Science of Computer Programming 184 (2019) 102316

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Dependency safety for Java – Implementing and testing failboxes

Dan Zhang [a,b,1], Dragan Bošnački [a], Mark van den Brand [a], Cornelis Huizing [a], Bart Jacobs [c], Ruurd Kuiper [a], Anton Wijs [a,*]

[a] Eindhoven University of Technology, NL-5600 MB Eindhoven, the Netherlands
[b] University of Twente, NL-7500 AE Enschede, the Netherlands
[c] Katholieke Universiteit Leuven, B-3001 Leuven, Belgium

ARTICLE INFO

Article history:
Received 12 January 2018
Received in revised form 23 September 2019
Accepted 25 September 2019
Available online 27 September 2019

Keywords:
Exception handling
Concurrency
Failboxes
Java

ABSTRACT

Exception mechanisms help to ensure that a program satisfies the important robustness criterion of dependency safety: if an operation fails in an execution sequence, any code depending on the successful completion of this operation should also fail in a controlled way. 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 in a compositional manner. Asynchronous exceptions occurring inside the failbox code are a serious challenge in achieving dependency safety. In this article we present a Java implementation which deals with this challenge by developing failboxes incrementally, through four increasingly robust iterations. For each incremental implementation step we analyze the vulnerabilities and argue the remedies in the next implementation. We also present a testing approach to investigate whether the vulnerabilities are realistic and the remedies proposed are effective. This testing approach enables us to generate asynchronous exceptions in a controlled manner for concurrent programs and the tests are repeatable in that they give the same results for runs that may differ in scheduling, even on different platforms.

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* [1]: if an operation A fails, any other operation B that depends on A's successful completion also fails. Preferably, the failure of B needs to be handled in a controlled way, striving to preserve as much as possible of the desired program properties, like consistency of the data or absence of indefinite blocking.

As argued in detail in the original work on failboxes [1], the exception handling mechanisms available in languages like Java [2] 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` instruction) may leave a data source in a corrupted state (we refer to this as violating *consistency dependency safety*). Problems on the liveness side are that a `wait` instruction may wait on the successful return of a failing operation, causing that thread to wait indefinitely (we refer to this as violating *wait dependency*

* Corresponding author.
E-mail address: a.j.wijs@tue.nl (A. Wijs).

[1] This work was sponsored by the China Scholarship Council (CSC), Grant number 201306280006.

https://doi.org/10.1016/j.scico.2019.102316
0167-6423/

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

Listing 1: Motivating example.

```
1 fb1 = new Failbox()
2 fb2 = new Failbox()
3 fb1.addDesc(fb2)
4 fb2.enter {
5   val queue = new LinkedBlockingQueue[String]()
6   fork { fb1.enter{queue.put("Hello, world") }
7   queue.take()
8 }
```

Listing 2: Proposed fix with failboxes.

safety). An example of the latter, taken from [3], is the Scala program in Listing 1. There, the main thread indefinitely blocks on `take` (line 3) if the forked thread, launched at line 2, fails.

A mechanism to achieve dependency safety, called *failbox*, is introduced by Jacobs et al. [1]. This language extension allows dealing with exceptions compositionally. During its execution, a thread may enter and leave various failboxes based on the dependencies of the instructions it is executing. Failboxes are placed in a tree structure. We say that a failbox f is a *descendant* of a failbox f' if either $f = f'$ or f is a descendant node of f' in the failbox tree. As soon as an instruction of one thread executing in a failbox fails (leading, e.g., to that thread terminating abruptly with an unchecked exception), all descendants of that failbox are marked as failed. Failed failboxes in turn notify and terminate all their threads, i.e., the threads that are at that time executing in them. To achieve dependency safety, if an operation B depends on an operation A it must be ensured that B runs in a failbox that is a descendant of the failbox in which A runs.

An example how one can use the failbox mechanism to fix the problem with the program in Listing 1 is given in Listing 2. At lines 1-2, two failboxes `fb1` and `fb2` are instantiated, and `fb2` is added as a descendant of `fb1` at line 3. The main thread `enters` `fb2` at line 4, and the forked thread `enters` `fb1` at line 6. Since at that point, the main thread runs in a descendant of the failbox in which the forked thread runs, 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 is informed about the failure and terminates.

Providing a failbox implementation that fully achieves dependency safety is far from straightforward. Identifying the requirements that a failbox should satisfy necessitates careful analysis of the vulnerabilities, overcoming those vulnerabilities in an implementation is challenging, and testing whether or not an implementation is robust w.r.t. those vulnerabilities calls for a methodology to perform precisely targeted tests. We present the development of a robust failbox implementation in an incremental way to make the issues involved explicit and accessible.

There are several technical limiting assumptions with respect to the actual failbox implementations provided by Jacobs et al. [1,3] which are related to the occurrence of *asynchronous exceptions*. These exceptions occur as a result of calling the stop method or because of an internal error or resource limitation of the Java Virtual Machine. Unlike synchronous exceptions (such as null pointer and array out of bounds) that occur when executing a specific instruction, asynchronous exceptions can happen anywhere in the program. Hence, asynchronous exceptions cannot be completely dealt with by `try-catch` instructions. 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. In this work we incrementally develop increasingly more robust failbox implementations. For each implementation we analyze the vulnerabilities and argue for the remedies in the next implementation.

We also investigate whether the vulnerabilities are realistic in the sense that there are tests for them that give a positive result, and also whether the remedies proposed are effective in the sense that these tests turn out negative on the improved version. This is challenging, as the properties to be tested involve subtle coordination issues, requiring instrumentation of Java and also Java Native Interface (JNI) [4] code. We introduce incrementally more powerful tests, guided by the demands of testing incrementally more robust failbox implementations.

This article is an extension of previously published work [5], in which we presented four different Java implementations of the failbox concept and reasoned about their strengths and weaknesses. The goal in that work is to construct an implementation that is robust w.r.t. asynchronous exceptions. Compared to that work, this article provides the following contributions:

1. We formally introduce and discuss relevant notions, such as multi-threaded program executions, thread failure and dependency safety.
2. We have added support for nested failboxes, that is, trees of failboxes, to handle asymmetric thread dependencies.
3. We discuss and demonstrate how to test the robustness of multi-threaded programs w.r.t. asynchronous exceptions.

4. In addition to discussing the four different implementations of the failbox concept and reasoning about their weaknesses, we apply the test setup to demonstrate that those weaknesses are indeed present, and that proposed remedies to remove those weaknesses are indeed effective.

Structure of the article. In Section 2, we discuss related work. Section 3 formally presents the different notions used in this article. Section 4 introduces the general test setup for testing failboxes on providing dependency safety. In Sections 5 to 8, four incrementally more robust implementations of the failbox concept are presented and tested using our general test setup. Section 9 provides an overview of the tests and test results. Section 10 contains conclusions and future work.

2. Related work

The failbox mechanism as explained in [1] offers a solution to the dependency safety problem which is less involving in terms of programming effort and less error-prone compared to the use of separate threads [6], manually ensuring non-failure while executing critical sections [7], and manually guarding dependent code [8]. Furthermore, the failbox mechanism induces less run-time overhead than the technique of Jacobs et al. [8].

Recent alternative approaches are the following. Bagherzadeh et al. [9] present a mechanism specifically for languages supporting implicit invocation, i.e., that one module invokes another implicitly and without knowing about it. In our case, targeting Java, we are not concerned with such invocations. An error recovery technique that enables the application of runtime assertion checkers is presented by Rebêlo et al. [10], but the authors leave the enforcement of dependency safety for future work.

Software and hardware transactional memory (TM) mechanisms [11] provide a means to define that a particular instruction block should be executed as an atomic transaction, inspired by the field of databases. Contrary to the use of locks, transactional memory provides an *optimistic* approach to parallelism: instead of refraining from executing a block of protected code B until all required locks are obtained, a thread immediately tries to execute B, and rolls back the effects of B in case another thread is interfering. In Software TM approaches, the required constructs to achieve this are implemented in software [12,13], while in Hardware TM, hardware constructs are provided [14]. While in general more robust, the drawback of Hardware TM is the limited size transactions can have; in particular, Software TM approaches have been developed in an attempt to mitigate this limitation, allowing for transactions of arbitrary size.

TM mechanisms traditionally do not address exception handling. Some proposals have been made to combine exception handling with TM. Felber et al. [15] focus on recovering to or achieving a consistent state using the TM technique by Shavit et al. [12]. Their technique supports both sequential and multi-threaded programs, and they experimentally compare the performance of their solution with the failbox approach, using for the latter the original implementation in Scala [3]. However, they ignore wait dependency safety, which is addressed by failboxes: in their interpretation of dependency safety, when a transaction fails, it has to be ensured that no transactions depending on the failing one will ever execute. The original definition of dependency safety is stronger, namely that the depending transactions fail as well. Consider again Listing 1. Following their interpretation, if one thread runs `queue.put` in a transaction and another thread `queue.take` in another transaction, then in case the `queue.put` method fails before the receiving thread has started to execute the `queue.take` method, the latter is allowed to block indefinitely. In contrast, failboxes ensure that this will not happen. Other proposals to combine exception handling and TM [16–19] also do not address wait dependency safety issues in which no inconsistent state is ever reached.

That proposals to combine TM techniques and exception handling so far have not addressed wait dependency safety is not surprising, considering what TM techniques have initially been designed for. Guerraoui and Kapałka [20] define the notion of *opacity* to formally define a correctness criterion for TM implementations. Informally, it requires that 1) operations performed by a committed, i.e., successfully performed, transaction appear as if they happened at a single point in time during the execution of the program, 2) no operation of an aborted transaction is ever globally visible, and 3) every transaction, also when aborted, always observes a consistent state. Note that this criterion does *not* address wait dependency safety. Therefore, any exception handling mechanism directly embedded in a TM technique is bound not to address that issue.

However, it should be noted that Felber et al.'s handling of consistency dependency safety issues can be seen as being more sophisticated, rolling back to a previous, consistent state as opposed to terminating the involved threads. For future work, it will be interesting to try to involve transactional memory for the handling of consistency dependency safety issues, while keeping support for handling wait dependency safety issues.

Lagario et al. [21] consider strong exception safety of single threaded programs, which requires recovery to the state before the unsuccessful operation started; extending this approach to multi-threaded programs is identified as challenging.

Matsakis et al. [22], rather than grouping dependent threads as in our approach, use intervals and an explicitly given happens-before relation to partition the computation of a program into phases. How the happens-before relation is actually respected by the program is left to the runtime system; for instance, locks are not explicitly used in the code. It would be interesting to experimentally compare their approach to ours, to establish which of the two tends to provide the best performance for commonly used patterns of parallel programs.

The testing approach is based on the Wait-notify patterns as discussed by Tasharofi et al. [23]; we also took inspiration from the online post of Komanov [24] for the application of countdown latches for synchronization in testing.

3. Preliminaries

3.1. Basic definitions

In this section we formally discuss the notions relevant for this article. The given definitions are based on those presented by Jacobs et al. [1].

First of all, we assume that for a given program, we can identify *program states* that contain all necessary information about a program execution and the corresponding thread states, e.g., a mapping of program variables (including the program/location counters) to their values, the sequences of instructions to be executed by each of the threads, activation records, etc. Using the concept of program state, we can define *executions*.

Definition 1 *(Execution).* Given a program π in program state C_0, we say that a (finite or countably infinite) sequence $C_0 \xrightarrow{st_0} C_1 \xrightarrow{st_1} C_2 \xrightarrow{st_2} \cdots$ is an *execution* of π iff the C_i are possible program states of π and in each C_i, program instruction st_i can be executed, leading to program state C_{i+1}.

Using the indices of the states in a given execution, it is possible to refer to particular execution points. For this reason, we say that i is an *execution point* in a given execution if the latter contains a state C_i.

When reasoning about multi-threaded programs, it is relevant to keep track of executions of the individual threads, in addition to which instructions are being executed. For this reason, we extend the notion of executions to *multi-threaded executions*.

Definition 2 *(Multi-threaded execution).* Given a multi-threaded program π with a set of thread identifiers T in program state C_0, we say that a sequence $C_0 \xrightarrow{t_0:st_0} C_1 \xrightarrow{t_1:st_1} C_2 \xrightarrow{t_2:st_2} \cdots$ is a *multi-threaded execution* of π iff $C_0 \xrightarrow{st_0} C_1 \xrightarrow{st_1} C_2 \xrightarrow{st_2} \cdots$ is an execution and for all t_i we have $t_i \in T$.

With $C_i \xrightarrow{t_j:st_j} C_{i+1}$ we denote that in program state C_i, thread t_j can execute program instruction st_j, leading to program state C_{i+1}. Similar to how we refer to execution points in executions, we can refer to execution points in multi-threaded executions. However, when we wish to make explicit which thread is going to execute the next instruction, we refer to a *thread execution point*. A thread execution point is a tuple (i, t), with i an execution point and t a thread identifier. We say that a given multi-threaded execution E has thread execution point (i, t) iff C_i is a program state in E and there exists an instruction st such that $C_i \xrightarrow{t:st} C_{i+1}$ is an execution step in E.

For a multi-threaded execution E, a *happens-before* relation can be defined between execution points, denoted by $\xrightarrow{hb_E}$. The happens-before relation is an irreflexive partial order (i.e., an antisymmetric and transitive relation) on thread execution points that captures their underlying causality relations. As such a happens-before relation closely depends on the programming language. The following definition is based on the one given in [1], which assumes a formally defined programming language supporting locks. We do not involve thread forking, since it is not used in the remainder of this article.

Definition 3 *(Happens-before).* The *happens-before relation* $\xrightarrow{hb_E}$ on thread execution points of a multi-threaded execution $E = C_0 \xrightarrow{t_0:st_0} C_1 \xrightarrow{t_1:st_1} C_2 \xrightarrow{t_2:st_2} \cdots$ is the smallest transitive relation that satisfies the following properties:

- Any thread execution point of a thread t happens-before any subsequent thread execution point of t:

$$i < j \Rightarrow (i, t) \xrightarrow{hb_E} (j, t)$$

- If execution step $t_i : st_i$ is a release of some lock o by thread t_i ($t_i : \mathsf{rel}(o)$), and subsequent execution step $t_j : st_j$ is an acquire of o by thread t_j ($t_j : \mathsf{acq}(o)$), then (i, t_i) happens-before all $(k, t_j) \in E$, with $k > j$:

$$C_i \xrightarrow{t_i:\mathsf{rel}(o)} C_{i+1} \wedge C_j \xrightarrow{t_j:\mathsf{acq}(o)} C_{j+1} \wedge i < j \Rightarrow (i, t_i) \xrightarrow{hb_E} (k, t_j)$$

The happens-before relation is partial in the sense that, for example, two different locks may be acquired by two different threads t_1, t_2 at execution points (i_1, t_1) and (i_2, t_2) that are not related by the happens-before relation. For such unrelated thread execution points, both execution orders, i.e., (i_1, t_1) followed by (i_2, t_2) and vice versa, are present in some possible executions of the multi-threaded program.

In the current article, we use, for testing purposes, a multi-threaded Java program in which threads try to access a queue (in fact, a `LinkedBlockingQueue`). This is done via the methods `put` and `take`. These methods make use of a lock to correctly access the queue. The `put` method releases the queue lock once a message has been put on the queue, and `take` acquires the lock before a message is taken. Hence, by the happens-before relation of Definition 3, there is also a happens-before relation between executions of `put` and subsequent executions of `take`.

Thread failure is represented in the execution points of an execution E. In other words, a failure of a thread t_j is recorded in an execution point i_{j+1} if execution step $i_j \overset{t_j:st_j}{\rightarrow} i_{j+1}$ fails.

Dependency safety is the property that if a thread instruction fails, instructions of other threads depending on its successful execution fail as well. To achieve this, we need to capture the dependencies between the execution points of those instructions. For this we use a *dependency relation* D which we define as a binary relation on thread execution points. This relation is application-specific, examples follow.

Now we can define dependency safety more formally:

Definition 4 *(Dependency safety).* Given a dependency relation D we say that a multi-threaded program π is *dependency safe* iff for all possible multi-threaded executions E and all pairs of thread execution points (i_1, t_1), (i_2, t_2) of E with $\langle (i_2, t_2), (i_1, t_1) \rangle \in D$ and (i_1, t_1) happens-before (i_2, t_2), we have that if t_1 is failing in i_1, then also t_2 is failing in i_2.

A dependency relation can be seen as a parameter of Definition 4 and is defined per program, depending on the correctness properties one is interested in. By different instantiations of the dependency relation D we can obtain different kinds of dependency safety relationships. For instance, by designing a dependency relation capturing dependency between updates of global objects (e.g., variables, arrays, database entries) we can realize *consistency dependency safety* which ensures that for each operation B that depends on operation A in the sense that both access the same object, if A fails, B does not subsequently operate on an inconsistent state of the object.

Another example is *wait dependency safety* which is the main notion considered in this article. It is obtained by defining D such that it expresses the fact that a wait operation B depends on an operation A since it is blocked awaiting a signal from A. For instance, in the Java-like language from [1] used in the definition of the happens-before relation (Definition 3), such a pair of operations are rel and acq. In Java, a *wait* instruction is dependent on a corresponding *notifyAll* instruction related to the same object. A multi-threaded program is wait dependency safe iff for all possible multi-threaded executions the following holds: if an operation B depends (by being blocked) on receiving an (unblocking) signal from an operation A but A unsuccessfully terminates before sending the signal to B, then B unsuccessfully terminates. In the Java-like language a possible concrete instantiation of the dependency relation D could be (with $|E|$ being the length of execution E, which is ∞ in case of an infinite execution):

Definition 5 *(Relation **D** for wait dependency safety).* Given an execution $E = C_0 \overset{t_0:st_0}{\rightarrow} C_1 \overset{t_1:st_1}{\rightarrow} C_2 \overset{t_2:st_2}{\rightarrow} \cdots$. For all locks o used in E, and all thread execution points (i_j, t_j), $(i_k, t_k) \in E$ with $t_j \neq t_k$, $st_j = st_k = \mathsf{acq}(o)$ and $next(k) \neq |E|$, we define that $(next(i_k), t_k)\ D\ (i_m, t_j)$ for all $i_m \in CS(i_j)$, with

- $CS(u) = \{v \mid v > u \wedge v < nrel(u) \wedge t_v = t_u\} \cup \{nrel(u) \mid nrel(u) \neq |E|\}$;
- $next(u) = min(\{v \mid v > u \wedge t_v = t_u\} \cup \{|E|\})$;
- $nrel(u) = min(\{v \mid v > u \wedge t_v = t_u \wedge st_v = \mathsf{rel}(o)\} \cup \{|E|\})$.

The definition establishes a general dependency of an acquire of a lock o by a thread t_k on the thread execution points at which other threads (t_j) hold the same lock, from the moment they acquire the lock to the moment they release the lock. With $next(u)$, we refer to the next execution point after u at which thread t_u executes a step, i.e., $(next(u), t_u)$ is a thread execution point in E. If this point does not exist, $next(u)$ refers to the last execution point in E. With $nrel(u)$, we refer to the next execution point after u at which t_u is about to execute $\mathsf{rel}(o)$. Again, $nrel(u)$ refers to the last execution point if such a point does not exist. Finally, $CS(u)$ is the set of execution points between u and $nrel(u)$ (including $nrel(u)$ if it does not refer to the last execution point in E) at which t_u is about to execute a step. For an execution point i_j at which t_j is about to execute $\mathsf{acq}(o)$, $CS(i_j)$ consists of all execution points at which t_j executes a step and holds lock o. Therefore, another thread t_k trying to acquire o is dependent on all those execution points. By applying the definition of D in Definition 5 on Definition 4. we obtain a concrete formal definition of the wait dependency safety relation for the Java-like language from [1].

Finally, a few words on the difference between the happens-before relation and dependency relation D. The happens-before relation captures causality between thread execution points and is dictated by the semantics of the programming language. The combination of D and happens-before in Definition 4, enables to separate dependency and causality into account and allows to conveniently define D at the level of instructions. For instance, D as defined in Definition 5 essentially relates instruction $\mathsf{acq}(o)$ (more precisely, all its executions) with (all executions of) $\mathsf{rel}(o)$. We define this formally as a relation $\hat{D}$ between pairs of instructions and threads. For two instruction-thread pairs (st_1, t_1) and (st_2, t_2), $\langle (st_1, t_1), (st_2, t_2) \rangle \in \hat{D}$ defines that for all execution points j, k of some execution E at which st_1 and st_2 are executed, respectively, we have $\langle (j, t_1), (k, t_2) \rangle \in D$.

3.2. Failboxes

It is worth noting that wait and consistency dependency safety described above are not achieved together in current implementations. A solution for one of these two types of dependency safety increases the risk of violating the other type.

This can be seen in the way languages like Java and .NET treat locks in the presence of exceptions. Namely, if an exception is raised by a thread t, locks owned by t are released before the exception is processed. Among other things, this is done to avoid other threads being blocked waiting to acquire the locks owned by t, as a prerequisite for wait dependency safety. However, releasing the locks creates a possibility that another thread t' will acquire the locks and process an inconsistent state, before the exception has been processed, i.e., before t' has been properly notified by t about the possible inconsistency, which may jeopardize consistency safety. Hence, a (partial) solution for the one dependency kind jeopardizes the other.

Nevertheless, the above mentioned aims are not irreconcilable. In previous work, it was shown that both wait and consistency dependency safety of a concurrent system with respect to a dependency relation D, that relates both wait dependent and consistency dependent execution points, can be ensured using so-called *failboxes* [1,3]. Failboxes are constructs containing threads, and failboxes can be placed in a hierarchy to reflect D. In case a thread crashes, the failbox in which it is executing at that moment is marked as failing, and all descendant failboxes are marked as well, causing those failboxes to notify the threads currently executing in them, by which those threads can react appropriately. By respecting D when executing instructions in failboxes we can achieve dependency safety. Respecting D means that for all pairs of thread execution points (i_1, t_1), $(i_2, t_2) \in E$ such that $(i_2, t_2)\ D\ (i_1, t_1)$, st_1 is running in a failbox f and st_2 is running in a descendant of f.

The failbox theory [1,3] states that if in a multi-threaded program π, for all pairs of operations A, B with B depending on A sending a signal, B is executed in a descendant of the failbox containing A, then π is wait dependency safe. In an analogous way, consistency dependency safety can be ensured by using failboxes correctly. Therefore, for both types of dependency safety, the following definition addresses the correct use of failboxes.

Definition 6 *(Correct use of failboxes).* We say that a multi-threaded program *uses failboxes correctly with respect to a dependency relation D*, if for all possible multi-threaded executions E and pairs of thread execution points (i_1, t_1), (i_2, t_2) with $\langle (i_2, t_2), (i_1, t_1) \rangle \in D$ and $C_{i_1} \overset{t_1:st_1}{\rightarrow} C_{i_1+1}$, $C_{i_2} \overset{t_2:st_2}{\rightarrow} C_{i_2+1}$ execution steps in E, st_1 is executed in a failbox f_1, st_2 is executed in a failbox f_2, and f_2 is a descendant of f_1.

Note that as long as the dependencies between *instructions* can be derived syntactically (as addressed at the end of Section 3.1, relation $\hat{D}$), it can be determined statically from the source code whether failboxes are used correctly. To ensure that using failboxes correctly with respect to D implies dependency safety of a (Java) program π the failbox construct needs to satisfy the following property (Main Lemma from [1]):

Lemma 1. *Consider an execution E of π containing two thread execution points (i_1, t_1) and (i_2, t_2) in E, such that (i_1, t_1) happens-before (i_2, t_2), t_1 is running at execution point i_1 in a failbox f_1, t_2 is running at i_2 in a failbox f_2, and f_2 is a descendant of f_1. Then, if t_1 is failing in i_1, t_2 is failing in i_2.*

Definition 6 and Lemma 1 together ensure that Definition 4 is satisfied. Definition 6 takes care of the D relation, whereas Lemma 1 ensures compliance of the instructions ordered by an execution E.

Therefore, in the case of wait dependency safety a failbox implementation needs to be designed such that whenever a wait operation W of thread t_2 running in a failbox f_2 at execution point i_2 depends on operation A of thread t_1 running in a failbox f_1 at execution point i_1, with f_2 being a descendant of f_1 and i_1 happens-before i_2, then if A fails at i_1, t_2 is notified about it (by sending a stop signal, for instance). Since, by the definition of the happens-before relation, W cannot be executed at i_2 before A at i_1 has been executed, W will eventually fail if A failed, either because of a stop signal or because it has already failed (independently of A).

In particular for wait dependency safety, such a design of the failbox relies on two aspects: successful detection of a failure and notification of descendant failboxes and all threads executing in the failing failbox. If a failure happens before the detection mechanism is activated or during the notification process the other threads are not properly notified. This observation also forms the basis for the design of our test suite that we use in this article.

To implement the use of failboxes, for consistency dependency safety the synchronizing mechanisms, e.g., synchronized instructions, locks, semaphores or transactional memory transactions, need to be adapted such that they become failbox aware. In particular, they need to be able to detect that a failbox has failed. For a more elaborate discussion on this, see Section 2 and the work of Jacobs et al. [1].

4. Testing the robustness of multi-threaded programs

In this section, using the definitions of Section 3 we provide a test setup to experimentally determine whether multi-threaded programs are wait dependency safe in the presence of asynchronous exceptions.

4.1. Test setup

When implementing the failbox concept, the challenge is to achieve correct behavior of the mechanism in the presence of asynchronous exceptions. Since these can occur at any moment during program execution, and hence at any stage of executing the code inside a failbox, both the detection and notification phases of a failbox can be jeopardized.

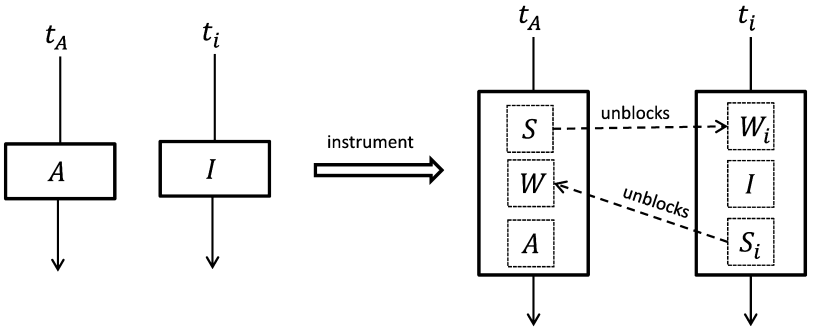

Fig. 1. Instrumentation.

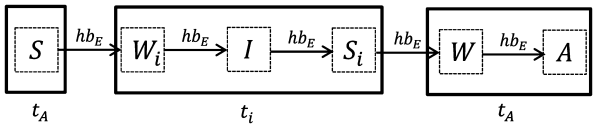

Fig. 2. The happens-before sequence.

To test failboxes, we provide a test setup that is representative for the dependency safety property in the presence of asynchronous exceptions. In what follows we focus on wait dependency safety, although the setup can be adapted for data consistency dependency safety. The setup consists of two threads t_A and t_B that execute operations A and B, respectively. The operations are dependent, i.e., $\langle (B, t_B), (A, t_A) \rangle \in \hat{D}$. For convenience, we say that A happens-before B in each considered execution sequence instead of stating the more precise relation between the corresponding execution points. We refer to this code as the *tested program*. For the test, we also need one thread t_i which generates asynchronous exceptions. We call the code corresponding to t_i the *test program*.

Testing in a concurrent setting is challenging because the scheduling of threads may influence the outcome. To enforce a particular testing scenario, i.e., occurrence of asynchronous exceptions in particular program states, synchronization is needed. This is done by preceding the critical operations (locations) with dummy synchronization operations.

For our tests we need to enforce a scenario in which operation A fails, and if as a result operation B fails too, we say that the test is *negative*, otherwise the test is *positive*, in the sense that it was able to demonstrate that the implementation is not correct, i.e., dependency safety does not hold.

To enforce that operation A fails due to an asynchronous exception, we need to execute an operation I triggering such an exception. Furthermore, we want to ensure a happens-before relation between I and A, which in general does not exist, if I and A are performed by different threads. (Note that I and A need not be dependent.) To this end, as shown in Fig. 1 we instrument the code by replacing operation I by the sequence $W_i; I; S_i$, where W_i and S_i are blocking and unblocking operations, respectively. Also a sequence of operations $S; W$ is added immediately before the operation A. S unblocks W_i whereas W is a blocking operation which can be unblocked only by S_i.

Now, using the definition of happens-before and its transitivity, it is easy to establish for an arbitrary execution E that $S \overset{hb_E}{\rightarrow} W_i \overset{hb_E}{\rightarrow} I \overset{hb_E}{\rightarrow} S_i \overset{hb_E}{\rightarrow} W \overset{hb_E}{\rightarrow} A$ in Fig. 2. This is because operation S executed by thread t_A unblocks W_i executed by thread t_i, implying $S \overset{hb_E}{\rightarrow} W_i$; operations W_i, I and S_i are executed by thread t_i, implying that $W_i \overset{hb_E}{\rightarrow} I \overset{hb_E}{\rightarrow} S_i$. Similarly, we can obtain relations $S_i \overset{hb_E}{\rightarrow} W$ and $W \overset{hb_E}{\rightarrow} A$. Obviously, the above implies I happens-before A. Moreover, since $S \overset{hb_E}{\rightarrow} I \overset{hb_E}{\rightarrow} A$, we achieve that after execution of I, the first operation performed by t_A will be A.

Note that the synchronization scheme introduced above does not introduce a deadlock as a side effect. This is because there is never circular waiting between the synchronized threads. The happens-before sequence of an arbitrary execution E will respect $S \overset{hb_E}{\rightarrow} W_i \overset{hb_E}{\rightarrow} S_i \overset{hb_E}{\rightarrow} W$, as shown in Fig. 3. Since W does not happen-before S, there is no circular waiting between threads t_A and t_i, implying that no deadlock is introduced by the synchronization scheme. Finally, note that since I interrupts thread t_A and W is waiting for a signal that is never supposed to arrive, there is technically no need to introduce S_i in the program.

We conjecture that our experimental setup is sufficient to ensure that if the implementation passes the test for safety dependency, it will pass that test for any program. We assume that thread t_B runs in a descendant of the failbox in which t_A runs. Threads t_A and t_B with the corresponding dependent operations A and B are sufficient abstractions of any thread and an environment in which dependent operations are executed. So, if the implementation works within our test setup, the dependency safety definition will be satisfied for any set of threads executed in a program. A rigorous proof of this generalization though remains beyond the scope of this work.

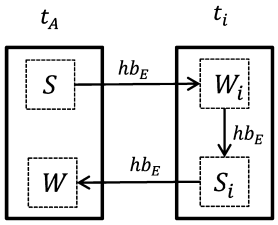

Fig. 3. Non-circular waiting.

```
1 class InterruptingThread extends Thread {
2   Thread interruptedThreadId;
3   CountDownLatch latchA;
4   InterruptingThread(Thread interruptedThreadId, CountDownLatch latchA) {
5     this.interruptedThreadId = interruptedThreadId;
6     this.latchA = latchA;
7   }
8   @Override
9   public void run() {
10    try {
11      latchA.await();
12    } catch (InterruptedException e) {
13      e.printStackTrace();
14    }
15    interruptedThreadId.stop();
16    System.out.println(Thread.currentThread().getName() + " stops "+interruptedThreadId.getName());
17  }
18 }
```

Listing 3: Class InterruptingThread with Latch.

4.2. Test setup implementation

To practically achieve the required order in which instructions S and W_i on the one hand, and S_i and W on the other hand, are executed in multi-threaded Java code, we use `CountDownLatch` based synchronization. Class `CountDownLatch` from the Java concurrency API is a construct that implements countdown latches. The construct provides a mechanism to make one or more threads wait for a given set of operations to complete. A `CountDownLatch` is initialized with a given count. This count is decremented by calls to the `countDown` method of class `CountDownLatch`. Threads that need to wait for this count to reach zero can do so by calling one of the `await` methods of a `CountDownLatch`. That is, invocation of the `await` method blocks the thread until the count reaches zero.

By using the `CountDownLatch` mechanism, a specific ordering of operations between multiple threads can be forced to occur. This way to achieve a deterministic ordering can make tests reliable, as it is not sensitive to the current workload of the machine or particular scheduler behavior. Moreover, different from delay based approaches, this approach helps to get fast running tests because the additional synchronization operations do add time, but latches save time with regard to delays. The latch methods `CountDown` and `await` correspond to the S and W operations in the abstract set up described above.

Test program. The interrupting thread t_i from the abstract setup in Section 4 is implemented by the class `InterruptingThread` in Listing 3. Operation I is implemented using method `stop` of class `Thread`. This method generates an asynchronous exception in thread t_A (`threadPut`).

As mentioned above, the blocking and releasing/signaling operations from the abstract setup are replaced with operations on countdown latches. In particular, we instrument the interrupting thread (Listing 3) by adding the call `latchA.await()` corresponding to the W_i operation.

Tested program. Thread t_A is implemented using class `RunnableBlockPut` in Listing 4. The `put` instruction (line 15) corresponds to operation A from the setup whose execution needs to be prevented.

The method calls `latchA.countDown()` at line 13 and `latchB.await()` at line 14 correspond to the signaling operation S and the blocking operation W from the abstract setup, respectively. Note that `latchB` is not decreased by any thread.

Thread t_B is implemented by class `RunnableBlockTake` in Listing 5. Operation B which depends on A (`put`) corresponds to method `take` (line 9).

The components of the test and tested program are run from the main program in Listing 6. Two instances of class `CountDownLatch`, `latchA` and `latchB`, are initialized with a given count 1, respectively at line 4 and 5, and passed to the constructor of class `RunnableBlockPut` at line 6. The `latchA` is passed to the constructor of class `InterruptingThread` at line 10. Via the test and tested program, we aim to check if the thread `threadTake` (a thread executing

```
1  class RunnableBlockPut implements Runnable {
2    LinkedBlockingQueue<String> queue;
3    CountDownLatch latchA;
4    CountDownLatch latchB;
5    RunnableBlockPut(LinkedBlockingQueue<String> queue, CountDownLatch latchA, CountDownLatch latchB) {
6      this.queue = queue;
7      this.latchA = latchA;
8      this.latchB = latchB;
9    }
10   @Override
11   public void run() {
12     try {
13       latchA.countDown();
14       latchB.await();
15       queue.put("hello");
16     } catch (InterruptedException e) {
17       e.printStackTrace();
18     }
19   }
20 }
```

Listing 4: Class RunnableBlockPut with Latch.

```
1  class RunnableBlockTake implements Runnable {
2    LinkedBlockingQueue<String> queue;
3    RunnableBlockTake(LinkedBlockingQueue<String> queue) {
4      this.queue = queue;
5    }
6    @Override
7    public void run() {
8      try {
9        queue.take();
10     } catch (InterruptedException e) {
11       e.printStackTrace();
12     }
13   }
14 }
```

Listing 5: Class RunnableBlockTake.

```
1  public class QueueExample {
2    public static void main(String[] args) {
3      LinkedBlockingQueue<String> queue = new LinkedBlockingQueue<String>();
4      CountDownLatch latchA = new CountDownLatch(1);
5      CountDownLatch latchB = new CountDownLatch(1);
6      Thread threadPut = new Thread(new RunnableBlockPut(queue, latchA, latchB));
7      threadPut.start();
8      Thread threadTake = new Thread(new RunnableBlockTake(queue));
9      threadTake.start();
10     new InterruptingThread(threadPut, latchA).start();
11   }
12 }
```

Listing 6: Class QueueExample with Latch.

`RunnableBlockTake`, Listing 5 line 7) will be notified when the `threadPut` (a thread executing `RunnableBlockPut`, Listing 4 line 11) crashes before reaching the `put` operation. Clearly, if this does not happen, wait dependency safety is violated, since $(\texttt{take}, \texttt{threadTake})\ \hat{D}\ (\texttt{put}, \texttt{threadPut})$ and the execution point at which thread `threadPut` executes `put` happens-before the execution point at which `threadTake` executes `take` (Definition 4).

The interrupting thread awaits a release signal via the countdown latch from the `put` thread. Immediately before executing the `put` operation the thread decrements the latch counter, thereby releasing the interrupting thread – and subsequently blocks, awaiting a signal from the interrupting thread. The interrupting thread sends a stop message to the `threadPut` thread which arrives while the `threadPut` thread is blocked. Therefore, the `put` operation is actually never executed. This simple handshaking protocol guarantees that an asynchronous exception always occurs before the `put` operation has completed.

Output and results. To observe sequences of the execution of the program, we add calls of the `println` method at critical places to show its outputs. For instance, a `println` method call is added at line 16 in Listing 3 to indicate whether thread `InterruptingThread` sends a "stop" signal to thread `threadPut` and another `println` method call is added at line 11 in Listing 5 to indicate whether thread `threadTake` is interrupted. In outputs of all tests demonstrated in this article, `Thread-0`, `Thread-1`, and `Thread-2` are strings denoting the names of threads `threadPut`, `threadTake`, and `InterruptingThread`, respectively.

If the line "Thread-1 is interrupted" is shown as output, we say that the test result is negative; if we do not get this line as output, the test result is positive, i.e., the dependency safety of this program is violated. As wait dependency safety is a liveness property, it is in general difficult to know when the property is definitely violated. However, for the test program, we know there is a violation of wait dependency safety (Definition 4 with (take, threadTake) $\hat{D}$ (put, threadPut)) as soon as we observe that no more than one thread is running, namely Thread-1, which can therefore never be interrupted anymore. With the Linux ps command, we can make such an observation. For all test cases described in Sections 5 to 8 that produce a positive result, we have made such an observation, and hence we can conclude that they demonstrate violations of wait dependency safety.

In Table 2, the outputs and results of all the tests conducted in this article are summarized. Table 1 provides an overview of vulnerability points of different failbox implementations tested throughout our stepwise development of the failbox concept.

In the next section we introduce a first, basic implementation of failbox and add it to the test setup.

5. A basic failbox implementation

In this section, we describe the first iteration of our failbox implementation. It is based on the Scala implementation of Jacobs [3]. 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), a list of descendant failboxes, and a list of threads. Furthermore, it provides the method addDesc, to add a descendant failbox, a method enter, which is called by a thread to add itself to the failbox, passing as an argument the code to be executed, and a method fail, to be executed when a failure happens. If the code passed to the enter method fails, the fail method is called to signal failure to all descendants and notify all other threads in the failbox. The Java implementation is shown in Listing 7. Initially, ignore the lines colored gray.

In Listing 7, the boolean variable failed (line 3) has initial value false (line 8) indicating that the failbox is operational. The ArrayList descendants (line 4) contains all descendants of the current failbox. The ArrayList threads (line 5) contains all threads that are running within this failbox.

Whenever a thread t needs to execute a code block and other threads depend on this succeeding, t calls the method enter of a failbox of which its descendants contain the other threads, passing the code block as a parameter wrapped in a Runnable object body (line 17). The method (lines 17-44) first checks the state of the failbox (line 27). 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 28). 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 34-36). If an exception occurs while executing the code block, the fail method is called (line 40). This method (lines 46-57) marks the failbox as failed (line 48), calls the fail method of all descendants (lines 49-51) and notifies all threads in the failbox (lines 52-55), but only if the current failbox was not already previously marked as failed (line 47). Consistency of fields descendants and threads is ensured via the synchronized mechanism.

We added println method calls at lines 18, 39, 43 and 54. The output of the println method call at line 18 prints the names of threads entering a failbox. The println method call at line 39 indicates that a thread starts interrupting other threads running inside descendant failboxes. When threads exit the failbox after no exception has occurred during their executions inside the failbox, the println method call at line 43 prints their name. Finally, the println method call at line 54 indicates that a thread has been interrupted. These messages help developers to keep track of the current status. We use the convention that test code such as these println instructions in the various failbox listings of this article is colored gray. In the next subsection, we discuss lines 19-25.

5.1. Testing with the basic failbox implementation

Next, we incorporate the basic failbox implementation in our test setup.

Test program. The code of the interrupting thread remains the same as in the previous section.

Tested program. The two thread implementations remain the same as in the previous section.

We use class FailboxedThread to wrap the invocation of the enter method of class Failbox in Listing 7. The updated class QueueExample with the failboxes and latches is shown in Listing 8. In this test case, threads threadPut and threadTake run inside failbox f1 and f2, respectively, with f2 being a descendant of f1 (line 5). Synchronization between threads InterruptingThread and threadPut is implemented by latches latchA and latchB, as before.

A new class FailboxedThread (Listing 9) is added to wrap the invocation of the enter method of class Failbox. Via the run method of class FailboxedThread, the two dependent operations, i.e. put and take, are passed to the enter method of their corresponding failboxes.

```
class FailboxException extends RuntimeException {}
class Failbox {
  private boolean failed;
  private ArrayList<Failbox> descendants;
  private ArrayList<Thread> threads;

  Failbox() {
    this.failed = false;
    this.descendants = new ArrayList<Failbox>();
    this.threads = new ArrayList<Thread>();
  }

  public void addDesc(Failbox f){
    this.descendants.add(f);
  }

  public void enter(Thread currentThread, Runnable body){
    System.out.println(currentThread.getName() + " enters a failbox");
    if (currentThread.getName().equals("Thread-0")) {
    try {
      currentThread.latchA.countDown();
      currentThread.latchB.await();
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
    synchronized (this) {
      if (failed) throw new FailboxException();
      threads.add(currentThread);
    }
    try {
      try {
        body.run();
      } finally {
        synchronized (this) {
          threads.remove(currentThread);
        }
      }
    } catch (Throwable t) {
        System.out.println(currentThread.getName() + " starts interrupting threads in failbox(es)");
        this.fail();
        throw t;
    }
    System.out.println(currentThread.getName() + " exits a failbox");
  }

  public synchronized void fail() {
    if (!failed) {
      failed = true;
      for (Failbox f : descendants) {
        f.fail();
      }
      for (Thread tr : threads) {
        tr.interrupt();
        System.out.println(tr.getName() + " is interrupted by " + Thread.currentThread().getName() + "
            inside a failbox");
      }
    }
  }
}
```

Listing 7: Java implementation of a Failbox.

```
public class QueueExample {
  public static void main(String[] args) {
    Failbox f1 = new Failbox();
    Failbox f2 = new Failbox();
    f1.addDesc(f2);
    LinkedBlockingQueue<String> queue = new LinkedBlockingQueue<String>();
    CountDownLatch latchA = new CountDownLatch(1);
    CountDownLatch latchB = new CountDownLatch(1);
    Thread threadPut = new FailboxedThread(f1, new RunnableBlockPut( queue, latchA, latchB));
    threadPut.start();
    Thread threadTake = new FailboxedThread(f2, new RunnableBlockTake( queue));
    threadTake.start();
    new InterruptingThread(threadPut, latchA).start();
  }
}
```

Listing 8: Class QueueExample with Failbox and Latches.

```
1  class FailboxedThread extends Thread {
2    Runnable runnable;
3    Failbox failbox;
4    public FailboxedThread(Failbox failbox, Runnable runnable) {
5      this.runnable = runnable;
6      this.failbox = failbox;
7    }
8    @Override
9    public void run() {
10     failbox.enter(this, runnable);
11   }
12 }
```

Listing 9: Class FailboxedThread with Failbox.

Output and results. The test produces the following output, with Thread-0 being the name of the `threadPut` thread, Thread-1 being the name of the `threadTake` thread, and Thread-2 being the name of the `InterruptingThread`:

1. Thread-0 enters the failbox
2. Thread-1 enters the failbox
3. Thread-2 stops Thread-0
4. Thread-0 starts interrupting threads in failbox(es)
5. Thread-1 is interrupted by Thread-0 inside a failbox

This output of the program shows that `threadPut` called the interrupt method of `threadTake` after the `InterruptingThread` stopped `threadPut`. Therefore, the result of this test is negative, i.e. the basic implementation of failbox helps to preserve the dependency safety of the program.

However, the basic failbox implementation may still crash, i.e., not function as intended in the two cases described below.

Limitations analysis. The 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 in `enter` (Listing 7) before the outer `try` block at lines 30-38, then the `catch` block (lines 38-42), which is supposed to mark descendants and notify the other threads, will not be executed.
2. If an asynchronous exception occurs while executing instructions in the `catch` part the notification may be interrupted.

Hence the limiting assumption: no asynchronous exceptions occur when execution of a failbox's `enter` method has not reached yet the outer `try` block or is inside the `catch` block. Note that the first vulnerability relates to the inability to detect a failure, while the second one relates to notifying the other threads, i.e., the two aspects identified in Section 3.

In the next section we will adjust our tests accordingly to point to the first vulnerability of the basic implementation. The second vulnerability will be addressed once the first has been remedied.

5.2. Testing the limitation of the basic failbox implementation

Test program. We take over the implementation of the interrupting thread from the previous section (Listing 3).

Tested program. The implementations of the threads remain the same, except for the Latch operations, which are now moved to the basic failbox code.

The instrumented basic failbox implementation of the `enter` method is shown in Listing 7 if we involve lines 19-25. If a thread's string name equals `Thread-0` (line 19), i.e., it is the `threadPut` thread, then the instructions at lines 20-25 are executed. The synchronization protocol with latches works as in the previous section. It ensures that an asynchronous exception occurs in `threadPut` before it can enter the outer `try` block at line 30. In this way, the test execution sequence, revealing a failbox crash vulnerability, can be enforced.

The class `QueueExample` in this test is similar to the one with the failbox and latches in Listing 8. We only need to pass the `latchA` and `latchB` to the constructor of class `FailboxedThread` instead of the one of class `RunnableBlockPut`, as shown at line 9 in Listing 10. This is because the synchronization in Listing 4 has been moved to the `enter` method of class `Failbox` (lines 19-25 in Listing 7) to simulate the failing scenario above. The class `FailboxedThread` is updated as shown in Listing 11.

Output and results. In this test, the execution of `threadPut` fails immediately before the synchronized instruction inside the basic failbox implementation. The output shows that `threadTake` running inside a descendant of the failbox in which `threadPut` runs is not notified when `threadPut` fails. So, the result of the test is positive.

Some of the vulnerabilities of the basic failbox implementation can be alleviated by using the mechanism of uncaught exception handlers in Java.

```
1  public class QueueExample {
2    public static void main(String[] args) {
3      Failbox f1 = new Failbox();
4      Failbox f2 = new Failbox();
5      f1.addDesc(f2);
6      LinkedBlockingQueue<String> queue = new LinkedBlockingQueue<String>();
7      CountDownLatch latchA = new CountDownLatch(1);
8      CountDownLatch latchB = new CountDownLatch(1);
9      Thread threadPut = new FailboxedThread(f1, new RunnableBlockPut(queue), latchA, latchB);
10     threadPut.start();
11     Thread threadTake = new FailboxedThread(f2, new RunnableBlockTake(queue), null, null);
12     threadTake.start();
13     new InterruptingThread(threadPut, latchA).start();
14   }
15 }
```

Listing 10: Postive testing: class QueueExample with Failbox and Latch.

```
1  class FailboxedThread extends Thread {
2    Runnable runnable;
3    Failbox failbox;
4    CountDownLatch latchA;
5    CountDownLatch latchB;
6    public FailboxedThread(Failbox failbox, Runnable runnable,CountDownLatch latchA, CountDownLatch latchB) {
7      this.runnable = runnable;
8      this.failbox = failbox;
9      this.latchA = latchA;
10     this.latchB = latchB;
11   }
12   @Override
13   public void run() {
14     failbox.enter(this, runnable);
15   }
16 }
```

Listing 11: Class FailboxedThread with Failbox and Latch.

When a thread needs to be terminated because of an uncaught exception, the method `getUncaughtExceptionHandler` of class `Thread` is called on the thread and the `uncaughtException` method is invoked with the thread and the exception as arguments.

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 terminated. We use this feature to provide additional robustness to the failbox implementation.

In the next section we show the failbox implementation using Java uncaught exception handlers and also demonstrate its improvement via our testing approach. However, as the failbox implementation using Java uncaught exception handlers still has possible vulnerabilities, we adjust our tests accordingly to point to the possible vulnerabilities of this implementation.

6. An implementation using Uncaught Exception Handler

Next, we use 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 12 is an implementation of the `Thread.UncaughtExceptionHandler` interface.

The fields `failed`, `descendants` and `threads` 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 it 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 terminated.

The association between the thread and the failbox, the latter becoming its uncaught exception handler, is made in line 17 in Listing 12. 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 `fail` method is invoked by the exception handler which we discuss shortly. This means that the processing of an interrupt is no longer confined to `try-catch` instructions. 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 7 (and than its Scala counterpart in [3]). A restriction for this implementation is that `enter` should never be called within the scope of an exception handler. However, this is a reasonable restriction, as calling the `enter` method should exclusively be done as the very first step of a newly created thread, and launching new threads is typically not done when handling an exception.

```
1  public class Failbox implements Thread.UncaughtExceptionHandler {
2    private boolean failed;
3    private ArrayList<Failbox> descendants;
4    private ArrayList<Thread> threads;
5
6    Failbox() {
7      this.failed = false;
8      this.descendants = new ArrayList<Failbox>();
9      this.threads = new ArrayList<Thread>();
10   }
11
12   public void addDesc(Failbox f){
13     this.descendants.add(f);
14   }
15
16   public void enter(Thread currentThread, Runnable code){
17     currentThread.setUncaughtExceptionHandler(this);
18     System.out.println(currentThread.getName() + " enters a failbox");
19     if (currentThread.getName().equals("Thread-0")) {
20     try {
21       currentThread.latchA.countDown();
22       currentThread.latchB.await();
23     } catch (InterruptedException e) {
24       e.printStackTrace();
25     }
26     synchronized (this) {
27       if (failed) throw new FailboxException();
28       threads.add(currentThread);
29     }
30     code.run();
31     System.out.println(currentThread.getName() + " exits the failbox");
32     currentThread.setUncaughtExceptionHandler(null);
33   }
34
35   @Override
36   public void uncaughtException(Thread th, Throwable t) {
37     System.out.println(th.getName() + " starts interrupting threads in failbox(es)");
38     fail();
39   }
40 }
```

Listing 12: Failbox class definition implementing UncaughtExceptionHandler.

The definition in Listing 12 requires an implementation of an uncaught exception handler method, which is given at lines 35-39. This simple method forwards all occurrences of uncaught exceptions to the `fail` method. The `fail` method itself remains as implemented in Listing 7, lines 46-57.

6.1. Testing the improvement of the new failbox implementation

Compared with the basic implementation, the robustness of the implementation of failboxes with uncaught exception handler is improved. So the test case should also be able to distinguish between the basic implementation of failbox and the one that uses the uncaught exception handler.

Test and tested programs. All programs remain the same as in the previous section, except for the presented new failbox implementation (Listing 12). In order to compare the two implementations, we need to choose a fixed location to add the same synchronizing test code. In this way, the location of the test code does not influence the distinguishing capability of the test. In the `enter` method of the basic implementation, the test code is added before the synchronized instruction at line 26 in Listing 7. Therefore, we add the same test code at lines 19-25 before the synchronized instruction at line 26 in Listing 12.

Output and results. Like the test in Subsection 5.2, the execution of `threadPut` fails at the same vulnerable place, i.e., immediately before the synchronized instruction inside the failbox code of the new implementation. However, the output of the test case for the new implementation shows that `threadTake` is interrupted by `threadPut` when it crashes. So the result of the test in this section is negative. Different results in Subsection 5.2 and Subsection 6.1 imply that the new failbox implementation that uses the uncaught exception handler is more robust than the basic one.

However, the new failbox implementation (Listing 12) may still crash, i.e., not function as intended in two cases as follows.

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 instructions at lines 26-29 in Listing 12 are now handled correctly. However, if an asynchronous exception happens in `enter` before the exception handler is set (line 17) 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.

Similar to the weaknesses identified in Section 5, the first point addresses the inability to detect a failure and the second one addresses the notification of other threads.

According to the above observations, the following assumption is still required: asynchronous exceptions do not occur when a thread is executing the code in the failbox `enter` method but has not yet set the exception handler, or when the handler is being executed.

In the next subsection we adjust our tests accordingly to point out that the new failbox implementation has the first vulnerability. Again, the second one will be addressed later.

6.2. Testing the limitation of the new failbox implementation

Test and tested programs. The implementation of the failbox with uncaught exception handler in Java may crash if an asynchronous exception happens in `enter` before the exception handler is set (line 17 in Listing 12). The only modification that we need to test this crashing point is to shift the synchronization code at lines 18-25 in Listing 12 before the point where the exception handler is set. Besides this modification, the rest of the setup is the same as in the previous section.

Output and results. The output of the test shows thread `threadTake` is not interrupted, i.e., the `fail` method is not called by `threadPut`.

7. 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 the 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 the guarantee that this method is executed regardless of the occurrence of uncaught interrupts. The code of the new `enter` method given in Listing 13 is a direct translation of the Java method in Listing 12, i.e., the corresponding Java calls are mapped to JNI calls.

The wrapper method calls with self-explanatory names in Listing 13 are used to 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, like `jboolean` at line 12. For instance, the method `do_SetUncaughtExceptionHandler_method` at line 10 is used to wrap the call of Java method `setUncaughtExceptionHandler`. The method `do_get_FailboxField_failed` at line 12 is used to retrieve the value of the field `failed` of `failbox` and its return value is assigned to the variable `failed_value` which is defined as a `jboolean` type in JNI.

The Java `synchronized` mechanism (line 26 in Listing 7) is replaced with JNI monitors to implement critical sections. A monitor is entered and exited by calling functions `do_MonitorEnter` (line 11 in Listing 13) and `do_MonitorExit` (line 18 in Listing 13), respectively. 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 at lines 12-17, corresponding to lines 26-29 in Listing 12.

Crucial for this implementation is the `ExceptionCheck()` method at line 19 in Listing 13. It is responsible for checking in the JVM whether any exceptions have occurred during the execution. In this way, we can check whether an asynchronous exception occurred.

The aforementioned functions `do_MonitorEnter` and `do_MonitorExit` are wrappers for the corresponding JNI functions `MonitorEnter` and `MonitorExit`. In the remainder of this article, 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 14.

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. For instance, the method `abort` is called at line 3 in Listing 14 when the call of JNI method `MonitorEnter` fails. If the call to the method `MonitorEnter` is successful, its return value is 0. Otherwise, its return value is a negative value.

In an analogous way, the `fail` method is implemented in JNI, see Listing 15, thereby further improving robustness. The `do_Failbox_fail` method called at line 13 in Listing 13 calls this JNI `fail` method. The failbox is marked as failed at line 5 (if it was not previously marked as failed), descendants are marked at lines 6-11, and threads are interrupted at lines 12-22. Compared to the Java version in Listing 7, the JNI implementation replaces the `synchronized` qualifier with the JNI `do_Monitor` methods to protect the shared fields. Also, after sending a notification to a thread in the failbox at line 16, the uncaught exception handler is unset at line 17.

```
JNIEXPORT void JNICALL Java_Failbox_enter(JNIEnv* env, jobject failbox, jobject currentThread, jobject
    body) {
  jstring name = do_Thread_getName_method(env, currentThread);
  const char* str = (*env)->GetStringUTFChars(env, name, NULL);
  if (strcmp(str, "Thread-0") == 0) {
    do_CountDownLatch_countDown_method(env, currentThread);
    do_CountDownLatch_await_method(env, currentThread);
  }
  printf("JNI: %s enters a failbox\n", str);
  fflush(stdout);
  do_SetUncaughtExceptionHandler_method(env, currentThread, failbox);
  do_MonitorEnter(env, failbox);
  jboolean failed_value = do_get_FailboxField_failed(env, failbox);
  if (failed_value == JNI_TRUE) {
    do_Throw_FailboxException(env, "FailboxFailed");
  }
  jobject threads = do_get_FailboxField_threads(env, failbox);
  do_ArrayList_add(env, threads, currentThread);
  do_MonitorExit(env, failbox);
  if ((*env)->ExceptionCheck(env) == JNI_TRUE) {
    printf("JNI: %s starts interrupting threads in failbox(es)\n", str);
    fflush(stdout);
    do_Failbox_fail(env, failbox);
  }
  do_Runnable_run(env, body);
  if ((*env)->ExceptionCheck(env) == JNI_FALSE)
    do_SetUncaughtExceptionHandler_method(env, currentThread, NULL);
  printf("JNI: %s exits a failbox\n", str);
  fflush(stdout);
}
```

Listing 13: The Java_Failbox_enter() C method.

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

Listing 14: The do_MonitorEnter() C method.

```
JNIEXPORT void JNICALL Java_Failbox_fail(JNIEnv* env, jobject failbox) {
  do_MonitorEnter(env, failbox);
  jboolean failed_value = do_get_FailboxField_failed(env, failbox);
  if (failed_value == JNI_FALSE) {
    do_set_FailboxField_failed(env, failbox, JNI_TRUE);
    jobject descendants = do_get_FailboxField_descendants(env, failbox);
    int arrayListSize = do_ArrayList_size(env, descendants);
    for (int i = 0; i < arrayListSize; i++) {
      jobject arrayElement = do_ArrayList_get(env, descendants, i);
      do_Failbox_fail(env, arrayElement);
    }
    jobject threads = do_get_FailboxField_threads(env, failbox);
    arrayListSize = do_ArrayList_size(env, threads);
    for (int i = 0; i < arrayListSize; i++) {
      jobject arrayElement = do_ArrayList_get(env, threads, i);
      do_Thread_interrupt(env, arrayElement);
      do_SetUncaughtExceptionHandler_method(env, arrayElement, NULL);
      jstring nameThread = do_Thread_getName_method(env, arrayElement);
      const char* strName = (*env)->GetStringUTFChars(env, nameThread, NULL);
      printf("%s is interrupted inside a failbox\n", strName);
      fflush(stdout);
    }
    do_ArrayList_clear(env, threads);
  }
  do_MonitorExit(env, failbox);
}
```

Listing 15: The Java_Failbox_fail() C method.

7.1. Testing the improvement of the new failbox implementation

The positive test result of the previous implementation can be remedied by introducing native C code via the Java Native Interface (JNI) to the implementation of the failbox with uncaught exception handler, as shown in Listing 13.

```
1  public class QueueExample {
2    public static void main(String[] args) {
3      Failbox f1 = new Failbox();
4      Failbox f2 = new Failbox();
5      f1.addDesc(f2);
6      LinkedBlockingQueue<String> queue = new LinkedBlockingQueue<String>();
7      CountDownLatch latchFirstA = new CountDownLatch(1);
8      CountDownLatch latchFirstB = new CountDownLatch(1);
9      CountDownLatch latchSecondA = new CountDownLatch(1);
10     CountDownLatch latchSecondB = new CountDownLatch(1);
11     Thread threadPut = new FailboxedThread(f1, new RunnableBlockPut(queue, latchFirstA, latchFirstB),
           latchSecondA, latchSecondB);
12     threadPut.start();
13     Thread threadTake = new FailboxedThread(f2, new RunnableBlockTake(queue), null, null);
14     threadTake.start();
15     new InterruptingThread(threadPut, latchFirstA, latchFirstB).start();
16     new InterruptingThread(threadPut, latchSecondA, latchSecondB).start();
17   }
18 }
```

Listing 16: Positive test: class QueueExample with Failbox and Latch.

Test and tested programs. In Listing 13 at lines 2-7, synchronization code is inserted at the potential crashing point before the exception handler is set.

Output and results. The thread `threadPut` calls the method `Java_Failbox_fail` in Listing 15 via the method `uncaughtException` of Listing 12 when an asynchronous exception occurs during its execution. That is, `threadTake` is notified by `threadPut` via the failboxes.

The improvement of the new implementation is demonstrated via our testing approach, i.e., Case 1 from the limitations analysis of the previous section is remedied. However, the JNI level approach presented in this section still retains one weak spot. We summarize the improvement and the limitation of the new implementation as follows:

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. The JNI level approach still has one weak spot. Namely, 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, as in Listing 12. If an asynchronous exception occurs in this method before the native implementation of `fail` has been started (line 38 in Listing 12), dependent 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.

7.2. Testing the remaining limitation of the new failbox implementation

In order to reveal the remaining weak spot in the implementation with the uncaught exception handler partially in native code, we modify the test scenario such that thread `threadPut` crashes twice: one crash occurs within the inner `try` block inside the failbox and the failbox is able to catch it; Another one happens within the method `uncaughtException` before the `fail` method is executed.

To be able to test these two crashes, we again use the synchronization approach in the `run` method of class `RunnableBlockPut` (lines 13-14 in Listing 4), remove the test code at lines 4-7 in Listing 13, and add synchronization points to the method `uncaughtException` (lines 4-11 in Listing 17). The updated `QueueExample` in Listing 16 is similar to Listing 10. In Listing 16, two `InterruptingThread` threads are created (lines 15-16) and two more latches are initialized (lines 7-10). The first interrupting thread at line 15 stops `threadPut` when it is waiting at line 14 in Listing 4. Then `threadPut` crashes and calls the `uncaughtException` method in Listing 17. During execution of this method, `threadPut` blocks again at line 9 until the second interrupting thread (line 16 in Listing 16) calls the `stop` method of `threadPut`. Note that in Listing 17, we need to call the `interrupted` method of thread `t` at line 5 to clear the interrupted flag of the thread. Otherwise, this thread, `threadPut` in our test scenario, will not block at line 9, as the interrupted flag will still be `true` once the first execution of the `stop` method of `threadPut` has completed.

The result of the test is positive, i.e., the output shows that `threadTake` is not notified about the failure of `threadPut`.

```
1  @Override
2  public void uncaughtException(Thread t, Throwable e){
3    System.out.println(th.getName() + " starts interrupting threads in failbox(es)");
4    if (t.getName().equals("Thread-0")) {
5      t.interrupted();
6      try {
7        FailboxedThread thread = (FailboxedThread) t;
8        thread.latchA.countDown();
9        thread.latchB.await();
10     } catch (InterruptedException ee) { }
11   }
12   fail();
13 }
```

Listing 17: Positive test: an Uncaught Exception Handler method.

8. A JNI implementation without Uncaught Exception Handler

To resolve the last remaining vulnerability of the JNI implementation with the uncaught exception handler from the previous section, we reconsider the initial implementation in Java given in Listing 7. We translate the `enter` method of this Java implementation into JNI as given in Listing 18.

This JNI implementation closely follows its Java counterpart from the basic implementation in Listing 7.

The `try-catch` instruction is replaced by an exception check by means of the JNI method `ExceptionOccurred` (line 15) and a conditional instruction (line 19). In particular, the inner `try` part (line 32 in Listing 7) is translated using the `do_Runnable_run` wrapper method (line 14). After checking for exceptions, the exception flag is cleared (line 16) such that new exceptions can be registered. The `finally` part (lines 33-26 in Listing 7) is merged with the `catch` part of the `try-catch` instruction into a critical section implemented using the wrapper methods already discussed in Section 7. In the critical section the current thread is removed unconditionally from the failbox (line 18). After that, if an exception has occurred (line 19), the original `catch` part is performed by calling the `do_Failbox_fail` method implementing the `fail` method on the JNI side (line 27). In the `do_Failbox_fail` method (lines 37-57), the `failed` flag is set at line 40, if it was not previously set, descendants are marked at lines 41-46 and the threads in the current failbox are interrupted at lines 47-56.

Since the native code cannot be interrupted by JVM (asynchronous) interrupts, this code ensures the correct notification of all dependent threads. Of course, a logical question that arises is “What if the execution of the native C code fails?”. For example, if the execution dereferences a null pointer or if it overflows the C stack. It 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. 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.

8.1. Testing the improvement of the new failbox implementation

To be able to compare the implementations with partially and fully native code, the test code should be added to the same location inside these two different implementations. In the previous case, the test code is added to the start of the handling exceptions part (lines 4-11 in Listing 17), i.e., the start of the Uncaught Exception Handler. In a similar way, the test code needs to be added to the implementation with fully native code, as shown at lines 22-26 in Listing 18 when the implementation in fully native code starts handling exceptions.

Output and results. The output of the test case for the implementation with fully native code shows that `threadTake` is notified by `threadPut`, i.e., the result is negative.

It is worth emphasizing that, although we test for an exception only in two locations of the code, in fact this covers occurrences of exceptions in all locations. Since the C code cannot be interrupted by an exception, it is irrelevant where during the execution of this code the exception has occurred. It is only important that an exception is registered and processed. During execution of the body (line 12) an exception is registered whenever it occurs. Applying our test schema such that it will enforce exceptions in (between) each of the instructions of the body does not increase the confidence in the test.

9. Overview of tests and results

In Sections 5-8, we demonstrated how to test implementations of the failbox concept using a synchronization mechanism with latches. With these tests, the improvements of each implementation can be demonstrated and also the limitations of each implementation (except for the full one in JNI) can be assessed. In this section, we show an overview of the tests and their outputs.

In fact, the combined test scenarios now cover all points where a failbox might not correctly handle exceptions.

```
1  JNIEXPORT void JNICALL Java_Failbox_enter(JNIEnv *env, jobject failbox, jobject currentThread, jobject
       body) {
2    jstring name = do_Thread_getName_method(env, currentThread);
3    const char* str = (*env)->GetStringUTFChars(env, name, NULL);
4    printf("JNI: %s enters a failbox\n", str);
5    fflush(stdout);
6    do_MonitorEnter(env, failbox);
7    jboolean failed_value = do_get_FailboxField_failed(env, failbox);
8    if (failed_value == JNI_TRUE) {
9      do_Throw_FailboxException(env, "FailboxFailed");
10   }
11   jobject threads = do_get_FailboxField_threads(env, failbox);
12   do_ArrayList_add(env, threads, currentThread);
13   do_MonitorExit(env, failbox);
14   do_Runnable_run(env, body);
15   jthrowable exception = (*env)->ExceptionOccurred(env);
16   (*env)->ExceptionClear(env);
17   do_MonitorEnter(env, failbox);
18   do_ArrayList_remove(env, threads, currentThread);
19   if (exception != NULL) {
20     printf("JNI: %s starts interrupting threads in failbox(es)\n", str);
21     fflush(stdout);
22     if (strcmp(str, "Thread-0") == 0) {
23       do_Thread_interrupted(env, currentThread);
24       do_CountDownLatch_countDown_method(env, currentThread);
25       do_CountDownLatch_await_method(env, currentThread);
26     }
27     do_Failbox_fail(env, failbox);
28   }
29   do_MonitorExit(env, failbox);
30   if (exception != NULL) {
31     do_Throw(env, exception);
32   }
33   printf("JNI: %s exits a failbox\n", str);
34   fflush(stdout);
35 }
36
37 void do_Failbox_fail(JNIEnv *env, jobject failbox) {
38   jboolean failed_value = do_get_FailboxField_failed(env, failbox);
39   if (failed_value == JNI_FALSE) {
40     do_set_FailboxField_failed(env, failbox, JNI_TRUE);
41     jobject descendants = do_get_FailboxField_descendants(env, failbox);
42     int arrayListSize = do_ArrayList_size(env, descendants);
43     for (int i = 0; i < arrayListSize; i++) {
44       jobject arrayElement = do_ArrayList_get(env, descendants, i);
45       do_Failbox_fail(env, arrayElement);
46     }
47     jobject threads = do_get_FailboxField_threads(env, failbox);
48     arrayListSize = do_ArrayList_size(env, threads);
49     for (int i = 0; i < arrayListSize; i++) {
50       jobject arrayElement = do_ArrayList_get(env, threads, i);
51       do_Thread_interrupt(env, arrayElement);
52       jstring nameThread = do_Thread_getName_method(env, arrayElement);
53       const char* strName = (*env)->GetStringUTFChars(env, nameThread, NULL);
54       printf("%s is interrupted inside a failbox\n", strName);
55       fflush(stdout);
56     }
57   }
58 }
```

Listing 18: The Java_Failbox_enter() and do_Failbox_fail() JNI methods of Class Failbox.

Table 1 demonstrates that the robustness of the implementation of the failbox concept is incrementally improved. The abbreviations in the first row represent different failbox implementations. **BF (Java)** represents the basic failbox implementation in Java. **UEHF (Java)** represents the failbox implementation using the Java uncaught exception handler. In **UEHF (JNI)**, the JNI is used together with the Java uncaught exception handler. Finally, **NUEHF (JNI)** indicates the failbox implementation in JNI without using the Java uncaught exception handler. The test at row 1 shows that if the test code, i.e., the synchronized mechanism with latches, is added immediately before the synchronized instruction inside each implementation, then the basic one can not handle the asynchronous exception but the one with the uncaught exception handler in Java can. We also use this test case to check the other two implementations, with and without the uncaught exception hander in JNI, which are not discussed in this article. The outputs of the tests show that these two implementations are also able to catch the asynchronous exception. At row 2, the test in Section 6 shows that the implementation with uncaught exception handler in Java is not able to catch the asynchronous exception when it occurs at the start of the `enter` method inside the implementation. However, this limitation is removed in the implementation with the uncaught exception handler in JNI, which is shown via the same test in Section 7.

Table 1
Overview of Tests with latches on Implementations of failboxes.

Location of the Test Code inside the failbox	BF (Java)	UEHF (Java)	UEHF (JNI)	NUEHF (JNI)
Immediately before the synchronized instruction	✗(III)	✓(IV)	✓	✓
At the start of the enter method	✗	✗(IV)	✓(V)	✓
At the start of the catch block	✗	✗	✗(V)	✓(VI)

[1] The checkmarks indicate the implementation is able to catch the asynchronous exception.
[2] The tests for different failbox implementations in gray cells are discussed in the corresponding sections from III to VI.
[2] The tests for different failbox implementations corresponding to white cells are not shown in this article.

Table 2
Overview of Outputs and Results on Tests.

Test	Output	Result
T_NBF	Thread-2 stops Thread-0	Positive
T_BF(Java)	Thread-0 enters a failbox Thread-1 enters a failbox Thread-2 stops Thread-0 Thread-0 starts interrupting threads in failbox(es) Thread-1 is interrupted by Thread-0 inside a failbox Thread-1 exits a failbox	Negative
T_L_BF(Java)	Thread-0 enters a failbox Thread-1 enters a failbox Thread-2 stops Thread-0	Positive
T_UEHF(Java)	Thread-0 enters a failbox Thread-1 enters a failbox Thread-2 stops Thread-0 Thread-0 starts interrupting threads in failbox(es) Thread-1 is interrupted by Thread-0 inside a failbox Thread-1 exits a failbox	Negative
T_L_UEHF(Java)	Thread-0 enters a failbox Thread-1 enters a failbox Thread-2 stops Thread-0	Positive
T_UEHF(JNI)	JNI: Thread-0 enters a failbox JNI: Thread-1 enters a failbox Thread-2 stops Thread-0 Thread-0 starts interrupting threads in failbox(es) JNI: Thread-1 is interrupted inside a failbox JNI: Thread-0 is interrupted inside a failbox JNI: Thread-1 exits a failbox JNI: Thread-0 exits a failbox	Negative
T_L_UEHF(JNI)	JNI: Thread-0 enters a failbox JNI: Thread-1 enters a failbox Thread-0 starts interrupting threads in failbox(es) Thread-3 stops Thread-0 Thread-2 stops Thread-0	Positive
T_NUEHF(JNI)	JNI: Thread-0 enters a failbox JNI: Thread-1 enters a failbox JNI: Thread-0 starts interrupting threads in failbox(es) JNI: Thread-1 is interrupted inside a failbox JNI: Thread-0 exits a failbox Thread-2 stops Thread-0 Thread-3 stops Thread-0 JNI: Thread-1 exits a failbox	Negative

Table 2 provides all results of the tests in this article. From the outputs printed on the console, we identify whether or not the notification part inside the failbox is executed, in other words, whether the implementation is able to handle the asynchronous exception. Note that the abbreviation **T_NBF** in the first row represents the test for testing without the use of failboxes.

Except for this abbreviation, the rest of abbreviations in the first column represent tests for testing the improvements and limitations of different failbox implementations in Java or in JNI. For instance, **T_BF(Java)** represents the test for testing

the basic Java implementation of failbox while **T_L_BF(Java)** represents the test for testing the limitation of the basic Java implementation of failbox.

10. Conclusions and future work

We presented a Java implementation of the concept of failbox from [3] and incrementally improved it by eliminating the assumption required in the original failboxes implementation. The assumption is eliminated in an incremental manner through four distinct increasingly more robust implementations. For each implementation we analyzed the vulnerabilities and argued the remedies in the next implementation.

We also presented a testing approach to investigate whether the vulnerabilities of each implementation of the failbox concept are realistic and the remedies proposed in the next implementation are effective. This testing approach enables us to generate asynchronous exceptions in a controlled manner for concurrent programs. The tests are repeatable in that they give the same results for runs that may differ in scheduling, even on different platforms.

Currently we are working in a project on transforming models written in a domain specific language to robust multi-threaded Java programs [25–31]. Future work involves integrating the final failbox implementation, as proposed in this article, into the project framework. Additionally, we plan to formally verify the proposed failbox implementations, and compare 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 [32]. VeriFast is a code verification tool that can be used to verify both multi-threaded Java and C programs. VeriFast does not currently support exception instructions like `try-catch`, but since they are not present in the C translation, this makes verification possible. Furthermore, VeriFast allows the verification of deadlock absence, which is an important aspect of dependency safety [3].

A direction for further research would be to make the approach more general, e.g., determine where synchronization can be applied, and which effect it will have: how to on the one hand precisely target vulnerable points and on the other hand not incur false positives, like getting deadlock as a test result because of a side effect of synchronization by the test rather than because of a property of the program.

Another direction to further develop the approach would be to automatically generate synchronization code, and even the complete tests, for given code with places in it that are identified as possibly vulnerable for asynchronous exceptions.

In [33], a language and tool are presented to provide test scenarios that, at a rather abstract level, are similar to ours. It would be interesting to investigate if our tests could be generated by that tool. To improve the modularity and generality of our scenario descriptions we consider using tools like AspectJ [34].

As already emphasized, in this article we aimed at an implementation ensuring wait dependency safety which only partially covers consistency dependency safety. The crucial point is that for wait dependency safety the dependent instructions are blocked and as a consequence the threads that attempt to execute them are notified on time. However, for consistency dependency safety it is not necessary that the dependent instructions are blocked, so race conditions may occur and a thread can execute a instruction on inconsistent data. In the original failbox paper [1] this is avoided by a clever definition of the failbox semantics and the fact that a simpler language than Java is used, as a proof of concept. To implement dependency safety with failboxes in Java and similar languages, probably one will need to change the semantics of the language features implementing critical sections, like locks, semaphores and monitors. More precisely, before entering a critical section a thread will need to check the status of the failbox and enter the section only in case the failbox has not failed.

References

[1] B. Jacobs, F. Piessens, Failboxes: provably safe exception handling, in: ECOOP, Springer, 2009, pp. 470–494.
[2] J. Gosling, B. Joy, G. Steele, G. Bracha, A. Buckley, The Java Language Specification, Java SE 8 edition, 2015.
[3] B. Jacobs, Provably live exception handling, in: FTfJP, ACM, 2015, pp. 7:1–7:4.
[4] Java Native Interface Specification, http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/jniTOC.html. (Accessed 17 November 2011).
[5] D. Bošnački, M. van den Brand, P. Denissen, C. Huizing, B. Jacobs, R. Kuiper, A. Wijs, M. Wilkowski, D. Zhang, Dependency safety for Java: implementing failboxes, in: PPPJ: Virtual Machines, Languages, and Tools, ACM, 2016, pp. 15:1–15:6.
[6] J. Armstrong, Making Reliable Distributed Systems in the Presence of Software Errors, Ph.D. thesis, Royal Institute of Technology, Stockholm, Sweden, 2003.
[7] S. Toub, Keep your code running with the reliability features of the .NET framework, in: MSDN Magazine, October, 2005.
[8] B. Jacobs, P. Müller, F. Piessens, Sound reasoning about unchecked exceptions, in: SEFM, IEEE, 2007, pp. 113–122.
[9] M. Bagherzadeh, H. Rajan, M.A.D. Darab, On exceptions, events and observer chains, in: AOSD, ACM, 2013, pp. 185–196.
[10] H. Rebêlo, R. Coelho, R.M.F. Lima, G.T. Leavens, M. Huisman, A. Mota, F. Castor, On the interplay of exception handling and design by contract: an aspect-oriented recovery approach, in: FTfJP, ACM, 2011, pp. 7:1–7:6.
[11] T. Harris, J. Larus, R. Rajwar, Transactional Memory, 2nd edn., Synthesis Lectures on Computer Architecture, Morgan & Claypool Publishers, 2010.
[12] N. Shavit, D. Touitou, Software transactional memory, in: PODC, ACM, 1995, pp. 204–213.
[13] M. Herlihy, V. Luchangco, M. Moir, W.N. Scherer III, Software transactional memory for dynamic-sized data structures, in: PODC, ACM, 2003, pp. 92–101.
[14] M. Herlihy, J. Moss, Transactional memory: architectural support for lock-free data structures, in: ISCA, ACM, 1993, pp. 289–300.
[15] P. Felber, C. Fetzer, V. Gramoli, D. Harmanci, M. Nowack, Safe exception handling with transactional memory, in: Transactional Memory. Foundations, Algorithms, Tools, and Applications, 2015, pp. 245–267.
[16] B. Cabral, P. Marques, Implementing retry - featuring AOP, in: Proc. LADC 2009, IEEE Computer Society, 2009, pp. 73–80.
[17] T. Harris, Exceptions and side-effects in atomic blocks, Sci. Comput. Program. 58 (2005) 325–343.

[18] T. Harris, S. Marlow, S. Peyton-Jones, M. Herlihy, Composable memory transactions, in: PPoPP, ACM, 2005, pp. 48–60.
[19] A. Shinnar, D. Tarditi, M. Plesko, B. Steensgaard, Integrating Support for Undo with Exception Handling, Technical report msr-tr-2004-140, Microsoft Research, 2004.
[20] R. Guerraoui, M. Kapałka, On the correctness of transactional memory, in: PPoPP, ACM, 2008, pp. 175–184.
[21] G. Lagorio, M. Servetto, Strong exception-safety for checked and unchecked exceptions, J. Object Technol. 10 (1) (2011) 1–20.
[22] N.D. Matsakis, T.R. Gross, Handling errors in parallel programs based on happens before relations, in: PDP, IEEE, 2010, pp. 1–8.
[23] S. Tasharofi, R. Johnson, Patterns in Testing Concurrent Programs with Non-deterministic Behaviors, Tech. rep., Department of Computer Science, University of Illinois at Urbana-Champaign, 2011.
[24] Testing Asynchronous Code, https://www.javacodegeeks.com/2015/10/testing-asynchronous-code.html. (Accessed 20 September 2016).
[25] D. Zhang, D. Bošnački, M. van den Brand, L. Engelen, C. Huizing, R. Kuiper, A. Wijs, Towards verified Java code generation from concurrent state machines, in: AMT, CEUR-WS.org, 2014, pp. 64–69.
[26] D. Bošnački, M. van den Brand, J. Gabriels, B. Jacobs, R. Kuiper, S. Roede, A. Wijs, D. Zhang, Towards modular verification of threaded concurrent executable code generated from DSL models, in: FACS, in: LNCS, vol. 9539, Springer, 2015, pp. 141–160.
[27] A. Wijs, M. Wiłkowski, Modular indirect push-button formal verification of multi-threaded code generators, in: SEFM, in: LNCS, vol. 11724, Springer, 2019, pp. 410–429.
[28] A. Wijs, L. Engelen, Efficient property preservation checking of model refinements, in: TACAS, in: LNCS, vol. 7795, Springer, 2013, pp. 565–579.
[29] A. Wijs, L. Engelen, REFINER: towards formal verification of model transformations, in: NFM, in: LNCS, vol. 8430, Springer, 2014, pp. 258–263.
[30] S. de Putter, A. Wijs, D. Zhang, The SLCO framework for verified, model-driven construction of component software, in: FACS, in: LNCS, vol. 11222, Springer, 2018, pp. 288–296.
[31] S. de Putter, A. Wijs, A formal verification technique for behavioural model-to-model transformations, Form. Asp. Comput. 30 (1) (2018) 3–43.
[32] B. Jacobs, D. Bosnacki, R. Kuiper, Modular Termination Verification: Extended Version, CW Reports 680, Katholieke Universiteit Leuven, 2015.
[33] V. Jagannath, M. Gligoric, D. Jin, Q. Luo, G. Rosu, D. Marinov, Improved multithreaded unit testing, in: ACM SIGSOFT/FSE, ACM, 2011, pp. 223–233.
[34] AspectJ, http://www.eclipse.org/aspectj. (Accessed 17 November 2011).