Exceptions and side-effects in atomic blocks

Tim Harris
University of Cambridge Computer Laboratory
15 JJ Thomson Avenue
Cambridge, UK, CB3 0FD
tim.harris@cl.cam.ac.uk

ABSTRACT

In our paper at OOPSLA 2003 we discussed the design and implementation of a new `atomic` keyword as an extension to the Java programming language. It allows programs 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. Both extensions are based on a single 'external action' abstraction, allowing code running within an atomic block to request that a given pre-registered operation be executed outside the block.

1. INTRODUCTION

In recent work we have been investigating the use of Software Transactional Memory as a mechanism for implementing language-level concurrency control features [6]. In our system, developed as an extension to the Java programming language, we have 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.

In this paper we expand the range of operations which can be performed within `atomic` blocks in two different ways. The first extension we consider is what behaviour to provide when an `atomic` block terminates early by an exception being thrown. The dilemma here is whether to roll-back 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. We discuss this in Section 2 and propose a hybrid model in which certain exceptions cause `atomic` blocks to be aborted and in which the exception thrown outside the block is a deep copy of the exception raised within it.

The second area we investigate is how to deal with I/O performed within an `atomic` block: our original design forbade any native method invocations which made most I/O operations unavailable. In Section 3 we discuss a number of ways in which I/O could be supported 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 automatic support for native methods.

```
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;
    }
  }
}
```

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

Our approach for supporting both of these new features is based on a single 'external action' abstraction which we introduce in Section 4. An external action object exports an operation which can be invoked from within an `atomic` block but which is executed within the context that the object was instantiated.

In Section 5 we discuss our experience using external actions to implement our exception-propagation model and I/O system. Finally, Section 6 discusses related work and Section 7 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

We informally define the semantics of non-nesting `atomic` blocks 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.

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 fragements behave equivalently *after blocking until the guard* `E` *is presciently known to yield true or terminate with an exception*:

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

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 [6]. 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 [4].

There are numerous subtleties which we elide here. These include dynamically nesting blocks, 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 we do not envisage that that would form the basis of an implementation. Instead, our current implementation is designed to allow most non-conflicting atomic blocks to execute concurrently.

The system is built in two layers. The lower layer is a word-based software transactional memory (STM). This allows groups of memory accesses to be performed within transactions which commit atomically. The STM is implemented in C within the Java Virtual Machine and provides operations for starting a new transaction (`STMStart`), aborting the current transaction (`STMAbort`), committing the current transaction (`STMCommit`), for reading a word within the context of the current transaction (`STMRead`) and for 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.

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

Figure 2: Code of the form `atomic { statements; }` expressed using STM management operations. In practice exception propagation is complicated by the fact that the translated code must throw the same set of exceptions as the original `statements`. Heap accesses within the `statements` (and within any methods they call) are performed using the STM.

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). The intermediate Java bytecode format is unchanged.

Our previous paper describes these two levels in detail [6]. 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.

2. 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. However, there are examples where it would seem more convenient for programmers to be able to *roll-back* any updates made within the `atomic` block up to the point where the exception is thrown.

For illustration, consider code to move an object between two collections. The source collection provides a `remove` method. The destination collection provides an `add` method that fails with 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. Furthermore, when `R1` has to be counteracted by `A2`, the underlying transaction may involve numerous updates even though the abstract state of the two collections is unchanged. This is a problem in concurrent systems because it increases contention in the memory hierarchy. It may even be necessary

```
boolean move(Collection s, Collection d, Object o)
{
  atomic {
    if (!s.remove(o)) { /* R1 */
      return false; /* Could not find object */
    } else {
      try {
        d.add(o); /* A1 */
      } catch (RuntimeException e) {
        s.add(o); /* A2 */
        throw e;  /* Move failed */
      }
      return true; /* Move succeeded */
    }
  }
}
```

Figure 3: A collection-to-collection move implemented within an atomic block using manual roll-back. The `add` operation `A2` compensates for `R1` if the object is removed from the source collection but cannot be inserted into the destination.

to consider exceptions raised by `A2` if the object is rejected by both collections.

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 `A2` with a request that the STM simply discards any heap updates performed within the `atomic` block.

2.1 Problems

Although the underlying STM provides an abort operation, this cannot be used directly to roll-back an `atomic` block before propagating the exception which caused the block to be aborted. The problem is that aborting would undo *all* of the updates made in the transaction: if the exception object was instantiated or modified in it then retaining that object is incompatible with rolling back the modifications. In the general case the exception object could be interlinked with other data structures, making it unclear which modifications to retain and which to lose.

There are two, more subtle problems with blindly using exceptions to trigger roll-back. The first is that it could destroy invariants assumed by existing code. For example, a library may ensure that a particular kind of exception is only thrown once a data structure has reached a given state. This guarantee would be broken if changes leading up to the exception were rolled back but the exception object was retained.

The second 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 locking rather than an STM, or a scheme which includes optimizations for single-threaded use.

2.2 Design

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 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.

```
boolean move(Collection s, Collection d, Object o)
{
  try {
    atomic {
      try {
        if (!s.remove(o)) { /* R1 */
          return false; /* Could not find object */
        } else {
          d.add(o); /* A1 */
          return true; /* Move succeeded */
        }
      } catch (RuntimeException e) {
        throw new AtomicAbortException();
      }
    }
  } (catch AtomicAbortException e) {
    return false; /* Move failed */
  }
}
```

Figure 4: A collection-to-collection move using an `AtomicAbortException` for roll-back.

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.

We use 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 it is possible to identify that as equivalent through static analysis.

3. 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.

3.1 Problems

It is not possible to allow native methods to be called from `atomic` blocks by simply ensuring that JNI heap accesses are performed using the STM. That would provide no control over system calls invoked from native 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, a highly stylized server implementation may be written as a loop:

```
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) {
            throw new AtomicAbortException(t);
          }
        }
      }
    };
    t.start();
  }
}
```

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.

In other cases it might be appropriate for external interactions to be deferred until the block has completed, or for corresponding compensatory operations to be issued if it does roll back.

3.2 Design

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.

```
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);
    }
  }
}
```

Figure 5: An example class instantiating and using an `AtomicOutputStream` wrapper to buffer output made within the `atomic` block in the `print_sum` method.

For console I/O we have implemented simple wrapper classes `AtomicInputStream` and `AtomicOutputStream` which provide example buffering layers for use above the ordinary I/O streams. Figure 5 shows an example of how 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.

4. 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 4.1 and 4.2 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 marshall 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 are 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.

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 6 illustrates a set of nested contexts.

The key challenge in Java in designing a mechanism for temporarily ‘stepping outside’ the current context is making it impossible to circumvent encapsulation enforced by language-based protection. In particular, code executing in

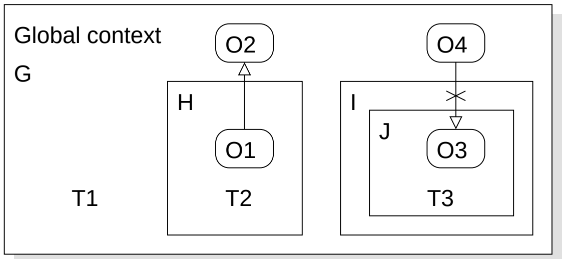

Figure 6: Thread T1 is executing in the global context G. Thread T2 is executing in context H within G. Thread T3 is executing within context J, nested two levels deep. Objects allocated in one context can only contain references to objects allocated in enclosing contexts, for instance O1 can refer to O2, but O4 cannot refer to O3.

a given context must not be able to access objects instantiated in an enclosed context – otherwise, for example, there is no guarantee that the code would even see the objects correctly initialised.

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, `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 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. However, we have investigated two ways of exposing them safely to ordinary code such as applications and I/O library implementations. The first of these, which we describe in Section 4.1, allows a single operation to be defined at a time. The second design, in Section 4.2, exports a whole `interface` of external actions: it is more verbose for short examples but is more convenient for non-trivial cases.

```
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();
    }
  }
}
```

Figure 7: An example code fragment defining and invoking an external action.

4.1 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 7 illustrates this: the `VoidExternalAction` class is extended with an `action` method that is called from the context created in `increment_x` but which is executed in the global context that was active when `printX` was initialized.

Generic types and variable-length argument lists can simplify the infrastructure for defining this form of external actions by avoiding the proliferation of separate kinds of action for different parameter and return types. Aside from actions with `void` return type, a single parametric definition would suffice.

However, with this approach, defining external actions which can throw checked exceptions remains problematic: the definition cannot be made parametric on a *set* of exceptions. In general the programmer has to follow inelegant approaches such as hiding checked exceptions within unchecked wrappers.

4.2 Interface-based external actions

The second way of defining external actions is more suitable for use in larger settings 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 exist-

```
// Definition of interface exported
interface printXIfc {
  public void printX();
}

// Signature of export operation
public class ExternalAction {
  static <F> F export(F imp) {
    ...
  }
}

// Invocation of external action
public class ExampleActionCall {
  static int x = 0;

  static printXIfc printer =
    ExternalAction.export(
      new printXIfc() {
        public void printX() {
          System.out.println(" x=" + x);
        }
      });

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

Figure 8: An external action defined using an interface.

ing implementations of interfaces such as `UserTransaction`, `PreparedStatement` and `Connection`.

Figure 8 illustrates how the earlier `increment_x` example from Figure 7 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.

5. IMPLEMENTATION EXPERIENCE

In this section we consider the use of external actions in providing a mechanism for managing exceptions (Section 5.1) and for performing external I/O operations (Section 5.2).

5.1 Propagating exceptions

The exception-propagation mechanism proposed in Section 2 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 context. The definition of this action is simply:

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

The actual copying of the exception object to the global context is performed by the marshalling of the exception object when `promoteException` is invoked. 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.

5.2 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` call-backs to trigger rebuffering of unused input (when aborting an input operation) or to trigger the actual output of buffered data (when committing an output operation).

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` in Figure 5. 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 (JTA) [11]. 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 the JTA `UserTransaction` interface with an additional `prepare()` operation. This issue would have to be addressed more methodically in a full-strength implementation of our system.

6. RELATED WORK

This `atomic` construct builds on designs for Conditional Critical Regions [7] and on the concurrency control features of languages such as DP [2], Edison [3], Lynx [10] and Argus [8].

Stack-like memory usage disciplines have been investigated in several other settings, most notably region-based memory management [12]. Regions have been proposed as

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

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.

The Real-Time Specification for Java (RTSJ) [1] defines a way of allocating objects within 'scoped memory areas' in order to allow storage reclamation without a run-time garbage collector. 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.

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 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.

7. CONCLUSIONS AND FUTURE WORK

This paper has shown how we have extended our `atomic` regions for concurrent Java programs to suppport 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 7.1) and a number of extensions for future work (Section 7.2).

7.1 Early dead-ends

Although these final abstractions are individually simple, developing them has 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 [5].

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 ultimate 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 exectued in the global context then the initialization of the object `s` refers is not visible. 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.

7.2 Future work

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? 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.

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. It will also be instructive to see whether the roll-back mechanisms triggered by `AtomicAbortException` objects can simplify sequential programs by automating the management of compensatory actions – this may be particularly useful when developing I/O-processing code with a wide variety of possible failure points.

A further point for future investigation will be the relationship between this work and the `java.util.concurrent` library[2] anticipated in 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 for extending them to provide aggregate atomic operations may encourage more wide-scale adoption of concurrency in applications.

7.3 Acknowledgments

This work has been supported by a donation from the Scalable Synchronization Research Group at Sun Labs Massachusetts.

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

8. REFERENCES

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

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

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

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

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

[6] Harris, T., and Fraser, K. Language support for lightweight transactions. In *Object-Oriented Programming, Systems, Langauges & Applications (OOPSLA '03)* (Oct. 2003), pp. 388–402.

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

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

[9] *Conference Record of the Twenty-first Annual ACM Symposium on Principles of Programming Languages* (Jan. 1994), ACM SIGPLAN Notices, ACM Press.

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

[11] Singh, I., Stearns, B., and Johnson, M. *Designing enterprise applications with the J2EE platform*, 2nd ed. Addison Wesley, 2002.

[12] Tofte, M., and Talpin, J.-P. Region-based memory management. *Information and Computation* (Feb. 1997). An earlier version of this was presented at [9].