Exceptions and side-effects in atomic blocks

Tim Harris

Microsoft Research, 7 JJ Thomson Avenue, Cambridge, UK, CB3 0FB

Abstract

In recent work we showed how to implement a new `atomic` keyword as an extension to the Java programming language. It allows a program to perform a series of heap accesses atomically without needing to use mutual exclusion locks. We showed that data structures built using it could perform well and scale to large multi-processor systems. In this paper we extend our system in two ways. Firstly, we show how to provide an explicit 'abort' operation to abandon execution of an atomic block and to automatically undo any updates made within it. Secondly, we show how to perform external I/O within an atomic block. During our work we found it was surprisingly difficult to support these operations without opening loop-holes through which the programmer could subvert language-based security mechanisms. Our final design is based on a 'external action' abstraction, allowing code running within an atomic block to request that a given pre-registered operation be executed outside the block.

Key words: Atomic blocks, exceptions, non-blocking synchronization, software transactional memory.

1 Introduction

Recently, along with other research groups, we have been investigating the design and implementation of new programming language features for concurrency control (1). In our system, developed as an extension to the Java programming language, we introduced a new keyword `atomic` which allows a group of statements to execute atomically with respect to the operation of other threads. As well as updating objects' fields, these statements can perform a wide range of operations including invoking methods and instantiating new objects. We also allow `atomic` statements to be guarded by boolean conditions, with execution blocking until the condition is satisfied. Figure 1 illustrates this by showing the implementation of a single-cell shared buffer.

Email address: `tharris@microsoft.com` (Tim Harris).

```
class Buffer {
  private boolean full;
  private int value;

  public void put(int new_value)
                  throws InterruptedException
  {
    atomic (!full) { // Wait until buffer is empty
      full = true;
      value = new_value;
    }
  }

  public int get() throws InterruptedException
  {
    atomic (full) { // Wait until buffer is full
      full = false;
      return value;
    }
  }
}
```

Fig. 1. A single-cell shared buffer implemented using atomic blocks.

Atomic blocks are an attractive alternative to using locks, primarily because they make thread-safe operations *composable*. For example, consider a hash table that supports thread-safe insert and delete operations. Now suppose that we want to delete one item `A` from table `t1`, and insert it into table `t2`; but the intermediate state (in which neither table contains the item) must not be visible to other threads. Using locks, unless the implementor of the hash table anticipates this need, there is simply no way to satisfy this requirement. Even if the programmer anticipates this need, all that can be done is to expose methods such as `LockTable` and `UnlockTable` – but as well as breaking the hash-table abstraction, they invite lock-induced deadlock, depending on the order in which the client takes the locks. In contrast, using `atomic` blocks, the insert and delete operations can simply be composed in sequence in a single block in the same way as in a single-threaded system.

Although constructs like `atomic` blocks have been proposed since at least 1977 (2), our implementation is the first to offer scalable performance for multi-processor machines. In particular, we mean that threads executing non-conflicting atomic blocks can generally run concurrently without synchronization. Furthermore, our implementation is *non-blocking* meaning that it does not suffer from low-level lock-induced deadlocks or priority inversion.

In this paper we address two problems which exist with our existing form of `atomic` blocks: what happens when an exception reaches the edge of an atomic block, and what happens when a thread attempts to perform I/O operations within an atomic block? As we discuss in Section 2, the solutions we have developed carry over to other designs beyond our own.

The problem raised by exceptions is whether to undo updates made in the `atomic` block or whether to retain them and propagate the exception. Unfortunately there is a Catch-22 situation: if we roll back the updates then the exception object itself could be lost, leaving nothing to propagate (or worse, creating a dangling pointer if its allocation is rolled back). We discuss this in Section 3 and propose a hybrid model in which certain exceptions cause `atomic` blocks to be aborted and in which the exception thrown outside the block behaves as if it is a deep copy of the exception raised within it.

The problem with I/O operations is that they will generally become immediately visible to other threads, destroying the illusion of atomicity. In Section 4 we discuss a number of alternative ways to support I/O and propose a model in which communication libraries must be adapted for use within `atomic` blocks. This places an onus on the library's implementer but, we argue, allows better performance and scalability than a generic mechanism.

Our solutions to both of these problems are based on a single 'external action' abstraction which we introduce in Section 5. These actions provide a form of inter-transaction-context method calls, in which an external action exports an operation which can be invoked from within an `atomic` block but which is directly executed outside it. This provides a way to marshall exceptions that leave `atomic` blocks and to perform situation-specific buffering when building atomic I/O. We discuss our experience with this approach in Section 6.

Finally, Section 7 discusses related work and Section 8 concludes, highlighting a number of areas for future work along with dead-ends we explored in developing the 'external action' abstraction.

In the remainder of this introduction we briefly review the intended semantics of atomic blocks in Section 1.1 and outline their implementation over a software transactional memory in Section 1.2.

1.1 Intended semantics of atomic blocks

The semantics of `atomic` blocks are defined by (*i*) specifying their behaviour when executed by a single thread running in isolation and (*ii*) requiring that, in a multi-threaded system, they behave as-if the executing thread ran in isolation while within the block.

For non-nesting blocks, there are two cases to consider based on whether or not the atomic block contains a guard condition. If there is no guard condition then the following two code fragments are equivalent:

```
atomic {
  S;                                { S; }
}
```

Similarly, if a guard condition is present then the following two code fragments behave equivalently *after blocking until the guard* `E` *is presciently known to yield true or terminate with an exception*:

```
atomic (E) {
  S;                                { E; S; }
}
```

Nested atomic blocks are considered to be flattened into the blocks that enclose them, with the entire assembly of blocks appearing to run at a point where all of the guard conditions executed will yield true or terminate with an exception.

These definitions have three major consequences. Firstly, they mean that if a system is genuinely single-threaded then the contents of an `atomic` block can be executed directly when its guard is satisfied. Secondly, these definitions lead to the semantics for exception propagation in our original paper – that is, if `E` or `S` terminates with an exception then the updates made up to that point are retained (1). Thirdly, these definitions allow the guard expression `E` to have side effects – this may be important in practice if, for example, the guard accesses a self-organizing data structure such as a splay tree (3).

There are numerous Java-specific subtleties which we elide here. These include, interruption while waiting, the interaction between class-loading and atomic block execution, thread creation within atomic blocks and the use of condition variables within atomic blocks. These issues are ones which would need to be considered carefully if incorporating `atomic` blocks into the design of a new language.

1.2 Implementation overview

Although we define the semantics of `atomic` blocks in terms of single-threaded execution it is not necessary to actually serialise them. Instead, we use a software transactional memory (STM) which allows groups of memory accesses to be performed within transactions which commit atomically.

```
boolean done = false;
while (!done) {
  STMStart ();
  try {
    statements;
    done = STMCommit ();
  } catch (Throwable t) {
    done = STMCommit ();
    if (done) {
      throw t;
    }
  }
}
```

Fig. 2. Translation of `atomic { statements; }` into STM operations.

The particular STM used in our Java prototype allows transactions to execute in parallel so long as the addresses accessed by different transactions do not collide under a hash function which forms part of the STM's implementation – in general this means that they execute in parallel unless they attempt conflicting operations.

The STM is implemented in C as part of a modified Java Virtual Machine and provides operations for starting a new transaction (`STMStart`), aborting the current transaction (`STMAbort`), committing the current transaction (`STMCommit`), reading a word within the context of the current transaction (`STMRead`) and updating a word within the context of the current transaction (`STMWrite`). There are two further operations to validate transactions and to block threads while waiting for conditions to become true – these are not relevant to the current paper, but are described in detail in our earlier work (1).

The higher layer of the implementation maps the `atomic` keyword onto a series of STM operations. For example, entering an atomic block requires `STMStart` to be invoked, and accesses to shared fields within a block require that `STMRead` and `STMWrite` be used in place of direct heap accesses. This translation is implemented in the source-to-bytecode compiler (for transaction management operations) and the bytecode-to-native compiler (for individual field accesses). Of course, the bytecode-to-native compiler must also ensure that appropriate STM operations are used in methods called from within `atomic` blocks. This is done by dynamically producing specialised versions of those methods.

As an example, Figure 2 summarises how a basic non-nesting `atomic` block without a guard condition may be expressed in terms of these explicit transaction management operations. Again, our previous paper describes these two levels more thoroughly (1).

2 Programming abstractions for atomic operations

There are several recent proposals for alternative abstractions for concurrent programming and alternative implementation techniques for building them.

Herlihy *et al* designed an object-based software transactional memory for Java which, unlike our design, works with an unmodified JVM (4). Transaction management operations are made through a library that the STM provides and the objects manipulated in transactions must be explicitly *opened* for transactional access before the first time they are used. Fraser designed a similar system as a library for programs written in C (5).

In a series of papers, Welc *et al* showed how existing Java programs using `synchronized` blocks can be executed using STM-like techniques, either forming per-thread logs of updates that they propose to make to the shared heap, or per-thread roll-back logs of updates that need to be undone if conflicts are detected (6; 7). This allows existing programs to be executed without being re-written to use alternative constructs like `atomic`.

Our techniques for dealing with exceptions and I/O are applicable to both of these approaches because they share a common strategy of *optimistic* execution in which threads execute potentially-conflicting operations while allowing their tentative updates to be aborted. If an operation completes without conflict then its tentative updates can be made permanent. Otherwise, if a conflict may have occurred, it can discard its tentative updates and re-try its operation. This means that, as in our system. operations with externally visible effects (such as output) must be deferred until the eventual commit / roll-back decision is made.

Welc *et al* suggest an alternative implementation method when the programmer describes concurrency control using mutual exclusion locks: if an operation with external I/O side-effects is attempted then the optimistic execution scheme can be disabled for the locks that are held (7). In their design this means that it is no longer necessary to support roll-back for those locks and so the I/O operations can be executed directly. This works well when using existing code, but it is not directly applicable using only `atomic` blocks rather than explicit synchronization.

A further alternative, which we have been exploring in ongoing work with the Haskell programming language, is to completely forbid I/O operations within atomic transactions (8). In Haskell we use the type system to distinguish operations that may have transactional side effects on the heap from operations that may have unrestricted side effects. The Haskell type we give to `atomic` guarantees that it contains only transactional operations and pure computation. This provides a robust guarantee that I/O will not be attempted.

3 Managing exceptions

The semantics defined in Section 1.1 mean that if an atomic block terminates with an exception, then any heap updates made within the block are retained and the exception is propagated. This allows single-threaded code to be directly re-used in a multi-threaded environment by inserting atomic blocks around related accesses to the heap, without having to think about whether automatic roll-back would be correct.

However, as Shinnar *et al* have argued, there are many examples where it is more convenient for the system to undo any updates that have led to an exception being raised: this reduces the need for programmers to write error-recovery code which is often intricate and difficult to test (9).

For illustration, consider code to move an object between two collections in which the source provides a `remove` method and the destination provides an `add` method. The `add` method throws an exception if the target collection cannot hold the item supplied. Figure 3 shows how a move operation can be implemented using an `atomic` block. The code is not elegant; the programmer must manually implement fix-up operations if the destination cannot contain the item supplied. In fact, in a full solution, it would be necessary to consider exceptions raised by the compensating `add` if the object is rejected by both collections. Furthermore, although the compensating operation means that the abstract state of the two collections is unchanged, the physical representation in memory is subject to numerous updates. This is a problem in concurrent systems because it increases contention in the memory hierarchy: the aborted move ends up forming an expensive no-op which may impede other threads' operation.

Of course, these same observations would hold if the `move` method was implemented using mutual exclusion locks. However, building the system over a STM allows the more convenient option of replacing the compensating operation with a request that the STM simply discards any heap updates performed within the `atomic` block.

However, there are problems with simply using the `STMAbort` operation that to roll back an `atomic` block before propagating an exception. The main problem is that aborting would undo *all* of the updates made in the transaction: we cannot roll back the creation of the exception object because we may need it to signal the kind of problem that arose. Even worse, if we roll back the instantiation of the exception object then we would be left with a dangling pointer if we then tried to propagate it – this could happen with hardware implementations of transactions in which all operations, including those performed in allocation functions, would be logged by the processor (10; 11; 12).

```
boolean move(Collection s, Collection d, Object o)
{
  atomic {
    if (!s.remove(o)) { /* Try to remove object */
      return false; /* Could not find object */
    } else {
      try {
        d.add(o); /* Add to target collection */
      } catch (RuntimeException e) {
        s.add(o); /* Compensating add */
        return false;  /* Move failed */
      }
      return true; /* Move succeeded */
    }
  }
}
```

Fig. 3. A collection-to-collection move using manual roll back.

Unfortunately, retaining the exception object while reverting other changes is not a viable alternative: what if the exception object refers to objects instantiated in the atomic block? What if it refers to objects that have been modified in the atomic block? What if the object thrown is actually a pre-existing one that is modified in the atomic block before being thrown? In general, the exception object could be interlinked with other data structures, making it unclear which modifications to retain and which to discard.

We avoid this problem by using object serialization to define what happens when aborting a block while retaining the exception object which triggered the abort. This is because the serialized byte-array form of an object is meaningful between JVMs and therefore meaningful between an `atomic` block and its enclosing context. If a block terminates by throwing an exception `e` whose serialized representation would be a byte-array `b` then the effect of executing the block is equivalent to de-serializing a byte-array with the same contents as `b` and then throwing the resulting exception. Of course, this 'as if' definition allows the exception object to be retained and thrown directly if static analysis can show that the behaviour is equivalent.

A further problem is that if all exceptions trigger roll back then it precludes alternative implementations of `atomic` blocks which, unlike our STM, do not produce the logging information necessary to abort a transaction – this might be true of a scheme based on automatic locking rather than an STM, or a scheme which includes optimizations for single-threaded use.

Our approach is to introduce a new `AtomicAbortException` class and to have instances of that, or its subclasses, trigger roll back. This is a checked exception

```
boolean move(Collection s, Collection d, Object o)
{
  try {
    atomic {
      try {
        if (!s.remove(o)) { /* Try to remove object */
          return false; /* Could not find object */
        } else {
          d.add(o); /* Add to target collection */
          return true; /* Move succeeded */
        }
      } catch (RuntimeException e) {
        throw new AtomicAbortException(e);
      }
    }
  } (catch AtomicAbortException e2) {
    return false; /* Move failed */
  }
}
```

Fig. 4. A collection-to-collection move using automatic roll back.

class and so the programmer must indicate where it may be thrown, allowing a non-abortable implementation to be used for blocks where these exceptions are not present.

Figure 4 shows how an atomic collection-to-collection move could be implemented using roll back: it is no longer necessary to include explicit compensatory code, and failed moves will lead to aborted lower-level transactions, reducing contention. As is typical, the `AtomicAbortException` which crosses the boundary of the `atomic` block carries the original exception raised by `remove` in order to indicate the root cause of the failure.

4 Managing I/O operations

The second area which we consider in this paper is how to support `atomic` blocks with external side effects. In our original design we prohibited blocks from invoking any `native` method – that is, any method that is not implemented in Java bytecode. This ultimately precludes the availability of most I/O operations.

Unfortunately, it is not possible to allow native methods to be called from `atomic` blocks by trapping heap accesses made across the Java Native Interface (JNI). That would provide no control over system calls invoked from native

```
void serverLoop(ServerSocket s) {
  while (true) {
    Socket c = s.acceptConnection(); /*M1*/
    Thread t = new Thread() {
      public void run() {
        atomic {
          try {
            dealWithClient(c); /*M2*/
          } catch (Throwable t) {
            // Roll back updates made by clients
            // whose actions cause exceptions
            throw new AtomicAbortException(t);
          }
        }
      }
    };
    t.start();
  }
}
```

Fig. 5. Stylized server execution using an atomic block to isolate each client – in practice the `run` method would need to handle the `AtomicAbortException` and perhaps close down the client's connection, log errors and so on.

methods, or on code within the JVM which uses internal lower-level interfaces to bypass JNI.

Of course, there are some operations for which the JVM cannot guarantee atomicity. For example, the programmer may define an `atomic` block to swap the names of two files by a series of `renameTo` method calls. Operating system support would be needed to make these operations appear atomic to other processes; all that can reasonably be provided is atomicity in the sense that either all of the operations in the block appear to occur, or none of them occurs. Again, this is consistent with our intended 'as-if single threaded' semantics from Section 1.1.

Furthermore, different behaviour is appropriate for different kinds of I/O operation. For instance, consider the highly stylized server loop shown in Figure 5. Connections from clients are received at method call `M1` and each is dealt with in an `atomic` block in a separate thread at `M2`. If an exception occurs in `M2` then the effect of the atomic block is discarded. In this case it may be appropriate for the external interactions performed between the client and the server to be carried out directly while executing the block and for the roll back to only discard updates to the state within the server: the exception may indicate an internal error in the server or one that has been triggered by a maliciously formed request from a client.

```
public class ExampleOutput {
  static PrintStream out =
     new PrintStream(
        new AtomicOutputStream(System.out));

  static void print_sum(int x, int y) {
    atomic {
      int result = x + y;
      out.println ("Result is " + result);
    }
  }
}
```

Fig. 6. Using an `AtomicOutputStream` to buffer output from an `atomic` block.

Rather than directly supporting unmodified native methods, the approach we take is to provide a set of Java-based interfaces with which an I/O library can implement appropriate buffering semantics. These allow a thread to determine whether it is in an atomic block and to register call-backs for when the transaction underlying the block attempts to commit or abort.

This allows a wide range of behaviour to be implemented. For instance, an output library can perform its own buffering of the deferred output, register a callback on commit to flush the output and register a callback on abort to discard the buffered state. Similarly, a library performing input can register a callback on abort to re-buffer the input which had been presented to the aborted transaction. This approach allows device-specific forms of buffering to be used – for example, to distinguish between stream-based input which cannot be re-ordered and datagram-based input in which datagrams may be re-ordered.

For console I/O we have implemented simple wrapper classes which provide example buffering layers for use above the ordinary I/O streams. Figure 6 shows an example of how such an `AtomicOutputStream` can be used. If these I/O features were integrated fully into the environment then these wrappers could be provided as the default I/O streams.

Of course, such application-agnostic approaches only work for simple situations in which the input received by the block does not depend on the output that it generates. More complex cases would require call-backs to engage in a distributed commit protocol, or to perform compensation actions using techniques like BPEL (13).

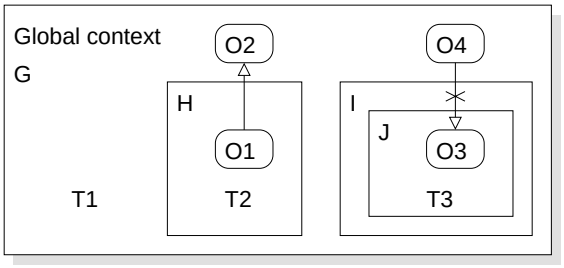

Fig. 7. Permitted (O1-O2) and forbidden (O4-O3) inter-context references. The instantiation of O3 may be buffered in transaction-private logs for T3, and so the reference would appear dangling from the point of view of T1.

5 External actions

In this section we introduce the 'external action' abstraction with which we implement our exception propagation model and I/O support libraries. In Sections 5.1 and 5.2 we discuss two ways of exposing external actions to programmers; we have implemented the first of these options and, although we have a thorough design for the second option, we have not yet tested it in practice.

External actions provide a controlled way in which code within an `atomic` block can temporarily perform operations directly on the heap rather than within the context of the current transaction. External actions are used in propagating exceptions in order to marshal the exception object so that it is available after the transaction is aborted. External actions are used during I/O to invoke native operations and to perform device-specific buffering to give transactional behaviour.

The behaviour of external actions is defined in terms of *contexts* which represent the different views that threads may have on the heap at any given moment. Contexts are hierarchical and a single *global context* exists as the root. Heap updates are said to occur *within* a given context, meaning that they are guaranteed to be visible to threads executing in that context, or executing within any context nested inside it. Conversely, updates made in one context are not guaranteed to be visible outside the context – for instance, they may be buffered in a thread-local log, as with our STM-based implementation.

When a thread enters an atomic block it creates a new context nested within its current one. When a thread leaves an atomic block then the nested context is discarded after *promoting* any heap updates made within it up to its parent context.

Figure 7 illustrates a set of nested contexts. Thread T1 is executing in the global context G. Thread T2 is executing in context H within G. Thread T3

is executing within context J, nested two levels deep. Objects allocated in one context can only contain references to objects allocated in enclosing contexts, for instance O1 can refer to O2, but O4 cannot refer to O3.

This rule ensures that a thread following a pointer is guaranteed to be executing in a context which can see the referent. The key challenge in Java in designing a mechanism for temporarily 'stepping outside' the current context is making it impossible to create references which circumvent this rule.

We deal with this problem by representing external actions as designated `ExternalAction` objects and ensuring that (*i*) actions are executed in the context within which the object is instantiated, and (*ii*) actions' parameters are passed by serialization. The first property ensures that free variables occurring within an action's definition will refer to data that is accessible in the context within which the action executes. The second property ensures that any incoming parameters received by the action have been copied and re-created within the context that the action executes.

We expose contexts to Java programmers as immutable `Context` objects which uniquely identify an active context and allow traversal from it to its enclosing context object. A static method returns the caller's current context. A thread can register a `ContextListener` with any context that is contained within its current one. Context listeners receive three call-backs:

```
boolean validToCommit(Context c);
void actionOnCommit(Context c);
void actionOnAbort(Context c);
```

These three operations are used to perform a two-phase commit of updates that external actions have associated with a context. The first of these operations, `validToCommit`, is called when deciding whether the context should be destroyed or whether, at the end of an `atomic` block, updates made within it should be merged into its parent context. If any context listener returns `false` then the context must be destroyed. The second and third call-backs are called to inform the listener of the outcome of this voting.

External actions are implemented by extending the STM interface with two context-control operations: a method for setting the current transactional context used by STM operations and a method for doing an inter-context copy of arrays of bytes when serializing parameters to external actions. The remainder of the implementation is Java-based; the `STMCommit` operation becomes a Java method which calls `validToCommit` on any `ContextListener` objects before attempting to commit the underlying STM transaction.

The two context-control operations are available only to trusted code because they may be used to create outer-to-inner inter-context references. We have

```
public class ExampleActionCall {
  static int x = 0;

  static VoidExternalAction printX =
    new VoidExternalAction() {
      public void action(Context caller_context) {
        System.out.println(" x=" + x);
    }};

  static void increment_x() {
    atomic {
      printX.doAction();
    }
  }
}
```

Fig. 8. An example code fragment defining and invoking an external action.

investigated two ways of building safe mechanisms through which to expose them to applications and libraries. The first of these, which we describe in Section 5.1, allows a single operation to be defined at a time. The second design, in Section 5.2, exports a whole `interface` of external actions: it is more verbose for short examples but is more convenient for non-trivial cases.

5.1 Operation-based external actions

The first way of defining external actions uses a simple mechanism in which the action is defined by overriding an `action` method on an `ExternalAction` class. A separate trusted `doAction` method uses the context-control extensions to marshal parameters for the action and to invoke it in the appropriate context.

Figure 8 illustrates this: an anonymous subclass of `VoidExternalAction` is created with an `action` method that outputs the value of `x`. When `doAction` is called from the context created in `increment_x`, the action is executed in the global context that was active when `printX` was initialized.

Variable-length argument lists can simplify the infrastructure for defining this form of external actions by avoiding the proliferation of separate kinds of action class. Similarly, aside from actions with `void` return type, a single parametric definition would suffice.

However, with this operation-based approach, defining external actions which can throw checked exceptions remains problematic: we cannot represent the relationship between the set of exceptions that can be raised in the user's `action` method and the set of exceptions that may result from the call to `doAction`.

When defining and calling these actions, we often needed to use inelegant techniques such as hiding checked exceptions within unchecked wrappers.

5.2 Interface-based external actions

The second way of defining external actions is more suitable for use in larger examples where the entire set of existing methods on an object are to be encapsulated as external actions. The approach is to allow an object to be exported from one context and for *all* method invocations on it to be made via stubs which behave as external actions.

The need for this kind of interface-based design became particularly apparent while creating wrappers for use around the Java Transaction API in which large numbers of boilerplate actions otherwise had to be written to wrap existing implementations of interfaces such as `UserTransaction`, `Connection` and `PreparedStatement`.

Figure 9 illustrates how the earlier `increment_x` example from Figure 8 could be expressed in this alternative form. As before, the example ultimately prints the contents of a field `x` in the global context. This operation is performed by (*i*) providing an interface `printXIfc` which defines the signatures of the methods to be exported as external actions, (*ii*) defining an implementation of these operations to be exported, (*iii*) invoking `ExternalAction.export()` to produce a set of stubs to perform the inter-context calls.

The stubs are constrained to implement an identical interface to one implemented by the original, retaining `throws` clauses for checked exceptions as well as the details of return types and parameters.

6 Implementation experience

In this section we consider the use of external actions in providing a mechanism for managing exceptions (Section 6.1) and for performing external I/O operations (Section 6.2).

6.1 Propagating exceptions

The exception-propagation mechanism proposed in Section 3 can be implemented by a single external action that takes the exception object created within the `atomic` block and returns a deep copy of it created in the global

```
//.....................................................
//
// Definition of interface exported

interface printXIfc {
  public void printX();
}

//.....................................................
//
// Signature of export operation.  This is parameterized
// by F which is expected to be an interface implemented by
// the exported object.

public class ExternalAction {
  static <F> F export(F imp) {
    ...
  }
}

//.....................................................
//
// Construction and invocation of an external action

public class ExampleActionCall {
  static int x = 0;

  // Implementation of the external action
  static printXIfc printer =
    (printXIfc) ExternalAction.export(
      new printXIfc() {
        public void printX() {
          System.out.println(" x=" + x);
        }
      });

  // Atomic call to the external action
  static void increment_x() {
    atomic {
      printer.printX();
    }
  }
}
```

Fig. 9. An external action defined using an interface.

context. In fact, the actual copying of the exception object to the global context is performed by the marshaling of the exception object when it is passed to the external action.

The design in Figure 2 for implementing an `atomic` block using STM operations is extended to propagate exceptions by adding an exception handler of type `AtomicAbortException` and having this promote the exception, abort the transaction and then re-throw the copy the exception.

The definition of the action is therefore simply:

```
static ObjectExternalAction promoteException =
  new ObjectExternalAction() {
    public Object action
               (Context caller_context,
                Serializable aae) {
      return aae;
    }
  };
```

Programmers defining sub-classes of `AtomicAbortException` need to be aware that, by default, they will receive a deep clone of the exception and the objects reachable from it. This means that they must either ensure that all of these object are themselves serializable – they may find that, for instance, instances of a singleton class may not be. If they need finer control then they can, of course, define custom serialization methods – for instance, if they wish to preserve references to a unique instance of a singleton class then they can do so when deserializing the exception.

We do not believe this to be a problem in practice where exceptions are used to signal error conditions and rarely carry data other than stack-traces and error messages.

6.2 Performing I/O

I/O operations are implemented using external actions to perform any native method invocations necessary for the I/O and using `ContextListener` callbacks to trigger re-buffering of unused input (when rolling back a transaction that has performed input) or to trigger the actual output of buffered data (when committing a transaction that has performed output).

For example, when reading from standard input, an external action is used to perform the read. It calls a native read method from within the global context and buffers the value read, again within the global context. In this

case a context listener is registered to re-buffer the data if the `atomic` block is aborted, or to discard the buffer if the `atomic` block completes successfully.

We define a set of utility classes which simplify the implementation of abstractions such as the `AtomicOutputStream` wrapper. These hold ordered collections of objects that are buffered until an atomic block commits, and collections of input items that have been received by an `atomic` block and must be held for potential re-buffering in case the block aborts.

Integration with external database transactions is not so straightforward. We have built a prototype system based on the Java Open Transaction Manager (JOTM) [1] , although this relies on modifications to the JOTM implementation rather than being made through the established Java Transaction API (14). The fundamental problem is that both the STM and the JOTM system want to make the final decision of whether or not to commit a set of operations; neither allows the other to perform a separate 'prepare' phase. We chose to extend JOTM's `UserTransaction` interface with an additional `prepare()` operation.

7 Related work

This `atomic` construct builds on designs for Conditional Critical Regions (CCRs) (15) and on the concurrency control features of languages such as DP (16), Edison (17), Lynx (18) and Argus (19).

The Real-Time Specification for Java (RTSJ) defines a way of allocating objects within 'scoped memory areas' in order to allow storage reclamation without a run-time garbage collector (20). Scoped memory areas must obey similar constraints to the `Context` objects proposed here: objects within one area may not refer to objects in less permanent areas.

Stack-like memory usage disciplines have been investigated in several other settings, most notably region-based memory management (21). Regions have been proposed as an alternative or adjunct to traditional garbage collection, allowing objects to be allocated within a stack of regions and allowing space to be reclaimed by removing an entire region from the top of the stack. Safety requires that references do not occur from more permanent regions into less permanent ones.

There are three main areas in which differences exist between our scheme, regions and scoped memory areas. The first is in whether the prevention of

[1] `http://jotm.objectweb.org`

illegal references is done statically or dynamically: our system, as with conventional region-based ones, takes the former approach whereas RTSJ takes the latter. The second point of comparison is the direction in which contexts are entered: our system must support transitions both from an outer context to an enclosed one (by entering an atomic block) and from an enclosed context to an outer one (by invoking an external action). The final point is that the stack of `Context` objects in our system should be viewed as 'overlays' on the same heap, with objects at one layer being shadowed by objects at enclosed layers, whereas the identities of objects in different regions or scoped areas are considered distinct.

8 Conclusions and future work

This paper has shown how we have extended our `atomic` regions for concurrent Java programs to support explicit abort operations and I/O. The design presented here introduces a notion of nested execution contexts and an abstraction for performing inter-context method calls. In this final section we highlight a number of dead-ends we followed in earlier designs (Section 8.1) and a number of extensions for future work (Section 8.2).

8.1 Early dead-ends

Although these final abstractions are individually simple, designing them highlighted a number of problems which we had not originally foreseen. These all relate to the need to be careful about passing object references into a context in which the initialisation of the objects' fields will not have been visible.

The original design we sketched proposed control methods through which reads or writes could be performed outside the current software transaction (22). This approach is not safe with respect to the language-based protection provided by Java: for example, `final` fields are intended to be constant once initialised, but using these methods a programmer could cause the initialisation to happen within a transactional context and subsequent accesses to take place outside that context and therefore without the initializations visible.

In subsequent designs we considered introducing a form of 'global action' which would always execute in the global context. As with our method-based design for external actions, these would be defined by instantiating an anonymous inner class, for example:

```
atomic {
  final String s = new String("Erroneous example");
  GlobalAction g = new GlobalAction() {
    public void doAction(Context caller_context) {
      System.out.println ("s=" + s); /*P1*/
  }};
  g.doAction();
}
```

Unfortunately if `P1` is executed in the global context then the initialization of the fields in the object `s` refers to is not visible – for instance, the updates may still be buffered in transaction-local storage. Note how our decision to execute external actions within the context within which they are instantiated avoids this problem without the need for dynamic checks. It also deals naturally with the case of nested contexts.

8.2 Future work

The key direction for future work is evaluating the practical utility of the techniques that we have developed: we have now considered atomic blocks with an armoury of features, but we have not exercised these features in earnest in a large system.

Object finalizers still pose a problem: if an object is instantiated in an `atomic` block and that block is subsequently rolled back by an exception then should finalizer methods be invoked on the objects that are lost? What happens if those methods loop or invoke external actions? There appear to be two options: the first is to consider the destruction of the atomic block's context to entirely undo the creation of the objects and therefore to not run finalizers on them. The second option is to execute the finalizers within the context that the objects were instantiated – i.e. to execute them just before destroying the context. These two options have different behaviour if the finalizers loop or perform external actions. We favour the first option because it is simpler to implement and because it is consistent with the semantics of Section 1.1.

A further point for future investigation will be the relationship between this work and the `java.util.concurrent` library [2] of J2SE 1.5. For instance, once there are benchmarks targeting JSR-166 features, then it will be interesting to compare the implementation of collections and queues built using `atomic` blocks with those built using the virtual machine's existing abstractions. We hope that our work is an excellent counterpart to JSR-166 and that the combination of well-engineered high-level abstractions and an effective mechanism

[2] JSR-166, `http://www.jcp.org/en/jsr/detail?id=166`

for extending them to provide aggregate atomic operations may encourage more wide-scale adoption of concurrency in applications.

8.3 Acknowledgments

The work reported here was undertaken while the author was at the University of Cambridge Computer Laboratory and supported by a donation from the Scalable Synchronization Research Group at Sun Labs Massachusetts and access to the Cambridge-Cranfield High Performance Computing Facility.

References

[1] T. Harris, K. Fraser, Language support for lightweight transactions, in: Object-Oriented Programming, Systems, Langauges & Applications (OOPSLA '03), 2003, pp. 388–402.

[2] D. B. Lomet, Process structuring, synchronization and recovery using atomic actions, in: D. B. Wortman (Ed.), Proceedings of an ACM Conference on Language Design for Reliable Software, ACM, ACM, 1977, pp. 128–137.

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, Introduction to Algorithms, MIT Press, Cambridge, Mass., 1990.

[4] M. Herlihy, V. Luchangco, M. Moir, W. N. Scherer, III, Software transactional memory for dynamic-sized data structures, in: Proceedings of the 22nd Annual ACM Symposium on Principles of distributed computing, ACM Press, 2003, pp. 92–101.

[5] K. Fraser, Practical lock freedom, Ph.D. thesis, University of Cambridge Computer Laboratory (2003).

[6] A. Welc, S. Jagannathan, T. Hosking, Transactional monitors for concurrent objects, in: Proceedings of the European Conference on Object-Oriented Programming (ECOOP), 2004, pp. 519–542.

[7] A. Welc, A. L. Hosking, S. Jagannathan, Preemption-based avoidance of priority inversion for java, in: Proceedings of the 2004 International Conference on Parallel Processing (ICPP), 2004, pp. 529–538.

[8] T. Harris, M. Herlihy, S. Marlow, S. Peyton-Jones, Composable memory transactions, in: Proceedings of the ACM Symposium on Principles and Practice of Parallel Programming, 2005.

[9] A. Shinnar, D. Tarditi, M. Plesko, B. Steensgaard, Integrating support for undo with exception handling, Tech. Rep. MSR-TR-2004-140, Microsoft Research (Dec. 2004).

[10] M. Herlihy, J. E. B. Moss, Transactional memory: Architectural support for lock-free data structures, in: Proceedings of the 20th Annual Interna-

tional Symposium on Computer Architecture, IEEE Computer Society Press, 1993, pp. 289–301.

[11] R. Rajwar, J. R. Goodman, Transactional lock-free execution of lock-based programs, ACM SIGPLAN Notices 37 (10) (2002) 5–17.

[12] L. Hammond, B. D. Carlstrom, V. Wong, B. Hertzberg, M. Chen, C. Kozyrakis, K. Olukotun, Programming with transactional coherence and consistency (TCC), in: ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems, ACM Press, 2004, pp. 1–13.

[13] F. Curbera, F. Leymann, T. Storey, D. Ferguson, S. Weerawarana, Web Services Platform Architecture: Soap, WSDL, WS-Policy, WS-Addressing, WS-Bpel, WS-Reliable Messaging and More, Prentice Hall, 2005.

[14] I. Singh, B. Stearns, M. Johnson, Designing enterprise applications with the J2EE platform, 2nd Edition, Addison Wesley, 2002.

[15] C. A. R. Hoare, Towards a theory of parallel programming, in: Operating Systems Techniques, Vol. 9 of A.P.I.C. Studies in Data Processing, 1972, pp. 61–71.

[16] P. Brinch Hansen, Distributed processes: A concurrent programming concept, Communications of the ACM 21 (11) (1978) 934–941.

[17] P. Brinch Hansen, Edison – a multiprocessor language, Software – Practice and Experience 11 (4) (1981) 325–361.

[18] M. L. Scott, Language support for loosely coupled distributed programs, IEEE Transactions on Software Engineering SE-13 (1) (1987) 88–103.

[19] B. Liskov, R. Scheifler, Guardians and actions: linguistic support for robust, distributed programs, ACM Transactions on Programming Languages and Systems 5 (3) (1983) 381–404.

[20] G. Bollella, B. Brosgol, P. Dibble, S. Furr, J. Gosling, D. Hardin, M. Turnbull, R. Belliardi, The Real-Time Specification for Java, Addison Wesley, 2000.

[21] M. Tofte, J.-P. Talpin, Region-based memory management, Information and Computation.

[22] T. Harris, Design choices for language-based transactions, Tech. Rep. UCAM-CL-TR-572, University of Cambridge, Computer Laboratory (Aug. 2003).