Journal of Universal Computer Science, vol. 13, no. 8 (2007), 1047-1072
submitted: 2/2/07, accepted: 23/4/07, appeared: 28/8/07

Improving Program Correctness with Atomic Exception Handling

Christof Fetzer
(TU Dresden, Germany
christof.fetzer@tu-dresden.de)

Pascal Felber
(University of Neuchâtel, Switzerland
pascal.felber@unine.ch)

Abstract: Exception handling is a powerful mechanisms for dealing with failures at runtime. It simplifies the development of robust programs by allowing the programmer to implement recovery actions and tolerate non-fatal errors. Yet, exception handling is difficult to get right! The complexity of correct exception handling is a major cause for incorrect exception handling. It is therefore important to reduce the complexity of writing exception handling code while, at the same time, making sure it is correct. Our approach is to use atomic blocks for exception handling combined with optional compensation actions.
Key Words: exception handling, transactional memory
Category: D.2.m, D.3.3.

1 Introduction

Developing robust software is a challenging, yet essential, task. A robust program has to be able to detect and recover from a variety of faults such as the temporary disconnection of communication links, resource exhaustion, and memory corruption. Ideally, robust software has to be able to tolerate runtime errors without a substantial increase in the code complexity. Indeed, this would augment the probability of design and coding faults and thus decrease the robustness of the application. Of course, code complexity and robustness are not antonymous if one can avoid or remove design and coding faults in the error handling code.

Language-level *exception handling* mechanisms allow programmers to handle errors with only one test per block of code. In programming languages without exceptions, such as C, programmers have to check for error return codes after each function call. The use of exception handling mechanisms can thus simplify the development of robust programs.

Unfortunately, exception handling is no panacea. First, it is *difficult to be concise*: a large percentage of an application's code is dedicated to exception handling because it needs to take into account all the possible causes of errors and perform various recovery actions. Second, it is *difficult to be right*: although the use of exceptions simplifies the detection of failures, the elegance of language-level exception handling mechanisms might lead to the neglect of recovery issues [Cargill 1994] and produce buggy code. Notably, the premature exit of a method due to an exception might leave an object in an inconsistent state because it does not guarantee atomicity, i.e., "all-or-nothing" semantics. If this inconsistency is not resolved in the error handling code, it might prevent a later recovery and thus, decrease the robustness of the program. Other sources of problems include nested exceptions and concurrency.

In this paper, we argue that atomic block constructs, as provided by software transactional memory, provide effective mechanisms to implement concise and correct exception handling code. They take care of rolling back partial effects on the application state between the beginning of a method execution and the throwing of an exception when necessary, thus freeing the programmer from writing complex and error-prone recovery code. Optional compensations actions can be added to take care of the "external" effects of the partial execution of the atomic block. Also, forward error recovery with the help of alternatives in the form of `do/or else` blocks is supported.

Our main emphasis is on the composability of exception handling code. Experience indicates that good exception handling code often requires that the programmer knows details about other components' failure causes and has a strategy to handle them. This usually breaks information hiding—which is the most powerful mechanism we have to deal with complexity. In this paper, we promote a novel approach where we want to increase the composability of exception handling code by hiding the details of failures from the programmers. The selection of a good recovery strategy is assigned to a recovery manager which will typically use statistical knowledge to select an adequate recovery strategy.

The rest of the paper is organized as follows: Section 2 gives an overview of exceptions and presents their limitations. In Section 3, we propose an alternative approach to implement error handling using atomic blocks and we illustrate it by the means of examples in Section 4. Finally, Section 5 concludes.

2 Exception Handling (and why it is no Panacea)

2.1 Exceptions

Modern programming languages, such as C++ and Java, provide explicit exception handling support (see Figure 1). When a semantic constraint is violated or when some exceptional error condition occurs, an exception is *thrown* (line 23). This causes a non-local transfer of control from the point where the exception occurred to a point, specified by the programmer, where the exception is *caught* (line 10). An exception that is not caught in a method is implicitly propagated to the calling method (line 17). The use of exceptions is a powerful mechanism

```
void main() {
  ...
  try {
    openFile();
    readFile ();
    testFormat();
  }
  catch(SecurityException e) { ... }
  catch(FileNotFoundException e) { ... }
  catch(EOFException e) { ... }
  catch(IOException e) { ... }
  catch(InvalidFormatException e) { ... }
  ...
}
void readFile() {
  while(true) {
    readLine();
    ...
  }
}
void readLine() {
  if(read (...) == −1)
    throw EOFException;
  ...
}
```

Figure 1: Error handling based on exceptions.

that separates functional code from the error handling code and allows a clean path for error propagation. It facilitates the development of applications that are robust and dependable by design. Compare the clean error handling based on exceptions in Figure 1 with the intricate error handling based on return codes in Figure 2.

```
void main() {
  ...
  int err = openFile();
  if (err == NO_SUCH_FILE) { ... }
  else if (err == NOT_ALLOWED) { ... }
  else {          // OK, file open
    err = readFile();
    if (err == READ_ERROR) { ... }
    else {        // OK, read it
      err = testFormat();
      if (err == INVALID_FORMAT) { ... }
      else {    // OK, keep going
        ... } } } }
```

Figure 2: Error handling based on return codes.

2.2 Exception handling code is buggy

Despite its elegance and relative simplicity (as compared to return codes), exception handling still accounts for a large portion of critical software because (1) exceptions introduce significant complexity in the application's control flow, depending on their type and the point where they are thrown, and (2) sophisticated recovery actions must be taken upon exception to preserve state consistency. Cristian [Cristian 1995] notes that often more than two thirds of the code is devoted to detecting and handling errors and exceptions. According to Utas [Utas 2004], three quarters of carrier grade code is dedicated to error and exception handling. Weimar and Necular also observe [Weimer and Necula 2004] that between 3% and 46% of an application's code is reachable from exception handling blocks and this percentage grows with the size and the maturity of the code.

It has also been noted [Cristian 1995] that exception handling code is more likely to contain software bugs (called *exception errors* [Maxion and Olszewski 2000]) than any other part of an application, because in addition to its complexity, exception handling code is rarely exercised and hence not well tested: two third of system failures have been traced to design failures in exception handling code in telephone switching systems. A study [Traupman et al. 2002] of several prominent open source applications (emacs, apache, BerkeleyDB) has shown that, when forcing a system call to fail and return a documented error code, the behavior of the application ranges from correct error handling to dropped requests, halting, crashing, and even database corruption. This highlights that exception handling code does indeed contain severe bugs. Another study [Maxion and Olszewski 2000] has shown that reducing the occurrence of exception handling failures would eradicate a significant proportion of security vulnerabilities. Therefore, removing errors in exception handling code would not only lead to more robust programs, but also more secure programs. One can use static analysis [Weimer and Necula 2004] or dynamic testing tools based on exception injection [Fetzer et al. 2004] to verify the correctness of exception handling code, but such solutions are typically time-expensive, achieve only partial coverage, and are not widely available.

2.3 Why it is hard to get it right

There are several reasons why it is hard to write correct exception handling code. First, exceptions modify the flow of control of the application. A `throw` statement behaves similarly to a `goto` statement—known to be dangerous—but the destination of the jump is not known a priori. Exceptions propagate from callee to caller in a controlled manner, which implicitely disposes of data on the stack, but there is no guarantee that all resources are indeed cleaned up

```
1  class BoundedStack {
2    ...
3    void push(Object o) {
4      if (items < MAX)
5        array[items++] = o;
6      else ...
7    }
8    Object pop() {
9      if (items > 0)
10       return array[--items];
11     else ...
12   }
13   ...
14 }
```

Figure 3: Bounded stack backed by an array with error in garbage collection code.

even in garbage-collected languages such as Java, for keeping a reference to an object is sufficient to prevent it from being freed. Consider the bounded stack implementation of Figure 3: popping elements from the stack does not fully dispose of them, because references are kept in the underlying array (line 10) until they are overwritten. Therefore, calling `pop()` is not sufficient to undo the effect of a previous call to `push()` during exception handling. Avoiding such problems requires discipline from the programmer to make sure that exceptions are caught and handled at the right place, and that cleanup code is correct and complete.

Second, exception handling code must be programmed carefully to ensure that the application is in a consistent state after catching an exception. Recovery is often based on retrying failed methods. Before retrying, the program might first try to correct the runtime error condition to increase the probability of success. However, for a retry to succeed, a failed method also has to leave changed objects in a consistent state. Consistency is ensured if any modification performed by the method prior to the occurrence of the exception is reverted before the exception is propagated to the calling method, i.e., we need atomic (all-or-nothing) semantics. Note that this is typically not the case when using exceptions to simply report on partial results.

Consider the stack implementation of Figure 4. The call to method `push()` on line 28 may throw on exception if the argument is `null` (this can be the case if the user hits `CTRL-D` in the terminal) and the developer has correctly enclosed the call within a `try-catch` block. Unfortunately, the stack is buggy because the number of elements is incremented (line 6) before the construction of the new node that throws the exception (line 7). Consequently, the stack is left in an inconsistent state, with the size updated but the element not inserted. The exception handling code on line 30 cannot take corrective actions because the

```
1  class Stack {
2    int nb_elements;
3    Node head;
4     ...
5    void push(Object o) {
6      nb_elements++;
7      head = new Node(o, head);
8    }
9    Object pop() throws EmptyStackException {
10     if (nb_elements == 0)
11       throw new EmptyStackException();
12      ...
13   }
14    ...
15 }
16 class Node {
17    ...
18   Node(Object o, Node next) {
19     if (o == null)
20       throw new IllegalArgumentException();
21      ...
22   }
23 }
24  ...
25 Stack stack = new Stack();
26 try {
27   String s = in.readLine();
28   stack.push(s);
29 } catch(Exception e) {
30   // Handle exception (how?)
31 }
```

Figure 4: Stack (version 1).

internal state of the stack is corrupted. Lines 6 and 7 should be swapped to solve this problem.

Consider now the corrected version of the stack in Figure 5 where lines 4 and 5 are in the correct order. There is a new method, `pushAll()`, that adds several elements to the stack at once. The method call on line 18 pushes two elements on the stack and may throw an exception if any of them is `null`. Atomicity is violated if the second element is `null` because, even though the stack is consistent, the effect of the `pushAll()` is only partial: the first element has been pushed, but not the second, breaking all-or-nothing semantics. Again, the exception handling code on line 20 cannot take corrective actions.

Typically, one would use *compensation actions* to restore a consistent state, but these actions must be taken at the right place (e.g., in method `push()` of Figure 4 and in method `pushAll()` of Figure 5). And even then, atomic behavior is hard to implement. Consider the method in Figure 6 that moves an item from one stack to another. The code includes compensation actions in case an exception is thrown during the move, but it does not take into account the fact that the call to `push()` on line 8 might as well throw an exception. As a general rule, exception handling code should not throw exceptions because recursive

```
class Stack {
  ...
  void push(Object o) {
    head = new Node(o, head);
    nb_elements++;
  }
  void pushAll(Object[] a) {
    for(Object o : a) push(o);
  }
  ...
}
...
Stack stack = new Stack();
try {
  Object[] a = new Object[2];
  a[0] = in.readLine();
  a[1] = in.readLine();
  stack.pushAll(a);
} catch(Exception e) {
  // Handle error (how?)
}
```

Figure 5: Stack (version 2).

```
void move(Stack a, Stack b) {
  Object item = null;
  try {
    item = a.pop();     // Might throw exception
    b.push(item);       // Might throw exception
  } catch (Exception e) {
    if (item != null)  // Compensation
      a.push(item);
    throw e;
  }
}
```

Figure 6: Moving an object from one stack to another.

exception handling is extremely complex to deal with. A programmer has to consider all possible places where an exception might be thrown, and has to make sure that none of these exceptions can cause a state inconsistency.

Figure 7 illustrates the difficulty to compose exception handling code. The `swap()` method exchanges the top items of two stacks. It necessitates four calls to `pop()` and `push()`, each of which might throw an exception. Depending on which method fails, the exception handling code must take different compensation actions and the resulting code is intricate and hard to follow. Again, it is not trivial to determine the actions to take if an exception occurs in the error handling code on lines 12 to 14: should we retry the compensation action or the original action in the `try` block in the hope that one of them succeeds, or simply abort? The complexity of nested exception handling illustrates why compensation actions should not throw exceptions.

Finally, consider that concurrency might interfere with compensation actions.

```
void swap(Stack a, Stack b) {
  Object item1 = null;
  Object item2 = null;
  boolean a_pushed = false;
  try {
    item1 = a.pop();  // Might throw exception
    item2 = b.pop();  // Might throw exception
    a.push(item2);    // Might throw exception
    a_pushed = true;  // Item pushed on a
    b.push(item1);    // Might throw exception
  } catch (Exception e) {
    if (a_pushed)        try { a.pop();   }        catch {...}
    if (item2 != null) try { b.push(item2); } catch {...}
    if (item1 != null) try { a.push(item1); } catch {...}
    throw e;
  }
}
```

Figure 7: Exchanging the topmost objects of two stacks.

In Figure 7, consistency will be violated if another thread modifies the top elements of one of the stacks between the failed action and the compensation code, even if the latter does not throw exceptions. Therefore, the `catch` block should be properly synchronized, similar to the `try` block, to avoid inconsistency resulting from concurrent accesses.

```
1  void move_mt(Stack a, Stack b) {
2    Object item = null;
3    mutex.lock();
4    try {
5      item = a.pop();     // Might throw exception
6      b.push(item);       // Might throw exception
7    } catch (Exception e) {
8      if (item != null)  // Compensation
9        a.push();
10     throw e;
11   }
12   mutex.unlock();
13 }
```

Figure 8: Multi-threaded version of the move method of Figure 6.

In general, concurrency introduces subtle problems that are already complex to manage in regular code, and become much harder when introducing exception handling that modifies the flow of control. A classical example is that of a lock being kept because an exception occurs before it is released. Consider the multi-threaded `move_mt()` method in Figure 8. A lock correctly synchronizes both the `try` and `catch` blocks, but it is not released when an exception is thrown from the `catch` block (on line 9 or 10). In that case, the lock should be released in a `finally` block.

In summary, the complexity of exception handling code results from several factors: the modifications of the control flow induced by exceptions; the difficulty of preventing resource leaks and inconsistent state after a failure during exception propagation; the problems with nested exceptions and the complexity to compose error handling routines; concurrency issues that interfere with compensation actions; and the difficulty to test exception handling code because failures are hard to reproduce.

3 Implementing Atomic Exception Handling

Language-level transactions have become popular recently. They allow programmers to perform arbitrary operations on in-memory data with transactional semantics. Notably, updates are atomic and are rolled back in case the transaction cannot commit. We propose to leverage transactional memory to implement atomic exception handling.

3.1 Transactional Memory

The concept of transaction has recently been proposed as a lightweight and safe mechanism to manage concurrent accesses to shared (in-memory) data in multi-threaded applications. *Software transactional memory* (STM) [Shavit and Touitou 1995] provides programmers with constructs to delimit transactional operations and implicitly takes care of the correctness of concurrent accesses to shared data. Transactions typically execute in a loop: if the transaction cannot commit, e.g., because it conflits with another transaction, it aborts and restarts. This process is generally automated and hidden behind a higher-level construct of *atomic block*. Conflict resolution is often taken care of by a configurable module, the *contention manager*, that defines the strategy for dealing with two conflicting transactions.

STM has been an active field of research over the last few years, e.g., [Harris and Fraser 2003, Herlihy et al. 2003, Herlihy 2005, Scherer III and Scott 2005, Marathe et al. 2005, Cole and Herlihy 2005, Guerraoui et al. 2005b, Guerraoui et al. 2005a]. It provides the programmer with a high-level construct—simple to use, familiar, efficient, and safe—to delimit the statements of its application that need to execute in isolation. In the following, we propose to use variants of STM atomic blocks to implement atomic exception handling. Most related to our work is that of Harris [Harris 2005]. Harris proposes to commit by default all changes when an exception is thrown. We argue that by default an exception should result in a rollback and extend the approach of Harris in several ways. Our approach also also has commonalities with coordinated atomic actions (CAAs) [Xu et al. 1995], a framework for achieving fault tolerance under cooperative and competitive concurrency, although there are notable difference in the focus and design. The scope of CAAs as wider as they supports distributed systems and deal with hardware faults, they require the use of explicit APIs, and they rely on full-fledge ACID transactions.

3.2 Exception Handling using Atomic Blocks

```
// Guarded atomic block
atomic (C) { // Wait until C is true
  S;            // Sequential code.
}

// Atomic block with retry
atomic {  // Wait or throws exception
  if (!C)
    retry; // Retry until C holds or alternative path found
  S;        // Sequential code.
}
```

Figure 9: Two variants of atomic blocks [Harris and Fraser 2003, Harris et al. 2005].

Harris proposes language support for atomic blocks using a new `atomic` keyword [Harris and Fraser 2003] (see Figure 9). Atomic blocks had originally a condition that must be valid for the program to execute the atomic block. In case no guard is necessary, the `atomic` keyword takes no argument. The block contains a sequence of statements and the block executes atomically and in isolation. A newer variant of an atomic block [Harris et al. 2005] uses a `retry` statement instead of a condition. Whenever the execution reaches this statement the atomic block is retried. Alternatives can be specified with the help of `do/or else` blocks. The execution of a `retry` within the `do` block will lead to the execution of the `or else` block before the enclosing atomic block is retried.

The semantics of an atomic block when an exception is thrown from within an atomic block is not obvious. On one hand, one could argue that an exception terminates the execution of an atomic block and hence, all changes done so far within the atomic block should be committed. On the other hand, one could argue that an exception indicates that a block is only partially executed and hence, the execution of the atomic block should be aborted by the exception, i.e., all changes done within the atomic block until the exception is thrown should be rolled back. We argue that the latter, which we call failure atomicity, should be the default behavior: as we have seen in the previous examples, some of the complexity of exception handling is caused by having to either deal with partial changes performed by lower level functions or having to manually roll back partial changes. Hence, if we can automatically roll back partial changes, one should be able to reduce the complexity of exception handling.

Ensuring Failure Atomicity

Clearly, atomic blocks have the potential to alleviate some of the problems of `try/catch` constructs. Indeed, upon failure, an STM can take care of rolling back the modifications performed in memory by the statements inside the atomic block. This frees the developer from the complex and error-prone task of identifying which ones of the statements in the `try` block have been executed and writing compensation code to undo their effect.

```
void swap(Stack a, Stack b) {
  atomic {
    Object item1 = a.pop();  // Might throw exception
    Object item2 = b.pop();  // Might throw exception
    a.push(item2);           // Might throw exception
    b.push(item1);           // Might throw exception
  }
}
```

Figure 10: Exchanging the top most objects of two stacks using an atomic block.

As a first step, we can simply use atomic blocks as as a substitute to `try/catch` constructs. The `swap()` method from Figure 7 that required to compose exception handling code can be advantageously replaced by the simpler and cleaner implementation shown in Figure 10. If any of the methods called from within the atomic block throws an exception, the modifications performed since the beginning of the block's execution are automatically undone without necessitating complex compensation code. In particular, an STM takes care of disposing of objects allocated on the heap without risks of resource leakage. It also allows several threads to access shared data in isolation without necessitating locks.

Atomic blocks alone are however not sufficient to offer a complete solution for exception handling. For instance, the general pattern employed for ensuring isolation with STMs is to transparently abort and retry a transaction if it cannot commit because of a concurrency conflict (race condition). In contrast, if the code executed in the context of a transaction explicitely throws an exception, e.g., because an overdraft occurs when transferring money between two accounts, it is not clear what should be the proper behavior:

– Should we automatically abort the transaction and retry? Unless the problem is due to a race condition, the same exception is in this example likely to be thrown again. We might also want to take corrective actions before retrying.

- Should we abort the transaction and propagate the exception? The main issue is that an exception is thrown but its cause has been rolled back. Further, this imposes strong restrictions on exceptions because they cannot refer to data that has been allocated or modified in the atomic block (and rolled back on exit).
- Should we commit the partial changes and propagate the exception? This corresponds to the traditional behavior expected by the programmer for `try` blocks, but it conflicts with the atomicity (all-or-nothing) property expected from a transaction. However, this can be useful, for example, in situations in which one can provide some graceful degradation by providing some weaker semantics.

As we have previously argued, the first strategy should be preferred most of the times as is helps writing more robust exception handling code. Yet, other behaviors are useful in some contexts and should be supported as well. Essentially, we need to know whether the execution of the atomic block has completed succesfully; in case of a failure, we should be able to learn the cause of the problem and retry the same block or an alternative execution path; finally, we must have a way to exit an atomic block either discarding or commiting partial changes.

We must also be able to intervene in the roll back process to handle "external" actions (e.g., writing to a file), which are not automatically reverted by STM. Conversely, we should be able to let some information (e.g., details about error conditions) leak out of atomic blocks upon failure and, hence, prevent automatic undo on specific data items.

These various aspects are discussed in the rest of this section.

Atomic Block Syntax and Semantics

We propose using an extended form of atomic blocks for improving correctness of exception handling. The general structure is shown in Figure 11. An atomic block executes in the context of a transaction. When starting in the context of another atomic block (line 4), it maps to a nested transaction: the inner atomic block may fail without aborting the outer block, while a failure of the latter will roll back the former.

If an atomic block cannot commit because of a concurrency conflict resulting from optimistic scheduling in the underlying STM, then it will restart *automatically* (after rolling back changes and possibly executing compensation actions). If the transaction aborts due to an exception being thrown in the atomic block via `leave` (see line 8), partial changes are rolled back by the underlying STM and execution continues in the optional `on failure` block (line 10), which runs outside of the scope of the aborted transaction but inside that of the enclosing

```
1  atomic {
2    // Sequential code executed atomically
3    ...
4    atomic {
5      // Atomic blocks may be nested
6      ...
7      if (error)
8        leave;  // Roll back and jump to on failure block
9      ...
10   } on failure {
11     // Executed if nested block fails
12     if (canCorrectErrorCondition())
13       retry;  // Retry nested block
14   }
15   ...
16 } on failure {
17   // Executed if outer block fails or nested block leaves
18 }
```

Figure 11: Atomic blocks for exception handling.

transaction, if any. If there is no `on failure` block, the exception propagates to the enclosing atomic block (as a `next`—see below). Such propagation also occurs if an exception is thrown from within an `on failure` block.

In the `on failure` block, one will typically try to fix runtime error conditions that prevented the atomic block from succeeding, possibly modifying the environment, and retry its excecution. To that end, one can use the `retry` keyword (line 13) to restart the associated atomic block.[1]

```
void swap(Stack a, Stack b) {
  atomic {
    Object item1 = a.pop();  // Might throw exception
    Object item2 = b.pop();  // Might throw exception
    a.push(item2);           // Might throw exception
    b.push(item1);           // Might throw exception
  } on failure {
    a.rejuvenate();
    b.rejuvenate();
    retry;
  }
}
```

Figure 12: Exchanging the topmost objects of two stacks using an atomic block and error handling code.

An implementation of the previously discussed `swap()` method using atomic

[1] Note the semantics of `retry` is different to that of [Harris et al. 2005]: the latter corresponds to our `next`. It also differs from the semantics of `retry` in Eiffel in that it implies a rollback and it can appear at any point within an atomic block or an `on failure` block.

blocks and error handling code is shown in Figure 12. If the exchange fails, we try to rejuvenate each of the stack objects (e.g., to correct their internal state that might be corrupted) before retrying (lines 8 and 9).

Specifying Alternative Execution Paths

```
1  atomic {
2    // Do/or else blocks must be in atomic blocks
3    ...
4    do {
5      // First alternative (a)
6      ...
7      if(error()) // Error cannot be fixed in (a)
8        next;    // Try next alternative
9      ...
10   } or else {
11     // Second alternative (b)
12     ...
13     do {
14       // Do/or else blocks may be nested (b.a)
15       ...
16       if (fatalError()) // Cannot deal with this error
17         // remaining alternatives do not fix this either
18         leave;
19         ...
20     } or else {
21       // (b.b)
22       ...
23         if (localFix()) // Might be fixed by retrying (b.b)
24           retry;
25       ...
26
27     }
28   } or else {
29     // Third alternative (c)
30     ...
31   }
32   ...
33 } on failure {
34   // Executed if all alternatives fail
35 }
```

Figure 13: Specifying alternative execution paths.

Failure blocks allow us to take different recovery actions. However, they do not permit explicitly to take a different execution path in the atomic block after a failure. To support this behavior, we introduce another language construct, `do/or else`, also proposed by Tim Harris *et al.* [Harris et al. 2005], that specifies alternatives to try before reporting a failure. The structure of this construct is shown in Figure 13.

A `do/or else` block can only occur in the context of an atomic block. The `do` clause contains the code of the default execution path (line 4); subsequent `or else` clauses describe alternative execution paths (lines 10 and 28). One can have several `do/or else` block in the same atomic block, and it is even possible to nest them (line 13). All alternatives are explored sequentially before giving up and executing the `on failure` block. In Figure 13, assuming that failures systematically occur during execution of the `do/or else` blocks, the following sequence of alternatives will be tried: the first alternative of the outer block (line 4); the second alternative of the outer block (line 10) with the first path in the inner block (line 13); the second alternative of the outer block with the second path in the inner block (line 20); the third alternative of the outer block (line 28).

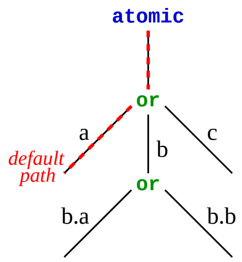

Figure 14: Execution tree representing the alternatives of Figure 13.

This process can be visualized as an execution tree (see Figure 14) in which children correspond to nested blocks and sibling branches to alternatives, and a complete successful execution maps to a root-to-leaf path. The leftmost path corresponds to the default execution of the atomic block.

If an atomic block fails by the abort of a nested atomic block while executing one of the alternatives of a `do/or else` block, it *automatically* executes the next alternative. The programmer can control the selection of the next alternative explicitly with the use of the keywords `next`, `retry`, and `leave`. Keywords `retry` and `leave` are introduced for performance reason only. Ideally, the system would chose automatically what alternative to execute next (see Section 5) and the programmer would only need to use one keyword, e.g., `next`.

One can use the explicit `next` keyword to move to the next alternative. For example, the `next` on line 8 will result in alternative (a) to be aborted and instead alternative (b) is tried. In other words, a `next` tells the system that the current alternative is not succeeding and the system should try the next one.

If the execution hits a `retry`, the programmer tells the system that a retry of the same block might be successful. For example, the `retry` on line 24 indicates that the system should retry block (b.b). An implementation has the choice to either (1) rollback the outermost atomic block and execute again path (b, b.b) or (2) it could just rollback the effects of (b.b) and retry (b.b).

A `leave` tells the system that the current alternative cannot succeed and also all successive alternatives will also fail. In other words, this is similar to a `next` but it tells the system that it can skip executing the remaining alternatives and immediately jump to the `on failure` block if it exists (and otherwise, just abort the atomic block). For example, a `leave` on line 18 results in an abort of alternative (b.a) and the `on failure` block on line 33 is tried next.

```
void swap(Stack a, Stack b) {
  atomic {
    do {
      Object item1 = a.pop();  // Might throw exception
      Object item2 = b.pop();  // Might throw exception
      a.push(item2);           // Might throw exception
      b.push(item1);           // Might throw exception
    } or else {
      Object value = a.top().getValue();
      a.top().setValue(b.top.getValue());
      b.top().setValue(value);
    }
  } on failure {
    a.rejuvenate();
    b.rejuvenate();
    retry;
  }
}
```

Figure 15: Exchanging the topmost objects of two stacks using an atomic block and alternative execution paths.

The `swap()` method can also be implemented using alternative execution paths, as shown in Figure 15. If the exchange fails in the `do` block, it is automatically retried by executing the `or else` clause that uses another strategy. If both fail, we rejuvenate the stack objects and retry the complete atomic block (starting with the default execution path).

Semantics of On Failure

There are subtle differences between the semantics of the `on failure` and `do/or else` constructs. Both specify alternatives that are executed when the primary alternative fails. However, the `on failure` block is executed outside of the scope of the transactions associated with the enclosing atomic block: its objective is to make changes visible in the next retry of the atomic block. One can view `on failure` as a special type of `or else` block.

For example, one reasonable approach for recovery is to perform changes to the environment before performing a retry (e.g., see [Qin et al. 2005]). Figure 16 depicts a wrapper for methods that tries to change the environment before a retry

```
Object atomicWrapper(Object o, Method m, Object[] args) {
  atomic {
    Object ret = m.invoke(o, args);
    if (checkState(o))   // Consistent state?
      next;              // No: failure
  } on failure {
    if (changeEnv())     // Try changing environment
      retry;             // Worked: retry
  }
}
```

Figure 16: On failure semantics.

is performed. Since the environment changes should be visible in the next retry, the semantics of the `on failure` block can be emulated with nested `do/or else` blocks as depicted in Figure 17.

```
Object atomicWrapper(Object o, Method m, Object[] args) {
  atomic {
    do {
      Object ret = m.invoke(o, args);
      if (checkState(o))   // Consistent state?
        next;              // No: failure
    } or else {
      if (changeEnv()) {
        Object ret = m.invoke(o, args);
        if (checkState(o))   // Consistent state?
          next;              // No: failure
      }
    }
  }
}
```

Figure 17: On failure semantics.

Controlling Rollback

Often, one needs to combine implicit rollback performed by STM with explicit compensation actions, either when the code performs external actions (e.g., I/O) or to improve performance (e.g., when some changes do not impact application consistency and do not have to be rolled back).

To indicate that some fields are neither saved nor restored by the underlying STM, we use the `transient` keyword.[2]

An example of a controlled rollback to handle external actions is shown in Figure 18. This class allows us to read character from the terminal and put

[2] The `transient` keyword is used in Java to indicate that a field should not be saved nor restored by the standard serialization mechanisms.

```
class CharacterInput {
  transient char[] buf = new char[N];  // Kept on abort!
  transient int nb = 0;                // Kept on abort!
  int idx = 0;                         // Rolled back

  char getCharacter() {
    while (idx >= nb)
      fillBuffer ();
    return buf[idx++ % N];
  }

  void fillBuffer () {
    int err = System.in.read(buf, nb % N, 1);
    if (err <= 0)      // Couldn't read at this time
      next;            // Use next alternative, if any
    nb++;
  }
}
```

Figure 18: Using transient fields to handle external actions.

them back in a buffer upon rollback for being subsequently read again. We use transient fields to manage the buffer. For simplicity, this code assumes that there are at most N calls to `getCharacter()` within an atomic block.

```
CharacterInput terminal;
Stack s;
...
atomic {
  // Read two characters and push them on stack
  s.push(terminal. getCharacter());
  s.push(terminal. getCharacter());
  ...
} on failure {
  // State of stacks is automatically reverted and
  // characters are returned to the input buffer, but
  // they are not returned to the operating system
  ...
}
```

Figure 19: Reading characters from the terminal using a class that handles external actions.

Our intention is that the use of `transient` is limited to libraries written by experts who are aware of the exact semantics of the underlying transactional memory. Most programmers should restrict themselves to the use of libraries that rollback external actions on abort of an atomic block. For example, Figure 19 illustrates how one can easily compose atomic blocks with the help such library methods. We use the previously defined class `CharacterInput` to read two characters within an atomic block. Whether the character reading method uses an atomic block internally does not need to be known by the callee. In this case, neither `getCharacter()` nor `fillBuffer()` need to be executed in a separate atomic block. Also, they can be called from within atomic blocks but also from code outside atomic blocks (but the latter would not support automatic rollback). Our goal is it to effectively combine atomic (and also non-atomic) blocks by nesting them in an enclosing atomic block. This composability feature allows us to perform complex operations on several objects without having to know how each of them implements error handling.

4 Examples

In this section, we present some examples on how one can use atomic blocks to recover from errors. To do so, we show how one can implement graceful degradation, selective retries, and recovery blocks using atomic blocks.

4.1 Graceful Degradation

Graceful degradation means that an application can provide some desired semantics (with possibly degraded quality of service) even in case that some components exhibit failures. Consider, for example, an application that writes its log messages to disk (see Figure 20). In case the disk is full, the application should not abort only because the application cannot write to the log. Instead, log messages might be written to the console. In case this fails too, e.g., because the console output is redirected to the same full disk, log messages should be ignored, i.e., instead of aborting the application or dropping requests.

Figure 20 shows that function `log()` uses function `writeString()` to write the log message either to disk or to console, i.e., `System.err`. Note that function `writeString()` can fail for various reasons but at the level of function `log()` we do not care why: we assume that `writeString()` has already done all that is needed to ensure that it succeeds whenever possible, e.g., attempted to free some disk space. Function `writeString()` fails by issuing a `leave`. Hence, one can use an atomic block with a nested `do/or else` block to switch to a degraded service in case writing to the log fails.

4.2 Selective Retries

With *selective retries* we refer to the retry of an atomic block only in cases where there is a chance that a retry succeeds, i.e., a transient failure prevented the atomic block to commit. For example, a failure caused by too few available resources might succeed on retry—if there are fluctuations in resource usage. Of course, for some other types of failures a retry might not have any chance of success, e.g., retrying a function with the same wrong arguments.

```
void log(String msg) {
  static PrintStream appLog = ...;
  atomic {
    do {          // Write to application log
      writeString(appLog, msg);
    } or else {   // If it fails , write to console
      writeString(System.err, msg);
    } or else {   // Otherwise, do not log
      ;
    }
  }
}
```

Figure 20: Graceful degradation: log messages are written to the application log when possible. If that fails, messages are written to the console. If that fails too, the messages are just ignored.

Figure 21 shows a selective retry when writing a string to a stream fails. In this example, function `writeBytes()` returns the number of bytes written. If the return value is different from the number of bytes that were requested to be written, an error has occurred. Function `getError()` returns the reason why not all bytes were written and function `canRetry()` uses this information to decide if a retry is reasonable. If yes, `retry` automatically rolls back all changes, including those of `writeString()`, and restarts the atomic block. If a retry is not considered sensible, the atomic block is aborted via `leave` and recovery needs to be attempted by the caller of function `writeString()`.

```
void writeString(PrintStream out, String msg) {
  atomic {
    byte[] bytes = msg.getBytes();
    if (writeBytes(bytes) != bytes.length) {
      // Could not write all bytes: check error
      int err = getError();
      if (canRetry(err))  // Temporary problem?
        retry;            // Yes: retry
      else
        next;             // No: give up
    }
  }
}
```

Figure 21: Selective retries: if there is a chance that a function can succeed during retry, we restart the atomic block. Otherwise, we leave it and let the enclosing block perform recovery at a higher level.

4.3 Repair and Retry

Selective retries are good to mask transient failures like temporary resource shortages. However, in some situations, e.g., when there are longer lasting resource shortages like a full disk, one would like to perform some repair before retrying an atomic block. The kind of repair that leads to successful execution of an atomic block would typically depend on the root cause of the atomic block's failure.

Figure 22 shows that one could use a classification of the cause of an error to decide what kind of environment changes might be needed. Note that the environment needs to be modified within the `on failure` block to make sure that changes are not rolled back before the next retry. We will show in Section 5 how this code can be simplified by delegating the task of identifying the cause of an error and the selection of the appropriate recovery strategy.

```
void writeString(PrintStream out, String msg) {
  transient int env = 0;
  atomic {
    byte[] bytes = msg.getBytes();
    if (writeBytes(bytes) != bytes.length) {
      // Could not write all bytes: check error
      int err = getError();
      // Fixable by a local change to the environment?
      env = needsLocalEnvChange(err);
      if (canRetry(err))  // Temporary problem?
        retry;            // Yes: retry
      else
        next;             // No: give up
    }
  } on failure {
    if (changeEnv(env))  // Try changing environment
      retry;             // Worked: retry
  } // otherwise abort atomic block
}
```

Figure 22: Repair and retry recovery strategy: before the atomic block is retried, one tries to increase the likelihood of success by changing the environment.

4.4 Recovery Block

A recovery block [Randell 1975] uses software diversity to mask software bugs. A recovery block has a post-condition (the *acceptance test*) that needs to be satisfied. Alternative implementation are tried in sequence. If an alternative does not satisfy the post-condition, the state changes are rolled back and the next alternative is tried. One can use atomic blocks to implement a recovery block in a straightforward manner. Figure 23 shows an example where we use alternative sort implementations to satisfy the post/condition.

```
sort(Object[] array) {
  atomic {
    do {
      quickSort(array);      // Fast sort
      if (!sorted(array))    // Check if sorted
        next;                // No: try next alternative
    } or else {
      mergeSort(array);      // Fast sort
      if (!sorted(array))    // Check if sorted
        next;                // No: try next alternative
    } or else {
      bubbleSort(array);     // Slower but correct
      if (!sorted(array))    // Check if sorted
        next;                // No: try next alternative
    }
  }
}
```

Figure 23: Recovery block: the atomic block terminates with success if one of the three sort functions returns normally and the postcondition `sorted()` is satisfied. The alternative sorting functions are executed in sequence and if none succeeds, the function returns via a `leave`.

```
sort(Object[] array) {
  atomic {
    do {
      quickSort(array);      // Fast sort
      if (!sorted(array))    // Check if sorted
        next;
    } or else {
      mergeSort(array);      // Fast sort
      if (!sorted(array))    // Check if sorted
        next;
    } or else {
      bubbleSort(array);       // Slower but correct
      // Trust that bubble sort is correct
    }
  }
}
```

Figure 24: Recovery block variant: we assume that `bubbleSort` is a very mature function but not very fast while the other functions, i.e., `quickSort` and `mergeSort` and `sorted`, are less mature. Hence, we could drop the post-condition for `bubbleSort` to tolerate possible bugs in `sorted`.

From a syntactical perspective, a recovery block has only one post-condition while the use of an atomic block would need to specify the post-condition for all alternatives. One could simplify the syntax by associating an atomic block with post-conditions. In general, it might be a good idea to specify contracts for atomic blocks, i.e., pre- and post-conditions and invariants. Note, however, that the explicit specification of post-conditions has also advantages. For example, experience indicates that pre- and postconditions are buggy too. Hence, one could use redundant assertions to be able to tolerate bugs in assertions. For example, in Figure 24 we drop the post-condition for one of the alternatives (`bubbleSort()`) in case the likelihood that `bubbleSort()` fails is much smaller than the likelihood that `sorted()` fails.

5 Concluding Remarks

We proposed the use of atomic blocks for error and exception handling. One of the main advantages of atomic blocks is that state changes are rolled back automatically. External changes are rolled back via the use of special "atomic" library functions. In this way, one can avoid common programming errors that result in inconsistent states after an exception was thrown. Atomic blocks can be nested and, in particular, error handling code itself runs safely in an atomic block. In some sense, error handling code is just an alternative (similar to the alternative implementations in the recovery block concept) that is executed if the default alternative fails. Therefore, we not only avoid synchronization issues during recovery in multi-threaded applications but we can also gracefully handle errors during the execution of error handling code itself. By using a combination of nested blocks and alternatives, we can effectively get the benefits of both backward and forward error recovery.

The second major advantage of atomic blocks for error handling is the increased composability. Traditional exception handling usually exposes some of the internals of components to external calls through the exception objects returned. Our approach is simple: atomic blocks can either request to retry the atomic block or to leave the atomic block (i.e., error recovery should be attempted on a higher level). They do not carry explicit information about the cause of a failure, unlike exceptions. A retry corresponds to the traditional resumption semantics of exceptions while a leave corresponds to termination semantics of exceptions. This restricted interface might however lead to situations in which programmers want to optimize the recovery by passing information from within an aborted atomic block via, e.g., transient variables as we demonstrated in Figure 22. While this might be acceptable when done locally and by expert programmers, such optimizations should in general be avoided. Instead, the decision of what alternative to execute next should be performed by a *recovery manager*. The recovery manager decides based on statistical and/or analytical data which alternative should be executed. In other words, recovery decisions that are difficult to decide based on information at design time are delegate to the recovery manager that can use up-to-date runtime statistics to select the recovery action that will most likely succeed.

The interface of the recovery manager should be as simple as possible. In our proposal, the recovery manager implements a function `select()` that picks one

```
...
RecoveryManager rm;
...
void writeString(PrintStream out, String msg) {
  atomic {
    byte[] bytes = msg.getBytes();
    writeBytes_a(bytes);        // Atomic method
  } on failure {
    int choice = rm.select({NO_RETRY, FREE_MEM, FREE_FD...});
    if (choice != NO_RETRY) { // Change environment?
      changeEnv(choice);   // Note: might issue leave
      retry;               // Retry with new environment
    } // otherwise abort
  }
}
```

Figure 25: Repair and retry using a recovery manager to select the next recovery alternative.

```
1  ...
2  RecoveryManager rm;
3  ...
4
5  void writeString(PrintStream out, String msg) {
6    atomic {
7        byte[] bytes = msg.getBytes();
8        writeBytes_a(bytes);  // Atomic method
9    } on failure {
10     int choice = rm.select({false, true});
11     if (choice)              // Should we retry?
12        retry;                // Yes: retry atomic block
13   }
14 }
```

Figure 26: Semantics of atomic blocks: each atomic block has an implicit call to select. Lines 9 to 13 are implicitly added to an atomic block.

of a given set of alternatives. For example, the code from Figure 22 can in this way be simplified as depicted in Figure 25. Since retries can be performed on different levels of abstractions, nested atomic blocks can result in an exponential increase of retries (i.e., with the nested level). Similarly, if retries of a certain function `f()` have not succeeded recently, e.g., because none of the needed resources are freeing up, retrying `f()` might not make sense. Therefore, the decision how many times an atomic block is retried should be delegated to the recovery manager.

Note that, if the programmer needs to decide explicitly if a block should be retried, s/he needs to gather information about the cause of an error (e.g., see Figure 21). For an increased information hiding, the recovery manager should instead decide if a retry or a leave should be performed. This would simplify the code because we could remove the `on failure` block altogether from Figure 26 (i.e., one could discard lines 9 to 13).

While the advantage for the composability with respect to error handling should be obvious, how to implement a recovery manager in an efficient and general way is less obvious. We are currently working on evaluating several design alternatives for recovery managers.

References

[Cargill 1994] Cargill, T. (1994). Exception handling: A false sense of security. *C++ Report*, 6(9).

[Cole and Herlihy 2005] Cole, C. and Herlihy, M. (2005). Snapshots and software transactional memory. *Science of Computer Programming*. To appear.

[Cristian 1995] Cristian, F. (1995). Exception handling and tolerance of software faults. In Lyu, M., editor, *Software Fault Tolerance*, pages 81–107. Wiley.

[Fetzer et al. 2004] Fetzer, C., Felber, P., and Högstedt, K. (2004). Automatic detection and masking of non-atomic exception handling. *IEEE Transactions on Software Engineering*, 30(8):547–560.

[Guerraoui et al. 2005a] Guerraoui, R., Herlihy, M., Kapalka, M., and Pochon, B. (2005a). Robust contention management in software transactional memory. In *Proceedings of SCOOL*.

[Guerraoui et al. 2005b] Guerraoui, R., Herlihy, M., and Pochon, S. (2005b). Toward a theory of transactional contention managers. In *Proceedings of PODC*.

[Harris 2005] Harris, T. (2005). Exceptions and side-effects in atomic blocks. *Sci. Comput. Program.*, 58(3):325–343.

[Harris and Fraser 2003] Harris, T. and Fraser, K. (2003). Language support for lightweight transactions. In *Proceedings of OOPSLA*, pages 388–402.

[Harris et al. 2005] Harris, T., Herlihy, M., Marlow, S., and Peyton-Jones, S. (2005). Composable memory transactions. In *Proceedings of PPOPP*.

[Herlihy 2005] Herlihy, M. (2005). The transactional manifesto: software engineering and non-blocking synchronization. In *Proceedings of PLDI*.

[Herlihy et al. 2003] Herlihy, M., Luchangco, V., Moir, M., and Scherer III, W. (2003). Software transactional memory for dynamic-sized data structures. In *Proceedings of PODC*, pages 92–101.

[Marathe et al. 2005] Marathe, V. J., III, W. N. S., and Scott, M. L. (2005). Adaptive software transactional memory. In *Proceedings of DISC*, pages 354–368.

[Maxion and Olszewski 2000] Maxion, R. and Olszewski, R. (2000). Eliminating exception handling errors with dependability cases: a comparative, empirical study. *IEEE Transactions on Software Engineering*, 26(9):888 – 906.

[Qin et al. 2005] Qin, F., Tucek, J., Sundaresan, J., and Zhou, Y. (2005). Rx: treating bugs as allergies—a safe method to survive software failures. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 235–248, New York, NY, USA. ACM Press.

[Randell 1975] Randell, B. (1975). System structure for software fault tolerance. In *Proceedings of the international conference on Reliable software*, pages 437–449, New York, NY, USA. ACM Press.

[Scherer III and Scott 2005] Scherer III, W. and Scott, M. (2005). Advanced contention management for dynamic software transactional memory. In *Proceedings of PODC*, pages 240–248.

[Shavit and Touitou 1995] Shavit, N. and Touitou, D. (1995). Software transactional memory. In *Proceedings of PODC*.

[Traupman et al. 2002] Traupman, J., Broadwell, P., and Sastry, N. (2002). Fig: A prototype tool for online verification of recovery mechanism.

[Utas 2004] Utas, G. (2004). *Robust Communications Software: Extreme Availability, Reliability and Scalability for Carrier-Grade Systems*. Wiley.

[Weimer and Necula 2004] Weimer, W. and Necula, G. C. (2004). Finding and preventing run-time error handling mistakes. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 419–431, New York, NY, USA. ACM Press.

[Xu et al. 1995] Xu, J., Randell, B., Romanovsky, A. B., Rubira, C. M. F., Stroud, R. J., and Wu, Z. (1995). Fault tolerance in concurrent object-oriented software through coordinated error recovery. In *Symposium on Fault-Tolerant Computing*, pages 499–508.