Venari/ML Interfaces and Examples

Jeannette M. Wing Manuel Faehndrich Nick Haines
Karen Kietzke Darrell Kindred J. Gregory Morrisett
Scott Nettles

March 1993
CMU-CS-93-123

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

This research was sponsored in part by the Avionics Laboratory, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U.S. Air Force, Wright-Patterson AFB, Ohio 45433-6543 under Contract F33615-90-C-1465, ARPA Order No. 7597, and in part by National Science Foundation Fellowships for D. Kindred and J. G. Morrisett..

Keywords: Transactions, threads, skeins, persistence, recovery, undoability, serializability, Standard ML, modules

Abstract

Transactions are a well-known and fundamental control abstraction that arose from the database community. Application programmers can treat a sequence of operations as an atomic ("all-or-nothing") unit and rely on the runtime environment to guarantee serializability of concurrent transactions and persistence of effects of committed transactions. In this report, we present interfaces, expressed in Standard ML, for creating and controlling transactions. 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. We present a set of simple examples that show how to use Venari/ML interfaces individually and also in some useful combinations. We also present a larger example of searching a database of BibTeX entries. This report is intended primarily for use by an SML programmer whose application requires transactional properties.

Our work on transactions in the context of SML led to our invention of a new control abstraction, called a *skein*, which is a group of threads that cooperate on a single task. Skeins take as parameters initialization and completion functions; transactions are easily constructed as a special case of skeins.

The Venari/ML interfaces are cast in terms of SML's modules facility. Modules support a separation of concerns, e.g., persistence from undoability, that are often tightly integrated in other transaction-based programming languages. We make extensive use of closures in SML, allowing us at runtime to compose different functions, each of which supports a different feature of transactions.

Contents

Chapter 1

Introduction

This report documents the current status of the Venari/ML interfaces. The main `VENARI` interface provides a way for application programmers to create and manipulate *concurrent multi-threaded transactions*. This interface is built up from others, each of which supports a separable feature of transactions, e.g., persistence, undoability, and isolation. We give many small examples and one larger one to show how programmers can use our interfaces. We implemented our interfaces for Standard ML of New Jersey. This report is meant to serve as a user's guide; hence, it does not elaborate on how we implemented the interfaces.

1.1 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). Transactions can be nested. 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.

1.1.1 Separation of concerns

Systems like Tabs [15] and Camelot [6] demonstrate the viability of layering a general-purpose transactional facility on top of an operating system. Languages such as Argus [9] and Avalon/C++ [4] 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 [14] 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. Ideally, we want to provide separable components to support transactional semantics for different settings, e.g., in the absence or presence of concurrency. 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.

1.1.2 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. In Chapter 4 we present a simplified version of this search problem, that of searching a database of bibliographic entries such as those in `.bib` files.

Another problem in software development is version management. Many people working on a large software project need to coordinate updates to different components of the software under development. Systems like RCS [16] provide some configuration management help. The Venari Project is currently implementing a configuration management facility similar to RCS; it uses the Venari/ML transactional interfaces described in this document.

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. As yet, however, the Venari Project has not done a careful or extensive performance analysis of our implemented features.

In the remainder of this section we discuss why we chose SML as our target language, summarize other related Venari/ML documents, and summarize the contributions of our work so far. We then motivate in Section 1.2 the separation between threads and transactions and in Section 1.3 give a high-level view of the most important Venari/ML interfaces and some of interesting combinations. Chapter 2 describes each interface more fully. We give simple examples in Chapter 3 and an extended example in Chapter 4, showing how to use our interfaces, especially to illustrate the orthogonality of concepts we provide.

1.1.3 Why SML?

We cast our approach concretely in the context of programming languages. Instead of designing a brand new language from scratch, we target an existing language as a basis for extension. For technical and pragmatic reasons, we chose Standard ML as our base language. SML is a strongly-typed, mostly functional, programming language. At its core, it supports functions as first-class values, exceptions, and polymorphism. SML's modules facility supports information hiding, data abstraction, and parameterized modules. Most notably, SML has a published formal semantics [11], which means that any extension has the potential of being formally defined and can be objectively evaluated in terms of how much it perturbs the existing semantics. One important pragmatic reason for choosing SML as our base language is that a decent compiler and runtime were readily available and relatively easy to extend. Another pragmatic reason is that SML has a growing local (Carnegie Mellon) and international user community. Finally, we chose to target the New Jersey implementation of SML because SML/NJ supports continuations[1] and it runs on different architectural and operating system platforms.

In the design and implementation of our own extensions, we gain additional leverage from SML's high-level language features and SML/NJ's well-modularized design. SML makes a type distinction between immutable and mutable values (`ref`s); 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. We exploit SML/NJ's highly-phased compiler by not modifying its front-end at all. We modify its back-end with additions that fit into its garbage collection scheme and take advantage of its simple runtime representation of data; we use the storage allocation algorithm unchanged.

Henceforth, we assume the reader has a reading knowledge of SML.

1.1.4 Related Venari/ML Documents

In a series of three ML workshop abstracts and papers, we incrementally reported on our design and implementation of support for transactions:

- First, we designed, along with others (namely Eric Cooper, Bob Harper, and Peter Lee) at Carnegie Mellon, a Threads interface for SML/NJ [3].. We reported on this work at the Edinburgh 1990 ML Workshop.
- Second, we designed and implemented support for the simple case of single-site, single-threaded, nested transactions. We separate persistence and undoability as orthogonal properties of transactions, and support each in a separate SML module. A third module is built in terms of those two to create one that provides transactions. We reported on this work at the

[1] SML as defined in [11] does not feature continuations, but see [5] for a formal description.

Pittsburgh 1991 ML workshop; Nettles and Wing's HICSS paper provides implementation details and preliminary benchmarks [12].

- Finally, we combined the above work to handle concurrency, which we address in two ways: making an individual transaction multi-threaded and allowing multiple transactions to run concurrently. A combination of these two lets us build *concurrent multi-threaded transactions*. We reported on this work in the San Francisco 1992 ML workshop [18]. This document gives the details of the interfaces mentioned in the workshop paper.

1.1.5 What is New about Venari/ML?

To the transaction and database community, our work is novel because it casts within a programming language a model of computation that supports multi-threaded transactions. Other transactional systems like Encina[2] and Quicksilver [7] support multi-threading and transactions, but not in the context of an advanced programming language like SML, and thus they cannot take advantage of other advanced language features (e.g., strong typing). Other transactional programming languages like Argus [9] and Avalon/C++ [4] support only single-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 and extensive use of closures should be of particular interest.

One novel technical contribution of Venari/ML is the invention of a new control abstraction, which we call a *skein*. A skein is a group of threads that cooperate on a common task. We use skeins to build multi-threaded transactions; however, skeins are more generally applicable than for just building transactions. For example, with skeins we can build multi-threaded systems that support undo but not persistence. Chapter 2 gives details.

Our design approach has been pragmatic and bottom-up; by prototyping individual features (e.g., persistence, undoability, read/write locking, nesting, threads, skeins, and transactions) incrementally and then combining them in various ways, we are able to explore a rich design space. Our concern has primarily been to provide reasonably efficient run-time mechanism to give system builders flexibility in deciding policy. This flexibility comes at a price—safety. We do enforce some safety guarantees (e.g., top-level transactions always obey the two-phase locking protocol thus guaranteeing serializability), but not others (e.g., since threads may execute outside any transaction they may violate the isolation principle).[3] To disciplined programmers, however, who always manipulate only "safe state" (Sections 2.2.4 and 2.4), we guarantee complete safety.

We have not thought greatly about the "ideal" programming interface to provide the SML end-user. We believe that we need to gain more experience using our current interface before embarking on a more complete language design effort. We also have not given a formal semantics of our extensions in the operational style followed in [11]; this work remains to be done. The multi-threaded transactional model of computation that Venari supports is new and we are currently working on a more formal semantic description of the model.

[2] Encina is a software product of Transarc Corporation.

[3] One solution is never to have threads execute outside a transaction or to guarantee they never interfere with any data accessed by a transaction.

1.2 Keeping Threads and Transactions Separate

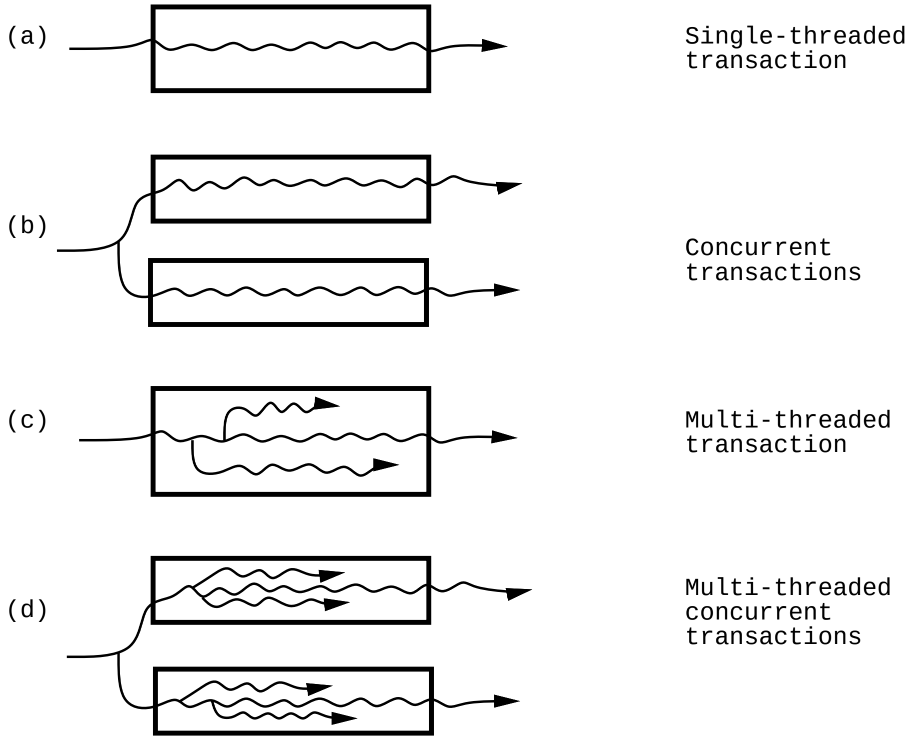

Figure 1.1: Threads and transactions are separate control abstractions.

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 1.1a and 1.1b depict the traditional model, where we use a wavy line to denote a thread and a thick-lined box to denote a transaction; time advances from left to right. Figure 1.1a shows a single thread executing, first entering a transaction and then leaving successfully (i.e., committing). Figure 1.1b shows two single-threaded transactions executing concurrently. Figure 1.1c depicts our new model where multiple threads execute within a single transaction. And finally, Figure 1.1d depicts concurrent multi-threaded transactions, the “composition” of Figures 1.1b and 1.1c. The goal of Venari's version of SML is to support Figure 1.1d through module composition.

1.2.1 Application Programmer's Interface

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 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.

1.2.2 Why separate threads and transactions?

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 be transactional (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.

At the same time, 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 uniquely with a single transaction, then Figure 1.2a 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 1.2b), 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 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 concurrency 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), program structuring, and performance benefits that transactions do not.

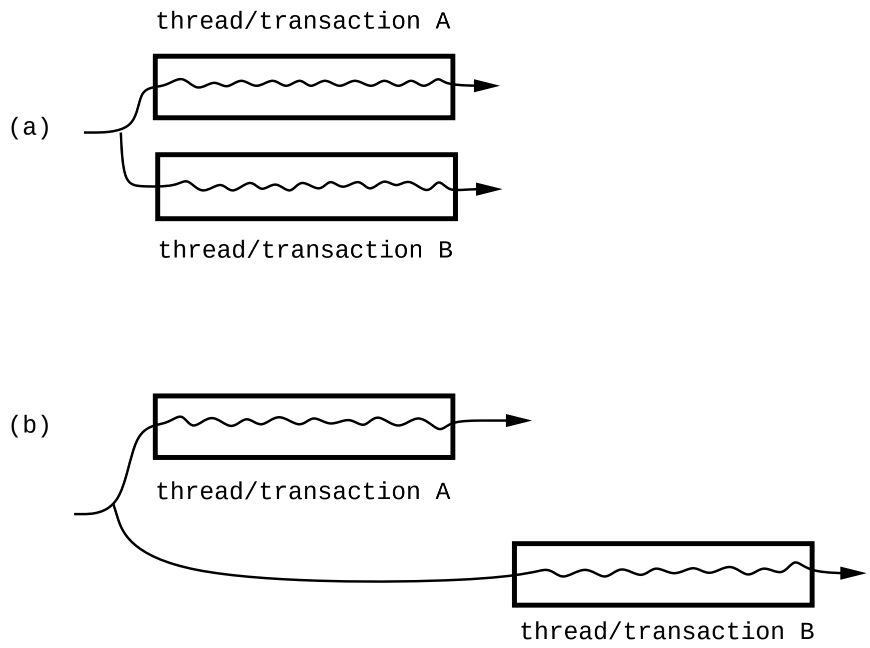

Figure 1.2: Transactions are serializable.

1.3 A Bird's Eye View of the Venari/ML Interfaces and Model

By teasing apart the usual atomicity, serializability, and persistence properties rolled into transactions, and adding the ability for transactions to be multi-threaded, we provide support for the following features, each as a separable component. (The name of the Venari/ML signature and section that discusses it are given in parentheses.)

- Persistence (`PERS`, Section 2.7).
- Undoability (`UNDO`, Section 2.6).
- Reader-writer locks (`RW_LOCK`, Section 2.4).
- Threads (`THREADS`, Section 2.2).
- Skeins (`SKEINS`, `FULL_SKEIN`, Section 2.3).

Our 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 multi-threaded transactions.

Here are some of the more interesting combinations of these pieces, each supporting a slightly different model of computation.

- 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)
  The `VENARI` interface 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. Thus, we commit (to disk) a nested persistent skein's effects when its outermost persistent or undo skein completes. We need to guarantee this behavior to make sense of the case when a persistent skein is nested within an undo skein: if we write to disk upon exit from the persistent skein, then it would be difficult for us to undo changes already made permanent to disk.

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.

Chapter 2

Venari/ML Interfaces

2.1 Top-Level Interface

We have extended Standard ML by providing a *multi-threaded transaction* control abstraction, and various related facilities. This chapter describes the user interface, which is an SML structure with the following signature:

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

        structure Threads : THREADS

        structure Skeins : SKEINS

        structure RW_Lock : RW_LOCK
        structure RW_Ref : RW_REF
        structure RW_Array : RW_ARRAY

        structure Undo : UNDO
        structure Pers : PERS

    end
```

Each element of this structure is described in a separate section below. Roughly speaking, a transaction is a *locking skein* of *threads* whose effects are *undone* if the transaction aborts or made *persistent* if it terminates.

2.2 Threads

The `Threads` structure provides the essential functions of our SML/Threads interface, reviewed briefly here and described fully in [3].[1] It has the following signature:

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

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

        structure M_Ref : M_REF
        structure M_Array : M_ARRAY

        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
        val wait            : condition -> unit

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

The function `fork` starts an invocation of its argument executing as an independent thread of control. No value is returned; the child function is executed for effect. Results can be communicated between threads via shared mutable objects. The function `exit` terminates the current thread, and never returns. The two types `mutex` and `condition` and associated functions provide basic thread synchronization primitives, as described in Sections 2.2.1 and 2.2.2. The type `'a var` provides per-thread mutable values, as described in Section 2.2.3. The two structures `M_Ref` and `M_Array`, which provide "safe refs" and "safe arrays" (refs and arrays protected by mutexes), are discussed in Section 2.2.4.

[1] For hints on programming with the threads abstraction, see [2].

2.2.1 Mutual Exclusion

A `mutex` is a mutual-exclusion lock. The function `mutex` creates a new mutex value. The function `acquire` attempts to lock a mutex and blocks the calling thread until it succeeds. At most one thread may hold a given mutex at any time. Attempting to acquire a mutex already held by the current thread causes an indefinite block. The function `try_acquire` is similar to `acquire`, except that it will not block: if the mutex is already locked it returns `false`. The function `release` unlocks a mutex, giving other threads a chance to acquire it. The function `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` to unit, and then releases `m`. The mutex `m` is released even if an exception is raised in `f`, so use of `with_mutex` can substantially simplify the writing of correct code.

The "mutex refs" and "mutex arrays" of Section 2.2.4 are protected by mutexes.

2.2.2 Conditions

A *condition variable* allows one thread to wait until another thread indicates that some event has occurred. The event is typically a change to shared data, and requires some application-specific test to detect. A mutex is used to prevent one thread from testing the shared data while another is updating it; this mutex is specified at the time the condition variable is created.

The function `condition` creates a new condition value, to be used under the protection of the specified mutex. The `with_condition` function is simply `with_mutex` applied to the condition's mutex. The `signal` operation indicates that an event has occurred; if any threads are waiting on the condition, at least one of them is woken; `broadcast` is similar but wakes all threads waiting on the condition. Both `signal` and `broadcast` can be called without holding the mutex. The function `wait` assumes and checks to see that the mutex is held when called; it atomically releases the mutex and waits to be signaled (the mutex is reacquired before returning). Other threads may execute between the signal and the return from `wait`, so the shared data should be checked in any case, and this is the function of `await` and `vwait`: `await c f` waits until `f ()` evaluates to `true` after a signal; `vwait c f` waits until `f` returns `SOME v`. The mutex is held while `f` is applied for both `await` and `vwait`. Both `await` and `vwait` try their test before they first wait for the signal.

2.2.3 Per-Thread Values

The `var` type constructor provides per-thread state. A `var` is similar to a `ref`, but contents are not shared between threads. A `var` defined in one thread may be undefined in another, so dereferencing may raise the exception `Undefined`. Note the use of imperative type variables.

2.2.4 Mutex Refs and Arrays

Mutex refs and *mutex arrays* provide a degree of safety beyond regular SML refs and arrays. Mutex refs (M_REF) and mutex arrays (M_ARRAY) are protected by mutexes. A thread must hold the mutex in order to read from or write to these objects. This property is enforced by a check at runtime. The functions `with_m_ref` and `with_m_array` call `with_mutex` (Section 2.2.2) on the mutex associated with their respective objects. The functions `pm_ref`, `pm_array`, `pm_arrayoflist`, and `pm_tabulate` create private refs and arrays that can be used by only the thread that creates them. `M_Ref` and `M_Array` parallel the pervasive `Ref` and `Array` structures.

```
signature M_REF =
    sig
        type 'a m_ref
        type mutex

        exception NotOwner

        val m_ref       : '_a * mutex -> '_a m_ref
        val pm_ref      : '_a -> '_a m_ref
        val m_get       : 'a m_ref -> 'a
        val m_set       : 'a m_ref -> 'a -> unit
        val m_inc       : int m_ref -> unit
        val m_dec       : int m_ref -> unit

        val mutex_of    : 'a m_ref -> mutex
        val with_m_ref  : 'a m_ref -> (unit -> 'b) -> 'b
    end

signature M_ARRAY =
    sig
        type 'a m_array
        type mutex

        exception M_Size
        exception M_Subscript
        exception NotOwner

        val m_array           : int * '_a * mutex -> '_a m_array
        val m_arrayoflist     : '_a list * mutex -> '_a m_array
        val m_tabulate        : int * (int -> '_a) * mutex -> '_a m_array
        val pm_array          : int * '_a -> '_a m_array
        val pm_arrayoflist    : '_a list -> '_a m_array
        val pm_tabulate       : int * (int -> '_a) -> '_a m_array
        val m_length          : 'a m_array -> int
        val m_sub             : 'a m_array * int -> 'a
        val m_update          : 'a m_array * int * 'a -> unit

        val mutex_of          : 'a m_array -> mutex
        val with_m_array      : 'a m_array -> (unit -> 'b) -> 'b
    end
```

2.3 Skeins

A *skein* is a new control abstraction that groups together a set of threads. *Full skeins* additionally take as parameters initialization and completion functions; multi-threaded transactions are thus easily constructed as a special case of full skeins.

2.3.1 Simple Skeins

```
signature SKEINS =
    sig
        structure Full_Skein : FULL_SKEIN
        structure Skein_ID : SKEIN_ID

        val skein : ('a -> '_b) -> 'a -> '_b
        val peer_skein : Skein_ID.skein_id -> ('a -> unit) -> 'a -> unit
        val top_skein : ('a -> unit) -> 'a -> unit
    end
```

A *skein* is a group of one or more threads cooperating on some task. Within a skein some ML function (the *body* of the skein) is executed. It may fork threads, but when it returns a value all other extant threads within the skein will be killed; only one thread ever leaves a skein. All held mutexes should be released before return. The function call `skein f a` creates a skein with body `f a`. Figure 2.1 shows the main graphical language we use to describe skein-based systems. A wavy horizontal line represents a thread. A vertical dashed line shows a forking of threads. A solid vertical line denotes the termination of a thread by either completing the execution of its functional argument or calling `Threads.exit`. A thin-lined rectangular box is a skein.

If any thread (including the body thread) running inside a skein raises an uncaught exception, the skein ends. The exception is propagated to the outside and any extant forked threads are killed. See Figure 2.2.

Skeins can contain child skeins. These are terminated when the parent skein finishes. See Figure 2.3.

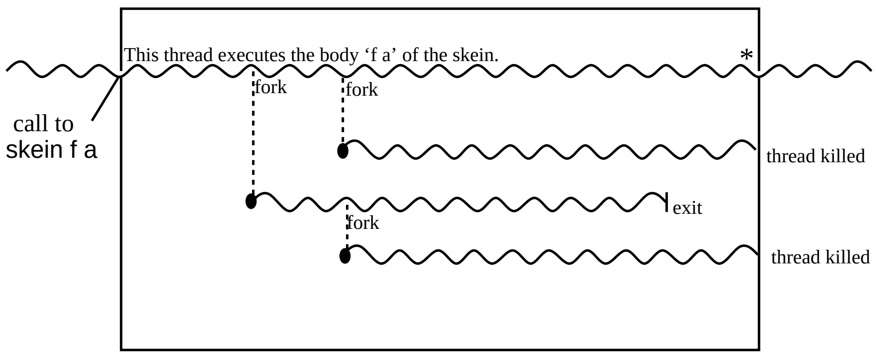

Figure 2.1: A thread executing a function within a skein. The body forks two additional threads, one of which forks again before calling `exit()`. At *, the body returns a value, so the remaining threads are killed and the skein finishes.

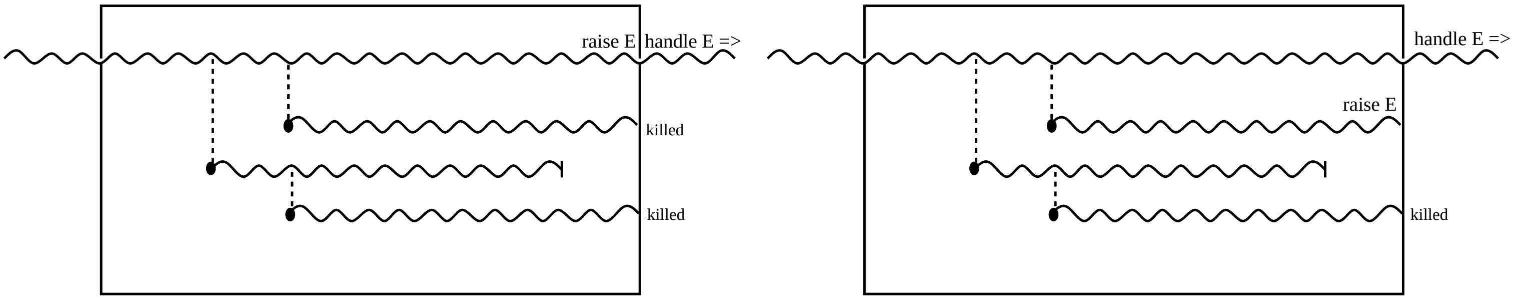

Figure 2.2: Two skeins end with uncaught exceptions. In the example on the left, the exception is in the body thread. In the example on the right, the exception is in a sub-thread. In both cases the exception is passed to the handler.

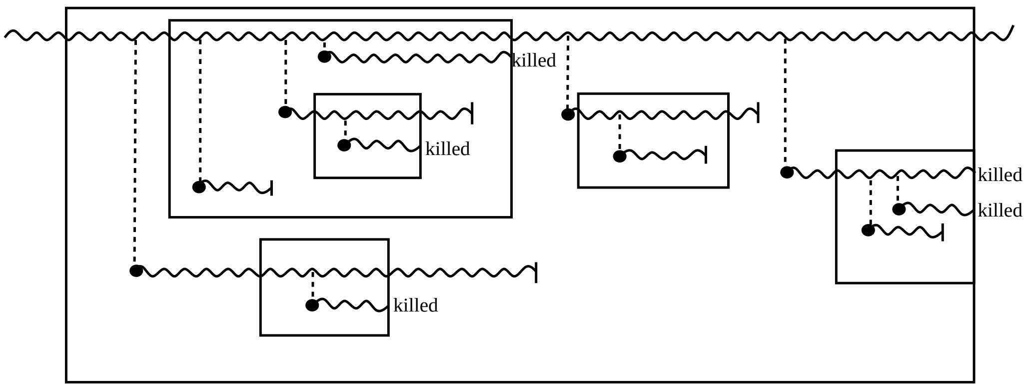

Figure 2.3: A skein with several children. Note that one child is still active when the parent finishes, and is terminated.

There are two functions that allow threads to start skeins which are not child (nested)skeins: `peer_skein` and `top_skein`. The function `peer_skein` creates a sibling of the specified skein (see Section 2.3.3 for a description of skein IDs) and runs the body in the newly created sibling. This sibling skein behaves exactly as if the parent skein created it. [2] The function `top_skein`, a special case of `peer_skein`, creates a new skein at the top level.

Since the skeins created by `peer_skein` and `top_skein` do not run in the calling thread, they are not killed when the calling thread terminates, and they do not necessarily share the calling thread's undo state (see Section 2.6). These functions should be used with care to avoid unexpected side effects.

2.3.2 Full Skeins

Although the skeins abstraction as described above is generally useful for work with threads (for instance it deals gracefully with threads performing speculative computation), it is insufficient for the purposes of implementing transactions. A transaction must execute certain code within a skein, but *after* all threads within that skein have completed or died. (For example, this code might commit persistent changes to disk or release reader-writer locks.) Allowing for such code within skeins also turns out to be generally useful, so we provide the user with this abstraction:

```
signature FULL_SKEIN =
    sig
        datatype 'a result =
            Result of 'a
          | Exception of exn

        exception Abort

        val full_skein :
            (unit -> unit) ->                (* initializing function *)
            ('_b result -> '_b result) ->    (* completing function *)
            ('a -> '_b) ->                   (* body *)
            'a -> '_b
    end
```

The body of a *full skein* is executed in a sub-thread within the skein, while a *control thread* waits for it to complete. Two extra arguments are given to `full_skein`: first, an initializing function, which is called in the control thread before the body thread is forked; and second, a completing function which is called in the control thread after the body has returned and any extant threads have been killed. The completing function is applied to the result of the body, and returns a value which is in turn presented as the result of the call to `full_skein`.

Since the body may complete either by returning a value or by raising an exception, the result is encapsulated with the datatype `result` before passing to the completing function. If the body is successful and returns `v`, the completing function is applied to `Result v`. If the body fails with an uncaught exception `E`, or if a sub-thread raises an uncaught exception `E`, the completing function is applied to `Exception E`. See Figure 2.4.

[2] For example, it follows the same locking rules of Section 2.4 as any other skein that is a child of that parent.

If the body of a skein finishes while sub-skeins are still executing, the sub-skeins are terminated, calling their completing functions with `Exception Abort`. The parent skein's completing function is not called until all sub-skeins have completed. See Figure 2.5.

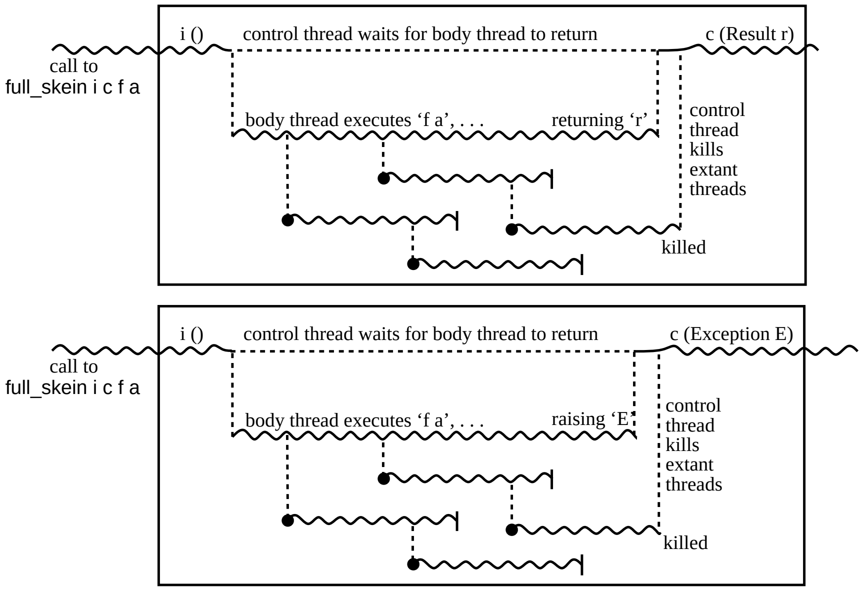

Figure 2.4: Full skeins. Note the initializing and completing functions. Extant threads are killed before the completing function is called.

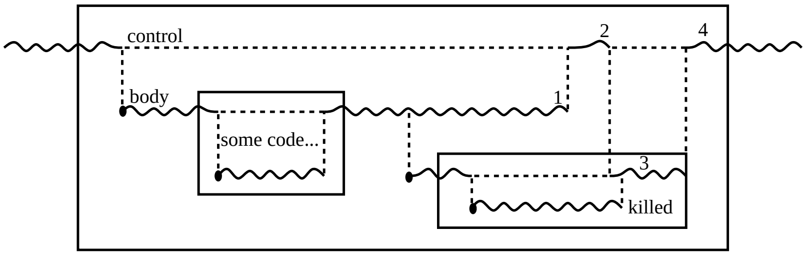

Figure 2.5: A skein that completes while a child is still active. At 1, the body of the parent skein returns some value `v`. At 2, the control thread signals the child skein to complete. At 3, the child skein kills any remaining threads and calls its completing function with `Exception Abort`. At 4, the parent control thread calls its own completing function with `Result v`.

If an exception is raised during the execution of the completing function, or if it returns a value `Exception e`, that exception is reraised, propagating out to the caller of `full_skein`.

Simple skeins are in fact implemented by using full skeins:

```
val skein = Full_Skein.full_skein (fn () => ())
                                  (fn r => r)
```

2.3.3 Skein IDs

Skeins have various attributes that can be read and in some cases written by the user. The structure `Skein_ID` provides functions to access these attributes for the current skein and for its ancestors.

```
signature SKEIN_ID =
    sig
        type skein_id

        exception NoSkein
        val skein_id : unit -> skein_id

        val set_name : string -> unit
        val skein_name : skein_id -> string
        val skein_no : skein_id -> int

        val make_locking : unit -> unit
        val get_locking : skein_id -> bool

        exception Parent
        val parent : skein_id -> skein_id

        val skein_path : skein_id -> string list
    end
```

Each skein has a value of type `skein_id` associated with it. The function `skein_id` returns this value for the current skein, or raises `NoSkein` if called outside any skein. The other attributes are readable given the `skein_id`, but cannot be written for any other than the current skein.

The first attribute is a name, which is simply a string, initially set to the empty string for every new skein. The function `set_name s` sets the name to `s`, while `skein_name sid` returns the name of the skein with ID `sid`. The function `skein_no` returns an integer uniquely identifying a skein. (It is useful for debugging purposes to avoid having to name each skein explicitly.)

The second attribute is a "locking" flag, initially set to `false`. This is connected with reader-writer locks, and will be explained in Section 2.4. The function `make_locking` sets the flag to `true`, while `get_locking` returns the value of the flag. This flag cannot be set to `false`: to do so would violate the desired properties of locks. The function `make_locking` is intended for use within the initializing function of a full skein.

The third attribute is the `skein_id` of the parent skein. The function `parent` obtains this, or raises `Parent` if the given skein is top-level (i.e., has no parent).

The function `skein_path` returns the list of skein names from the given skein up to the top level. It is defined as follows:

```
fun skein_path sid = (skein_name sid) ::
                        (skein_path (parent sid) handle Parent => [])
```

2.4 Reader-Writer Locks

We provide *reader-writer locks* to enable the user to enforce isolation and serializability of given skeins. They are used primarily in transactions (see Section 2.8), which must be serializable, but the user may find reader-writer locks valuable in other contexts. Analogously to using mutexes to protect mutex refs and mutex arrays, we use reader-writer locks to protect the "reader-writer refs" and "reader-writer arrays" of Section 2.5.

```
signature RW_LOCK =
    sig
        eqtype rw_lock

        exception NotLocking
        exception Read
        exception Write

        val create        : unit -> rw_lock
        val acquire_read  : rw_lock -> unit
        val acquire_write : rw_lock -> unit
        val read          : rw_lock -> ('a -> 'b) -> 'a -> 'b
        val write         : rw_lock -> ('a -> 'b) -> 'a -> 'b
    end
```

Assume at first that all skeins have the "locking" flag set (see Section 2.3.3); we will describe the general case later.

Locks are held on a per-skein basis. A lock is created by a call to `create`. It is acquired for reading or writing by a call to `acquire_read` or `acquire_write` respectively. A (thread within a) skein can perform reads and writes on the data protected by a lock, subject to the following simple conditions:

- A skein may read if it holds the lock in read or write mode, and all writers are ancestors of the skein.
- A skein may write if it holds the lock in write mode, and all readers *and* writers are ancestors of the skein.

Here, "readers" and "writers" refer to the extant skeins that hold the lock in read or write mode, respectively. The acquiring functions block until the corresponding condition is satisfied, and when they return the calling skein holds the lock in the specified mode. The user should pay close attention to the order in which locks are acquired, so as to avoid deadlock.

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 corresponding condition will hold 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 a skein calls `read` or `write` without holding the lock in the appropriate mode, the exception `Read` or `Write` will be raised; otherwise, `read` and `write` will block until the condition is satisfied.

When a skein completes successfully (i.e., with a value other than an exception), all currently-held locks are handed off to the parent skein (or released if there is no parent). If the skein finishes with an exception (i.e., if the completing function of the full skein either raises an exception or returns `Exception e`), all currently-held locks are released. Note that these actions take place *after* the completing function is executed. These "anti-inheritance" rules apply to all kinds of skeins, including undo skeins (Section 2.6), persistent skeins (Section 2.7), and transactions (Section 2.8).

For the general case, in which some skeins do not have the "locking" flag set, the rules extend simply by considering *families* of skeins, which consist of a single locking skein and its non-locking descendants. Locks are held by a family (actually, by the locking skein at its root), and acquired on behalf of the family. Locks are handed off to the enclosing family. This enables users to wrap functions with `skein` for purposes of flow-control without confusing the locking behavior of their programs.

All functions in this structure except for `create` must be called from within a locking family of skeins; otherwise the exception `NotLocking` is raised.

2.5 Safe State

Two structures are provided to help the user manipulate state safely. *Reader-writer refs* (RW_REF) and *reader-writer arrays* (RW_ARRAY) are protected by reader-writer locks; in order for a thread to access these objects, its skein must hold the `rw_lock` (for reading or writing, as appropriate). The accessing functions below will call `RW_Lock.read` or `RW_Lock.write` to ensure that the read or write condition from Section 2.4 holds.

Because they provide an additional level of safety using reader-writer locks, reader-writer refs and reader-writer arrays are "safer" than mutex refs and mutex arrays (Section 2.2.4).

```
signature RW_REF =
    sig
        type 'a rw_ref
        type rw_lock

        val rw_ref    : '_a * rw_lock -> '_a rw_ref
        val rw_get    : 'a rw_ref -> 'a
        val rw_set    : 'a rw_ref -> 'a -> unit
        val rw_inc    : int rw_ref -> unit
        val rw_dec    : int rw_ref -> unit

        val lock_of   : 'a rw_ref -> rw_lock
    end


signature RW_ARRAY =
    sig
        type 'a rw_array
        type rw_lock

        exception RW_Size
        exception RW_Subscript

        val rw_array         : int * '_a * rw_lock -> '_a rw_array
        val rw_arrayoflist   : '_a list * rw_lock -> '_a rw_array
        val rw_length        : 'a rw_array -> int
        val rw_sub           : 'a rw_array * int -> 'a
        val rw_tabulate      : int * (int -> '_a) * rw_lock -> '_a rw_array
        val rw_update        : 'a rw_array * int * 'a -> unit

        val lock_of          : 'a rw_array -> rw_lock
    end
```

RW_Ref and RW_Array parallel the pervasive Ref and Array structures. The lock_of functions return the lock associated with an rw_ref or rw_array. If any of the accessing functions (rw_get, rw_set, rw_inc, rw_dec, rw_sub, or rw_update) are called when the lock is not held in the appropriate mode, the RW_Lock.Read or RW_Lock.Write exceptions will be raised.

2.6 Undoability

This structure allows users to make undoable changes to the store, an essential feature of transactions. It does so by providing a specialized form of skein, an *undo skein*, created by providing particular initializing and completing functions to (a partial application of) Full_Skein.full_skein.

```
signature UNDO =
    sig
        val undo_skein : ('a -> '_b) -> 'a -> '_b

        exception Restore of exn

        val exn2restore_skein : ('a -> '_b) -> 'a -> '_b
        val exn2restore : ('a -> 'b) -> 'a -> 'b
        val restore2exn : ('a -> 'b) -> 'a -> 'b
    end
```

The expression undo_skein f a evaluates f a inside an undo skein. If the exception Restore E is raised (and not caught) within f, the skein will end and all changes to the store made up to that point will be undone. The exception Restore E will be propagated out to the caller of undo_skein. Note that the changes undone include those done within any sub-skeins.

Undo skeins are just a special kind of full skein; they are easily implemented using full skeins as follows:

```
        val undo_skein = Full_Skein.full_skein init_undo (complete_undo false)
```

where init_undo does appropriate initialization (e.g., setting the "locking" flag). When the boolean argument to complete_undo is false, the skein will only restore if the Restore exception is raised from within it.

The functions exn2restore_skein, exn2restore, and restore2exn are used to manipulate exceptions in the context of an undo skein. They are defined as follows:

```
        val exn2restore_skein =
            full_skein
              (fn ()=>())
              (fn Exception exn => Exception (Restore exn)
                | Result x => Result x)
        fun exn2restore f a = (f a) handle exn => raise Restore exn
        fun restore2exn f a = (f a) handle Restore exn => raise exn
```

and have their most obvious use in a piece of code like this:

```
        fun restore_on_exn f = restore2exn (undo_skein (exn2restore_skein f))
```

which defines a function that will execute inside an undo skein and restore the state if any exceptions are raised. The `exn2restore` function is similar to `exn2restore_skein`, except that an exception raised in a subthread within f will not be converted to a `Restore` exception, and thus will abort the undo skein without causing changes to be undone. For this reason, exn2restore_skein is the correct choice in most circumstances.

Undo skeins have the "locking" flag set (Section 2.4). If the persistent store is initialized, an undo skein that completes successfully and has no undo skeins or persistent skeins among its ancestors will commit any changes to the persistent store (Section 2.7).

Note that the semantics of undo is defined only with respect to the store. In particular, it is not defined with respect to I/O, for example, reading from a file or printing to the terminal. Hence, programmers should take care when doing I/O within an undo skein.

2.7 Persistence

The other major feature of our work on transactions is the persistent value store. A persistent value is one that outlives the computation that created it. Any first-class SML value can be made persistent. A top-level *persistent skein* is a group of threads whose changes to the store are made permanent.

```
signature PERS =
    sig
        exception CommitFailed
        val pers_skein : ('a -> '_b) -> 'a -> '_b

        exception PersInitFailed
        val init : string * string * bool -> unit

        type identifier
        exception Unbound
        val make_id : string -> identifier
        val bind : identifier * 'a -> unit
        val unbind : identifier -> unit
        val retrieve : identifier -> 'a
    end
```

The function `init` initializes a persistent store, and has the effect of obtaining a pointer, which we call the *persistent handle*, to a persistent store. Persistence is implemented through the RVM system[10], and the first two arguments are the names of the RVM log and data files respectively; from Venari/ML's viewpoint, these two files represent a persistent store. If the third (boolean) argument is `true`, the handle points to a new, empty persistent store; otherwise, the handle points to a previously saved one.

The expression `pers_skein f a` evaluates `f a` in a skein. If the persistent store is initialized and the skein has no undoable or persistent ancestors, then when it completes changes are committed to disk. The "locking" flag is set (Section 2.4).

Both `init` and `pers_skein` may raise an exception because of I/O problems like file access errors or other rare events encountered by RVM.

The other functions deal with identifiers. The persistent store is a map from identifiers (which the user creates from strings) to values. `make_id` creates an identifier, `bind` adds a binding to the map, unbind removes a binding, and retrieve returns the bound value. The function retrieve raises the exception Unbound if the given identifier is not bound in the persistent store. Notice here a need for dynamic types, which SML does not currently support. SML cannot statically determine whether the type of the value returned by a retrieve of some identifier is the same as the type of the value when it was initially bound through a bind.

2.8 Transactions

The previous parts of the VENARI interface expose functions which are necessary for implementing transactions, and which have other more general uses. The various features are all used in VENARI's main function:

```
val transact : ('a -> '_b) -> 'a -> '_b
```

This function evaluates its argument within a skein, known as a *transaction*. The "locking" flag is set (Section 2.4) so the transaction holds its own locks. Within the transaction, further calls to transact will create nested transactions just as with skeins.

Note how we can succinctly implement the function transact using full skeins:

```
fun init_transact = (Pers.init_pers ();
                     Undo.init_undo () )

val complete_transact = Undo.complete_undo true

val transact = Full_Skein.full_skein init_transact complete_transact;
```

where setting Undo.complete_undo's boolean argument to true signifies that the transaction will always restore when it fails to complete successfully, regardless of whether the Restore exception is raised from within it.[3]

2.8.1 Transaction Guarantees

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.

[3]In contrast to the implementation of undo_skein in Section 2.6.

If the functions executed within transactions have no effects except through the use of the safe state described in Section 2.5, 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. Note that the guarantees above hold even if non-transactional skeins or threads are invoked within the transactions involved.

2.8.2 Hints at Using Transactions

Programmers must take care to avoid deadlock situations. Deadlock will arise when, for example, transaction A acquires lock L_1, transaction B acquires lock L_2, then A attempts to acquire L_2 and B attempts to acquire L_1. Programmers can prevent deadlocks by obeying strict lock-acquisition ordering, by using coarser-grain locking, or by some combination of the two.

By specifying the lock-acquisition sequence precisely, we can avoid these deadlocks entirely. If for every pair of locks (L_1, L_2), we decide that one will always be acquired before the other, then deadlocks of the type described above cannot occur. A related deadlock type involves lock-promotion (upgrading ownership of a lock from read mode to write mode). If several transactions acquire a lock in read mode, and then try to acquire it in write mode, all will block. It is best to avoid this by always acquiring a lock in write mode directly if it will eventually be needed in write mode, or by ensuring that only one transaction will attempt to acquire the lock in write mode. Unfortunately, these lock-ordering methods can complicate and restrict programs significantly.

Coarse-grain locking (using a single lock to protect large amounts of data) can make deadlocks less likely by reducing the number of locks a transaction must acquire to accomplish a task. For example, if two `rw_refs`, `r1` and `r2`, are nearly always accessed together, then we can make them share a single `rw_lock` to prevent possible deadlocks from transactions acquiring `r1`'s and `r2`'s locks in different orders. The disadvantage of coarse-grain locking is that it restricts the potential concurrency; if one transaction needs to access only `r1` and another needs to access only `r2`, they cannot proceed concurrently unless `r1` and `r2` have distinct locks.

Another factor to consider in determining the granularity of locking is speed. In our current implementation, the repeated lock creation and manipulation required with a fine locking granularity carries a heavy performance penalty. So even though we have increased the potential level of concurrency by allowing locking at a fine level, the observed latency of user requests may be greater.

Finally, as for undo, the semantics of transactional abort with respect to I/O is not defined. Hence, programmers should take care when doing I/O from within a transaction.

Chapter 3

Some Small Examples

In this chapter, we consider some small examples to show individual and combined uses of the Venari/ML interfaces.

3.1 Threads

A multi-threaded application might use a logical clock to establish an order of events. A signature for a logical clock is shown in Figure 3.1. The function `get_time` increments the clock and return a new, unique time.

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

Figure 3.1: Signature for a Logical Clock

```
structure SimpleClock : CLOCK =
    struct
        structure T = Venari.Threads
        structure M = T.M_Ref

        val time = M.m_ref (0, T.mutex())

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

Figure 3.2: A Simple Logical Clock

The clock might be implemented as shown in Figure 3.2. The logical time is stored in a `m_ref`, `time`. `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

Our use of a *mutex ref* (see Section 2.2.4) gives us this protection. To ensure that each caller is given a unique time, the function `get_time` wraps a `with_m_ref` around the calls to increment and read `time`.

3.2 Persistence

Suppose we want to keep the clock in Figure 3.2 in the persistent store. Since the times provided by the clock are unique as well as ordered, it could be used as a source of unique identifiers, which would be particularly useful in a persistent environment.

A persistent implementation of the clock is shown in Figure 3.3. To do this, we store the logical time in the m_ref `time`.

```
structure PersClock : CLOCK =
   struct
        structure T = Venari.Threads
        structure P = Venari.Pers
        structure M = T.M_Ref

        val time : int M.m_ref =
            let val time_id = P.make_id "*TIME*"
            in
                P.retrieve time_id
                handle P.Unbound =>
                       let val t = M.m_ref (0, T.mutex())
                       in
                           P.pers_skein P.bind (time_id, t);
                           t
                       end
            end

        fun get_time () =
            P.pers_skein (M.with_m_ref time)
            (fn () => (M.m_inc time; M.m_get time))
    end
```

Figure 3.3: A Persistent Logical Clock

We need to initialize `time` with its previous value, if there is one. To do this, we attempt to retrieve the value and watch for the exception `Unbound`, which will be raised by `retrieve` if this value is not in the persistent store. We handle the exception by initializing the value in the persistent store.

We also wrap a persistent skein (pers_skein) around the call to with_m_ref in the function get_time. We use the persistent skein to ensure that the new value for time is properly recorded before we return it. Top-level persistent skeins write out any changes to the persistent store before exiting.

3.3 Undo

Undoability can be very useful for backtracking. Suppose we have an unordered list of side-effecting functions and we want to find a valid ordering for them if such an ordering exists. This situation could arise if several people were cooperating in the creation of a database, for example. One person might be responsible for creating the initial entries; others would be responsible for filling in various fields, some of which might require someone else's fields to be filled in already. It would be nice if we could just let everyone add their functions to a list which would be executed later in an acceptable order.

The function valid_ordering, shown in Figure 3.4 takes a list of functions and tries to find a valid ordering for them. The functions should raise an exception if something goes wrong; otherwise, it will be assumed that everything is ok. If a valid ordering is found, the side effects remain; otherwise they are undone.

```
structure U = Venari.Undo
exception NotValid

fun valid_ordering function_list =
    let fun try result [] [] = rev result
          | try result retry [] = raise U.Restore NotValid
          | try result retry (f::rest) =
            U.undo_skein (fn () => (U.exn2restore_skein f ();
                                    try (f::result) [] (retry@rest))) ()
            handle U.Restore _ => try result (f::retry) rest
    in
        U.undo_skein (try [] []) function_list
        handle U.Restore NotValid => raise NotValid
    end
```

Figure 3.4: Find a valid ordering of function calls.

The function valid_ordering defines a function, try which does most of the work. Try takes three arguments. The first argument, the result list, is the list of functions executed so far, in reverse order. The second argument, the retry list, is the list of functions that have been unsuccessfully attempted. The third list, the function list, is the list of functions that have not been tried yet.

If both the retry and function lists are empty, a valid ordering has been found. We return the result list, calling rev first to return the list in its proper order. If the function list is empty but the retry list is not, we have run out of combinations to try. In this case, there is no valid ordering starting with the current result list. We raise restore to undo the effects of the function most recently added to the result list.

If the function list is not empty, we first try to execute the first function on the list, and then try to execute the rest of the list after putting the retry list back on the function list. If either of these fails, we put the function on the retry list and try again with the next function on the function list.

3.4 Transactions

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.

```
structure U = Venari.Undo;

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

Figure 3.5: Transfer money between bank accounts.

The function `transfer`, shown in Figure 3.5, 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))
```

3.5 Multi-Threaded Transactions

For a more complicated example of multi-threaded transactions, suppose we wanted to tally a list of votes where each vote may be for one of a number of candidates running for a particular office. We could do this as follows:

1. Number the candidates.
2. Create an array of integers, with each element initialized to 0. Each element corresponds to one of the candidates.
3. Walk through the list of votes, incrementing the appropriate element of the array for each vote.
4. The value of each element is the number of votes for the corresponding candidate.

The signature in Figure 3.6 shows an interface that supports this method of vote-counting. The function `voting_array` creates a new, properly-initialized array. The function `add_vote` takes a candidate number and increments the vote count for the corresponding candidate. The function `how_many` takes a candidate number and returns the number of votes for the corresponding candidate. The `Subscript` exception is raised if a candidate number is out of bounds. The numbering of candidates is assumed to be consecutive starting at 0.

```
signature VOTING_ARRAY =
    sig
        exception Subscript
        type voting_array

        val voting_array : int -> voting_array
        val add_vote : voting_array -> int -> unit
        val how_many : voting_array -> int -> int
    end
```

Figure 3.6: Signature for a Voting Array

Figure 3.7 shows an implementation of this interface. Each element of the array is a mutex ref which will hold the vote count for the corresponding candidate. The functions `add_vote` and `how_many` use `with_m_ref` to coordinate access to each candidate's vote count. This allows multiple threads to update the array at the same time without interfering with each other.

Figure 3.8 shows a function, `tally_votes` which takes a voting array and a list of votes, and records all the votes in the voting array. `Tally_votes` uses multiple threads to speed up the handling of long vote lists. It defines some local variables and functions to assist in this task. `Thread_count` contains the number of threads started. This value will be compared with `done_count` to determine when all the threads have finished. The condition `done_cond` is signaled whenever a thread finishes processing its list.

The function `process_votes` processes a vote list, indicating that it has finished by incrementing `done_count` and signaling `done_cond`. The function `launch_threads` breaks off pieces of the vote list and starts threads to handle them until the whole list has been handled. The unhandled part of the vote list is stored in `unprocessed_votes`. The function `first_n` removes the first `n` elements from `unprocessed_votes` and returns them.

The function `wait` waits for `done_count` to match `thread_count`. This is done to ensure that all the threads are finished updating the voting array before we return.

The body of the function `tally_votes` executes `launch_threads` and `wait` inside a `skein`. Normally, exceptions raised in a thread are not passed outside the thread. The `skein` will trap any

```
structure Voting_Array : VOTING_ARRAY =
    struct
        structure A = Array
        structure T = Venari.Threads
        structure M = T.M_Ref

        exception Subscript = A.Subscript

        type voting_array = int M.m_ref array

        fun voting_array size =
            A.tabulate (size, (fn _ => M.m_ref (0, T.mutex())))

        fun add_vote vote_array candidate =
            let val count = A.sub (vote_array, candidate)
            in
                M.with_m_ref count (fn () => M.m_inc count)
            end

        fun how_many vote_array candidate =
            let val count = A.sub (vote_array, candidate)
            in
                M.with_m_ref count (fn () => M.m_get count)
            end
    end
```

Figure 3.7: A Voting Array

```
structure T = Venari.Threads; structure M = T.M_Ref;
structure S = Venari.Skeins;

fun tally_votes voting_array vote_list =
    let val thread_count = M.m_ref (0, T.mutex())
        val done_count = M.m_ref (0, M.mutex_of thread_count)
        val done_cond = T.condition (M.mutex_of thread_count)

        fun process_votes vl =
            (app (Voting_Array.add_vote voting_array) vl;
             T.with_condition done_cond (fn () => M.m_inc done_count);
             T.signal done_cond)

        fun launch_threads () =
            let val unprocessed_votes = M.pm_ref(vote_list)
                fun first_n n =
                    let fun f_n 0 res = res
                          | f_n n res =
                            if null (M.m_get unprocessed_votes) then res
                            else let val (h::t) = M.m_get unprocessed_votes
                                 in
                                     M.m_set unprocessed_votes t;
                                     f_n (n-1) (h::res)
                                 end
                    in
                        f_n n []
                    end
            in
                while not (null (M.m_get unprocessed_votes))
                    do let val first_10 = first_n 10
                       in
                           T.fork (fn () => process_votes first_10);
                           M.with_m_ref thread_count (fn () =>
                                                  M.m_inc thread_count)
                       end
            end

        fun wait () =
            T.with_condition done_cond
            (fn () => T.await done_cond (fn () => (M.m_get done_count) =
                                                  (M.m_get thread_count)))
    in
        S.skein (fn () => (launch_threads (); wait())) ()
    end
```

Figure 3.8: Tally votes using a voting array.

such exceptions and reraise them. This allows the caller to know that something went wrong and prevents the `wait` function from waiting forever for any thread that dies prematurely.

If an exception is raised inside one of the threads, the results are invalid. We can prevent partial results from being added to a voting array by wrapping a transaction around the call to `tally_votes`:

```
Venari.transact (tally_votes vote_array) vote_list
handle _ => print "tally_votes failed!\n"
```

3.6 Concurrent Multi-Threaded Transactions

Now suppose we wanted to count votes for more than one office. We could do this with an array of voting arrays, where each voting array contains the votes for one office. Figure 3.9 shows the implementation of a function, `tally_offices`, which uses such an array to count votes on a number of offices. `Tally_offices` takes two arguments: `office_array` and `votes_list`.

`Office_array` is an array of pairs of reader-writer locks and voting arrays where the subscript corresponds to the office number. Reader-writer locks provide isolation between transactions. If two transactions were to update the same voting array at the same time, the results would not be serializable, and we would have problems if we needed to abort one of the transactions. Thus, each voting array in `office_array` has its own reader-writer lock to ensure that the transactions updating them are serializable.

`Votes_list` is a list of pairs of office numbers and vote lists. There is no restriction on the number of times an office may appear in the list; voting in several cities could generate several vote lists for a particular office.

The function `tally_offices` is similar to the function `tally_votes`, described in Section 3.5. The primary difference is in the function `process_office`. `Process_office` calls `tally_votes` on the vote list and voting array associated with a particular office. We do this inside a transaction to allow a single vote count to fail and be redone later without requiring a full recount. We acquire the write lock before calling `tally_votes` to ensure that multiple transactions do not update the same voting array at the same time. For simplicity, we handle aborts by printing a warning message; in real life, we would want to save the failed office and vote list somewhere for later consideration.

```
structure A = Array; structure L = Venari.RW_Lock;
structure S = Venari.Skeins; structure T = Venari.Threads;
structure M = T.M_Ref; structure U = Venari.Undo;
structure VA = Voting_Array;

fun tally_offices office_array votes_list =
    let val thread_count = M.m_ref (0, T.mutex())
        val done_count = M.m_ref (0, M.mutex_of thread_count)
        val done_cond = T.condition (M.mutex_of thread_count)

        fun process_office (office, vote_list) =
            let val (lock, voting_array) = A.sub (office_array, office)
            in
                (Venari.transact
                 (fn vl => (L.acquire_write lock;
                            L.write lock (tally_votes voting_array) vl))
                 vote_list)
                handle _ =>
                    print ("WARNING: vote count on office "^
                           (makestring office)^" failed!!!\n")
            end

        fun process_offices vl =
            (app process_office vl;
             T.with_condition done_cond (fn () => M.m_inc done_count);
             T.signal done_cond)

        fun launch_threads () =
            let val unprocessed_votes = M.pm_ref (votes_list)
                fun first_n n = {defined exactly as in tally_votes}
            in
                while not (null (M.m_get unprocessed_votes))
                    do let val first_10 = first_n 10
                       in
                           T.fork (fn () => process_offices first_10);
                           M.with_m_ref thread_count (fn () =>
                                                       M.m_inc thread_count)
                       end
            end

        fun wait () = {defined exactly as in tally_votes}
    in
        S.skein (fn () => (launch_threads (); wait())) ()
    end
```

Figure 3.9: The definition of tally_offices.

3.7 Skeins

In addition to their use in providing transactions, undoability, and persistence, `skeins` are useful in situations where threads need to guarantee that they will clean up after themselves. The function `run_in_xterm`, shown in Figure 3.10, is an example of this situation. It executes a function in a separate thread inside an `xterm` window. The `xpipe` structure, defined elsewhere, handles the details of the actual interface with the `xterm`.

We could just start a thread that would create the `xterm`, execute `f`, and then destroy the `xterm`, but such an implementation would leave the `xterm` orphaned if the thread were to exit prematurely. We avoid this problem by using the `full_skein` interface.

In addition to the normal `skein` arguments, a `full_skein` takes an `init` function and a `complete` function. The `init` function, which in our case sets up a new `xterm`, is executed before the body of the `full_skein`. The `complete` function, which we use to close down the `xterm`, is executed immediately after the body returns. We ignore the `xpipe.xpipeIo` exception, which will be raised if the user closes the `xterm` (by typing Control-D, for example) before the `complete` function is executed.

```
structure T = Venari.Threads
structure FS = Venari.Skeins.Full_Skein

fun run_in_xterm (f : xpipe.id -> unit) name =
    let val the_pipe = ref (xpipe.empty_id)

        fun init _ = (the_pipe := xpipe.create ("");
                      xpipe.set_name (!the_pipe) name;
                      xpipe.clear_screen (!the_pipe) ())

        fun cleanup _ = (xpipe.close (!the_pipe) ();
                         the_pipe := xpipe.empty_id;
                         FS.Result ()) handle xpipe.xpipeIo (s) => FS.Result ()

        fun run () = f (!the_pipe) handle xpipe.xpipeIo _ => ()
    in
        T.fork (FS.full_skein init cleanup run)
    end
```

Figure 3.10: Run a function in an xterm.

The function `hello_world`, shown below, uses `run_in_xterm` to print “Hello World!” in an `xterm` named “hello”. It waits for the user to press return, indicating that it is ok to return and remove the window.

```
fun hello_world () =
    let fun hw xp = (xpipe.prline xp "Hello World!"; xpipe.read xp (); ())
    in
        run_in_xterm hw "hello"
    end
```

3.8 A Concurrent Iterator

In this section, we consider the implementation of a concurrent iterator. We want to allow a group of threads to walk through a list, with each element being handled exactly once. If the "next" element is locked, another thread may be handling it and we want to move on instead of waiting for the lock. We do not assume that all threads are alike; some threads may not be able to handle some elements. If a thread is unable to find anything that it can handle, but was unable to see some elements because they were locked, it will wait for a lock to be released and try again. Each thread signals a waiting thread when it is done looking at the list.

The signature for our concurrent iterator is shown in Figure 3.11. The function `iterator` takes a list of objects and creates an iterator containing those objects. The function `next_item` takes an iterator and an "ok" function and either returns an acceptable (as defined by the "ok" function) object or raises the exception `Empty` if no such object exists.

```
signature CONCURRENT_ITERATOR =
    sig
        exception Empty
        type 'a iterator
        val iterator : 'a list -> 'a iterator
        val next_item : 'a iterator -> ('a -> bool) -> 'a
    end
```

Figure 3.11: Signature for a Concurrent Iterator

The `iterator` type, shown in Figure 3.12, consists of a list of elements, `elt_list`, a logical clock, `last_unlock`, which is incremented whenever a thread finishes its current pass through the element list and has released its locks, and a `condition`, `unlock`, which is signaled when the logical clock is incremented. Each element is protected by a `mutex` and is accompanied by a "done" flag to indicate whether it has been handled yet.

The function `next_item`, shown in Figure 3.13, attempts to find an object in the iterator that the calling thread can handle. It defines a few local variables and functions to assist in this task. `Unlock`, `last_unlock`, and `elt_list` contain the corresponding parts of the iterator, `iter`, and the current logical time is stored in `start_loop_time`. The function `signal_waiters` increments the logical clock, `last_unlock`, and signals any threads that are waiting on the condition `unlock`. The clock is incremented inside a `top_skein` to ensure that any later undoing will not generate an inconsistent value for the clock.

Most of the work is done by the function `get_item`. The first argument to `get_item` is `true` if a locked item has been skipped, `false` otherwise. The remaining argument is the part of the element list that we haven't seen yet.

If the element list is empty, we signal one of the waiting threads, if any. If we had not skipped any locked items, we just raise the `Empty` exception. If we had to skip some items because they were locked, we wait for someone else to release their locks and then try again. We determine whether or not someone else has released their locks by looking at the logical clock. We saved the "time" in the ref `start_loop_time` before starting our pass through the list, and we increment the clock ourselves before signaling any waiting threads. Thus, if no other threads have finished going through the list since we started, the value of the logical clock should be `start_loop_time` + 1. If

```
structure Concurrent_Iterator : CONCURRENT_ITERATOR =
    struct
        structure T = Venari.Threads
        structure S = Venari.Skeins

        exception Empty

        type 'a element = {elt : 'a,
                           lock : T.mutex,
                           done : bool ref}

        type 'a iterator = {unlock : T.condition,
                            last_unlock : int ref,
                            elt_list : 'a element list}

        fun iterator l =
            let val m = T.mutex()
            in
                {unlock = T.condition m,
                 last_unlock = ref(0),
                 elt_list =
                 map (fn e => {elt=e, lock=T.mutex(), done=ref(false)}) l}
            end
```

Figure 3.12: The first part of the Concurrent_Iterator structure.

```
fun next_item (iter : 'a iterator) (ok : 'a -> bool) =
    let val {unlock, last_unlock, elt_list} = iter

        val start_loop_time =
            ref (T.with_condition unlock (fn () => !last_unlock))

        fun signal_waiters () =
            S.top_skein (fn () =>
                         (T.with_condition unlock (fn () =>
                                                   inc last_unlock);
                          T.signal unlock))

        fun get_item false [] =
            (signal_waiters(); raise Empty)

          | get_item true [] =
            (signal_waiters();
             T.with_condition unlock
             (fn () =>
                  (T.await unlock (fn () =>
                                   (!start_loop_time + 1) <
                                   !last_unlock);
                   start_loop_time := !last_unlock));
             get_item false elt_list)

          | get_item found_locked ({elt,lock,done}::rest) =
            let val got_lock = T.try_acquire lock
                val got_one =
                    got_lock andalso (not (!done)) andalso (ok elt)
            in
                if got_one then done := true else ();
                if got_lock then T.release lock else ();
                if got_one then (signal_waiters(); elt)
                else
                    get_item (found_locked orelse not got_lock) rest
            end
    in
        get_item false elt_list
    end
end
```

Figure 3.13: The rest of the Concurrent_Iterator structure.

the value is greater than `start_loop_time` + 1, some other thread must have finished going through the element list, so we may see some previously locked elements if we try again.

If the element list is not empty, we try to find out if the current thread can handle the first item on the list. If we can get the lock, the element has not already been handled by another thread, and this thread can handle it, then we can return the element. We signal any waiting threads before returning.

If the current thread cannot handle the first item on the list for some reason, we call `get_item` on the rest of the list. If we cannot get the lock, we set `get_item`'s first argument to `true` to remind us that we had to skip an item.

A sample use of a concurrent iterator is shown below. Suppose we have an unordered list of changes to a database and two threads attempting to make those changes. Thread 1 can only handle simple operations, while thread 2 can handle anything. We assume that the functions `make_simple_change`, `make_change` and `is_simple` are defined elsewhere.

Thread 1 will go through the list looking for and handling simple changes. Thread 2 will go through the list, handling anything it finds. We do not have to worry about thread 2 reaching the end of the list and exiting while thread 1 is looking at a change that it cannot handle. The iterator will force thread 2 to wait until thread 1 is finished before declaring that the iterator is really empty; if thread 1 cannot handle the last change, it will be given to thread 2.

```
structure T = Venari.Threads;
structure CI = Concurrent_Iterator;

val change_iter = CI.iterator change_list;

fun thread_1 () =
    (while true do make_simple_change (CI.next_item change_iter is_simple))
    handle CI.Empty => ();

fun thread_2 () =
    (while true do make_change (CI.next_item change_iter (fn _ => true)))
    handle CI.Empty => ();

T.fork thread_1;
T.fork thread_2;
```

Chapter 4

A Larger Example

We now turn to a more substantial application of the Venari extensions. The application makes extensive use of nested, multi-threaded transactions, and is intended to demonstrate the practical use of our extensions. For brevity, we will always assume the declaration "`structure V = Venari`" in this section.

4.1 The Application

Recording and managing bibliographic information is a task which few enjoy. Many users of the LaTeX [8] and *Scribe* [17] document preparation systems use BibTeX [13] to facilitate bibliography-building. BibTeX derives bibliographic information from entries in `.bib` files, so most users enter this information by hand and maintain one or more personal `.bib` files containing frequently used references.

We want to provide a convenient way for BibTeX users to collect their bibliographic entries, share entries with others, and quickly locate desired entries.

4.2 The BIBS Interface

The BIBS application builds on the "set" abstraction, which is a mutable collection of bibliographic entries. These sets may be registered by name, to allow access in a later session or by other users. A single BIBS server provides operations on sets to multiple clients. In a more advanced implementation, the clients and server might run in different SML processes, on different machines, but at present the server and all clients live in a single SML process.

A BibTeX bibliographic entry (Figure 4.1) has a type (e.g., `book`, `article`, `manual`), a key used to refer to it in documents (e.g., `seuss88`), and a set of field names (`author`, `title`) with associated values.

The signature for a BIBS client appears in Figure 4.2. It provides functions to create and manipulate sets, along with the `run` function, which invokes an interactive BIBS interface. The interactive interface provides the same functionality, but with a more concise and convenient command language.

The `search` function takes an existing `set` (of `.bib` entries), a field specification, and a value to match. It returns a new set that contains all entries in the original set for which the indicated field(s) contain a word matching the `matchval`. The `matchval`, `Exact s`, matches only the word `s`, while `Prefix s` matches any word beginning with `s` (case is always ignored). So, for example,

```
@book{seuss88,
        author = "Dr. Seuss",
        title = "Green Eggs and Ham",
        publisher = "Beginner Books", year = 1960,
        series = "I Can Read It All By Myself",
}
```

Figure 4.1: A sample BibTeX entry.

```
signature BIBS_CLIENT =
    sig
        eqtype set

        datatype field = Allfields | Author | Title
        datatype matchval = Exact of string | Prefix of string

        val search : set -> field -> matchval -> set

        val union        : set -> set -> unit
        val intersection : set -> set -> unit
        val difference   : set -> set -> unit

        val copy        : set -> set

        val load_file : string -> set
        val save_file : set -> string -> unit
        val print_set : set -> unit
        val size      : set -> int

        exception Retrieve              (* no such named set *)
        val register : set -> string -> unit
        val retrieve : string -> set
        val destroy  : string -> unit

        val run : string -> unit
    end
```

Figure 4.2: The BIBS client signature.

```
fun run display_name =
    let
        val win = open_window display_name
        fun init () = ()
        fun complete result = (close_window win; result)
    in
        V.Threads.fork
        (fn () =>
         V.Skeins.Full_Skein.full_skein init complete cmd_loop win)
    end
```

Figure 4.3: The `run` function.

```
  search s Allfields (Exact "eggs")
```

will produce a set containing all members of `s` that contain the word "eggs" in any field.

The `union`, `intersection`, and `difference` functions take two sets and perform the corresponding set operations, modifying their first argument. A "new" copy set may be produced with `copy`.

The `load_file` and `save_file` functions convert sets to and from `.bib` files, and `print_set` displays a set in BibTeX format. The `size` function simply returns the number of entries in a set.

Sharing and long-term storage of BIBS sets is accomplished with `register`, `retrieve`, and `destroy`. These functions assign a name to a set, retrieve a set by its name, and destroy a named set, respectively.

4.3 Use of Venari Extensions in BIBS

We make use of the Venari extensions in the BIBS application for several purposes. They provide us with mechanisms for expressing concurrency inherent in the application, for ensuring the persistence and consistency of our registered sets of entries, and for protecting concurrent tasks from one another.

4.3.1 Concurrency

To allow several users to access BIBS sets simultaneously, we can take advantage of Venari's support for concurrency. Each client runs in a separate skein; clients can perform operations concurrently. Unfortunately, it is not yet possible to create multiple threads each running the SML top-level loop, so each user must run the interactive interface, which sends output to and receives input from each user through a separate X Window System[1] window. The `run` function takes an X display name and forks a thread in which the interactive interface is run (Figure 4.3). The `cmd_loop` is run in a skein in order to ensure that all threads that it creates are destroyed before the window is closed, and that all uncaught exceptions propagate back to the top level. The completion function includes `close_window` so that the window is closed whether the skein completes successfully or with an exception. We could use `V.transact` instead of `full_skein`, but as we will see later, this would

[1]X Window System is a trademark of MIT.

restrain the concurrency more than we want. Of course, we want to ensure that the clients do not interfere with each other in "undesirable" ways; we will address this later.

The other significant use of concurrency in BIBS is for implementing parallel algorithms within the application. Associated with each BIBS set is one or more indices that allow quick searches for entries containing a given word or words. The operations which maintain these indices allow some parallelism.

To index an entry based on all its words in parallel, we would like to write something like this:

```
fun index_entry idx entry =
    let
        val words = list_words entry
    in
        par_app (fn w => insert idx (w,entry)) words
    end
```

where `par_app`, like `List.app`, applies its first argument to each element in its second argument, but in parallel. The `par_app` function proves useful in many circumstances, so we build it, and its companion, `par_map` (like `List.map`). We can use these functions whenever the order in which the applications occur is unimportant. The `par_map` function (Figure 4.4) forks a thread for each element. Each of these threads computes the value of the function applied to one of the arguments, stores the result, and signals the original thread if all threads are done. Note that the decrementing and checking of the `to_go` counter is done under protection of the mutex associated with the `done` condition. The entire operation is enclosed in a skein, so if any thread raises an exception, all threads are killed and the exception is propagated out to the caller.

The indices themselves are *tries* [1]. A trie is a tree in which each edge is labeled with a letter, and each node corresponds to the word spelled along the path to that node from the root. Each node in the trie will contain

- a list of the entries in the set which contain its corresponding word, and
- a list of $(letter, trie)$ pairs representing the node's children.

See the example trie in Figure 4.5.

Given this kind of index, we can retrieve all records containing a given word easily, in time proportional to the length of the word. Inserting an entry under a given word is equally fast. We can index an entire entry in parallel simply by forking threads to insert each word in the entry. When we use `union` to merge one set into another, we must merge their indices as well, and this operation has a straightforward parallel implementation (Figure 4.6). This function makes a recursive call for each child of the merged node, and the `fork` in `merge_kids` causes the calls for each of the children to proceed in parallel.

Another use for threads which BIBS does not yet take advantage of is "background computation." If there is some low-priority work to be done, such as tree-balancing, trimming unnecessary data, or making backups, then a thread that runs constantly in the background could handle this work. Since there is no provision for priority in scheduling threads, however, such a background thread would have to be careful to avoid depriving more important tasks of processing cycles.

```
fun par_map func args =
    let
        val v = Vector.vector args
        val len = Vector.length v
        val results = Array.array (len,NONE)
        val to_go = ref len
        val done = T.condition (T.mutex ())
        fun do_item i =
            let
                val result = func (Vector.sub (v,i))
            in
                Array.update(results, i, SOME(result));
                T.with_condition done
                (fn () => (dec to_go;
                           if !to_go = 0 then T.signal done
                           else ()))
            end

        (* apply func to args i..len-1 *)
        fun do_all i =
            if i<len then
                (T.fork (fn()=> do_item i);
                 do_all (i+1))
            else ()

        (* build a list from results i..len-1 *)
        fun collect_results i =
            if i<len then
                case Array.sub(results,i) of
                    SOME x => x :: (collect_results (i+1))
                  | NONE => raise Bug "map: missing result"
            else []
    in
        V.Skeins.skein
        (fn ()=> (do_all 0;
                  T.with_condition done
                  (fn () => T.await done (fn()=>(!to_go = 0)));
                  collect_results 0))
        ()
    end
```

Figure 4.4: The `par_map` function, a parallel version of `List.map`.

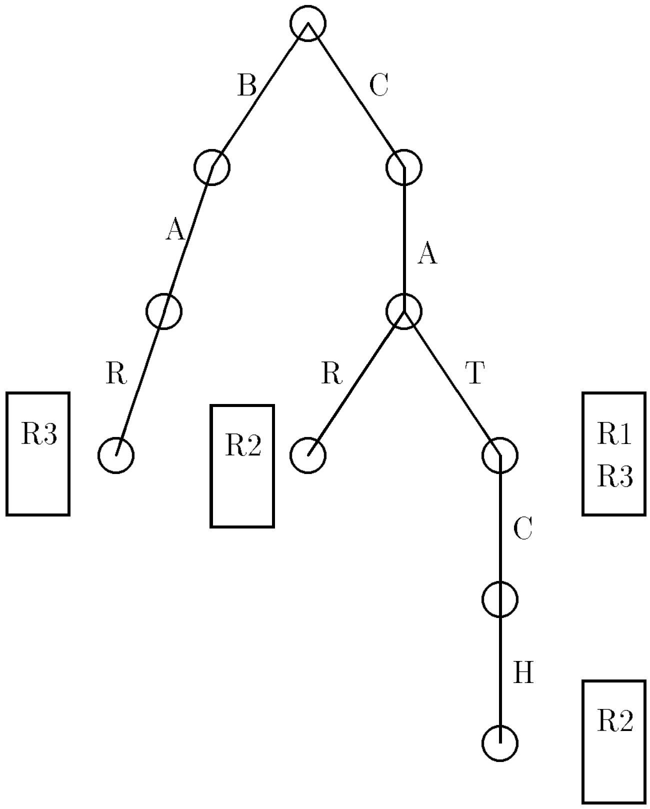

Figure 4.5: A sample trie. Record *R1* contains the word "cat," record *R2* contains "car" and "catch," and record *R3* contains "bar" and "cat."

```
(* A trie (children are always sorted) *)
datatype ''a trie = TRIE{entries : ''a list,
                         children : (string * trie) list}

fun merge (TRIE{entries=ent1, children=ch1})
          (TRIE{entries=ent2, children=ch2}) =
    let
        val new_entries = append_no_duplicates(ent1,ent2)

        fun pair_kids kids1 [] = map (fn (c,t) => (c,[t])) kids1
          | pair_kids [] kids2 = map (fn (c,t) => (c,[t])) kids2
          | pair_kids (kids1 as ((c1,t1)::tl1))
                      (kids2 as ((c2,t2)::tl2)) =
                      if (c1=c2) then
                          (c1,[t1,t2])::(pair_kids(tl1,tl2))
                      else if (c1<c2) then
                          (c1,[t1])::(pair_kids(tl1,kids2))
                           else
                               (c2,[t2])::(pair_kids(kids1,tl2))

        fun merge_pair (c,[t1,t2]) = (c,merge t1 t2)
          | merge_pair (c,[t1]) = (c,t1)
          | merge_pair _ = raise Bug

        fun merge_kids kids1 [] = kids1
          | merge_kids [] kids2 = kids2
          | merge_kids kids1 kids2
            let val pairs  = pair_kids kids1 kids2
            in
                par_map merge_pair pairs
            end
    in
        TRIE{entries=new_entries, children=sort_kids (merge_kids ch1 ch2)}
    end
```

Figure 4.6: The `merge` function on `tries`.

4.3.2 Persistence

The primary purposes of persistence in BIBS are to preserve named sets from one invocation of the system to the next and to make those sets reliable in the presence of failures[2]. Thus, we can put our named sets in the persistent store.

When the BIBS server is started, it must initialize the persistent store and retrieve the stored sets.

```
V.Pers.init ("Bibs.log","Bibs.data",false);
val bib_sets : set V.RW_Ref.rw_ref safe_hashtable
         = V.Pers.retrieve (V.Pers.make_id "BIBS:bib_sets");
```

We can `register`, `retrieve`, and `destroy` sets by accessing the `bib_sets` hash table, using the set name as the key. The buckets in this hash table are protected by `rw_locks`. By using a hash table rather than, for example, a list, we allow updates to a set in one bucket to proceed concurrently with reads or updates to sets in other buckets. (For another example of manipulating the persistent environment in a similar way, see [12].)

Now, to ensure that the persistent store is kept consistent and up-to-date, we should perform every operation that modifies sets within a persistent skein (`V.Pers.pers_skein` or `V.transact`). The set must be left in a consistent state when the outermost persistent skein completes as well, or we will commit the inconsistent state to disk. We can satisfy both of these requirements by enclosing the body of each top-level client function in a `V.transact`.

4.3.3 Safe State

Finally, we must impose some restrictions on the interactions between different clients using BIBS simultaneously, and make some guarantees regarding the correctness of our parallel algorithms. For these purposes we use transactions and "safe" mutable objects.

Users will certainly want some degree of isolation from each other's activities, but probably not full isolation. (If one user adds new entries to a shared set, others will want access to those new entries at some point.) We would like incomplete modifications to be undone in the event of a failure (loss of connection, user interrupt, etc.) or mistake on the user's part. If we assume that sets are composed of safe mutable objects, then we can accomplish all of this by enclosing the body of each top-level client function in a `V.transact`, as mentioned above. This will have the effect that changes to a set are made visible to other clients as soon as the top-level function making the changes completes. A user may "bundle" several operations by introducing an outer transaction. For example, if `soso_books` is a subset of `bad_books` and disjoint from `good_books`, then

```
V.transact (fn () => (union good_books soso_books;
                      difference bad_books soso_books))
           ()
```

will move the entries in `soso_books` from `bad_books` to `good_books`, with the guarantee that no other client will find the same entry in `good_books` and `bad_books` within a single transaction. Furthermore, if any exception is raised during this transaction, the entire operation will be undone.

A `set` consists of a mutable list of entries and several mutable indices on those entries. We can use the `Venari.RW_Ref` structure to provide safe, mutable entry lists and safe, mutable indices.

---

[2] Ideally, clients would rely on the persistent store as they would the file system.

```
(* A safe, mutable trie *)
datatype ''a safetrie = ''a trie V.RW_Ref.rw_ref
val safe-merge-mutex = V.Threads.mutex();

(* merge second trie into the first *)
fun safe-merge (m-trie1, m-trie2)
    let
        val _  = V.Threads.with_mutex safe-merge-mutex
         (fn ()=>
                   V.RW_Lock.acquire_write (V.RW_Ref.lock_of m-trie1)
                   V.RW_Lock.acquire_read (V.RW_Ref.lock_of m-trie2))
        val t1 = V.RW_Ref.rw_get m-trie1
        val t2 = V.RW_Ref.rw_get m-trie2
    in
        V.RW_Lock.rw_set m-trie1 (merge t1 t2)
    end
```

Figure 4.7: The `merge` function on “safe” mutable `tries`.

Figure 4.7 demonstrates how tries can be made safe by protecting each trie with an `rw_ref`, thereby providing the isolation guarantees we need. As mentioned in Section 2.8.2, care needs to be taken to avoid deadlock situations. For instance, in Figure 4.7, if we removed the safe-merge-mutex, then a deadlock could arise if two transactions called safe-merge with the same arguments, in reverse order, since both could complete the `acquire_write` successfully and then block on the `acquire_read`.

Chapter 5

For More Information

All source code is available in the directory /afs/cs/project/venari/projects/newxact, which contains subdirectories of the current and past versions. Send e-mail to ky@cs.cmu.edu for further information about source code. We welcome bug reports and suggestions for improvements to these interfaces. Send e-mail to wing@cs.cmu.edu about the Venari Project in general.

Bibliography

[1] A. Aho, J. Hopcroft, and J. Ullman. *Data Structures and Algorithms*, pages 163–169. Addison-Wesley, 1983.

[2] Andrew Birrell. An introduction to programming with threads. Technical Report Research Report 35, DEC/Systems Research Center, January 1989.

[3] 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.

[4] 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.

[5] Bruce F. Duba, Robert Harper, and David B. MacQueen. Typing first-class continuations in ML. In *Proc. of POPL*, 1991.

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

[7] R. Haskin, Y. Malachi, W. Sawdon, and G. Chan. Recovery Management in QuickSilver. *ACM Transactions on Computer Systems*, 6(1):82–108, February 1988.

[8] Leslie Lamport. LaTeX: *A Document Preparation System.* Addison-Wesley, 1986.

[9] 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.

[10] Hank Mashburn and M. Satyanarayanan. RVM: Recoverable virtual memory. Note in progress, March 1991.

[11] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* The MIT Press, 1990.

[12] Scott M. Nettles and J.M. Wing. Persistence + Undoability = Transactions. In *Proc. of HICSS-25*, January 1992. Also CMU-CS-91-173, August 1991.

[13] Oren Patashnik. BibTeXing. Documentation for general BibTeX users, 8 February 1988.

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

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

[16] W.F. Tichy. Design, implementation, and evaluation of a revision control system. In *Proc. of the 6th International Conf. on Software Engineering*, September 1982.

[17] Unilogic, Ltd., Pittsburgh. *Scribe Document Production System User Manual*, April 1984.

[18] 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.