4. Satoh, I., and Tokoro, M., *A Formalism for Real-Time Concurrent Object-Oriented Computing*, Proceedings of ACM OOPSLA'92, p315-326, October, 1992.
5. Satoh, I., and Tokoro, M., *A Timed Calculus for Distributed Objects with Clocks*, Proceedings of ECOOP'93, July, 1993.
6. Takashio, K., and Tokoro, M., *DROL: An Object-Oriented Programming Language for Distributed Real-time Systems*, Proceedings of ACM OOPSLA'92, p276-294, October, 1992.
7. Takashio, K., and Tokoro, M., *Time Polymorphic Invocation: A Real-Time Communication Model for Distributed Systems*, Proceedings of IEEE WPDRTS'93, p79-88, April, 1992.
8. Tokoro, M., *Computational Field Model: Toward a New Computing Model/-Methodology for Open Distributed Environment*, Proceedings of IEEE Workshop on Future Trends of Distributed Computing Systems, September, 1990.
9. Tokoro, M., *Toward Computing Systems for the 2000's*, Proceedings of Operating Systems in 1990's and Beyond, LNCS 563, December, 1991.
10. Tokoro, M. and Satoh, I., *Asynchrony and Real-Time in Distributed Systems*, Proceedings of Parallel Symbolic Computing: Language, Systems, and Applications, October, 1992.
11. Tokuda, H., and Mercer, C. W., *ARTS: A Distributed Real-Time Kernel*, ACM Operating System Review, Vol.23, No.3, 1989.
12. Yonezawa, A., and Tokoro, M., editors, *Object-Oriented Concurrent Programming*, MIT Press, 1987.
13. Yokote, Y., *The Apertos Reflective Operating System: The Concept and its Implementation*, Proceedings of ACM OOPSLA'92, p397-413, October, 1992.

Decomposing and Recomposing Transactional Concepts

Jeannette M. Wing [1]
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

1 Revisiting Transactions

Distributed systems are different from concurrent (and parallel) systems because they need to deal with failures, not just concurrency. Transactions are a way ol masking the distributed nature of a computation at the programming languag level by transforming all failures into aborted transactions. If a communicatio link goes down or a node crashes, the transaction simply aborts. Users may tr again later to rerun their computation, but they are at least guaranteed that th system is left in some consistent state.

Transactions are a well-known and fundamental control abstraction tha arose out of the database community. They have three properties that distin guish them from normal sequential processes: (1) A transaction is a sequenc of operations that is performed *atomically* ("all-or-nothing"). If it complete successfully, it *commits*; otherwise, it *aborts*; (2) concurrent transactions are *s rializable* (appear to occur one-at-a-time), supporting the principle of isolatio and (3) effects of committed transactions are *persistent* (survive failures).

1.1 Separation of concerns

Systems like Tabs [8] and Camelot [3] demonstrate the viability of layering general-purpose transactional facility on top of an operating system. Languag such as Argus [4] and Avalon/C++ [2] go one step further by providing lingui tic support for transactions in the context of a general-purpose programmi language. In principle programmers can now use transactions as a unit of enca sulation to structure an application program without regard for how they a implemented at the operating system level.

[1] This research is sponsored by the Wright Laboratory, Aeronautical Systems Cent Air Force Materiel Command, USAF, and the Advanced Research Projects Agen (ARPA) under grant number F33615-93-1-1330.
The views and conclusions contained in this document are those of the authors a should not be interpreted as necessarily representing the official policies or endor ments, either expressed or implied, of Wright Laboratory or the U. S. Governme
The U. S. Government is authorized to reproduce and distribute reprints for G ernment purposes notwithstanding any copyright notation thereon. This manuscri is submitted for publication with the understanding that the U. S. Government authorized to reproduce and distribute reprints for Governmental purposes.

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, 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 [10]. Ideally, we want to provide separable components to support transactional semantics for different settings, e.g., in the absence or presence of concurrency and/or distribution. Programmers are then free to compose those components supporting only those features of transactions they need for their application. Our approach also enables programmers to code some applications that cannot be done without an explicit separation of concerns.

We want to support this approach at the programming language level. The current status of the Venari Project is that we can support concurrent, multi-threaded, nested transactions in the context of Standard ML. Our implementation, however, does not yet run in a distributed environment.

1.2 Why SML?

To explore the feasibility of designing a language to support orthogonal transactional concepts, we chose not to design a brand new language from scratch. Instead, we decided we would target an existing language as a basis for extension; we chose Standard ML, and in particular the New Jersey implementation.

SML is not the obvious choice for building a transaction-based programming language, even less so for building an object-oriented distributed language. SML's heart is in functional (stateless) programming and transactions are very much a state-oriented concept. SML has no notion of subtype or inheritance and no direct support for concurrency, distribution, or persistence.

However, SML does give a good starting point. In the design and implementation of our extensions, we gained leverage from SML's high-level language features including strong typing, exceptions, first-class functions, and modules. SML makes a type distinction between immutable and mutable values (**refs** and **arrays**); we rely on strong typing to let the runtime system safely operate on addresses (without the programmer's knowledge). SML's support for first-class functions (closures) allow us to make transactions first-class. We use signatures to separate interface information from implementation and functors to compose parameterized modules. SML's modules facility enables us to support our "pick-and-choose" approach at the language level.

2 The Application Programmer's View of Venari/ML Transactions

If **f** is a function applied to some argument a, then to execute:

```
f a
```

in a transaction, programmers can write:

```
(transact f) a
```

or more probably,

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

where Foo is a user-defined exception. 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 any uncaught exception, e.g., Foo, then the transaction aborts and all of **f**'s effects are undone. Through SML's exception-handling, in the case of an aborted transaction, the programmer has control of what to do such as clean-up and/or retrying the transaction.

As a more compelling (and the canonical) example, suppose we want to transfer money from one bank account to another. This would involve withdrawing money from one account and depositing it in the other. We need to make sure that either both the withdrawal and the deposit succeed, or that neither of them occur. If only the withdrawal happened, the money would be lost, and we would be very unhappy. If only the deposit happened, the money would be "duplicated," and the bank would be very unhappy. So, we use a transaction to effect the desired behavior.

```
fun transfer (account_1, account_2, amount) =
    let fun do_transfer () =
        (withdraw (account_1, amount);
         deposit (account_2, amount))
    in
        transact do_transfer ()
    end
```

The function **transfer** transfers money from **account_1** to **account_2** with the guarantee that a partial transfer will not occur. The transfer itself occurs in the function **do_transfer**, which withdraws the money from **account_1** and deposits it into **account_2**. The functions **withdraw** and **deposit** are expected to raise an exception if something goes wrong, e.g., if **account_1** has insufficient funds or the bank's computer goes down. We wrap a transaction around the call to **do_transfer** so that if anything goes wrong, the whole transfer will be aborted. If the transfer is aborted, we reraise the exception that caused the abort.

We could make the transfer transaction multi-threaded by having one thread do the withdrawal while another does the deposit. All we would need to do is to replace the two-line definition of do_transfer with the starred lines below:

```
fun transfer (account_1, account_2, amount) =
    let fun do_transfer () =
        (fork (fn () => withdraw (account_1, amount));  *
                        deposit (account_2, amount))    *
    in
        transact do_transfer ()
    end
```

3 The Venari/ML Interfaces

In our design, we teased apart the usual atomicity, serializability, and persistence properties rolled into transactions, and added the ability for transactions to be multi-threaded. In particular, we provide support for the following features, each as a separable component—the name of the Venari/ML signature is given in parentheses.

- Persistence (**PERS**)
- Undoability (**UNDO**)
- Reader-writer locks (**RW_LOCK**)
- Threads (**THREADS**)
- Skeins (**SKEINS**)

The basic idea is that we want the individual pieces to compose in a seamless way to give us transactions. Persistence ensures permanence of effects of top-level transactions. Undoability allows us to handle aborted transactions. Reader-writer locks provide isolation of changes to the store, and hence ensure transaction serializability of concurrent transactions. Skeins let us group a collection of threads together, giving us the ability to make multi-threaded transactions.

Putting all these pieces together into a single ML module culminates in our main **VENARI** interface shown below. It provides a way for application programmers to create and manipulate concurrent multi-threaded transactions. What distinguishes our model from the more standard model of concurrent, nested transactions is our ability to identify multiple threads of control (not just one thread) with a single transaction.

```
signature VENARI =
    sig
        val transact : ('a -> '_b) -> 'a -> '_b

        structure Threads : THREADS
        structure Skeins : SKEINS
        structure RW_Lock : RW_LOCK
        structure Undo : UNDO
        structure Pers : PERS
    end
```

Roughly speaking, a transaction is a *locking skein* of *threads* whose effects are *undone* if the transaction aborts or made *persistent* if it terminates. (In SML 'a is a polymorphic type variable.)

By having separated transactional concepts from one another, we also provide the ability to put some pieces together, ignoring others. This separation of concerns enables direct support for different non-transactional models of computation. Here are some of the more interesting combinations:

- Multi-threaded persistence (threads + persistence = persistent skeins)
- Multi-threaded undo (threads + undo = undo skeins)
- Locking threads (threads + r/w locks = locking skeins)
- Concurrent persistence (threads + r/w locks + persistence = locking persistent skeins)
- Concurrent multi-threaded transactions
  (persistence + undo + r/w locks + threads = transactional skeins)
  As seen, the **VENARI** interface above supports this particular combination directly.

All skeins can be nested, hence each combination above can be nested. Permanence of a nested persistent skein's effects is relative to its parent. All mixes are possible. For example, a transaction can have an undo skein or locking skein within it, and vice versa. A skein can have nested within it concurrent skeins of different flavors. Finally, the single-threaded case of any of these is just a special case in which a skein has just one thread; Venari/ML does not explicitly provide interfaces for the single-threaded cases.

In previously published papers, we have already reported on various aspects of the Venari/ML interfaces. Details of the design and implementation of the Threads interface are reported in [1]; of the separation between persistence and undoability for single-threaded nested transactions, in [6]. An early design of concurrent multi-threaded transactions appears in [11]. The remainder of this paper focuses on the details of the synchronization primitives provided for our model of computation.

4 Synchronizing Concurrent Multi-threaded Transactions

Our generalized model of transactions requires a generalization of Moss's (traditional) locking rules used for nested transactions [5]. Between transactions we must of course guarantee isolation; but within a transaction, threads may freely execute and need not be serialized with respect to each other. Our model also allows parents and children to execute simultaneously; parents are not suspended. To deal with some of the semantic complexity of this model, we provide a notion of *safe state* for users and we implement certain runtime checks that guarantee the principle of isolation for transactions. We can make this guarantee only under the assumption that users access only safe state. Thus, we begin by describing safe state and then describe the transaction guarantees that we have implemented.

4.1 Safe State

As seen, we not only allow transactions to run concurrently and we also allow each transaction to be multi-threaded. What implications does this model of computation have on access to shared data? It requires a means to synchronize concurrent threads and a means to synchronize concurrent transactions. These two different requirements suggest having two different grains of locks for synchronizing access to shared mutable data.

First, as typical for multi-threaded programs, we rely on *mutex locks*; second, as typical for concurrent transactions, we rely on *reader-writer locks*. Mutex locks allow threads within a transaction to synchronize, in particular, to enforce mutual exclusion; reader-writer locks allow concurrent transactions to synchronize, in particular, to enforce isolation (i.e., serializability). Without transactions, a thread need only acquire a mutex lock before accessing shared data, but with multi-threaded transactions, it must acquire a reader-writer lock.

To provide programming language support for this model we were easily in Standard ML to provide *safe state* [9]: mutable data that is guaranteed to be always accessed by a thread that has acquired the right kind of lock. What is "safe" about safe state are the correctness guarantees it provides: mutual exclusion between threads and isolation between transactions. Disciplined programmers, those who use only safe state, will be assured that their concurrent multi-threaded transactions are serializable.

Let's look in some detail at the abstractions needed. First, we need two types of locks, **mutex** locks and **rw_locks**, respectively defined in **Threads** and **RW_Lock** modules. In particular, the **Threads** module declares (among others) the following type and function values:

```
type mutex
val mutex      : unit -> mutex
val acquire    : mutex -> unit
val release    : mutex -> unit
val owner      : mutex -> bool
val with_mutex: mutex -> (unit -> 'a) -> 'a
```

Evaluation of the function call **mutex()** creates a new mutex value. The function **acquire** attempts to lock a mutex and blocks the calling thread until it succeeds; **release** unlocks a mutex, giving other threads a chance to acquire it; **owner** returns true if and only if the mutex is currently held by the current thread. The evaluation of **with_mutex m f** acquires the mutex **m**, applies the function **f**, and then releases **m**.

The **RW_Lock** module includes (among others) the following type and function values:

```
type rw_lock
val create    : unit -> rw_lock
val read      : rw_lock -> ('a -> 'b) -> 'a -> 'b
val write     : rw_lock -> ('a -> 'b) -> 'a -> 'b
```

The **read** and **write** functions take a lock, a function, and its argument, and apply the function to the argument with the guarantee that the lock is held in the appropriate read or write mode during the execution of the function. In particular, no other *thread* may use the lock in read or write mode while the function executes. If some thread within a transaction calls **read** or **write** without holding the lock in the appropriate mode, we raise an exception; otherwise, **read** and **write** will block until the condition is satisfied.

Second, we exploit the fact that SML makes a type distinction between immutable and mutable values. In SML, only **refs** and **arrays** are mutable. Thus, only refs and arrays need to be protected by any kind of lock. (Since refs and arrays are treated similarly, we will talk about just refs for the remainder of this paper.) Below are the two relevant functions on **ref** types:

```
val !  : 'a ref -> 'a
val := : 'a ref * 'a -> unit
```

If **x** is of type **int ref** then **x := 5** assigns the integer 5 to **x**; subsequent evaluation of **!x** is 5.

Now, we can build *mutex refs*, safe state for multi-threaded programs. An **m_ref** is a regular (unsafe) ref protected by a mutex lock. The relevant functions that we can perform on mutex refs are akin to their unsafe counterparts:

```
val m_get:  'a m_ref -> 'a
val m_set:  'a m_ref -> 'a -> unit
```

But, what is so elegantly expressible in SML are their definitions (implementations):

```
datatype 'a m_ref = M_Ref of ('a ref * Threads.mutex)

fun m_get (M_Ref (uref,l)) =
    if Threads.owner l then (!uref) else raise NotOwner
fun m_set (M_Ref (uref,l)) v =
    if Threads.owner l then uref := v else raise NotOwner
```

Here we see explicitly how an **m_ref** is implemented—as a pair of a regular ref and a mutex lock. A call to **m_get** checks to see that the caller owns the associated lock `l` before performing the access (!) on the ref **uref** itself. If the check fails, an exception is raised. Similarly for **m_set**.

Finally, we can play the same game of making transactionally safe state by building *reader-writer refs*. As for mutex refs, we can perform get and set functions on reader-writer refs:

```
val rw_get:  'a rw_ref -> 'a
val rw_set:  'a rw_ref -> 'a -> unit
```

where again by looking at their definitions, we can see how we enforce safe access:

```
datatype 'a rw_ref = RW_Ref of ('a ref * RW_Lock.rw_lock)

fun rw_get (RW_Ref (r,l)) =
    RW_Lock.read l !r
fun rw_set (RW_Ref (r,l)) v =
    RW_Lock.write l (fn () => r:=v) ()
```

Since a **rw_ref** is a regular ref protected by a reader-writer lock, a call to **rw_get** guarantees safe access to the ref because it acquires the lock `l` in read mode before returning the value stored in the ref `r`. Similarly for **rw_set**.

To show a simple use of mutex refs (reader-writer refs would be similar), consider a multi-threaded application that uses a logical clock to establish the order of events. Here's an SML signature for such a clock:

```
signature CLOCK = sig
    val get_time : unit -> int
end
```

It exports only the one function, **get_time**, which increments the clock and returns a new, unique time. We can implement the clock and its **get_time** function as:

```
val time = m_ref (0, Threads.mutex())

fun get_time () =
    with_m_ref time (fn () => (m_inc time; m_get time))
```

where the mutex ref **time** stores the logical time, **with_m_ref** is the mutex ref counterpart to **with_mutex**, and **m_inc** increments the integer value of the clock by 1 (defined in the obvious way in terms of the increment function on **int refs**.). **Time** must be protected by a mutex to avoid the following incorrect sequence of events in which two threads would be given the same time:

```
thread A thread B
inc time
         inc time
!time
         !time
```

To ensure that each caller is given a unique time, the function **get_time** wraps **with_m_ref** around the calls to increment and read **time**.

To summarize, we guarantee the principle of isolation for transactions by making use of *safe state*. 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 a reader-writer lock is a safe ref. A read or write of a safe ref will fail unless the thread (transaction) holds the mutex (reader-writer lock) of the ref. Thus, it is impossible to violate the isolation principle if the programmer uses only safe state.

4.2 Transaction Guarantees

Given that within a (multi-threaded) transaction only safe state is accessed, we generalize traditional rules for managing nested transactions.

If the body thread or any sub-thread raises an uncaught exception, the transaction *aborts*. If the body evaluates successfully, the transaction *commits*.

When a transaction aborts,

- all changes to the persistent and volatile stores made by the transaction and its descendants are undone; and
- all reader-writer locks held by the transaction and its descendants are released.

When a transaction commits,

- if this is a top-level transaction (i.e., no ancestor skein is persistent, undoable, or a transaction), and the persistent store is initialized, any changes to the persistent store are committed to disk; and
- all reader-writer locks are handed to the nearest locking ancestor skein.

If the functions executed within transactions have no effects except through the use of the safe state, then we can make certain guarantees regarding the interaction of those transactions. Let T be a transaction, and let S and S' be any locking skeins (thus S and S' may be transactions as well). (T, S, and S' are all different from one another.) The following guarantees hold:

- If neither S nor T is a descendant of the other, then
  - if T aborts, S observes no effects of T or T's descendants;
  - the effects of T and its descendants appear atomic to S (i.e., S sees either all of their effects or none of their effects); and
  - the effects of S and T are serializable from the viewpoint of any other locking skein S'.
- If T is a descendant of S, then
  - the effects of T and its descendants appear atomic to S; and
  - the state which T observes will reflect a "snapshot" of S's effects (taken at the instant after T acquires its last reader-writer lock); and
  - if S's effects before and after the "snapshot" point are denoted E_S^{before} and E_S^{after}, and the effects of T and its decendants are denoted E_T, then these effects will appear to S' to take place in the order $(E_S^{before}, E_T, E_S^{after})$.
- The image of the persistent store on disk will always be ***consistent*** (partial effects of a transaction will never appear on disk).

One should consider a transaction T_2 which is a child of transaction T_1 to be doing work "on behalf of" T_1. The guarantees above hold even if non-transactional skeins or threads are invoked within the transactions involved.

5 Summary and Future Work

Unlike other transaction-based high-level programming languages such as Argus and Avalon, Venari/ML is the first to support multi-threaded transactions, where each transaction may have multiple threads of control executing within its scope. This generalization of the standard transactional model led to our support for concurrent multi-threaded transactions and some novel abstractions like skeins and safe state. Moreover, our model required rethinking the synchronization rules for nested transactions, leading to the transaction guarantees described in Section 4.2.

The Venari/ML interfaces are cast in terms of SML's modules facility. Our modules support a separation of concerns, e.g., persistence from undoability, that are often tightly integrated in other transactional systems. We also make extensive use of closures in SML, allowing us at runtime to compose different functions, each of which supports a different feature of transactions. E.g., the argument to the function **transact** is a closure.

There are two main directions that we would like to pursue in the future. First, 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. The Venari Project expects to do some careful performance analysis of our implemented features. Second, we hope to build a non-trivial and non-traditional application using our interfaces. Existing transactional facilities have been designed primarily for applications like electronic banking and airline reservations. We are more interested in applications like cooperative work environments and engineering design systems where the objects of interest are irregular in structure and the computations may span hours or even days.

6 Acknowledgments

This paper could not have been written without Greg Morrisett's work on the Threads package for SML/NL, Scott Nettles's work on persistence and undoability, Darrell Kindred's work on locks, and Nick Haines's work on putting it all together. Portions of this paper are from the technical report [10] that we jointly wrote.

References

1. E.C. Cooper and J. Gregory Morrisett. Adding threads to Standard ML. Technical Report CMU-CS-90-186, Carnegie Mellon School of Computer Science, 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 Publishers, Inc., 1991.
4. B. Liskov and R. Scheifler. Guardians and actions: Linguistic support for robust, distributed programs. *ACM Transactions on Programming Language and Systems*, 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, Laboratory for Computer Science, April 1981.
6. Scott M. Nettles and J.M. Wing. Persistence + Undoability = Transactions. In *Proc. of HICSS-25*, January 1992. Also CMU-CS-91-173, August 1991.
7. M. Satyanarayanan et al. Coda: A highly available file system for a distributed workstation environment. *IEEE Trans. Computers*, 39(4):447–459, April 1990.
8. A.Z. Spector et al. Support for distributed transactions in the TABS prototype. *IEEE Transactions on Software Engineering*, 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.
10. J.M. Wing and et al. Venari/ML interfaces and examples. Technical Report CMU-CS-93-123, CMU School of Computer Science, March 1993.
11. J.M. Wing, M. Faehndrich, J.G. Morrisett, and S.M. Nettles. Extensions to Standard ML to support transactions. In *ACM SIGPLAN Workshop on ML and its Applications*, June 1992. Also CMU-CS-92-132, April 1992.

Rachid Guerraoui Oscar Nierstrasz
Michel Riveill (Eds.)

Object-Based Distributed Programming

ECOOP '93 Workshop
Kaiserslautern, Germany, July 26-27, 1993
Proceedings

Springer-Verlag
Berlin Heidelberg New York
London Paris Tokyo
Hong Kong Barcelona
Budapest