Dependency Safety for Java – Implementing Failboxes

Dragan Bošnački [1]
d.bosnacki@tue.nl

Mark van den Brand [1]
m.g.j.v.d.brand@tue.nl

Philippe Denissen [1]
p.e.j.g.denissen@student.tue.nl

Cornelis Huizing [1]
kees.huizing@gmail.com

Bart Jacobs [2]
bart.jacobs@cs.kuleuven.be

Ruurd Kuiper [1]
r.kuiper@tue.nl

Anton Wijs [1]
A.J.Wijs@tue.nl

Maciej Wiłkowski [1]
m.wilkowski@student.tue.nl

Dan Zhang [1]
D.Zhang@tue.nl

[1] Eindhoven University of Technology
NL-5600 MB Eindhoven, The Netherlands

[2] Katholieke Universiteit Leuven
B-3001 Leuven, Belgium

ABSTRACT

Exception mechanisms help to ensure that a program satisfies the important robustness criterion of dependency safety: if an operation fails, no code that depends on the operation's successful completion is executed anymore nor will wait for the completion. However, the exception handling mechanisms available in languages like Java do not provide a structured way to achieve dependency safety. The language extension *failbox* provides dependency safety for Java in a compositional manner. So far, there only exists an implementation of failbox in Scala. It requires the assumption of absence of asynchronous exceptions inside the failbox code. In this paper, we are the first to provide an implementation without the above mentioned assumption, this time in Java. First, we present and discuss a direct reimplementation of failbox in Java that is still restricted. Then, we show that using uncaught exception handlers the earlier assumption can be essentially weakened to only concern code before setting the thread handler. Finally, we provide an implementation using the Java native interface that completely removes the assumption.

CCS Concepts

•Software and its engineering → Software reliability; Software safety; Concurrent programming languages; Concurrent programming structures;

Keywords

exception handling; concurrency; failboxes; Java

1. INTRODUCTION

During execution of a program some operations may fail. A way to mitigate the effect of such failures is to ensure that the program has the property of *dependency safety* [10]: if an operation fails, no thread that depends on the operation's successful completion continues its execution, nor waits for the completion of that operation.

As argued in detail in [10], the exception handling mechanisms available in languages like Java [6] do not provide a direct way to achieve, let alone verify, dependency safety. Problems on the safety side are that exiting a `try` block (as part of a `try-catch` statement) may leave a data source in a corrupted state. Problems on the liveness side are that a `wait` statement may wait on the successful return of a failing operation, causing that thread to wait indefinitely. An example of the latter, taken from [7], is the Scala [1] program in Listing 1. There, the main thread indefinitely blocks on `take` (line 3) if the forked thread, launched at line 2, fails.

Listing 1: Motivating Example

```
1 val queue = new LinkedBlockingQueue[String]()
2 fork { queue.put("Hello, world") }
3 queue.take()
```

An approach to achieve dependency safety in Java, called *failbox*, is introduced in [10]. This language extension allows dealing with exceptions compositionally. The mechanism of failboxes is as follows. All threads that depend on each other's successful execution of instructions are forced to run in the same failbox. As soon as an instruction executing in a failbox fails (leading, e.g., to its thread terminating abruptly with an unchecked exception), all threads in the same failbox will be notified, and terminated.

To achieve dependency safety, if an operation B depends on an operation A it must be ensured that they execute in the same failbox.

An example how one can use the failbox mechanism to fix the problem with the program in Listing 1 is given in Listing 2.

Listing 2: Proposed Fix with Failboxes

```
1 fb = new Failbox()
2 fb.enter {
3   val queue = new LinkedBlockingQueue[String]()
4   fork { fb.enter{queue.put("Hello, world") }
5 }
6 queue.take()
```

[1] http://www.scala-lang.org/files/archive/spec/2.11/.

PPPJ'16, August 29-September 02, 2016, Lugano, Switzerland

DOI: http://dx.doi.org/10.1145/2972206.2972216

Since the main and the forked thread run in the same failbox, if the forked thread crashes the failbox mechanism ensures that the main thread is notified about it. As a result, unlike in Listing 1, the main thread no longer blocks on `take`. Instead, it will be informed about the failure and terminate.
Failboxes are symmetric, dependencies can be asymmetric. This can be solved using nested failboxes. The extension is relatively straightforward, and not addressed in this paper.
There are several technical limiting assumptions with respect to the actual failbox implementations provided in [7, 10] which are related to the occurrence of *asynchronous exceptions*. Unlike synchronous exceptions (such as null pointer and array out of bounds) that occur when executing a specific statement, asynchronous exceptions can happen anywhere in the program. Hence, asynchronous exceptions cannot be completely dealt with by `try-catch` statements. These exceptions are caused by external factors, like the user interrupting (a part of) the program or another thread sending a stop signal.
The issues with asynchronous exceptions need to be resolved to guarantee dependency safety without limiting assumptions. This paper presents solutions to these problems in an incremental manner, and provides better insight into the subtle issues involved.
Our main contributions are as follows:

1. In [7] a failbox implementation in Scala was provided, which is used to translate to Java in this paper. This implementation is very intuitive because Scala turned out to be particularly amenable to implementing the failbox concept. As stated in that paper, this implementation provides the failbox functionality under the assumption that *no asynchronous exceptions occur in certain parts of the code implementing the failbox.*

   To better connect to Java and Java-tooling, a Java implementation is desirable. This is not a straightforward matter, because Java differs in various aspects from Scala that make the latter so suitable for implementing the failbox. A Java implementation that matches the Scala implementation, with the same limiting assumptions, is presented in Section 2.

2. It turns out that using Java's mechanism of uncaught exception handlers, the assumptions can be weakened to only concerning the *handlers of the failbox code.* An analysis concerning this observation and an implementation based on it is presented in Section 3. The analysis leads to the conclusion that, to further lift the limiting assumption, it might be necessary to use the Java Native Interface (JNI).[2]

3. An implementation pushing the uncaught exception handler approach to its limits, using JNI, that almost completely removes the assumption is provided in Section 4.

4. Finally, a precise analysis of the remaining restriction leads to a JNI level solution that does not use the uncaught exception handler and which is fully hardened against asynchronous interrupts; this is provided in Section 5.

Section 7 contains conclusions.

[2]http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/jniTOC.html.

Related work.
Apart from the general goal to produce (provably) robust code, to which the notion of failbox contributes, this research is specifically used in our model-based development framework, in which we use generic code to automatically generate concurrent Java programs based on state-machine-like models [3, 16]. We aim to fully verify correctness of this generic code, using the Java verification tool VeriFast.[3] VeriFast currently does not support the `try-catch` Java language feature. We use the failbox as a wrapper class to write robust code without using `try-catch` statements. This allows us to use VeriFast to verify robust code (modulo a proof by hand of the failbox implementation). Furthermore, the use of a wrapper class makes the code and its verification independent of changes to the failbox implementation, as long as the wrapper class specification remains satisfied.
The failbox mechanism offers a solution to the dependency safety problem which is less involving in terms of programming effort, and induces less run-time overhead compared to manually guarding dependent code [9] and the technique based on the use of separate threads [1]. Our approach requires less effort than the alternatives requiring to always maintain consistency [14, 5] or never fail during critical sections [15]. For more elaborate arguments on this subject we refer the reader to [10]. Recent alternative approaches are the following. [2] specifies the flow of exceptions between modules with implicit invocation relations. [13] presents an error recovery technique that enables to apply runtime assertion checkers. [4] focuses on recovering to or achieving a consistent state using the transactional memory technique from [14]. [11] deals with single threaded programs strong exception safety, requiring recovery to the state before the unsuccessful operation started; extension to multi-threading is identified as challenging. [12] rather than grouping dependent threads as in our approach, uses intervals and happens-before relations to reason about phases of computation to which work belongs.
In the present paper we concentrate on the failbox concept, improving the failbox technique by removing the limiting assumptions.

2. A BASIC FAILBOX IMPLEMENTATION

In this section, we describe the first iteration of our failbox implementation. It is based on the Scala implementation from [7]. Like its Scala counterpart, it is still vulnerable to asynchronous interrupts. It can be seen as a stepping stone towards the more robust implementations presented in subsequent sections.
A failbox is basically an object with a boolean variable, telling whether the failbox is operational (i.e., has failed or not), and a list of threads. Furthermore, it provides the method `enter`, which is called by a thread to add itself to the failbox, passing as an argument the code to be executed. If this code fails, the `enter` method notifies all other threads in the failbox. The Java implementation is shown in Listing 3.

Listing 3: Java Implementation of a Failbox

```
1 class FailboxException extends RuntimeException {}
2 class Failbox {
3   private boolean failed;
```

[3]https://people.cs.kuleuven.be/~bart.jacobs/verifast/.

```
4   private ArrayList<Thread> threads;
5   public void enter(Thread currentThread, Runnable body){
6     synchronized (this) {
7       if (failed) throw new FailboxException();
8       threads.add(currentThread);
9     }
10    try {
11      try {
12        body.run();
13      } finally {
14        synchronized (this) {
15          threads.remove(currentThread);
16        }
17      }
18    } catch (Throwable t) {
19      synchronized (this) {
20        failed = true;
21        for (Thread tr : threads) {
22          tr.interrupt();
23        }
24      }
25      throw t;
26    }
27  }
28 }
```

In Listing 3, the boolean variable `failed` (line 3) has initial value `false` indicating that the failbox is operational. The ArrayList `threads` (line 4) contains all threads which are running within the same failbox.
Whenever a thread t needs to execute a code block and other threads depend on this succeeding, t calls the method `enter` of the failbox containing the other threads, passing the code block as a parameter wrapped in a Runnable object `body` (line 5). The method (lines 5-27) first checks the state of `Failbox` (line 7). If the Failbox has already failed for some reason, a `FailboxException` is immediately thrown. Otherwise, the current thread is added to the list `threads` (line 8). When t has finished executing the code block, either in a regular way or due to an error, it is removed from the list `threads` (lines 14-16). If an exception occurs while executing the code block, all threads in `threads` are notified (lines 19-24). Consistency of field `threads` is ensured via the `synchronized` mechanism.

Limitations analysis.
This basic implementation is not fully resistant to asynchronous exceptions. It may fail to notify its threads about crashes in two cases:

1. If an asynchronous exception happens when in `enter` before the outer `try` block is entered (lines 6-9) the `catch` part, which would notify the other threads, will not be executed.

2. If an asynchronous exception occurs in the `catch` part the notification may be interrupted.

Hence the limiting assumption: no asynchronous exceptions occur when execution in a failbox's `enter` method is before the outer `try` block or in the `catch` block.

3. AN IMPLEMENTATION USING UNCAUGHT EXCEPTION HANDLER

The above mentioned vulnerabilities can be alleviated by using the mechanism of Uncaught Exception Handlers that was introduced in Java 1.5. Before terminating because of an uncaught exception the method `getUncaughtExceptionHandler` is called on the thread and the `uncaughtException` method is invoked with the thread and the exception as arguments.
The new `Failbox` class definition given in Listing 4 is an implementation of the `Thread.UncaughtExceptionHandler` interface.

Listing 4: Failbox class definition and construction

```
1 public class Failbox implements
        Thread.UncaughtExceptionHandler {
2   private boolean failed;
3   private ArrayList<Thread> threads;
4   Failbox() {
5     this.threads=new ArrayList<Thread>();
6     this.failed = false;
7   }
8 }
```

The fields `threads` and `failed` have the same meaning as before.
The `enter` method remains the core of the failbox implementation. In our context an important feature of the uncaught exception handler is that the method is executed no matter what happens with the thread that has set the handler. This is the last method that is being called before the thread is deleted. We use this feature to provide additional robustness to the failbox implementation. The code of the `enter` method is given in Listing 5.

Listing 5: The Failbox.enter() method

```
1 public void enter(Thread currentThread, Runnable code){
2   currentThread.setUncaughtExceptionHandler(this);
3   synchronized (this) {
4     if (failed) throw new FailboxException();
5     threads.add(currentThread);
6   }
7   code.run();
8   currentThread.setUncaughtExceptionHandler(null);
9 }
```

The association between the thread and the failbox, the latter becoming its uncaught exception handler, is made in line 2 in Listing 5. We assume that this is the only way for each thread to set its handler, i.e., no other method except `enter` sets it. The part of the previous implementation of `enter` that notifies all other threads in the failbox is moved to the exception handler which we discuss shortly. This means that the processing of an interrupt is no longer confined to `try-catch` statements. Instead, the interrupts can be caught and processed as soon as they happen. Besides improving robustness, this makes the implementation more efficient than the implementation in Listing 3 (and than its Scala counterpart in [7]). Note that `enter` should never be called within the scope of an exception handler.
The definition in Listing 4 requires an implementation of an uncaught exception handler method. We provide one in Listing 6.

Listing 6: An Uncaught Exception Handler method

```
1 public void uncaughtException(Thread th, Throwable t) {
2   fail();
3 }
```

This simple method forwards all occurrences of uncaught exceptions to the fail method provided in Listing 7.

Listing 7: The Failbox.fail() method

```
1 public synchronized void fail() {
```

```
2   for (Thread tr : threads) { tr.interrupt(); }
3   threads.clear();
4   failed = true;
5 }
```

Similarly to the previous Java implementation, the Java version needs to interrupt all the threads inside the failbox.

Limitations analysis.
For the handler-based implementation, there are two cases where asynchronous exceptions can still cause problems.

1. The robustness of the failbox is improved, since crashes occurring while executing the statements at lines 6-9 in Listing 2 are now handled correctly. However, if an asynchronous exception happens in `enter` before the exception handler is set (line 2 in Listing 5) we have again the problem that the exception will be missed.

2. If an asynchronous exception occurs in the handler itself, the problem with the incomplete notification remains.

The above observations require us to still assume the following: no asynchronous exceptions occur either when a thread is executing `enter`, but has not yet set the exception handler, nor when the handler is being executed.

4. AN IMPLEMENTATION USING UNCAUGHT EXCEPTION HANDLER AND JNI

The limitations mentioned at the end of the previous section can be partially removed by using native code (C or C++). Java supports native code via Java Native Interface (JNI). Using the JNI framework, Java methods of an application running in a Java Virtual Machine (JVM) can call or be called by JNI functions.
By using the JNI, we can further improve the failbox. This is because the execution of JNI instructions is completely independent of the JVM execution. This means that the JVM cannot stop JNI methods in the middle of execution. By reimplementing the `enter` method in JNI, we obtain a guarantee that this method is executed regardless of the occurrence of uncaught interrupts. The code of the new `enter` method given in Listing 8 is a direct translation of the Java method in Listing 5, i.e., the corresponding Java calls are mapped to JNI calls.

Listing 8: The Java_Failbox_enter() C method

```
1  JNIEXPORT void JNICALL Java_Failbox_enter(JNIEnv* env,
       jobject failbox, jobject currentThread, jobject
       body)
2  {
3    do_SetUncaughtExceptionHandler_method(env,
         currentThread, failbox);
4    do_MonitorEnter(env, failbox);
5    jboolean failed_value =
         do_get_FailboxField_failed(env, failbox);
6    if (failed_value == JNI_TRUE) {
7      do_Throw_FailboxException(env, "FailboxFailed");
8    }
9    jobject threads = do_get_FailboxField_threads(env,
         failbox);
10   do_ArrayList_add(env, threads, currentThread);
11   do_MonitorExit(env, failbox);
12   if ((*env)->ExceptionCheck(env) == JNI_TRUE)
13     do_Failbox_fail(env, failbox);
14   do_Runnable_run(env, body);
15   if ((*env)->ExceptionCheck(env) == JNI_FALSE)
16     do_SetUncaughtExceptionHandler_method(env,
           currentThread, NULL);
17 }
```

Listing 9: The do_MonitorEnter method

```
1 void do_MonitorEnter(JNIEnv *env, jobject failbox){
2   jint result = MonitorEnter(env, failbox);
3   if (result != 0) abort();
4 }
```

Listing 10: The Java_Failbox_fail() C method

```
1  JNIEXPORT void JNICALL Java_Failbox_fail(JNIEnv* env,
       jobject failbox)
2  {
3    do_MonitorEnter(env, failbox);
4    jobject threads = do_get_FailboxField_threads(env,
         failbox);
5    int arrayListSize = do_ArrayList_size(env, threads);
6    for (int i = 0; i < arrayListSize; i++) {
7      jobject arrayElement = do_ArrayList_get(env, threads,
           i);
8      do_Thread_interrupt(env, arrayElement);
9      do_SetUncaughtExceptionHandler_method(env,
           arrayElement, NULL);
10   }
11   do_ArrayList_clear(env, threads);
12   do_set_FailboxField_failed(env, failbox, JNI_TRUE);
13   do_MonitorExit(env, failbox);
14 }
```

The wrapper method calls with self-explanatory names in Listing 8, like `do_SetUncaughtExceptionHandler_method` at line 3 and `do_get_FailboxFiled_failed` at line 5, retrieve the various class types, as well as the methods, fields, and field values of Java classes which were defined at the Java level. The retrieved elements are stored in variables of corresponding JNI types.
The Java `synchronized` mechanism (line 6 in Listing 2) is replaced with JNI monitors to implement critical sections. A monitor is entered and exited by calling functions `do_MonitorEnter` (line 4 in Listing 8) and `do_MonitorExit` (line 11 in Listing 8), respectively. These functions are wrappers for the corresponding JNI functions `MonitorEnter` and `MonitorExit`. In the remainder of this paper, we use the convention that a method named `do_`*name* is a wrapper for the corresponding JNI method *name*. All wrapper methods follow the pattern of the `do_MonitorEnter` method given in Listing 9.
If the call to the original JNI function *name* is unsuccessful, the native code will cause a crash. The `abort` method is called to terminate the whole program when the original JNI method *name* fails.
Within the monitor, the status of `failed` is checked and a possible exception is thrown. Furthermore, the current thread is added to the failbox. These operations are done in lines 4-11, corresponding to lines 3-6 in Listing 5.
Crucial for this implementation is the `ExceptionCheck()` method in line 12 in Listing 8. It is responsible for checking in the JVM whether any exceptions have occurred during the execution. In this way, we can check if an asynchronous exception occurred before setting the exception handler.
In an analogous way, the code of the exception handler is implemented in JNI, see Listing 10, thereby further improving robustness. Compared to the Java version in Listing 7, the JNI implementation replaces the `synchronized` qualifier with the JNI `do_Monitor` to protect the shared fields. Also,

after sending a notification to each thread in the failbox, its uncaught exception handler is unset.

Limitations analysis.

1. The above discussion shows how case 1 from the limitations analysis of the previous section is remedied.

2. Case 2 of the previous section is also covered to a significant extent. However, the JNI level approach presented in this section still retains one weak spot. To call the uncaught exception handler, one needs to briefly go back to Java, i.e., back to the JVM. This is because the uncaught exception handler, which calls the native code of `fail`, has to be written in Java, see Listing 6. If an asynchronous exception occurs in this method before the native implementation of `fail` has been started (line 2 in Listing 6), the remaining threads will not be properly notified.

Hence the limiting assumption: no asynchronous exceptions occur when executing the (Java) uncaught exception handler, before the native method `fail` has been started.

5. A JNI IMPLEMENTATION WITHOUT UNCAUGHT EXCEPTION HANDLER

To resolve the last remaining vulnerability of the JNI implementation with the uncaught exception handler, proposed in the previous section, we reconsider the incomplete initial solution in Java given in Listing 3. We translate the `enter` method of this Java implementation into JNI as given in Listing 11.

Listing 11: The enter JNI method of Class Failbox

```
1  JNIEXPORT void JNICALL Java_Failbox_enter(JNIEnv *env,
       jobject failbox, jobject currentThread, jobject
       body) {
2    do_MonitorEnter(env, failbox);
3    jboolean failed_value =
         do_get_FailboxField_failed(env, failbox);
4    if (failed_value == JNI_TRUE) {
5      do_Throw_FailboxException(env, "FailboxFailed");
6    }
7    jobject threads = do_get_FailboxField_threads(env,
         failbox);
8    do_ArrayList_add(env, threads, currentThread);
9    do_MonitorExit(env, failbox);
10   do_Runnable_run(env, body);
11   jthrowable exception = (*env)->ExceptionOccurred(env);
12   (*env)->ExceptionClear(env);
13   do_MonitorEnter(env, failbox);
14   do_ArrayList_remove(env, threads, currentThread);
15   if (exception != NULL) {
16     do_set_FailboxField_failed(env, failbox, JNI_TRUE);
17     int arrayListSize = do_ArrayList_size(env, threads);
18     for (int i = 0; i < arrayListSize; i++) {
19       jobject arrayElement = do_ArrayList_get(env,
             threads, i);
20       do_Thread_interrupt(env, arrayElement);
21     }
22   }
23   do_MonitorExit(env, failbox);
24   if (exception != NULL)
25   do_Throw(env, exception);
26 }
```

This JNI implementation closely follows its Java counterpart from Listing 3.
The `try-catch` statement is replaced by an exception check by means of the JNI method `ExceptionOccurred` (line 11) and a conditional statement (line 15). In particular, the inner `try` part (line 13) in Listing 3 is translated using the `do_Runnable_run` wrapper method (line 10). After checking exceptions, the exception flag is cleared (line 12) such that new exceptions can be registered. The `finally` part (lines 14-18) in Listing 3 is merged with the `catch` part of the `try-catch` statement into a critical section implemented using wrapper methods such as `do_MonitorEnter` already discussed in Section 4. In the critical section the current thread is removed unconditionally from the failbox (line 14). After that, if an exception has occurred (line 15), the original `catch` part is performed to set the `failed` flag (line 16) and the interrupts are sent to each of the threads of the current failbox (lines 18-21).
Since the native code cannot be interrupted by JVM (asynchronous) interrupts, this code ensures the correct termination of all threads in the failbox. Of course, a logical question that arises is "What if the native C code fails?". For example, if the C code dereferences a null pointer or if it overflows the C stack. This can cause problems if such a native exception is caught by the JVM and turned into a Java exception. However, the Java documentation does not mention such a transformation of exceptions. In any case, we reason that assuming the absence of "machine" exceptions in the C code is more acceptable than assuming that no exceptions occur while executing Java instructions. In any case, it seems inevitable that one must assume reliable execution at some level of the application.

6. DEPENDENCY SAFETY REVISITED

Dependency safety can be interpreted as capturing two properties. The first one prevents situations when a thread remains blocked waiting for action of another thread which will never happen because of the crash. More precisely:

> For any threads t_1, t_2, if t_1 crashes, then t_2 is not blocked on an operation B which depends on operation A executed by t_1.

The second one prevents that a thread finds an inconsistent state because of the crash of another thread:

> For any threads t_1, t_2 which execute dependent operations that are mutually exclusive, if t_1 crashes, then t_2 does not execute operation B which depends on an operation A executed by t_1, after t_1 has crashed.

The solutions presented in the previous section were mainly oriented towards the first non-blocking version of the dependency safety. However, all of the presented implementations cover also the second inconsistency avoiding variant. The conditions have the same assumption with regard to the asynchronous interrupts.
If one is interested only in capturing the inconsistency version of the dependency safety, then the code of the solution can be simplified. A version of the Java solution (vulnerable to asynchronous interrupts) is given in Listing 12.

Listing 12: Failbox implementation for dependency safety for consistency only

```
1  class FailboxException extends RuntimeException {}
2  class Failbox {
3    private boolean failed;
```

```
4    public void enter(Runnable body) {
5       if (failed) throw new FailboxException();
6       try {
7          body.run();
8       } catch (Throwable t) {
9          failed = true;
10         throw t;
11      }
12   }
13 }
```

Since the operations are mutually exclusive we do not need the synchronisation blocks. Also, the other threads need not be notified about the crash, assuming that each dependent code block is entered separately into the failbox. The synchronisation happens via the `failed` field. Let t_1 and t_2 belong to the same failbox and let thread t_1 operation A depend on thread t_2 operation B. If t_1 crashes because of executing A, it will set `failed` to `true`. Then when executing method `enter`, protecting operation B, t_2 will find out that the failbox has failed and B will not be executed.
Of course, also the JNI implementation can be straightforwardly adjusted for this simple case.

7. CONCLUSIONS

We presented a Java implementation of the concept of failbox from [7]. Failboxes allow a simple way to deal with dependency safety problems in multithreaded programs. We presented four versions of the failbox implementation - two in Java and two in a combination of Java and C, using JNI. The Java implementations have the disadvantage that they are vulnerable to asynchronous interrupts. This weakness can be removed by implementing the crucial methods in C and using JNI.
Currently we are applying the failboxes concept in a project on verifying a model transformation from a domain specific language to Java [16, 3]. Future work involves formally verifying the proposed failbox implementations, and comparing the failbox approach to other mechanisms guaranteeing dependency safety. By setting objective criteria, a full evaluation of usability and performance could be performed.
Translating the failbox to JNI and C paves the way for an automated verification of the implementation. In particular, this can be done using VeriFast. (Besides Java, VeriFast can also verify C programs.) VeriFast does not currently support exception statements like `try-catch`, but since they are not present in the C translation, this makes the verification possible. Also, the new capabilities of VeriFast [8] allow to verify absence of deadlocks, which is an important aspect of the dependency safety [7].

8. REFERENCES

[1] J. Armstrong. *Making Reliable Distributed Systems in the Presence of Software Errors*. PhD thesis, Royal Institute of Technology, Stockholm, Sweden, 2003.

[2] M. Bagherzadeh, H. Rajan, and M. A. D. Darab. On Exceptions, Events and Observer Chains. In *AOSD*, pages 185–196, 2013.

[3] D. Bošnački, M. van den Brand, J. Gabriels, B. Jacobs, R. Kuiper, S. Roede, A. Wijs, and D. Zhang. Towards Modular Verification of Threaded Concurrent Executable Code Generated from DSL Models. In *Formal Aspects of Component Software - 12th International Conference, FACS 2015, Niterói, Brazil, October 14-16, 2015*, pages 141–160, 2015.

[4] P. Felber, C. Fetzer, V. Gramoli, D. Harmanci, and M. Nowack. Safe Exception Handling with Transactional Memory. In *Transactional Memory. Foundations, Algorithms, Tools, and Applications*, pages 245–267, 2015.

[5] C. Fetzer, K. Högstedt, and P. Felber. Automatic Detection and Masking of Non-Atomic Exception Handling. In *2003 International Conference on Dependable Systems and Networks, 22-25 June 2003, San Francisco, CA, USA*, pages 445–454, 2003.

[6] J. Gosling, B. Joy, G. Steele, G. Bracha, and A. Buckley. *The Java Language Specification*. Java SE 8 Edition, 2015.

[7] B. Jacobs. Provably Live Exception Handling. In *Proceedings of the 17th Workshop on Formal Techniques for Java-like Programs, Prague, Czech Republic, July 7, 2015*, pages 7:1–7:4, 2015.

[8] B. Jacobs, D. Bosnacki, and R. Kuiper. Modular Termination Verification: Extended Version. Cw reports, Katholieke Universiteit Leuven, 2015.

[9] B. Jacobs, P. Müller, and F. Piessens. Sound Reasoning about Unchecked Exceptions. In *Proceedings ICFEM*, 2007.

[10] B. Jacobs and F. Piessens. Failboxes: Provably Safe Exception Handling. In *ECOOP*, pages 470–494. Springer, 2009.

[11] G. Lagorio and M. Servetto. Strong Exception-Safety for Checked and Unchecked Exceptions. *Journal of Object Technology*, 10(1):1–20, 2011.

[12] N. D. Matsakis and T. R. Gross. Handling Errors in Parallel Programs Based on Happens Before Relations. In *24th IEEE International Symposium on Parallel and Distributed Processing, Atlanta, Georgia, USA, 19-23 April 2010*, pages 1–8, 2010.

[13] H. Rebêlo, R. Coelho, R. M. F. Lima, G. T. Leavens, M. Huisman, A. Mota, and F. Castor. On the Interplay of Exception Handling and Design by Contract: An Aspect-Oriented Recovery Approach. In *Proceedings of the 13th Workshop on Formal Techniques for Java-Like Programs, Lancaster, United Kingdom, July 25-29, 2011*, pages 7:1–7:6, 2011.

[14] N. Shavit and D. Touitou. Software Transactional Memory. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing, Ottawa, Canada, August 20-23, 1995*, pages 204–213, 1995.

[15] S. Toub. Keep Your Code Running with the Reliability Features of the .NET Framework. *MSDN Magazine*, October 2015.

[16] D. Zhang, D. Bošnački, M. van den Brand, L. Engelen, C. Huizing, R. Kuiper, and A. Wijs. Towards Verified Java Code Generation from Concurrent State Machines. In *AMT@ MoDELS*, pages 64–69, 2014.