot's other features.

The Venari Project at CMU is revisiting support for transactions by adopting a "pick-and-choose" approach rather than a "kit-and-kaboodle" approach. Ideally, we want to provide separable modules to support transactional semantics for different settings, e.g., in the absence or presence of concurrency. Programmers are then free to compose those modules supporting only those features of transactions they need for their application.

1.2 Non-traditional applications

A second problem with existing transactional facilities is that they have been designed primarily with applications like electronic banking, airline reservations, and relational databases in mind. Non-traditional applications such as proof support environments, software development environments, and CAD/CAM systems want transactional features, most notably data persistence, but have different performance characteristics. For example, these applications do not manipulate simple database records but rather complex data structures such as proof trees, abstract syntax trees, symbol tables, car engine designs, or VLSI designs. Also, users interact with these data during long-lived "sessions" rather than short-lived transactions; indeed we can view a "session" itself as a sequence of transactions. For example, during a proof session a user might explore one path in a proof tree transactionally; if the path begins looking like a dead-end the user may choose to abort, backing all the way up to the first node in the path or perhaps to some intermediate node along the way. Also, though multiple users may need to share these data, simultaneous access might be less frequent. For example, proof developers might work on independent parts of a proof tree, perhaps each proving auxiliary lemmas of the main theorem; software developers might modify different modules of a large program. Finally, these non-traditional applications typically support different update patterns. Whereas travel agents make frequent updates to airline reservations databases, we do not expect to make updates as frequently to proofs of theorems saved in proof libraries.

The Venari Project's application domain is software development environments; one specific problem we are addressing is searching large libraries, e.g., specification and program libraries, used in the development of software systems. We imagine the scenario in which a user searches a large library for a program module that "satisfies" a particular specification. We might wish to perform each query as a transaction, for example, to guarantee isolation from any concurrent update transaction or to abort the query after the first n modules are returned.

Our effort to support a "pick-and-choose" approach for transactions has the advantage of providing us with a way to take performance measurements on different combinations of our separable modules. We have the potential to do different kinds of performance tuning for the non-traditional applications we hope to support.

1.3 Contributions of this paper

SML's modules system lets us cleanly exemplify our "pick-and-choose" approach. In the case of single-site, single-threaded, nested transactions, we support separately the persistence and undoability properties of transactions in terms of two different modules; we then compose them to build a module for transactions [6]. We reported on this work at the Pittsburgh 1991 ML workshop. Also, along with others at Carnegie Mellon, we have separately designed and built a threads package for SML/NJ [1]. We reported on this work at the Edinburgh 1990 ML workshop. [1]

This paper reports on our progress for combining our support for threads and that for transactions. We address concurrency, in two ways: making an individual transaction multi-threaded and allowing multiple transactions to run concurrently. To the transaction and database community, our work is novel because it is the first to cast within a programming language a model of computation that supports multi-threaded transactions. To the programming language community, our work is among the few to extend the functional programming paradigm to support a traditionally imperative feature. To the SML community, our application of the modules facility should be of particular interest.

In the rest of this paper, we motivate the desire to keep threads and transactions as separate control abstractions (Section 2), give a snippet of the SML programmer's interface to multi-threaded concurrent transactions (Section 3), describe some details of our design (Section 4), discuss what features of SML helped in our design and implementation effort (Section 5), and close with a list of future work (Section 6).

We emphasize that this paper describes on-going work. We are taking a pragmatic, bottom-up approach; by prototyping individual features (e.g., persistence, undoability, read/write locking, nesting, threads, and transactions) incrementally and then combining them in various ways, we can explore a rich design space. This paper focuses on the hardest combination: threads and transactions. (Examples of other reasonable combinations are "multi-threaded persistence" and "multi-threaded undoability.") Thus, for now we are concerned with providing efficient enough run-time *mechanism* to give systems builders flexibility in deciding *policy*. Although the design described in Section 4 does reflect one particular policy, our runtime mechanism is general enough to support alternate policies. Finally, we have not thought greatly about the "ideal" programming interface to provide the SML end-user, but look forward to designing one in the near future.

2 Keeping Threads and Transactions Separate

In languages like Argus and Avalon, a single thread of control is associated with each transaction. But threads and transactions are orthogonal control abstractions. So, we would like to relax the restriction of identifying threads and transactions by allowing multiple threads of control to execute within, and on behalf of, a single transaction.

Figures 1a and 1b depict the traditional model, where we use a wavy line to denote a thread and a box to denote a transaction; time moves from left to right. Figure 1a shows a single thread executing, first entering a transaction and then leaving successfully (i.e., committing). Figure 1b shows two single-threaded transactions executing concurrently. Figure 1c depicts our new model where multiple threads execute within a single transaction. And finally, Figure 1d depicts multi-threaded concurrent transactions, the "composition" of Figures 1b and 1c. The goal of Venari's version of SML is to support Figure 1d through module composition.

[1] For this paper to be somewhat self-contained, we include the cited Pers, Undo, Trans, and Threads signatures in Appendices A and B.

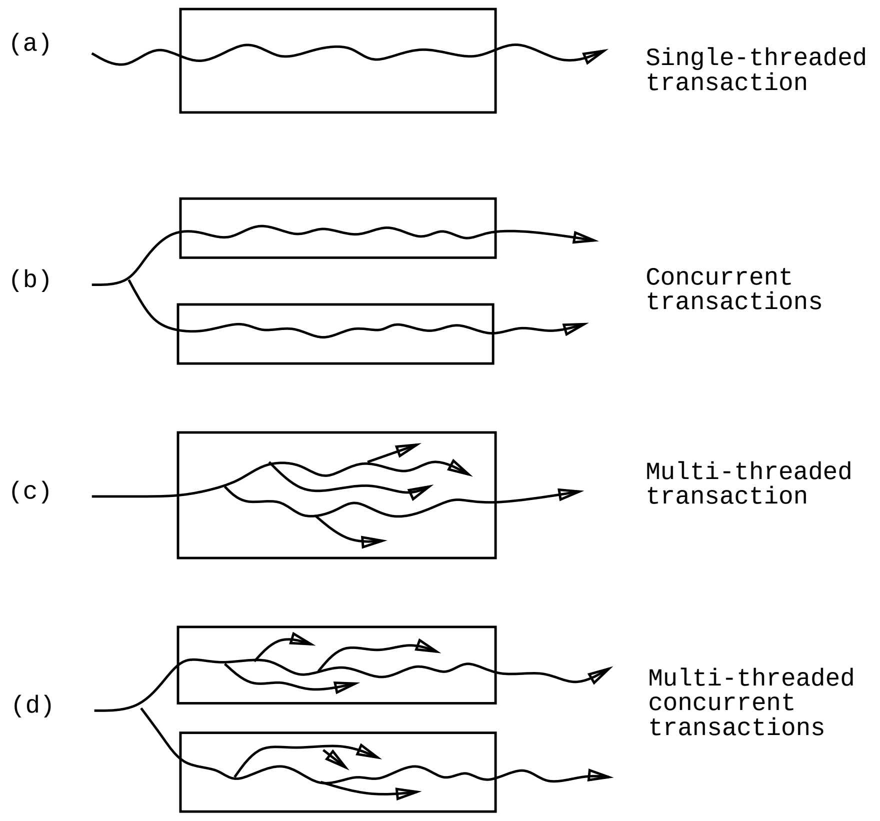

Figure 1: Threads and transactions are separate control abstractions.

2.1 Why have multiple threads within a transaction?

The most compelling argument for supporting multiple threads within a transaction is modularity. Consider the following kinds of multi-threaded programs: (1) a search procedure that uses multiple threads to find program modules satisfying a specification, returning when the first one is found; (2) a procedure with benign side effects, e.g., rebalancing a B-tree or doing garbage collection, that executes in the background of the main computation; (3) a netnews server that uses multiple threads to minimize latency.

We would like to able to run such a multi-threaded program from within a transaction without having to modify the source code. We would like to treat the program as a black box, reuse it in its entirety, but have its effects done transactionally (i.e., atomic, serializable, and persistent). Without being able to simply "wrap" a transaction around the program, we are forced to recode each separate thread as a concurrent subtransaction of a top-level transaction. This violates one aspect of modularity since the entire program has to be recoded.

2.2 Why have multiple threads at all?

Concurrent transactions have to be serializable. Thus, by definition, we can view transactions as happening one after another. On the other hand, threads are often used for two-way communication through shared, mutable resources (e.g., refs). If we identify each thread with a single transaction, then we can no longer do two-way communication between threads. For instance, assuming we associate each thread with a transaction, then Figure 2a shows thread/transaction A and thread/transaction B executing concurrently. Transaction semantics require that the effects of A and B executing concurrently are the same as that of either A executing first followed by B (Figure 2b), or vice versa. Suppose A sends a message to B and B wants to acknowledge A; we cannot put A's execution before B (since A will never get the acknowledgment) nor can we put B before A (since B will never get the message). Thus if we want to support two-way communication between processes, we need to support multiple threads independent of transactions.

Another argument for supporting both threads and transactions as orthogonal concepts is performance. In existing transactional systems, the runtime cost of creating and managing a transaction ("heavyweight" process) is not the same as that for a thread ("lightweight" process). Transactions require runtime mechanism to support protocols for locking, logging, committing/aborting, and crash recovery. There are cases when parallelism is desired without the performance overhead of transactions. Again, even if we were to recode one of our example multi-threaded programs with transactions, we probably do not want to incur the cost of making each thread a transaction.

In short, transactions provide features that threads do not: persistence, undoability, isolation of effects, atomicity of a sequence of operations, and crash recovery. Threads provide functionality, e.g., two-way communication, and performance benefits that transactions do not.

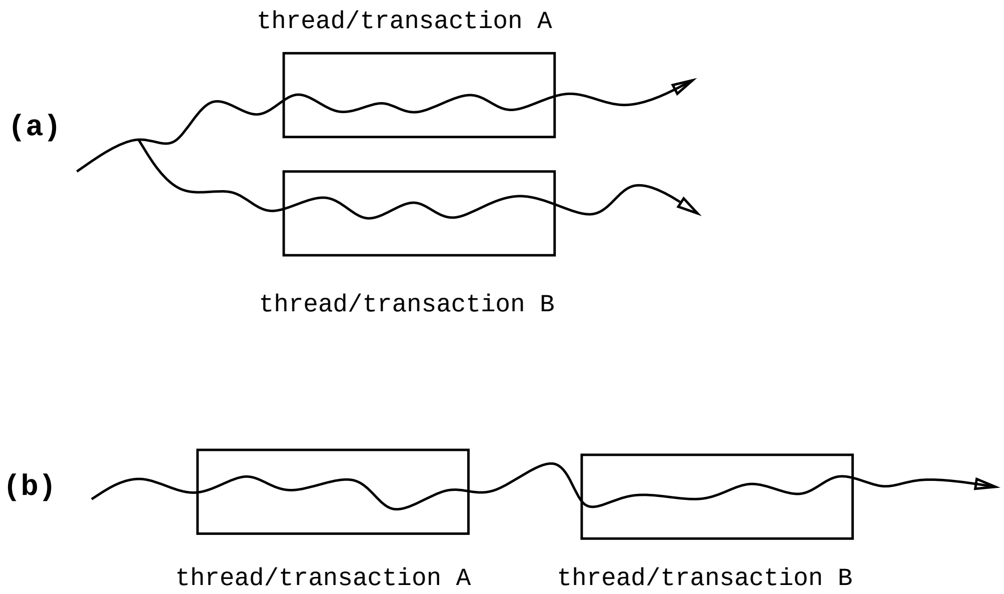

Figure 2: Transactions are serializable.

3 Design Overview

As in our design for single-threaded transactions for SML [6], if `f` is a function applied to some argument `a`, then to execute:

```
f a
```

in a transaction, we want programmers to be able to write:

```
(transact f) a
```

or more probably,

```
((transact f) a ) handle Abort => [some work]
```

Here `f` might be multi-threaded. Informally, the meaning of calling `f` with `transact` is the same as that of just calling `f` with the following additional side effects: If `f` returns normally, then the transaction commits, and if it is a top-level transaction, its effects are saved to persistent memory (i.e., written to disk). If `f` terminates by raising the exception Abort, then the transaction aborts and all of `f`'s effects are undone. Through SML's exceptional handling, in the case of an aborted transaction, the programmer has control of what to do such as clean-up and/or reraising Abort. Note that we support the usual model for subtransactions: the persistence of a child's effects is relative to the commit of its parent and aborting a child does not imply the abort of its parent.

We have implemented the interfaces shown in Figure 3. We use standard two-phase read/write locks to ensure serializability among concurrent transactions. We use Moss's locking rules for nested concurrent transactions [5].

Two items of note are visible through these interfaces. First, the TRANSACT signature shows the clear separation between threads (the substructure Thread_System) and transactions (the substructure Trans).[2] The functor header additionally shows how we have achieved modularization of our support: concurrency within a transaction is packaged in TRANS_THREAD; concurrency among transactions, in RW_LOCK; transaction undoability, in UNDO.

Second, we guarantee the principle of isolation for transactions by making use of "safe" refs [9] (and correspondingly "safe" arrays). In the context of just threads, a normal SML ref is unsafe, while a ref protected by a mutex is a safe ref. In the context of transactions, a ref protected by only a mutex is an unsafe ref, while a ref protected by both a mutex and a read/write lock is a safe ref. A read or write of a safe ref will fail unless the thread (transaction) holds the mutex (read/write lock) of the ref. Thus, it is impossible to violate the isolation principle if the programmer uses only safe refs.

[2] Appendix B shows parts of the THREAD_SYSTEM and other relevant signatures; see [1] for a discussion of threads in SML.

```
signature TRANSACT =
    sig
        structure Trans :
            sig
                exception Abort
                val transact : ('a -> 'b) -> 'a -> 'b
                val abort_top_level: unit -> 'a
                val abort: unit -> 'a

                eqtype rw_lock
                val rw_lock       : unit -> rw_lock
                val acquire_read  : rw_lock -> unit
                val acquire_write : rw_lock -> unit
            end

        structure SRef   : SREF
        structure SArray : SARRAY

        structure Thread_System : THREAD_SYSTEM

        sharing type SRef.lock = SArray.lock = Trans.rw_lock
        sharing type SRef.uref = Thread_System.SRef.sref
        sharing type SArray.uarray = Thread_System.SArray.sarray
    end

functor Transact (structure TT     : TRANS_THREAD
                  structure RW     : RW_LOCK
                  structure SRef   : SREF
                  structure SArray : SARRAY
                  structure Undo   : UNDO
                  sharing type SRef.lock = SArray.lock
                  sharing type SArray.lock = RW.T
                  sharing type SRef.uref = TT.TS.SRef.sref
                  sharing type SArray.uarray = TT.TS.SArray.sarray)
    : TRANSACT = struct ... end
```

Figure 3: Signature and functor modules for transactions.

4 Design Details

4.1 Simplifying assumptions

To simplify our model, and hence our design, we assume that there is exactly one thread that enters a transaction and exactly one that leaves a transaction. We do not lose any generality since we can always immediately fork off multiple children upon entry and we can always force all threads to join into one upon exit. Second, again without loss of generality, we will assume that conceptually the thread exiting is the same as that entering; we call this the transaction's *root thread*.[3] Finally, we assume no mutexes are held before a transaction begins. We make this assumption so we do not have to reacquire locks that were held upon entering a transaction in case an abort occurs. Doing so could cause a deadlock: Suppose the entering thread t holds a lock and then releases it sometime during the transaction. A thread s outside the scope of the transaction then acquires it. If the transaction now aborts, and we are to undo all of its effects, including reacquiring the lock, we may deadlock if s is waiting to acquire some other held lock.

4.2 Per transaction state

Just as there is per thread state [1], we assume there is *per transaction state*. This state includes four pieces of separable information:

- The data objects accessed by the transaction. Since the order of modifications to these data objects is important with respect to abort, we call this information the *(data) undo list*.
- The set of mutex locks held by threads within a transaction. We call this information the *mutex lock set*.
- The set of read and write locks held for the duration of the transaction. We call this information the *read-write lock set*.
- The set of threads running on behalf of the transaction.

The first piece of information (*data state*) is separable from the other three (*synchronization state*) which we need to maintain because of concurrency due to threads.

4.3 Who commits and who aborts?

Our design gives the root thread the privilege of committing the transaction and the responsibility of knowing when it is safe to do so. However, for abort, any thread may encounter a state in which the thread cannot back out of and knowingly wish to cause the abort of the entire transaction; the root thread need not be the only thread to determine that an abort is necessary. Thus, rather than requiring such a thread to communicate with the root thread who could then cause the abort, we permit any thread to cause an abort.

[3] We could relax this assumption by letting threads pass a "baton" among each other, where the baton's flow of control would reflect that of the root thread, but this relaxation is unnecessarily general.

4.4 What happens upon commit?

The effect of a commit is to preserve all data state changes made by the committing transaction. Upon commit, we do the following:

1. Stop all other active threads running on behalf of the transaction so they do not continue to modify state;
2. If the transaction is top-level, throw away the data undo list since we do not need to save the old data values; otherwise, anti-inherit the list to its parent.
3. Release all mutex locks held by non-root threads running on behalf of the transaction.
4. Anti-inherit all read/write locks to its parent [5].
5. If the transaction is top-level, save the state of persistent memory.
6. Exit the transaction and continue processing.

4.5 What happens upon abort?

A transaction may voluntarily abort or be involuntarily aborted (e.g., due to a system crash). Following our semantics for single-threaded `transact` [6], we mask any exception as an abort. Moreover, we treat any unhandled exception as an abort. The effect of an abort is to undo all changes to the data state made by all threads executing on behalf of the transaction. Upon abort, we do the following:

1. Stop all other active threads running on behalf of the transaction so they do not continue to modify state;
2. Follow the undo list backwards, rewriting all old data values.
3. Release all mutex locks held by threads running on behalf of the transaction.
4. Release all read/write locks.

It is critical that we first undo the data values, then release mutexes, and then release read/write locks. Data are protected by mutexes; read/write locks are implemented using them. If we were to release mutexes before undoing all data values, then a thread may modify a data object after the old value gets rewritten. In order to release a read/write lock, we need to be able to acquire other mutexes; if we were not to release mutexes before read/write locks, we could end up in a deadlock situation.

5 Where SML Made a Difference

The SML modules facility is key to our "pick-and-choose" approach. It lets us explicitly reflect the inherent orthogonality of concepts like persistence, undoability, multi-threading, and transactions by letting us define separate structures for each and then functors that compose them. The Transact functor that builds a structure for multi-threaded transactions is one example (Section 3). When we prototyped our implementation for single-threaded, non-concurrent transactions (Appendix A), we also used a functor parameterized over Pers and Undo structures, which respectively provide persistence and undoability.

We also parameterized the Thread_System structure itself so that the programmer can pick-and-choose among separable support for persistence, undoability, and multi-threading. The Fox Project at CMU, for example, needs only multi-threading; it does not have to use a separate threads module, but rather it just has to apply the Thread_System functor to Undo and Pers structures that are essentially "no-ops." Moreover, it is to SML's credit that modifying the original Thread_System structure to work with Undo required only two lines of additional code: to "turn off undo" while doing a thread operation (e.g., acquiring a mutex) and to "turn it back on" when the operation is completed.

Another way we are able to exploit the modules facility is in code reuse. For example, we use only one functor to implement both kinds of safe refs, that for just threads and that for transactions (q.v., `sharing type SRef.uref = TT.TS.SREf.sref` of Figure 3).

Having first-class functions in SML lets us easily support first-class transactions. This view of transactions is a radical departure from the more traditional view taken by other transaction-based programming languages. Programmers in Camelot, Argus, and Avalon write constructs like `begin_transaction ... end_transaction` to bracket transaction boundaries and cannot treat the compound statement as a value.

We relied on the "mostly functional" nature of SML in our implementation of undoability and persistence. For example, our implementation for undoability keeps a log of all modifications to the store and the old values originally assigned to the modified locations. For traditional imperative languages where modifications to the store would be frequent, maintaining and replaying such logs would be expensive. Such a log is inexpensive to maintain for SML since we can assume mutations are rare.

Though we greatly benefit from SML's static typing, one place where we need dynamic types is in our support for persistence. Our interface allows us to add bindings between identifiers and values to a persistent environment; SML cannot statically determine whether the type of the value returned by a subsequent retrieve (e.g., upon startup of a new SML session or upon crash recovery) on some identifier is the same as the type of the value when it was initially bound.

In summary, SML is a nice vehicle with which to express separable concepts. Though SML may not be the natural language of choice in which to support transactions, it is a natural language of choice for the non-traditional applications of transactions that we have in mind. Many of CMU's projects that involve reasoning about programs, Edinburgh's LEGO "proofs-as-programs" system, Cornell's NuPrl system, and AT&T Bell Labs's proof support environment all use SML as their implementation language. These applications need some, if not all, transactional features like data persistence, concurrency control, checkpointing, backtracking, and crash recovery. We hope to provide these potential users with a set of SML modules that they can use in a "pick-and-choose" fashion.

6 Status and Future Work

We have a working prototype of all the interfaces given in this paper, but much work remains:

- *Language design*: As mentioned in the introduction we have yet to do a design of an SML end-user's interface for multi-threaded concurrent transactions. We are also still exploring different policies that our mechanisms can support. We may export different end-user interfaces, each reflecting a different policy.

- *Semantics*: We have been negligent in working out any formal semantics for our extensions to SML. Some of the challenges specific to SML include a semantics for undoability and a semantics for the interactions between `callcc` and `transact`; specific to transactions, a model of computation and meaning of correctness for multi-threaded concurrent transactions.
- *Implementation*: After our prototype becomes stabler, we intend to build sample applications and perform experiments to measure the costs of our extensions.

Acknowledgments

We thank the rest of the Venari Group for their discussions: Gene Rollins, Amy Moormann Zaremski, Nick Haines, Darrell Kindred, and Drew Dean. Nick and Darrell, in consultation with Scott Nettles and Greg Morrisett, are now rebuilding the prototype implementation originally built by Greg and Manuel Faehndrich. Drew, who is building a file system in SML, may very well be Venari's first real client.

References

[1] E.C. Cooper and J. Gregory Morrisett. Adding threads to Standard ML. Technical Report CMU-CS-90-186, CMU, December 1990.

[2] D. L. Detlefs, M. P. Herlihy, and J. M. Wing. Inheritance of synchronization and recovery properties in Avalon/C++. *IEEE Computer*, pages 57–69, December 1988.

[3] J. Eppinger, L. Mummert, and A. Spector. *Camelot and Avalon: A Distributed Transaction Facility.* Morgan Kaufmann, 1991.

[4] B. Liskov and R. Scheifler. Guardians and actions: Linguistic support for robust, distributed programs. *ACM TOPLAS*, 5(3):382–404, July 1983.

[5] J.E.B. Moss. Nested transactions: An approach to reliable distributed computing. Technical Report MIT/LCS/TR-260, MIT, April 1981.

[6] Scott M. Nettles and J.M. Wing. Persistence + Undoability = Transactions. In *Proc. of HICSS-25*, January 1992.

[7] M. Satyanarayanan et al. Coda: A highly available file system for a distributed workstation environment. *IEEE Trans. Comp.*, 39(4):447–459, April 1990.

[8] A.Z. Spector et al. Support for distributed transactions in the TABS prototype. *IEEE TSE*, 11(6):520–530, June 1985.

[9] A.P. Tolmach and A.W. Appel. Debugging Standard ML without reverse engineering. In *Proceedings of the ACM Lisp and Functional Programming Conference*, pages 1–12, 1990.

Appendix A: Persistence and Undoability

Figure 4 illustrates persistence and undoability as orthogonal concepts; we compose them to form single-threaded, non-concurrent, nested transactions. In Figure 4a, as a thread executes, a call to `persist` has the effect of saving all values reachable from a persistent handle to disk. In Figure 4b, a call to `checkpoint` has the effect of "remembering" the store at the point of call; a call to `restore` has the effect of undoing the effects on the store, reverting back to that at the (dynamically) last call to checkpoint. Finally, Figure 4c shows how we compose `checkpoint` with `persist` to give us transaction commit; `checkpoint` with `restore` to give us transaction abort. The signatures for persistence, undoability, and transactions that follow are unfortunately slightly different from that described in [6], but do reflect the current working version of our implementation. The primary difference between the TRANSACT signature here and its analogue in Figure 3 is the absence of an interface for read/write locks, which is not needed in the absence of concurrency.

```
signature PERS = sig
     exception INIT_FAILED
     exception PERSIST_FAILED
     val init: string * string * bool -> unit
     val persist: ('a -> 'b) -> 'a -> 'b

     type identifier
     exception UnboundId
     val bind: identifier * 'a -> unit
     val unbind: identifier -> unit
     val retrieve: identifier -> 'a
  end

signature UNDO =  sig
    exception Restore of exn
    val checkpoint: ('a -> 'b) -> 'a -> 'b
    val restore: exn -> 'a

    val exn2restore: ('a -> 'b) -> 'a -> 'b
    val restore2exn: ('a -> 'b) -> 'a -> 'b
    val restore_on_exn: ('a -> 'b) -> 'a -> 'b
  end

signature TRANSACT = sig
    val transact: ('a -> 'b) -> 'a -> 'b

    exception Abort
    val abort_top_level: unit -> 'a
    val abort: unit -> 'a
  end
```

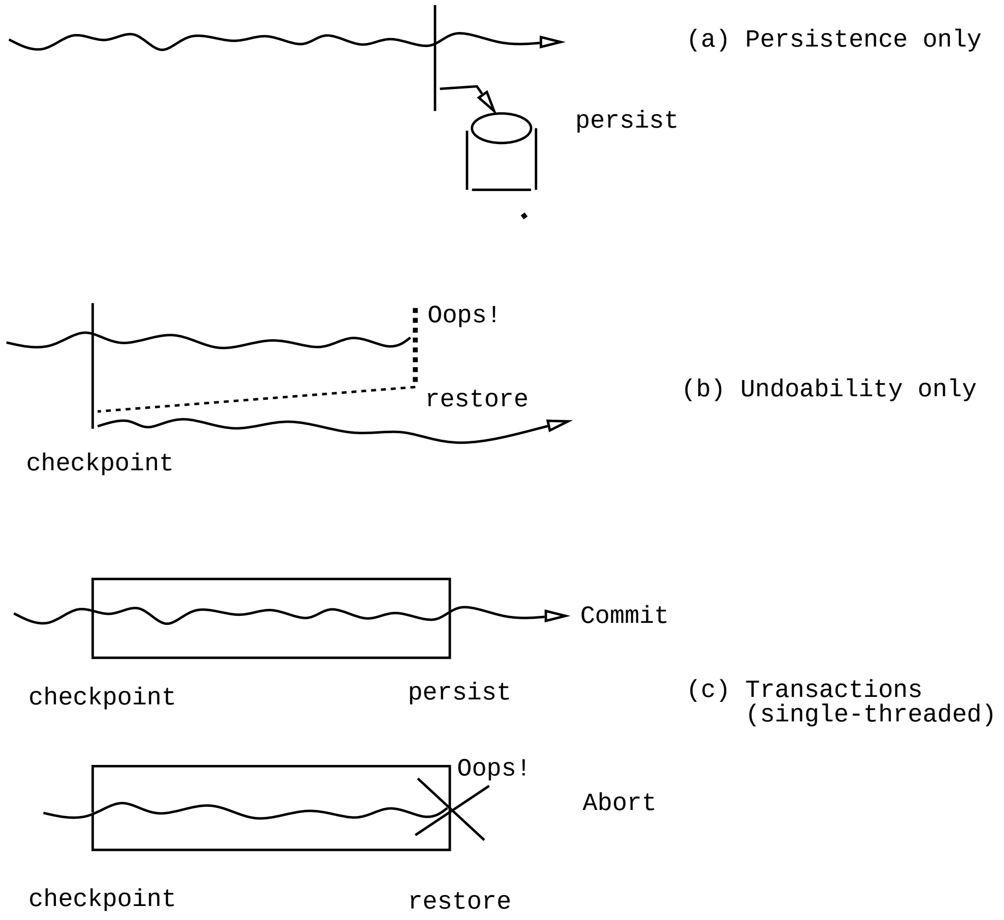

Figure 4: Persistence + Undoability = Transactions

Appendix B: Threads

Our Threads interface and parts of the TRANS_THREAD and THREAD_SYSTEM signatures:

```
signature THREAD = sig
    val fork : (unit -> unit) -> unit
    val exit : unit -> unit

    type mutex
    val mutex : unit -> mutex
    val with_mutex : mutex -> (unit -> 'a) -> 'a

    type condition
    val condition : mutex -> condition
    val with_condition : condition -> (unit -> 'a) -> 'a
    val signal : condition -> unit
    val broadcast : condition -> unit
    val await : condition -> (unit -> bool) -> unit
    val vwait : condition -> (unit -> 'a option) -> 'a

    exception Undefined
    type 'a var
    val var : unit -> '1a var
    val get : 'a var -> 'a
    val set : 'a var -> 'a -> unit
  end

signature TRANS_THREAD = sig
    structure TS : THREAD_SYSTEM
    structure TransID :  sig ... end
    ...
  end

signature THREAD_SYSTEM =  sig
    structure Thread : THREAD

    structure SRef :  sig ... end
    structure SArray :  sig ... end
    ...
  end
```