Composing Transactional Concepts

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

1. Revisiting Transactions

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).

1.1. Separation of concerns

Systems like Tabs [5] and Camelot [2] demonstrate the viability of layering a general-purpose transactional facility on top of an operating system. Languages such as Argus [3] and Avalon/C++ [1] 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 [4] 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 [6]. 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. We have not yet addressed distribution, but see that as our next big step. (We were more concerned to break apart separable transactional concepts before tackling distribution.)

---

[1]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.

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:

```
(fork (fn () => withdraw (account_1, amount));
              deposit (account_2, amount))
```

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 on the next page. 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.

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.

References

[1] 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.

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

[3] 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.

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

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

[6] J.M. Wing and et al. Venari/ML interfaces and examples. Technical Report CMU-CS-93-123, CMU School of Computer Science, March 1993.