Myrrh: A Transaction-Based Model for Autonomic Recovery

Guy Eddon
Department of Computer Science
Brown University
Providence, RI 02912, USA
geddon@cs.brown.edu

Steven Reiss
Department of Computer Science
Brown University
Providence, RI 02912, USA
spr@cs.brown.edu

Abstract

As software comes under increasing scrutiny for its lack of safety and reliability, numerous static and partially dynamic tools (including model checking) have been proposed for verifying large and complex systems of interacting components. However, because these tools have been largely unsuccessful, it is essential to develop dynamic mechanisms able to enforce runtime safety properties. We aim to address this goal through a runtime system, called Myrrh, that can provide broad safety and reliability guarantees. Myrrh uses transactions, an abstraction commonly associated with database systems, in order to create autonomic capabilities that support automated recovery within the context of a general purpose programming language. The resulting code is self-correcting; exceptions cause faulty transactions to rollback and thus return the system to its previous state.

1. Introduction

Modern applications consist of large and complex systems of interacting components. These systems are commonly built on component models and even web service infrastructures, which enable programmers to write code that effectively makes remote calls to other applications over the web, but hides that complexity from the programmer behind high-level interfaces. This complexity, however, is rather problematic, as these types of systems are prone to frequent failures. Because a call, which may appear to be a standard (local) procedure call, is actually a remote call, it becomes subject to the vagaries of network traffic, other machines' failing, and other people's software crashing (to name just a few potential failure points). As a result, applications that were once straightforward and relatively understandable become distributed and hence more vulnerable.

To address the fragility of complex systems of interacting components, we propose a based transactional model for failure control. Transactions are widely used in database systems to guarantee the atomicity of a set of actions [20]. When applied to a programming language, transactions make the program act as though either all desired actions were completed, in the case of a successful execution, or none were ever attempted, in the case of failure. We believe this approach can safeguard programs from unreliable components, and therefore lead to better self-correcting systems. As a result, even when things do go wrong, problems can be corrected automatically and before users notice anything amiss, thus achieving a central tenet of autonomic computing [4]. We refer to this new kind of safe program as "transactional software," as transactions pervade the entire application and guarantee its safety and reliability.

Section 2 critically discusses the current literature on concurrent programming and related work on nonblocking algorithms and transactional memories. In section 3, we outline our new approach to automatic transactions, define the primary algorithms in section 4, and describe solutions for dealing with concurrency in section 5. Section 6 provides a step-by-step demonstration of the overall system. We conclude in section 7 by summarizing our current experience with Myrrh and discussing several directions for ongoing work.

2. Concurrent programming

Concurrent programming has been a popular approach to programming since time sharing systems first emerged in the late 1960s. Indeed, even though concurrent programming has changed substantially since then, it still struggles with the challenge of providing concurrent use of shared resources, especial memory. Thus, current synchronization primitives built from basic mutual exclusion algorithms into higher level constructs that synchronize access to shared data (such as critical sections, mutexes, and semaphores), are defined by a common characteristic: when a desired lock is unavailable, they block. Blocking, however, is undesirable, because it means that some threads will waste time waiting for other threads to release their locks. Furthermore, in certain cases, it can also produce undesirable side effects, such as deadlock, livelock, and priority inversion.

In order to address these shortcomings, in the last decade a great deal of research has focused on a class of concurrent algorithms called nonblocking algorithms [7].

Proceedings of the Second International Conference on Autonomic Computing (ICAC'05)
0-7695-2276-9/05 $ 20.00 **IEEE**

These types of algorithms permit multiple threads to read and write shared data concurrently without corrupting it. However, while nonblocking algorithms are in principle far more appealing than their more conventional blocking counterparts, in practice they have been somewhat disappointing, both because their performance often does not compare favorably with highly optimized blocking versions of the same algorithms, and because nonblocking algorithms have proven very difficult to write compared with equivalent blocking algorithms.

The difficultly inherent in writing nonblocking algorithms can be traced back to the very low-level atomic primitives implemented in hardware, such as compare-and-swap (CAS) or load linked/store conditional (LL/SC). Both CAS and LL/SC enable a thread to read a value from memory, modify the value, and then later atomically save the new value back to its original location, if and only if no other thread modified the value stored at that location in the interim. (Note that, although CAS operates atomically, under no circumstances does it block.) The Java code below shows the basic semantics of a CAS operation:

```
public class register {
   private Object value;    // actual value
   public synchronized boolean CAS(
      Object oldV, Object newV) {
      if(value != oldV)
         return false;
      value = newV;
      return true;
   }
}
```

As the code above demonstrates, the CAS operation may be thought of as a very primitive transaction. (It operates atomically, such that all of its changes become visible to other threads at one instant known as the linearization point; and its work is also isolated, in the sense that, should it fail, no intermediate changes will ever have been visible to any other threads.) However, the main drawback of CAS is that it is too primitive to provide an adequate basis for the practical implementation of general purpose nonblocking algorithms. This suggests that it is necessary to develop a higher-level abstraction to facilitate concurrent programming.

2.1. Transactional memory

Transactional memory [10] represents a significant step in this direction. Transactional memory builds on the abstraction of a transaction to provide hardware instructions that atomically modify the value of an arbitrary number of memory locations. If the transaction succeeds, all updated values appear to change at one time; if it fails, no changes are ever recorded. Most hardware-based transactional memory systems utilize processor caches to store tentative changes, as well as the cache coherence protocol to advertise committed transactions to all other processors [19]. However, such approaches are limited, to the extent that a transaction can only modify as many values as will fit into a processor's cache. (Some researchers have proposed protocols that spill over into virtual memory after a processor's main cache has been filled.) Furthermore, because hardware-based transactional memory requires hardware modifications (new instructions, a second set of processor caches, as well as extra storage for cache lines involved in an active transaction) to implement successfully, it has yet to be supported by any commercially available system.

2.2. Software transactional memory

Software transactional memory (STM) is a programming toolkit that provides a software implementation of the ideas first developed in the context of hardware-based transactional memory. The great advantage of this approach is that it allows researchers to experiment with transactional memory protocols without waiting for these ideas to be implemented in hardware. The first STM proposal was static [19]: it required the developer to specify, in advance, what data would be accessed as part of the transaction. Later, however, dynamic STM [9] developed these ideas such that the system does not a priori require the user to specify which objects will be accessed from a transaction.

The insert method shown below is a fragment from a nonblocking singly-linked integer list implemented using DSTM. There are several aspects worth noting about this code: First, classes that will be used in transactions need to support cloning through a custom TMCloneable interface. Second, a target object that will be participating in a transaction must be wrapped by a proxy class called TMObject. Third, the body of the code resides in a while loop, which is required in order to retry the transaction in case of failure. Fourth, an explicit call is necessary to begin the transaction. Fifth, before accessing a transactional object, the developer must enlist it in the transaction by being opened in read or read/write mode. Finally, the transaction must explicitly end with an attempt to commit its work; if commit fails for any reason, such as a concurrency violation, the transaction is aborted and control flow returns to the while loop in step 3 to retry. Note that the insert method itself is essentially not transactional: while transactions begin and end by invoking methods of the current thread, only the set of objects specifically wrapped by a TMObject proxy, and then opened for reading and/or writing, actively participate in the transaction.

```
public boolean insert(int v) {
   // (1) List must implement TMCloneable
   List newList = new List(v);

   // (2) wrap target object
   TMObject newNode = new TMObject(newList);
```


Proceedings of the Second International Conference on Autonomic Computing (ICAC'05)
0-7695-2276-9/05 $ 20.00 **IEEE**

```
    TMThread thread =
       (TMThread)Thread.currentThread();

    // (3) retry loop - in case tx fails
    while(true) {
       // (4) start the transaction
       thread.beginTransaction();

       boolean result = true;
       try {
          // (5) open objects in rw mode
          List prevList =
            (List)first.open(TMObject.WRITE);
          List currList = (List)
          prevList.next.open(TMObject.WRITE);
          while(currList.value < v) {
             prevList = currList;
             currList = (List)
          currList.next.open(TMObject.WRITE);
          }
          if(currList.value == v)
             result = false;
          else {
             result = true;
             newList.next = prevList.next;
             prevList.next = newNode;
          }
       } catch(Denied d) { }
       // (6) attempt to commit the tx
       if(thread.commitTransaction())
          return result;
    }
    return false;
}
```

The example above suggests that DSTM code is far easier to understand than a similar nonblocking algorithm based on the far more primitive CAS operation, but the programming paradigm forced on the programmer by this model is still rather awkward and requires a solid grasp of the underlying concurrency issues. DSTM makes an original but limited effort to address this shortcoming by separating concurrency issues from the main programming model through the use of customizable contention managers. Contention managers are responsible for setting the policy followed when two or more transactions conflict through contention for the same resources. Some contention managers might abort all transactions in their path, potentially leading to livelock, while others might use a more "polite" scheme such as exponential back-off before aborting another transaction.

2.3. Transactions in programming languages

Against this background, there have been several attempts to integrate transactions directly into programming languages. An early approach developed at MIT in the 1980s was called Argus [15]. It supported distributed programming through dynamically created guardians and atomic actions, much like transactions. Argus, however, supported distributed transactions with a two-phase commit protocol and used the guardian abstraction to model programs that could even survive hardware failures. These far-reaching goals resulted in a language that, while impressive in its scope, was too inefficient to be applied to systems that did not benefit from its expensive machinery.

Other attempts to integrate distributed transactions have included heavyweight approaches such as COM+ [1] and .NET Enterprise Services, which provide hooks for a number of mainstream programming languages. Sun offers similar support for Java by way of the Java Transaction Service, which provides an implementation of the OMG Object Transaction Service (OTS) 1.1 specification. The Java Transaction Service uses standard CORBA ORB/TS interfaces and the Internet Inter-ORB Protocol (IIOP) for transaction context propagation between transaction managers.

More recently, a lighter-weight approach was taken in the design of an atomic keyword for Java [6]. This work builds on the idea of conditional critical regions (CCRs), first proposed for concurrency control by Anthony Hoare in 1972 [12]. In its Java incarnation, the atomic keyword supports a boolean guard condition that causes calling threads to block until the condition is satisfied. Instead of implementing this guard through mutexes, the atomic keyword in Java builds the CCR concept on top of a word-sized STM system that supports obstruction freedom. Thus, for example, the get method shown below is from a class that implements a shared buffer using the atomic keyword with a guard condition. Here, the guard condition is used to guarantee that the buffer is not empty when attempting a get. When the guard condition is satisfied, the atomic block ensures that both the *items* count is decremented and the last value is returned atomically:

```
public int get() {
   atomic(items != 0) {
      items--;
      return buffer[items];
   }
}
```

Note that this approach to transactions does not have any explicit concept of an abort and rollback due to failure of the user's code[1]. For example, imagine that you wanted to use the atomic keyword in the implementation of a move method that causes an object to migrate from one collection to another, as shown below. This implementation has a serious problem, in that the *remove* method might succeed only to see the *insert* fail, resulting in a lost object.

```
boolean move(Collection s, Collection d,
   Object o) {
   atomic {
```

[1] Among the numerous problems that can arise in concurrent programming, such as unprotected concurrent access to data, failures in user code, and poor performance, etc., the atomic keyword deal only with the first.

Proceedings of the Second International Conference on Autonomic Computing (ICAC'05)
0-7695-2276-9/05 $ 20.00 **IEEE**

```
        if(!s.remove(o))   // remove object o
           return false;
        d.insert(o); // might throw exception
        return true;
     }
  }
```

The atomic block with a guard condition, then, cannot prevent errors of this sort, forcing the wary programmer to write explicit code to deal with this possibility by catching errors thrown from the insert method, and then adding the removed object back to the source collection, as shown below.

```
boolean move(Collection s, Collection d,
   Object o) {
   atomic {
      if(!s.remove(o))
         return false;
      else {
         try {
            d.insert(o);
         }
         catch(Throwable t) {
            // couldn't insert the object
            s.insert(o);  // add it back
            throw t;      // move failed
         }
         return true;
      }
   }
}
```

What is missing to address this problem is a system for automatically catching such errors and rolling back all changes executed since the beginning of the transaction. Such support for implicit rollback is actually required for safe, autonomic programming to be feasible.

3. Automated recovery transform

We aim to improve the overall reliability of software systems by using a transactional model in order to automatically integrate recovery properties into compiled Java bytecode. To achieve this goal, Myrrh employs metaprogramming to inject transactional recovery code, therefore ensuring the system's portability through the use of a standard execution environment. Since this recovery method does not require programmer intervention, but, rather, transforms the code automatically, it can potentially simplify the design and implementation of self-healing autonomic systems and reduce the potential for failure in large-scale distributed applications. By tracking changes and then rolling back in the face of errors, the system can in many cases guarantee an accurate return to a previously correct state.

The operation of the Myrrh transform is shown below. The input target system consists of class files typically produced by a Java compiler. The Myrrh transform rewrites these classes subject to the parameters of an XML configuration file and outputs a JAR file containing the transactional version of the target system. This transactional version of the target system, what we refer to as "transactional software," is guaranteed to be semantically equivalent to the original[2].

Target system | <Myrrh transform>(
xml configuration file) →
transactional target system

3.1. Transactions

Implementing transactions in an already completed application is a challenging goal. The main difficultly consists in the fact that transactions must pervade the entire program; all changes made to global variables, including heap storage, need to be tracked and undone if the transaction subsequently fails. To address this challenge, Myrrh's approach builds on STM, but it differs from STM in several important respects. In contrast to STM, Myrrh puts forth an automated model that allows transactions to begin and end automatically, and enables target objects to automatically join the transaction without having to implement a special interface. Moreover, within this model, the locks required to access an object in a transaction are also acquired automatically. Finally, transactions that must abort due to conflicts detected by the concurrency manager are automatically restarted, avoiding the need for awkward retry loops.

The main focus of this approach to application recovery is to enable the programmer to define transactions at a high level of abstraction that are then implemented by the system automatically. Once the programmer identifies the transactional status of a given method[3], the system automatically analyzes the code to determine those data that need to be preserved if the transaction fails and the locks needed to ensure that different transactions don't conflict.

Thus, once the system identifies the set of target objects and variables, it instruments the code in all locations that access those objects and renders such access transaction dependent. Changes made to data during the transaction are recorded in the transaction's journal. Then, at the end of the transaction, depending on whether it ends in success or failure, the journal is either discarded or its changes replayed on the object. As a result, a call to the transactional method can now be viewed by the programmer as atomic. This means that, at runtime, if the method succeeds, all is well; if it throws an exception, the programmer can rest assured that no important state has been modified within the program.

Transparency is a fundamental consideration within this context. Both the beginning of a transaction and the

[2] With very minor exceptions such as infinite loops (where heuristics or a timeout can be set to terminate a transaction that encounters this condition).

[3] This is done through an optional XML configuration file (see section 6).

Proceedings of the Second International Conference on Autonomic Computing (ICAC'05)
0-7695-2276-9/05 $ 20.00 **IEEE**

point at which it will end must be automatically determined by the system with little, if any, input from the programmer. Transactions in this model represent the basic abstraction in which a thread can access data both safely and concurrently with other threads in the system. Thus, should an error (exception) be thrown out of a transaction's context, the transaction is automatically aborted and all mutations made in the course of that transaction are rolled back, consequently returning the system to a safe state. Only when a transaction completes successfully do its changes commit and become visible to all other transactions running in the system. In the event that the error is the product of Myrrh's concurrency manager, the transaction is aborted and then automatically restarted to yield exactly-once execution semantics.

Metaprogramming is employed to add transactional rollback capabilities to existing systems in an unobtrusive manner. This approach fundamentally involves two passes over the input code: the first pass statically analyzes the code for read and write operations in order to determine the extent and type of rewriting necessary; the second actually transforms and produces the output code containing the additional calls to acquire and release the necessary locks inserted before and after such access. These passes can be performed as a post-compilation step (static metaprogramming) or just prior to execution (dynamic metaprogramming). It should be noted that the system adds no new keywords to the target programming language [11], since that would conflict with the fundamental goal of automated recovery.[4] This approach is therefore substantively different and significantly more transparent than previous attempts (see, for example, Harris and Fraser who introduce a guarded atomic block to Java (discussed in section 2.3) or Flanagan and Qadeer [3], who use type checking to validate a similar construct).

3.1.1. Null transactions

Myrrh must also take into account code that is not running within the scope of a transaction. Such code is still subject to the rules that apply to transactional methods in order to ensure that the two don't conflict. Myrrh conceives of such methods as transactions that commit immediately and therefore can never be aborted. The primary task of Myrrh in this case is to instrument non-transactional code in order to acquire the necessary locks as the method reads and writes memory (see section 6) and later release those locks when the method/thread ends. In systems where standard two-phase locking is employed, the null transaction must take precedence over existing locks held by other transactions, which may require aborting other transactions that hold conflicting locks. Exceptions thrown by non-transactional methods also release the locks held. In this case, no rollback is possible.

3.2. Concurrency

Concurrency is a second but no less critical goal of the Myrrh recovery system. Transactional software implies concurrent software, and therefore requires good general purpose solution for concurrency management. Because this is an open research area where there are many possible (but few good) approaches to concurrency management, this issue is orthogonal to the transaction support that Myrrh provides. Like STM, which defines a separate interface for contention management, Myrrh defines an interface for concurrency management. However, concurrency, in this model, is not a direct extension of the transactional system, but rather its consequent. Thus, many different concurrency schemes are possible under a single transactional model. To date we have experimented with two-phase locking and obstruction-free [8] approaches. We are also exploring the possibility of a specialized merge operation that will resolve conflicts after they occur, in a manner somewhat analogous to that used for nonblocking data structures. The main objective of this part of the project is to define a general-purpose concurrency manager that provides acceptable performance for most applications and can be optimized through merge capabilities for special objects that require higher performance and improved concurrency.

4. Algorithms

There are four main algorithms in Myrrh that support automated recovery through transactions. These are: get, put, commit, and abort. The following subsections discuss the role of each of these algorithms in the context of the overall recovery model proposed. The algorithms described in the following subsections are overloaded to operate on fields or arrays of type <*type*>; <*type*> represents all primitive types recognized by the Java VM (int, float, reference, long, and double) [14].

4.1. Get

The get algorithm (defined below) retrieves a specific datum from the field of an object. In order to do this successfully, it follows a number of steps: (1) acquires a read lock for the field[5] (if unsuccessful, the lock method throws a ConcurrencyException); (2) checks whether the current transaction is a null transaction; if it is, the

[4] For an interesting discussion of several design choices facing implementers of language-based transactions see [5].

[5] The actual behavior of the lock operation depends on the concurrency manager implementation selected by the user. This option is discussed further in section 5.

Proceedings of the Second International Conference on Autonomic Computing (ICAC'05)
0-7695-2276-9/05 $ 20.00 **IEEE**

algorithm (3) enters a synchronized block[6] and (4) returns the current value of the requested field. If the algorithm is operating within a real transaction, (5) checks whether the requested field has been previously read or written within the current transaction; if it has, (6) returns the value of the specified field from the journal; if not, (7) retrieves the field's current value and stores the result in *val*; (8) adds *val* to the journal, and (9) returns *val* to the caller.

```
<type> transaction.get<type>(Object target,
   String field, int index) throws
   ConcurrencyException {
(1) conMan.lock(this, target, field,
      index, txlockmode.read, m_priority);
(2) if(m_priority == txpriority.txnull)
(3)    synchronized(getClass()) {
(4)       return getField<type>(target,
             field, index);
       }
(5) if(journal.contains(target, field,
       index))
(6)    return journal.get<type>(target,
          field, index);
(7) <type> val = getField<type>(target,
       field, index);
(8) journal.put(target, field, index, val,
       val);
(9) return val;
}
```

4.2. Put

The put algorithm (defined below) stores a value in the specified field of an object as part of the current transaction. As in the get algorithm, the first step (1) consists in acquiring a lock (in this case, a write lock); (2) checks whether the current transaction is a null transaction; if it is, the algorithm (3) enters a synchronized block and (4) puts the specified value in the requested field. If the algorithm is operating within a real transaction, (5) checks whether access to the specified field has already been recorded in the transaction's journal; if it has, (6) replaces the current value in the journal with the new value. Otherwise, (7) adds the specified value to the transaction's journal, while also recording the current actual value of the field for later use when attempting to merge conflicting transactions (see subsection 5.4).

Both the get and put algorithms are used to rewrite access of static (getstatic/putstatic) as well as of regular fields (getfield/putfield) and arrays[7]. In the case of static fields, the metaprogramming system passes a reference to the class object (getClass()) via the target parameter; the class object takes the place of the reference to the current object (this) used for normal fields.

```
void transaction.put(Object target,
   String field, int index, <type> val)
   throws ConcurrencyException {
(1) conMan.lock(this, target, field, index,
       txlockmode.write, m_priority);
(2) if(m_priority == txpriority.txnull)
(3)    synchronized(getClass()) {
(4)       setField(target, field, index,
             val);
       }
    else
(5)    if(journal.contains(target, field,
          index))
(6)       journal.put(target, field, index,
             val);
       else
(7)       journal.put(target, field, index,
             val, getField<type>(target,
             field, index));
}
```

4.3. Commit

The commit algorithm (defined below) commits all changes made within the transaction in a single atomic unit. It executes the following steps: (1) checks whether this is a null transaction; if it is, (2) releases held locks and (3) returns. Otherwise, (4) enters a synchronized block; (5) asks the concurrency manager to validate the transaction and resolve any potential conflicts (see subsection 5.3); (6) retrieves an iterator for the transaction's journal; and (7) loops through all the entries in the journal. For each entry (8), (9) retrieves the value from the transaction's journal and (10) sets the specified (object, field, index) tuple to the desired value. Finally, (11) releases all the locks held by the transaction.

```
void transaction.commit()
   throws ConcurrencyException {
(1) if(m_priority == txpriority.txnull) {
(2)    conMan.unlock(this);
(3)    return;
    }
(4) synchronized(getClass()) {
(5)    conMan.validate(this);
(6)    Iterator it =
          journal.values().iterator();

(7)    while(it.hasNext()) {
(8)       txJournalEntry je =
             (txJournalEntry)it.next();
(9)       Object val =
             journal.getRef(je.target,
             je.field, je.index);
(10)      setField(je.target, je.field,
             je.index, val);
       }
    }
(11) conMan.unlock(this);
}
```

4.4. Abort

[6] This synchronization block ensures that actual values are not read of written by null transactions during commit operations.

[7] The rewritten array instructions are divided into two categories: array loads (aaload, baload, caload, daload, faload, iaload, laload, and saload) and array stores (aastore, bastore, castore, dastore, fastore, iastore, lastore, and sastore). These instruction codes represent the types reference, byte/boolean, char, double, float, int, long, and short, respectively.

Proceedings of the Second International Conference on Autonomic Computing (ICAC'05)
0-7695-2276-9/05 $ 20.00 **IEEE**

The abort algorithm is the simplest of all the primitive transactional operations. The algorithm needs only to release the locks being held by the transaction (1). Java's garbage collection mechanism ensures that the transaction's journal will be discarded.

```
void transaction.abort() {
(1) conMan.unlock(this);
}
```

5. Concurrency management

In a transactional system for automated recovery, there are several possible approaches to the problem of overlapping transactions. Section 5.1 discusses traditional two-phase locking designed for database transactions. Section 5.2 covers obstruction freedom, a looser correctness condition designed for software transactional memory, and explores the possibility of conflict resolution through transactional merge semantics. Section 5.3 describes Myrrh's approach to supporting multiple, potentially user-defined, concurrency schemes through the definition of a concurrency manager, and section 5.4 describes an implementation of that interface that supports merge semantics for conflict resolution.

5.1. Two-phase locking

As pointed out in section 2, locking is the most common solution to the problem of concurrently executing transactions. According to this approach, a transaction that attempts to read a value must first acquire a read lock for that item, and a transaction that wants to modify a value must first acquire a write lock (read locks conflict only with write locks, while write locks conflict with both read and write locks). This is known as the two-phase locking theorem, which holds that a serializable transaction cannot acquire a lock, release it, and subsequently acquire it again. (Eswaran et al [2] provided the first proof of the two-phase locking theorem in 1976; a simpler proof was later given by Ullman [21]).

However, the two-phase locking theorem gives rise to a number of problems. Thus, for example, consistency comes at the expense of delays resulting from the possibility that transactions may block while waiting for other transactions to release locks; moreover, there is the potential for transactions to deadlock when a lock cycle occurs. In the context of this work, two-phase locking has proven largely impractical, due to the propensity of threads that access shared data to access and therefore to attempt to lock the same regions concurrently.

5.2. Nonblocking strategies: obstruction freedom and optimistic concurrency

To avoid the problems inherent in the database-oriented approach underlying the two-phase locking theorem, Herlihy et al provide a progress condition they call obstruction freedom, which guarantees that a halted thread will not prevent other threads from making progress and requires only the readily available compare-and-swap (CAS) instruction. However, the non-blocking progress condition of obstruction freedom is designed primarily to enable concurrent execution on symmetric multiprocessing architectures and thus may not be the best option for automated recovery, which is essentially a software engineering challenge.

Yet another approach to lock management proposes avoiding locking altogether [13]. In this view, transactions might be presumed not to conflict until proven otherwise. Reiss et al [16], for example, use triggers to notify transactions of any change in a value, thus enabling one transaction to write a value and one or more other transactions to read it without conflict. Another instance of transactions that may avoid locking is abort-only transactions, where the program has no intention of committing the results. This is a particularly useful approach for running tests procedures.

Finally, a transactional system can forgo locking if it believes that it can resolve potential conflicts after they occur. The idea here is that, even in cases where it is determined that transactions did in fact "conflict" under the traditional database-oriented interpretation, it may be possible to resolve the conflict through an intelligent merge operation that combines the changes made by two or more overlapping transactions that want to commit. As with nonblocking algorithms, such an approach takes into account the fact that most executions of transactions do not conflict and provides a framework for resolving conflict in those cases where it occurs. Nevertheless, actually merging the results of two conflicting transactions can be challenging because, except for timestamps, the system has little information from which to work.

5.3. The concurrencyManager interface

Given that there are a number of different and, under some circumstances, equally legitimate approaches to lock management and conflict resolution, Myrrh separates locking from transaction support via the concurrencyManager interface. With this interface, it is now possible to implement a number of different concurrency managers that take different approaches to the problem of contention management and resolution.

To date, we have implemented two concurrency managers: blockLocker, which provides typical two-phase locking for serializable transactions, and mergeLocker, which supports optimistic concurrency—conflict resolution is handled by attempting to merge the results during a commit. The concurrencyManager interface is defined below.

```
interface concurrencyManager {
```


Proceedings of the Second International Conference on Autonomic Computing (ICAC'05)
0-7695-2276-9/05 $ 20.00 **IEEE**

```
// attempts to acquire a lock of type
// lockmode on field for tx
(1) void lock(transaction tx,
       Object target, String field,
       int index, int lockmode,
       int priority)
       throws ConcurrencyException;
// releases all the locks held by this tx
(2) void unlock(transaction tx);
// attempts to resolve conflicts
// during a commit operation
(3) void validate(transaction tx)
       throws ConcurrencyException;
}
```

The concurrencyManager interface outlined above defines two methods for lock acquisition and release. The lock method (1) is called by a transaction attempting to acquire a read or write lock on a target field. This method should be implemented in such a way that, if the transaction already holds the desired lock, the method simply returns true. In case of contention, the lock method is allowed to block until a conflicting lock is released. Returning true indicates success; returning false indicates that the concurrency manager failed to obtain the requested access, which will abort the transaction immediately and signal the ConcurrencyException. The unlock method (2) releases all the locks held by a specified transaction.

In addition, a third method, validate, is designed for implementation by concurrency managers that do not serialize transactions, but, instead, ask the target object to help resolve conflicts after they occur. In standard locking equals blocking implementations, the concurrencyManager.validate method should always return true. Otherwise, the concurrency manager should check for and attempt to resolve any existing conflicts by asking the target object for help.

5.4. Merging conflicting transactions

To enable transactional merge support, Myrrh defines a second interface, mergeManager, which objects may implement in order to provide customized reconciliation code. An object's merge method (defined below) is invoked by the concurrency manager's validate method when it determines that a conflict has occurred. If successful, the method should return a value that resolves the conflicting results of the two transactions; it is this value that will be used to update the transaction's journal and later committed to the object as part of the transaction.commit algorithm. By contrast, if the transactions cannot be reconciled, the method is expected to throw the MergeFailedException, in which case the system will abort the last transaction.

```
interface mergeManager {
   Object merge(String field, int index,
      Object desiredValue,
      Object originalValue,
      Object currentValue)
      throws MergeFailedException;
}
```

Instead of failing when a conflict is detected, mergeManager's lock method always returns success. In case of conflict, this fact is simply noted in the journal entry for later reconciliation in the validate algorithm.

The mergeLocker.validate algorithm must determine whether conflicts have occurred when a transaction wants to commit, and, if so, whether those conflicts can be resolved. For this purpose, the validate algorithm acquires an iterator for the transaction's journal (1); for each entry (2), it checks whether a conflict was previously recorded (3); if it was, the algorithm gets a reference to the mergeManager interface of target object (4), and calls the object's merge method to resolve the conflict (5). Finally, the algorithm puts the resultant value back in the transaction's journal (6). If an error occurs or the target object does not implement the mergeManager interface, the algorithm throws a ConcurrencyException (7) to indicate that the transaction must abort.

```
void mergeLocker.validate(transaction tx)
   throws ConcurrencyException {
(1) Iterator it =
       tx.journal.values().iterator();
    while(it.hasNext()) {
(2)    txJournalEntry je =
          (txJournalEntry)it.next();
(3)    if(hadConflict(je.target, je.field,
          je.index))
       try {
(4)       mergeManager target =
             (mergeManager)je.target;
(5)       Object result =
            target.merge(je.field, je.index,
             je.value, je.originalValue,
             tx.getFieldRef(je.target,
             je.field, je.index));
(6)       tx.journal.put(je.target,
             je.field, je.index, result);
          }
       catch(Throwable t) {
(7)       throw new ConcurrencyException ();
       }
    }
}
```

6. Example

To provide a better understanding of the code rewriting done by the system at the metaprogramming stage, we offer the following changeName method in Java as an example:

```
public class student {
   String m_name;
   public boolean changeName(String newName)
        throws InvalidDataException {
      m_name = newName;
      helper h = new helper();
      if(h.hasDigits(m_name))
         throw new InvalidDataException();
      return true;
   }
```


Proceedings of the Second International Conference on Autonomic Computing (ICAC'05)
0-7695-2276-9/05 $ 20.00 **IEEE**

```
}

class helper {
   static String m_prevString;
   public boolean hasDigits(String s) {
      m_prevString = s;
      return !s.matches("^[A-Za-z]+$");
   }
}
```

The student.changeName method mutates the value of the student.m_name field and helper.hasDigits of the helper.m_prevString field. If the new name has any digits, the function throws the InvalidDataException; otherwise it returns true. In this simple example, the system rewrites the method to support automated recovery in the event of a failure. The resultant class, if translated back to Java, looks as follows (changes are highlighted in boldface):

```
public class student {
   String m_name;
   public boolean changeName_tx(String
      newName) throws InvalidDataException {
      myrrh_rt.transaction tx =
         myrrh_rt.transaction.currentTx();
      tx.put(this, "m_name", newName);
      helper h = new helper();
      if(h.hasDigits_tx((String)tx.getRef(
         this, "m_name")))
         throw new InvalidDataException();
      return true;
   }
}

class helper {
   static String m_prevString;
   public boolean hasDigits_tx(String s) {
      myrrh_rt.transaction tx =
         myrrh_rt.transaction.currentTx();
      tx.put(getClass(), "m_prevString", s);
      return !s.matches("^[A-Za-z]+$");
   }
}
```

In the rewritten version of the class, transactional methods have been renamed methodname_tx. The _tx method is the original method, rewritten to access all fields and arrays through the transaction class. In addition, a new proxy method is inserted with the original name of each of transactional method. The purpose of the proxy method is to create a new transaction and then call the _tx version of the method. The basic algorithm of the proxy methods is shown below:

```
public <type> proxy_method([<type> arg1,
   ...]) [throws <type>, ...] {
   if(myrrh_rt.transaction.inTx())
      return real_method_tx([arg1, ...]);
   while(true) {
      myrrh_rt.transaction tx =
         myrrh_rt.transaction.createTx();
      try {
         [<type> retval =]
            real_method_tx([arg1, ...]);
         tx.commit();
         return [retval];  // success
      }
      // encountered a concurrency error
      catch(myrrh_rt.ConcurrencyException e)
         {
         tx.abort(); // abort, then restart
      }
   // catch exceptions thrown by real_method
      [catch(<type> e) {
         tx.abort();    // abort
         throw e;       // rethrow exception
      }]
      catch(Throwable e) { // any error
         tx.abort();       // abort
         throw e;          // rethrow error
      }
   }
}
```

Any attempt to retrieve or change the value of a field is replaced by a call to the transaction object's get or put method, respectively. The helper object's hasDigits method joins the transaction begun by the changeName method (the current transaction is retrieved by the static transaction.currentTx method from a thread local variable). The static field m_prevString is stored by the transaction with a reference to its class object (retrieved with getClass instead of the instance reference this).

Note that, in the presence of virtual functions, it is not possible to guarantee that static analysis will correctly determine the actual target of a method invocation. Also, as discussed in section 2.2.1, methods that do not wish to operate as part of a transaction must be rewritten in order to correctly deal with contention. For both of these reasons, Myrrh must rewrite all methods in an application. For methods that do not wish to operate within a transaction, the m_priority flag is set to txpriority.txnull and the global "null" transaction (transaction.getNullTx()) is used by that code.

Finally, before the changeName method returns, the code is instrumented to call the transaction's abort or commit method, depending on the outcome of the function. The abort method, invoked automatically in response to a caught exception, simply discards the changes made in the transaction, while the commit method implements any changes to the fields that were recorded during the transaction. Moreover, when the instrumented changeName method is called, the function and its descendents are guaranteed not to cause side-effects unless and until the method completes successfully.

Users can use an optional XML configuration file to specify precisely those methods in a class that must be transactional. A sample file is shown below:

```
<?xml version='1.0' encoding='utf-8'?>
<autorecover>
   <class name="school.student">
      <method name="student" paramlist="()"
         status="no"/>
      <method name="main"
         paramlist="(java.lang.String[])"
         status="no"/>
```


Proceedings of the Second International Conference on Autonomic Computing (ICAC'05)
0-7695-2276-9/05 $ 20.00 **IEEE**

```
      <method name="changeName"
         paramlist="(java.lang.String)"
         status="tx"/>
   </class>
   <class name="school.helper">
      <method name="hasDigits"
         paramlist="(java.lang.String)"
         status="tx"/>
   </class>
</autorecover>
```

6.1. Bytecode metaprogramming

The Java bytecode below shows how the m_name field is retrieved and set in the current sample both before and after the metaprogramming pass. Note that in the "before" code fragment the bytecode simply loads the this reference into the stack and then executes getfield for the desired field. In the rewritten "after" version, by contrast, the this reference is saved in a local variable named target. Subsequently, the local transaction reference is loaded, followed by the target object and the name of the desired field. Finally, the transaction.get algorithm (described in subsection 4.1) is invoked to retrieve the requested value from the transaction's journal. The value returned is then cast to the desired type in preparation for whatever code may follow.

Before:

```
aload_0              // load this
getfield   m_name
```

After:

```
aload_0              // load this
astore     target    // save this in target
aload      tx        // load tx
aload      target    // load target
ldc        "student.m_name"   // load field
invokevirtual   myrrh_rt.transaction.getRef(
                        java.lang.Object,
                        java.lang.String)
checkcast  java.lang.String
```

Putfield operations, by contrast, are replaced with calls to the transaction.put algorithm (described in subsection 4.2). In this case the new value and target object are saved in temporary variables and then the parameters for the put algorithm are loaded onto the stack.

Before:

```
aload_0              // load this
ldc        "Roger"   // load the value
putfield   m_name
```

After:

```
aload_0              // load this
ldc        "Roger"   // load the value
astore     value     // save the value
astore     target    // save this in target
aload      tx        // load tx
aload      target    // load target
ldc        "student.m_name"   // load field
aload      value     // load the new value
invokevirtual   myrrh_rt.transaction.put(
                        java.lang.Object,
                        java.lang.String,
                        java.lang.Object)
```

7. Experience and future directions

Our experience with Myrrh shows that it is capable of successfully instrumenting moderate sized web services. Currently, our main test project is a zip code service consisting of several thousand lines of Java code that allows clients to search for all zip codes within a specified radius. We believe this is a good test system, as it contains many standard features of web service applications. While the current implementation of Myrrh produces autonomic code that is still too slow to be of practical use, we are currently exploring a number of optimizations that should bring its performance into an acceptable range for many applications. These approaches include data flow analysis to determine what parts of the code need to be instrumented and the use of Java synchronized blocks to provide hints to Myrrh regarding data that are shared across transactions.

The primary goal of this work is to automate the dynamic recovery process for software that encounters errors. Since programs often fail under slightly unusual circumstances because the assumptions made by the developer do not hold true, a robust model for automated recovery would be an ideal tool to step in, undo the damage, and return the system to a stable state. This approach is inspired by research in autonomic computing, where software systems are designed to be self-monitoring and self-correcting. The preliminary results, based on work with a moderately sophisticated web service, suggest that this approach holds great promise. If successful, we hope these ideas will eventually become an integral part of mainstream programming languages and virtual execution environments such as Java and .NET, much as garbage collection has slowly made its way into general-purpose software development.

8. References

[1] Eddon, G. COM+: The Evolution of Component Services. *Computer, IEEE* (July 1999).

[2] Eswaran, K. P., Gray, J. N., Lorie, R. A., and Traiger, I. L. The Notions of Consistency and Predicate Locks in a Database System. *Commun. ACM* (Nov. 1976), vol. 19, no. 11, pp. 624-633.

[3] Flanagan, C., and Qadeer, S. Types for atomicity. In *Proceedings of the Workshop on Types in Language Design and Implementation* (Mar. 2003), vol. 38(3) of *ACM SIGPLAN Notices*, pp. 1-12.

[4] Ganek, A., and Corbi, T. The dawning of the autonomic computing era. In *IBM Systems Journal* (2002), vol. 42 no. 1, pp. 5-18.

[5] Harris, T. L. Design choices for language-based transactions. In *University of Cambridge Computer*

Laboratory technical report No. UCAM-CL-TR-572 (Aug. 2003).

[6] Harris, T. L., and Fraser, K. Language Support for Lightweight Transactions. In *Proceedings of OOPSLA* (Oct. 2003), pp. 388-402.

[7] Herlihy M. Wait-Free Synchronization. In *ACM Transactions on Programming Languages and Systems,* 13(1):124-149, 1991.

[8] Herlihy, M., Luchangco, V., and Moir, M. Obstruction-free synchronization: Double-ended queues as an example. In *Proceedings of the 23rd International Conference on Distributed Computing Systems* (2003).

[9] Herlihy, M., Luchangco, V., Moir, M., and Scherer, W. Software transactional memory for dynamic-sized data structures. In *Proceedings of the 22nd Annual ACM Symposium on Principles of Distributed Computing* (July 2003), pp. 92-101.

[10] Herlihy, M., and Moss, J. E. B. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture* (May 1993), IEEE Computer Society Press, pp. 289-301.

[11] Hoare, C. A. R. Hints on Programming Language Design. In *Stanford University Computer Science Department technical report No CS-73-403* (Dec. 1973).

[12] Hoare, C. A. R. Towards a theory of parallel programming. In *Operating Systems Techniques, vol. 9 of A.P.I.C Studies in Data Processing,* Academic Press, pp. 61-71. (1972).

[13] Kung, H. T., and Robinson, J. T. On Optimistic Methods for Concurrency Control. In *ACM Transactions on Database Systems* (June 1981), vol. 6(2), pp. 213-226.

[14] Lindholm, T., and Yellin, F. *The Java Virtual Machine Specification*, 2nd ed. Addison-Wesley, Reading, MA, USA, 1999.

[15] Liskov, B. and Scheifler, R. Guardians and Actions: Linguistic Support for Robust, Distributed Programs. In *ACM Transactions on Programming Languages* (July 1983), pp. 381-404.

[16] Reiss, S. P, Skarra, A. H., and Zdonik, S. B. An Object Server for an Object-Oriented Database System. In *International Workshop on Object-Oriented Database Systems* (Sept. 1986), pp. 196-204.

[17] Reiss, S. P., and Eddon, G. Automated Recovery with Transactions. In *Parallel and Distributed Computing Systems (PDCS)* (Sept. 2004).

[18] Rudys A., and Wallach D. Transactional Rollback for Language-Based Systems. In *International Conference on Dependable Systems and Networks* (2002).

[19] Shavit, N. and Touitou, D. Software transactional memory. *Distributed Computing,* Special Issue(10):99-116, 1997.

[20] Traiger, I. L., Gray, J., Galtieri, C. A., and Lindsay**,** B. G. Transactions and consistency in distributed database systems. In *ACM Transactions on Database Systems* (Sept. 1982)**,** vol. 7 no. 3.

[21] Ullman, J. D. *Principles of Database Systems*, 2nd ed. Computer Science Press (1982), Potomac, MD.