

# Subsystems: Provably Safe Exception Handling (Status Report)

*Bart Jacobs*      *Frank Piessens*

*Report CW516, May 2008*



Katholieke Universiteit Leuven  
Department of Computer Science

Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

# Subsystems: Provably Safe Exception Handling (Status Report)

*Bart Jacobs*      *Frank Piessens*

*Report CW 516, May 2008*

Department of Computer Science, K.U.Leuven

## Abstract

The primary goal of exception mechanisms is to help ensure that when an operation fails, code that depends on the operation's successful completion is not executed (a property we call *dependency safety*). However, current exception mechanisms make it hard to achieve dependency safety, in particular when objects manipulated inside a try block outlive the try block.

To remedy this, we propose a language mechanism called subsystems. Programmers may create subsystems dynamically and execute blocks of code in them. Once any such block fails, all subsequent attempts to execute code in the subsystem will fail. To achieve dependency safety, programmers simply need to ensure that if an operation *B* depends on an operation *A*, then *A* and *B* are executed in the same subsystem. Furthermore, subsystems help fix the unsafe interaction between locks and exceptions and they enable safe cancellation and robust resource cleanup. We have implemented the mechanism as a C# library, and we show that the constructs have low performance overhead.

**Keywords :** Cancellation, data consistency, exception handling, locking, subsystems.

**CR Subject Classification :** D.3.3.

# Subsystems: Provably Safe Exception Handling (Status Report)

Bart Jacobs\*    Frank Piessens

Department of Computer Science, Katholieke Universiteit Leuven, Belgium

{bart.jacobs,frank.piessens}@cs.kuleuven.be

## Abstract

The primary goal of exception mechanisms is to help ensure that when an operation fails, code that depends on the operation's successful completion is not executed (a property we call *dependency safety*). However, current exception mechanisms make it hard to achieve dependency safety, in particular when objects manipulated inside a try block outlive the try block.

To remedy this, we propose a language mechanism called subsystems. Programmers may create subsystems dynamically and execute blocks of code in them. Once any such block fails, all subsequent attempts to execute code in the subsystem will fail. To achieve dependency safety, programmers simply need to ensure that if an operation  $B$  depends on an operation  $A$ , then  $A$  and  $B$  are executed in the same subsystem. Furthermore, subsystems help fix the unsafe interaction between locks and exceptions and they enable safe cancellation and robust resource cleanup. We have implemented the mechanism as a C# library, and we show that the constructs have low performance overhead.

**Categories and Subject Descriptors** D.3.3 [*Language constructs and features*]: Control structures

**General Terms** Reliability, Languages

**Keywords** Cancellation, data consistency, exception handling, locking, subsystems

## 1. Introduction

If a program is seen as a state machine, a programmer's job may be seen as writing code to deal with each of the states that the program may reach. However, programmer time is limited and some states are less likely to occur during

---

\* Bart Jacobs is a Postdoctoral Fellow of the Research Foundation - Flanders (FWO)

production than others. Therefore, in many projects it is useful to designate the most unlikely states as *failure states* and to deal with all failure states in a uniform way, while writing specific code only for non-failure (or *normal*) states.

An extreme form of this approach is to simply ignore failure states and not care what the program does when it reaches a failure state (i.e., when it *fails*). This is often what happens when subroutines indicate failure conditions as special return values, and programmers have no time to write code at call sites to check for them.

A major problem with this approach is that it is *unsafe*: a failure may lead to the violation of any and all of the program's intended safety properties. Specifically, the approach violates *dependency safety*, the property which says that when an operation fails, code that depends on the operation's successful completion is not executed.

To fix this, modern programming languages offer constructs that make it easy for programmers to indicate that a state is a failure state, and deal with failure states by terminating the program by default. The underlying assumption is that termination is always safe. For example, in Java, a failure state is indicated by throwing an unchecked exception. We will focus on the Java language in this paper; the related work section discusses other languages.

Whereas by default, when a program throws an exception it terminates immediately, the programmer can override this default through the use of try-catch statements and try-finally statements. Furthermore, in a multithreaded program, when a thread's main method completes abruptly (i.e., an exception was thrown and not caught during its execution), only that thread, not the entire program, is terminated. Also, when a synchronized block's body completes abruptly, the lock is released before the exception is propagated further.

These deviations from strict termination behavior are useful and are used for two reasons. Firstly, not all exceptions indicate failure. Sometimes, programmers throw and catch exceptions to implement the program's functional behavior. Typically, in Java, checked exceptions are used for this. Secondly, programmers sometimes wish to increase the program's robustness by not considering the program to be a single unit of failure but rather by identifying multiple smaller units of failure. Common examples are extensible programs, where poorly written or malicious plugins

(such as applets or servlets) should not affect the base system; and command-processing applications (such as request-response-based servers, GUI applications, or command-line shells) where a failure during the processing of a command should simply cause an error response to be returned, while continuing to process other commands normally.

However, by continuing to execute after a failure, the risk of safety violations reappears. In particular, safety violations are likely if the code that fails leaves a data structure in an inconsistent state and this data structure is then accessed during execution of a finally block or after the exception is caught, or by another thread. In other words, there is a safety risk if a try block manipulates an object that outlives the try block. More generally, dependency safety may be violated if pieces of code outside a try block depend on particular pieces of code inside the try block either not executing at all or executing to completion successfully. This is the problem addressed in this paper.

The rest of the paper is structured as follows. In Section 2, we illustrate the problem with an example and discuss existing approaches. In Section 3, we introduce our proposed approach to address this problem, called *subsystems*. We show additional aspects and benefits of the approach for multithreaded programs in Section 4. Sections 5 and 6 discuss how the approach enables safe cancellation and robust resource cleanup, respectively. We sketch a proof of the safety of the approach in Section 7. We end the paper with sections on implementation issues (Section 8), related work (Section 9), and a conclusion (Section 10).

This paper does not report on the usability and applicability of the approach for large programs. Such an assessment is important future work.

## 2. Problem Statement

Consider the example program in Figure 1. It shows a program that continuously receives commands and processes them. The code for processing commands is not shown, except that it involves calls of *compute* and calls of *addEntry* on a *Database* object *db* that is shared across all command executions. If the processing of a command fails, e.g. because it requires too much memory, the exception is caught, an error message is shown to the user, and the next command is received.

This program is unsafe. Specifically, some executions of this program violate the intended safety property that at the start of each loop iteration, object *db* is *consistent*, i.e., satisfies the property that *count* is not greater than the length of *entries*. In particular, consider an execution where method *addEntry* is called in a state where *entries* is full. This means *count* equals *entries.length*. As a result, after incrementing *count*, *addEntry* will attempt to allocate a new, larger array. Now assume there is not enough memory for this new array and an *OutOfMemoryError* occurs at location *A*. At this point, *count* is greater than the length

```
class Database {
    int count;
    int[] entries := new int[10];
    /* invariant: count ≤ entries.length */
    void addEntry(int entry) {
        count++;
        if (count = entries.length + 1) {
            int[] es := new int[count * 2]; // *** A ***
            System.arraycopy(
                entries, 0, es, 0, entries.length);
            entries := es;
        }
        entries[count - 1] := entry; // *** B ***
    }
    ...
}

class Program {
    public static void main(
        String[] args) {
        Database db := new Database();
        while (true)
            /* invariant: db is consistent */
            {
                String cmd := readCommand();
                try {
                    ... compute(cmd); ...
                    ... db.addEntry(...); ...
                } catch (Throwable e) {
                    showErrorMessage(e);
                }
            }
        ...
    }
}
```

**Figure 1.** An unsafe program. An unchecked exception (e.g. an *OutOfMemoryError*) in *compute* is handled correctly, but if an *OutOfMemoryError* occurs at location *A*, the *Database* object is left in an inconsistent state, violating the safety of subsequent loop iterations.

of *entries* and the *Database* object is inconsistent. Next, the exception is caught in method *main* and the loop is continued, violating the safety property.

Note: In this case, the safety violation results in an *ArrayIndexOutOfBoundsException* at location *B* in each subsequent call of *addEntry*; however, in general, safety violations might remain undetected and lead to data corruption, incorrect results, or sending incorrect commands to hardware devices.

The following approaches exist to deal with this complication:

- **Never catch unchecked exceptions.** Never catching unchecked exceptions makes it easier to preserve safety properties, since the many implicit control flow paths created by catching unchecked exceptions are avoided. However, catching unchecked exceptions can be useful, as in the example. Note also that **try-finally** blocks are equivalent to **try-catch** blocks that catch unchecked exceptions; specifically, assuming  $S_1$  does not jump out

of the `try` block, a statement

```
try { S1 } finally { S2 }
```

is equivalent to

```
try { S1 } catch (Throwable t) { S2 throw t; } S2
```

and is subject to the same complication:  $S_2$  might depend on the successful completion of certain sub-computations within  $S_1$ . Never catching unchecked exceptions would imply never using `try-finally` blocks, or modifying their semantics so that they ignore unchecked exceptions. The semantics of synchronized blocks would need to be updated similarly.

- **Always maintain consistency.** It is often possible to ensure that objects used across try-catch blocks, like the *Database* object in the example, are in a consistent state at all times. Often it is sufficient to reorder assignments; e.g., in the example, moving the count increment after the assignment to *entries* preserves consistency. Another approach is to use a functional programming-like approach, where a new object state is built up separately and then installed into the object using a single assignment. In the example, method *addEntry* would return a new *Database* object rather than updating the existing one. Yet another approach is to use transaction-like technologies, such as software transactional memory (Shavit and Touitou 1995; Fetzer et al. 2003). However, these approaches either require the programmer to perform non-trivial additional reasoning and/or programming work, or impose a non-trivial performance overhead.
- **Never fail during critical sections.** It might be possible in some cases to guarantee absence of failure at points where failure would violate safety. This requires careful programming to avoid operations that might encounter resource or implementation limitations, such as heap or stack memory allocations or operations on bounded integers, or to move these operations out of the critical section. Furthermore, this might require virtual machine support if the virtual machine may perform resource allocations implicitly. For example, the .NET Framework’s JIT compiler may allocate memory at any time to store a newly compiled piece of code. Therefore, starting with version 2, the .NET Framework offers constructs to “prepare” a piece of code that must execute without failure (Toub 2005). However, this approach imposes a significant burden on the programmer.
- **Ensure dependent code is not executed.** In this approach, steps are taken to ensure that if a computation fails with an unchecked exception, then no computations that depend on the failed computation’s successful completion ever get to run. There are at least two ways to achieve this:

- **Use separate threads.** In this approach, threads are adopted as the units of failure. Within a thread, unchecked exceptions are never caught; that is, an exception in the thread causes the entire thread to die. All data structures are local to threads. Instead of running a block of code in a try-catch block, it is run in a separate thread. During this time, the original thread waits for the termination of the child thread; additionally, the original thread may accept messages on a message queue. If the child thread needs to perform an operation whose failure should cause the parent thread to fail (such as an *addEntry* call on the *Database* object), the child thread may perform a remote procedure call into the parent thread via the parent thread’s message queue. This is more or less the approach used in operating systems, in the Erlang language (Armstrong 2003), and in the SCOOP multithreading approach for Eiffel (Meyer 1992).
- **Guard dependent code manually.** The programmer can manually arrange to ensure that dependent code is not executed. For example, the programmer could associate a boolean flag with each object used across try-catch blocks that tracks whether the object is in a consistent state, and check this flag before accessing the object (Jacobs et al. 2007). If the flag is false, an exception is thrown.

In this paper we present a new approach in the fourth category, which, like the use of separate threads and manually guarding dependent code, supports catching exceptions and does not require that consistency be maintained always or that failures be avoided, but which has less programming and run-time overhead than the use of separate threads and which has less programming overhead than manually guarding dependent code.

### 3. Subsystems

In our approach, the language is extended with a notion of *subsystems*. Constructs are added for creating a new subsystem and for running a piece of code in a designated subsystem. As soon as one such piece of code fails, any subsequent attempt to run code in the subsystem fails. To ensure dependency safety, the programmer simply needs to ensure that if a computation  $B$  depends on a computation  $A$ , then  $A$  and  $B$  run in the same subsystem.

The basic subsystems constructs are as follows. (Note: In this section, we consider only single-threaded programs. Furthermore, we present a simplified semantics. The next section deals with multithreaded programs and introduces the full semantics, including subsystem hierarchies and implicitly created subsystems.)

- A new subsystem is created using a `new Subsystem()` expression; this yields an instance of type *Subsystem* that reflects the newly created subsystem.

- A block  $S$  is executed in a subsystem  $s$  using an

`enter (s) S`

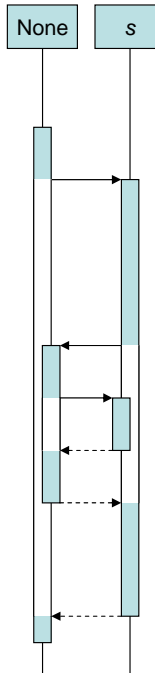
statement, where  $s$  is an expression of type *Subsystem*. Specifically, the semantics of this statement is as follows (using the terminology of the Java Language Specification):

- First, if  $s$  has been marked as failed, the statement completes abruptly with a *SubsystemException*.
- Otherwise,  $s$  is designated the current subsystem.
- Then,  $S$  is executed.
- Then, if some subsystem  $s'$  was designated the current subsystem prior to execution of the enter statement,  $s'$  is again designated the current subsystem.
- Then, if the execution of  $S$  completed abruptly because of an unchecked exception,  $s$  is marked as failed and the enter statement completes abruptly because of the same exception.
- Otherwise, if some subsystem  $s'$  was designated the current subsystem prior to the enter statement, and  $s'$  is marked as failed, the enter statement completes abruptly with a *SubsystemException*.
- Otherwise, if the execution of  $S$  completed normally, the enter statement completes normally.
- Otherwise, if the execution of  $S$  completed abruptly for a reason other than an unchecked exception (such as a checked exception or a return), the enter statement completes abruptly for the same reason.

Furthermore, the semantics of try-catch statements is adapted as follows. An execution of a try-catch statement `try S catch (T e) S'` occurs according to the following steps:

- First, if any subsystem is designated the current subsystem, this designation is cancelled and there is no longer a current subsystem.
- Then, the try block  $S$  is executed.
- Then, if some subsystem  $s$  was designated the current subsystem prior to the try-catch statement, the following steps are taken:
  - First,  $s$  is again designated the current subsystem.
  - Then, if  $s$  is marked as failed, the following steps are skipped and the try-catch statement completes abruptly with a *SubsystemException*.
- Then, if the execution of  $S$  completed abruptly because of an exception  $E$  that is assignable to type  $T$ , the catch block  $S'$  is executed with local variable  $e$  bound to  $E$ , and the try-catch statement completes in the same way as  $S'$ .

### Subsystem



```
class Program {
  public static void main(String[] args) {
    Subsystem s := new Subsystem();
    enter (s) {
      Database db := new Database();
      while (true)
        /* invariant: db is consistent */
        {
          String cmd := readCommand();
          try {
            ... compute(cmd); ...
            enter (s) {
              db.addEntry(...);
            }
            ...
          } catch (Throwable e) {
            showErrorMessage(e);
          }
        }
    }
  }
}
```

**Figure 2.** The example of Figure 1, fixed using subsystems. When an `addEntry` call fails, subsystem  $s$  is marked as failed. When control subsequently exits the try block, this is considered an attempt to enter  $s$ ; therefore, a *SubsystemException* is thrown. As a result, the catch block is skipped, the loop is exited, and the program terminates safely. The sequence diagram shows the subsystem transitions.

- Otherwise, the try-catch statement completes in the same way as  $S$ .

At any one time, at most one subsystem is designated the current subsystem. Initially, there is no current subsystem. We use the following terminology: We say that an event occurs *in a subsystem*  $s$  or a statement is executed (or executes) in  $s$  if the event occurs or the statement execution starts at a time when  $s$  is designated the current subsystem. We say that *a failure occurs* when an unchecked exception is thrown. We say that a statement execution *fails* if it completes abruptly because of an unchecked exception. We say that a subsystem  $s$  *fails* when a failure occurs in  $s$ . We say that an execution step *enters* a subsystem  $s$  if  $s$  is the current subsystem after the step and was not the current subsystem before the step. Similarly, we say that an execution step *leaves* a subsystem  $s$  if it is not the current subsystem after the step and was the current subsystem before the step.

Notice that the above semantics guarantees that if a subsystem fails, then it is marked as failed. Furthermore, it guarantees that if a subsystem fails, no code ever executes in that subsystem again.

The approach is illustrated and motivated by the example in Figure 2. It shows how the unsafe program of Figure 1 can be made safe using subsystems. A subsystem  $s$  is created and then both the main loop and calls of `addEntry` are executed in  $s$ . This ensures that if a call of `addEntry` fails, the main loop terminates.

The example motivates why on entry to a try block, the subsystem in which the try-catch statement executes is no longer considered the current subsystem. This ensures that failures in method `compute` are properly caught by the try-catch statement, and do not cause the program to terminate.

An implementation approach for the simplified subsystems approach of this section is shown in Figure 3. It shows how the approach could be implemented as a combination of a library class and a compiler extension. In this approach, the compiler expands the enter statements and try-catch statements of the approach to regular Java constructs. Note: This scheme does not take into account the possibility that a try block or the body of an enter statement completes abruptly because of a return, a break, or a continue. Also, it assumes that no asynchronous exceptions (i.e., `InternalError` or `UnknownError` instances in Java, which may occur at any time) occur within the code of the expansion. The implementation issues caused by asynchronous exceptions are discussed in Section 8.

Our approach can be used to ensure dependency safety. Specifically, we define dependency safety as the property that whenever two computations (e.g. statement executions)  $A$  and  $B$  occur in a program execution, in this order, and  $B$  depends on  $A$ , then  $A$  does not fail in this execution. The notion of whether a computation  $B$  depends on a computation  $A$  is application-dependent; it means that the safety of  $B$  depends on the successful completion of  $A$ , or, in other words, that executing  $B$  after  $A$  fails might violate the program's intended safety properties. A typical dependency relation is the relation between a computation that updates a data structure, and a computation that subsequently accesses this data structure.

Given a dependency relation, we say that a program *uses subsystems correctly* if whenever a computation  $B$  depends on a computation  $A$ ,  $A$  and  $B$  execute in the same subsystem. Furthermore, given a program that uses subsystems, we call the *erasure* of this program the program obtained by replacing each enter statement with its body. We then have that each execution of a program that correctly uses subsystems is an execution of the erased program and is dependency-safe. (This characterization assumes that the program does not distinguish between `UnknownError` instances and `SubsystemException` instances, so that if an execution of the original program throws a `SubsystemException`, this corresponds with an execution of the erased program that throws an `UnknownError` exception.) We call this property the soundness of the approach. What this means is that when reasoning about pro-

```

class Subsystem {
    static Subsystem current;
    boolean failed;
}

Expansion[[ try { S } catch ( T e ) { S' } ]] =
    Subsystem s := Subsystem.current;
    T t := null;
    try {
        Subsystem.current := null;
        try {
            S
        } finally {
            Subsystem.current := s;
        }
    } catch ( T e ) {
        t := e;
    }
    if ( s ≠ null ∧ s.failed )
        throw new SubsystemException();
    if ( t ≠ null ) { T e := t; S' }

Expansion[[ enter ( s ) { S } ]] =
    if ( s.failed )
        throw new SubsystemException();
    Subsystem olds := Subsystem.current;
    try {
        Subsystem.current := s;
        try {
            S
        } finally {
            Subsystem.current := olds;
        }
    } catch ( RuntimeException e ) {
        s.failed := true;
        throw e;
    } catch ( Error e ) {
        s.failed := true;
        throw e;
    }
    if ( olds ≠ null ∧ olds.failed )
        throw new SubsystemException();

```

**Figure 3.** A scheme for translating the proposed language extension of Section 3 to Java

```

class Program {
  public static void main(String[] args) {
    final Database db := new Database();
    while (true) {
      final String cmd := readCommand();
      new Thread() {
        public void run() {
          try {
            ... compute(cmd); ...
          } catch (Throwable e) {
            db.addEntry(...);
            showErrorMessage(e);
          }
        }
      }.start();
    }
  }
}

```

**Figure 4.** An unsafe program. A failure in *compute* is handled correctly, but if a failure occurs in method *addEntry* while the *Database* object is inconsistent, the object’s lock is released, causing threads that subsequently acquire the lock to see the object in an unexpected state, violating safety.

grams that use subsystems, one can reason using either the semantics described above, or the soundness property, or both. Another way to look at it is that, given a program that does not use subsystems and that is not dependency-safe, one can add enter statements to prune those executions that are not dependency-safe.

The soundness property can be applied to the example to prove that the loop invariant holds at each loop iteration. For this purpose, we consider each loop iteration to depend on each preceding *addEntry* call. Since both each loop iteration and each *addEntry* call execute in subsystem *s*, the program uses subsystems correctly. Therefore, by the soundness of the approach, one can derive, by reasoning about those executions of the erased program that are dependency-safe, that the loop invariant holds at each loop iteration of the program.

#### 4. Multithreading

One common way that the strict termination approach of dealing with failures is overridden, is through the use of *synchronized* blocks. A *synchronized* (*o*) *S* block in Java acquires the lock of object *o*, executes statement *S*, and then releases the lock of *o*, even if *S* failed. This helps prevent deadlocks, but it creates a safety risk. In particular, if a failure occurs while *o* is inconsistent, the commonly intended safety property that shared objects whose lock is not held are consistent, is violated.

The problem is illustrated by the example program in Figure 4. It is a multithreaded version of the original example in Figure 1. Rather than processing each command before receiving the next command, the program receives a com-

```

class Program {
  public static void main(String[] args) {
    final Subsystem s := new Subsystem();
    enter (s) {
      final Database db := new Database();
      while (true) {
        final String cmd := readCommand();
        new Thread() {
          public void run() {
            try {
              ... compute(cmd); ...
            } synchronized (db) {
              enter (s) { db.addEntry(...); }
            }
          } catch (Throwable e) {
            showErrorMessage(e);
          }
        }
      }.start();
    }
  }
}

```

**Figure 5.** The example of Figure 4, made safe using subsystems. If a call of *addEntry* fails, subsystem *s* is marked as failed and subsequent attempts by other threads to enter the subsystem will fail. Furthermore, by the Fail Fast feature, a *stop s* signal is sent to all threads running in the failed subsystem *s* or a descendant of *s*. In the example, this means the program terminates.

mand, spawns a thread to process it, and immediately receives the next command. The *Database* object is shared by all command processing threads; accesses to the object are synchronized using a *synchronized* block.

This program is unsafe. In particular, in some executions, the intended safety property that whenever a shared object’s lock is not held by any thread, the object is consistent, is violated. This property is relied on to guarantee that method *addEntry* is called only on objects that are consistent. Specifically, suppose a failure occurs in method *addEntry* while the *Database* object is inconsistent. This causes the lock to be released. Subsequent command processing threads that acquire the lock will then see the *Database* object in an inconsistent state.

Subsystems can be used to write safe lock-based multithreaded programs, by associating each shared object with a subsystem and running the code that accesses a shared object within the associated subsystem. This way, when a failure occurs, the subsystem is marked as failed, so that when another thread subsequently attempts to enter the subsystem in order to access the object, an exception is thrown and the thread is prevented from seeing inconsistent state. The modified safety property is that whenever no thread holds a shared object’s lock, either the object is consistent or its associated subsystem is marked as failed.

The approach is illustrated in Figure 5. It is the example of Figure 4, made safe using subsystems. The main thread creates a subsystem  $s$  and both the loop in the main thread and the `addEntry` calls in the command processing threads are executed in this subsystem. When an `addEntry` call fails,  $s$  is marked as failed before the lock is released. When another thread subsequently acquires the lock and attempts to enter  $s$ , an exception is thrown, so that the thread is prevented from unsafely calling `addEntry`.

In a multithreaded program, it is possible for computations in multiple threads to be executing in the same subsystem  $s$  concurrently. The question then arises as to what happens when one of these computations fails. In this matter, our approach adopts the Fail Fast principle (Shore 2004). The fact that the computations are running in the same subsystem is taken to mean that they depend on each other for useful progress. As a result, if one of them fails, there is no point for the others to continue. Therefore, in our approach, the thread where the failure occurred throws an asynchronous exception in the other threads that are running in  $s$ . We allow the signal to be delivered asynchronously, to allow efficient implementations.

Note that the Fail Fast feature serves only to stop computations that can no longer make useful progress; it is not intended and not needed for ensuring safety properties. Below we state the soundness theorem for the multithreaded approach; it does not rely on the Fail Fast feature.

The usefulness of this Fail Fast feature is illustrated by the example in Figure 5. Once subsystem  $s$  has failed, all subsequent attempts to access the database fail. Assuming most commands access the database, this means the program's functionality is severely degraded. Therefore, it seems appropriate to escalate the failure and terminate the program. This typically signals a system administrator or service management daemon to restart the program in a clean state, hopefully restoring full service. In the example, this behavior is achieved by running not just the `addEntry` calls, but the main loop as well, in subsystem  $s$ . When an `addEntry` call fails, an asynchronous exception is thrown in the main thread, which causes the loop to terminate.

In fact, since the existing command processing threads are unlikely to be able to run to completion successfully, it makes sense to terminate these as well. This is what happens in our approach, as a result of two interacting features: implicitly created subsystems and the subsystem hierarchy. Specifically, in addition to subsystems created explicitly through `new Subsystem()` expressions, subsystems are created implicitly in two places. Firstly, at the start of a try block, a new subsystem is created and the try block is executed in it. Secondly, when a thread (including the main thread) is started, a new subsystem, called the thread's root subsystem, is created, and the thread's main method is executed in this subsystem. Furthermore, a parent-child hierarchy exists among subsystems. Specifically, each subsystem

$s$  is considered a child of its creator subsystem, i.e. the subsystem that executed the code that created  $s$ . For a thread's root subsystem, this means it is a child of the subsystem that performed the `start` call. Note that this means that all subsystems are descendants of the main thread's root subsystem, which we call the global root subsystem. Note also that one can override an explicitly created subsystem's default parent by supplying a parent subsystem as an argument: `new Subsystem(parent)`.

When a subsystem fails, both it and all of its descendants (i.e., its transitive children) are marked as failed. Furthermore, all threads that are running in the failed subsystem or a descendant receive an asynchronous exception. It follows that if the global root subsystem fails, the program terminates.

In the example, the subsystem hierarchy is as follows. The global root subsystem has one child, subsystem  $s$ .  $s$  has one child for each active command processing thread: the thread's root subsystem. This, in turn, has one child: the subsystem associated with the try block. This means that when subsystem  $s$  fails, all command processing threads terminate and the main thread exits the enter block. Note that in this example, we could have used the global root subsystem instead of the explicitly created subsystem  $s$ , with the same effect. This could be achieved by replacing `new Subsystem()` with `Subsystem.getCurrent()`.

In the preceding section, we stated the soundness of our approach for single-threaded programs, and for a simplified semantics without the subsystem hierarchy. The soundness statement for the approach of this section is as follows: if a computation  $A$  happens-before a computation  $B$  and  $A$  and  $B$  are well-guarded, and  $A$  executes in some subsystem  $s$  and  $B$  executes in  $s$  or some descendant of  $s$ , then  $A$  is not a failed computation.  $A$  happens-before  $B$  means that  $A$  and  $B$  do not execute concurrently. A computation is well-guarded if, for every lock block that encloses it, there is an enter block that encloses it and that is itself enclosed by the lock block. Well-guardedness ensures that if  $A$  and  $B$  are enclosed by a lock block and  $A$  fails, then the subsystem and its descendants are marked as failed before the lock is released, and in the thread of  $B$ , it is checked after the lock is acquired that the subsystem has not failed. A simple way to ensure well-guardedness is to always use an enter statement as the body of a lock statement. Note that the soundness result does not rely on the Fail Fast feature.

## 5. Cancellation

This paper is about preserving safety after a failure. However, it turns out that the machinery introduced above for safely dealing with failures, also enables safe programmatic cancellation of computations.

An existing way to program safe cancellation of a computation by a supervisor thread is by introducing a variable shared between the computation and the supervisor to indi-

```

class Program {
    public static void main(String[] args) {
        final Database db := new Database();
        List<Thread> jobs := new ArrayList<Thread>();
        while (true) {
            final String cmd := readCommand();
            if (cmd.equals("cancelAll")) {
                for (Thread t : jobs) t.stop();
            } else {
                Thread t := new Thread() {
                    public void run() {
                        try {
                            ... compute(cmd); ...
                            ... synchronized (db) { db.addEntry(...); } ...
                        } catch (Throwable e) {
                            showErrorMessage(e);
                        }
                    }
                };
                jobs.add(t); t.start();
            }
        }
    }
}

```

**Figure 6.** A program with unsafe cancellation. A cancellation during *compute* is handled correctly, but if a cancellation occurs when a job is in method *addEntry* and the *Database* object is in an inconsistent state, the *Database* object’s lock is released, causing threads that subsequently acquire the lock to see the object in an unexpected state.

cate whether cancellation was requested, and by manually inserting code into the computation to poll this variable. The polling must occur sufficiently often to achieve good responsiveness to cancellation requests. This could be problematic if the computation uses library routines that do not poll the variable.

Apparently, the need for a better solution was recognized by the designers of Java, since they included a method called *Thread.stop* into the Java API. This method performs asynchronous cancellation: it throws a *ThreadDeath* exception in the target thread asynchronously; there is no need for manual polling. However, this method was later deprecated, because it was realized that it was difficult to use safely. Specifically, *Thread.stop* does not interact safely with locking. If the target thread is holding the lock of an object *o*, this lock is released, potentially exposing other threads that lock *o* to an inconsistent state of *o*.

The problem with *Thread.stop* is illustrated by the example in Figure 6. It is the example of Figure 4, extended with support for a command that cancels all currently running command executions. The command is implemented by calling *Thread.stop* on all command processing threads.

This program is unsafe, even in the absence of failures. Indeed, asynchronous cancellation causes the same problems that failure does. If a thread is stopped while it is exe-

```

class Program {
    public static void main(String[] args) {
        final Subsystem s := Subsystem.getCurrent();
        final Database db := new Database();
        List<Subsystem> jobs := new ArrayList<Subsystem>();
        while (true) {
            final String cmd := readCommand();
            if (cmd.equals("cancelAll")) {
                for (Subsystem job : jobs) job.cancel();
            } else {
                final Subsystem job := new Subsystem();
                jobs.add(job);
                new Thread() {
                    public void run() {
                        try {
                            enter (job) {
                                ... compute(cmd); ...
                                synchronized (db) {
                                    enter (s) { db.addEntry(...); }
                                } ...
                            }
                        } catch (Throwable e) {
                            showErrorMessage(e);
                        }
                    }
                }.start();
            }
        }
    }
}

```

**Figure 7.** The program of Figure 6, corrected to perform safe cancellation using subsystems. If a *job* subsystem is cancelled while the corresponding thread is executing method *addEntry*, the thread is not stopped. Only after method *addEntry* completes and the thread re-enters subsystem *job*, an exception is thrown.

cuting inside method *addEntry* and the *Database* object is inconsistent, the lock is released and other threads can see the inconsistent object.

The Fail Fast feature of our subsystems approach can be used for safe asynchronous cancellation of computations. By running a computation in a subsystem *s*, the computation can be cancelled by calling *s.cancel()*. This causes the subsystem to fail with a *SubsystemCancelledException*. The computation can protect sub-computations that should not be cancelled by running the sub-computations in a subsystem that is not *s* or a descendant of *s*.

The approach is illustrated in Figure 7. The program creates a subsystem for each command processing thread and executes the command processing code in it. To cancel a command, method *cancel* is called on the corresponding subsystem. As before, a separate subsystem is used in which to run both the main loop and the *addEntry* calls. Note that subsystem *s* is not a descendant of any job subsystem. As a result, cancelling a job subsystem does not affect a thread that is executing in subsystem *s*; only when the thread leaves *s* and enters the job subsystem, which is marked as failed, an exception occurs.

Notice that combining enter statements with the use of *Thread.stop*, e.g. by using *Thread.stop* in Figure 5, would

also preserve dependency safety. However, in that scenario, if a stop signal arrives during an `addEntry` call, subsystem `s` is marked as failed and the entire program is terminated, instead of only the targeted jobs being cancelled.

## 6. Cleanup

### 6.1 The problem

In the preceding sections, we have looked at ensuring dependency safety, where a piece of code outside a try block depends on a piece of code inside a try block either not executing at all, or executing to completion. Another common type of dependency is where a piece of code outside a try block depends on a piece of code inside a try block either not executing at all, or executing to completion *and being compensated* by successful completion of another piece of code (a *cleanup action*) before the try block is exited.

The property at stake may be a safety property, but more often it is a property that states the *absence of leaked resources*. Possible resources are operating system objects such as file handles or network sockets, or application-specific resources such as entries in shared data structures that outlive the try block. We call a resource *leaked* if the computation that allocated it (the *client computation*) has finished without de-allocating it. Resource leaks are problematic because they may lead to gradual and hard-to-diagnose degradation of performance in long-running programs.

Typically, Java programs use try-finally statements in an attempt to guarantee cleanup and prevent resource leaks in the event of the failure of a client computation. However, the typical try-finally-based pattern, where a resource is allocated, then client code is executed in a try block, and finally the resource is de-allocated in a finally block, does not fully prevent resource leaks, and, additionally, suffers from a number of safety risks. Specifically:

1. A failure during allocation may leave the resource provider (e.g., a shared data structure) in an inconsistent state.
2. If a failure occurs after allocation but before the try block is entered, e.g., during custom initialization of the resource, or as the result of a cancellation or a failure in operations performed implicitly by the virtual machine, the resource is leaked.
3. If a failure occurs during manipulation of the resource inside the try block, the cleanup action is performed in the finally block, violating safety.
4. A failure during cleanup may leave the resource provider in an inconsistent state.
5. If a failure occurs after control enters the finally block, but before the cleanup action is started, the resource is leaked.

Importantly, none of these failures would violate safety or cause resource leaks if they caused the resource provider

```

class Widget {
    WidgetManager m;
    Widget(WidgetManager m)
    { this.m := m; }
    public void dispose() { m.removeWidget(this); }
    ...
}
class WidgetManager {
    List<Widget> widgets := new ArrayList<Widget>();
    Widget allocWidget() {
        Widget w := new Widget();
        widgets.add(w);
        return w;
    }
    void removeWidget(Widget w) {
        widgets.remove(w);
    }
}
class Program {
    static Widget allocGreenWidget(WidgetManager m) {
        Widget w := m.allocWidget();
        w.setColor(Color.green);
        return w;
    }
    public static void main(String[] args) {
        WidgetManager m := new WidgetManager();
        while(true) {
            String cmd := readCommand();
            try {
                ...
                Widget w := allocGreenWidget(m);
                try {
                    ...
                } finally { w.dispose(); }
                ...
            } catch (Throwable t) {
                showErrorMessage(t);
            }
        }
    }
}

```

**Figure 8.** A program with a potential resource leak. If an exception occurs during the `setColor` call, or if the `dispose` call throws a `StackOverflowError`, the exception is caught and the widget is never removed from the widget manager. Furthermore, a failure during `add` or `remove` could violate safety.

(e.g., a shared data structure) itself to be de-allocated (or in the case of operating system resources, if the failure caused the process to terminate); the problem is that all of these failures are incorrectly treated like failures of the client computation, i.e., caught by the try-catch block enclosing the client computation.

These problems are illustrated by the example program in Figure 8. It is again a command processing program that continuously receives a command, processes it, and then receives the next command. As before, if a failure occurs during command processing, it is caught, an error message is shown, and the next command is received.

In this example, processing commands requires the use of widgets, which are allocated from a widget manager. To prevent the program using ever more memory, widgets must be disposed after use.

Specifically, the example illustrates all of the issues identified above (except for issue 3, which is similar to 1 and 4). A failure during the calls of *add* (issue 1) or *remove* (issue 4) might leave the *ArrayList* object, and therefore, the *WidgetManager* object, in an inconsistent state, causing subsequent command executions to violate safety. Furthermore, a failure during *setColor* (issue 2) or a *StackOverflowError* when calling *dispose* (issue 5) would leak the widget.

Some of these issues can be solved using existing approaches, and some can be solved using the subsystems machinery we introduced in the preceding sections. Specifically, *cleanup stacks* (Weimer and Nacula 2004) (see Section 6.3) solve the issue of a failure after allocation but before the try block is entered (issue 2). However, cleanup stacks introduce the issue of a failure after allocation but before the cleanup action is pushed onto the compensation stack. Furthermore, they do not solve any of the other issues. On the other hand, our subsystems proposal, as introduced in the preceding sections, solves the issues pertaining to data structure inconsistency (issues 1, 3, and 4), by associating a subsystem with the resource provider and executing the allocation, update, and de-allocation of resources within the provider's subsystem. Subsystems also solve the issue of a failure between allocation and registration of the compensation. It follows that together, cleanup stacks and subsystems solve all issues, except issue 5, which arises when a failure occurs after the finally block is entered but before the provider subsystem is entered. Therefore, to the best of our knowledge, no existing approach solves all five issues.

In the remainder of this section, we propose two approaches for performing cleanup that solve all five issues and therefore are safe and prevent leaks: the provider-controlled cleanup approach, and the client-controlled cleanup approach. We present both approaches because the former requires less new machinery and provides an interesting new insight into the problem, whereas the latter is more flexible and probably preferable in practice.

## 6.2 Provider-controlled cleanup

In this approach, in order to allocate and use a resource, the client computation encapsulates the operations it wishes to perform on the resource in a *closure*, and passes it to the resource provider. The resource provider then allocates a resource, executes the closure passing the resource as an argument, and finally de-allocates the resource.

To ensure safety and to ensure that if a failure occurs between any of these three steps, the resource provider's subsystem is terminated and the resource is not leaked, the resource provider enters its associated subsystem, using an enter block, for the entire duration of the three steps. However,

to ensure that a failure within the client code does not affect the resource provider, the closure is executed in a *reenter block* nested within the enter block. A reenter block is like an enter block in that it enters the subsystem specified as an operand; it differs from an enter block in that it does not propagate exceptions. If the body of a reenter block fails, the subsystem fails as usual, and the exception is registered with the subsystem, but the reenter block terminates normally.

In the provider-controlled cleanup approach, the provider subsystem is entered and then, after allocation of the resource, the client subsystem is re-entered using a reenter block, to execute the closure. If a failure occurs in the closure, the exception is registered with the client subsystem. After the resource is cleaned up, control attempts to re-enter the client subsystem, which is marked as failed, causing an exception to be thrown and causing the failure to propagate properly.

The approach is illustrated in Figure 9. It is the program of Figure 8, corrected using the provider-controlled cleanup approach. Notice that the allocation and de-allocation methods have been replaced with methods that take a closure.

The example shows how utility methods that allocate and initialize objects can be refactored to fit into the provider-controlled cleanup approach. Method *allocGreenWidget* has been replaced with a method *usingGreenWidget* which takes a closure and calls *usingWidget*, passing another closure that first initializes the resource before executing the closure passed as an argument to *usingGreenWidget*.

## 6.3 Client-controlled cleanup

Our client-controlled cleanup approach is an integration of the cleanup stacks approach into the subsystems approach.

The *cleanup stacks* approach (Weimer and Nacula 2004) solves the issue of a failure that occurs after allocation but before entering the try block of the try-finally statement as follows. Before the try block is entered, a *CleanupStack* object is created. The resource allocation method is called inside the try block instead of before the try block, and the *CleanupStack* object is passed as an argument to this method. After allocating the resource, the allocation method pushes a cleanup action (a closure in (Weimer and Nacula 2004)) onto the cleanup stack. In the finally block, the cleanup stack's *run* method is called, which pops the cleanup actions from the cleanup stack in LIFO order and executes them.

The cleanup stack approach solves issue 2 above: as soon as the cleanup action is pushed onto the cleanup stack, no failure prior to the exit from the try block can prevent the cleanup action from being executed. However, the approach introduces a new issue: that of a failure after the resource is allocated and before the cleanup action is pushed onto the cleanup stack. Furthermore, cleanup stacks do not solve any of the other four issues identified above.

As noted above, a straightforward combination of the subsystems approach of the preceding sections and cleanup

```

class Widget {
    ...
}
class WidgetManager {
    Subsystem s := Subsystem.getCurrent();
    List<Widget> widgets := new ArrayList<Widget>();
    void usingWidget({Widget => void} body) {
        Subsystem caller := Subsystem.getCurrent();
        enter (s) {
            Widget w := new Widget();
            widgets.add(w);
            reenter (caller) { body(w); }
            widgets.remove(w);
        }
    }
}
class Program {
    static void usingGreenWidget(
        WidgetManager m, {Widget => void} body) {
        m.usingWidget({Widget w =>
            w.setColor(Color.green);
            body(w);
        });
    }
    public static void main(String[] args) {
        WidgetManager m := new WidgetManager();
        while(true) {
            String cmd := readCommand();
            try {
                ...
                usingGreenWidget(m, {Widget w =>
                    ...
                });
                ...
            } catch (Throwable t) {
                showErrorMessage(t);
            }
        }
    }
    ...
}

```

**Figure 9.** The program of Figure 8, corrected to perform safe cleanup using subsystems, using provider-controlled cleanup. If the `setColor` call fails, the widget is still cleaned up correctly, since the `reenter` block does not propagate exceptions. If the `remove` call fails, subsystem `s` (i.e., the global root subsystem) is marked as failed and the program terminates, preventing a resource leak. For conciseness, the example uses the proposed syntax for closures and function types in Java.

stacks solves all but one of the five issues that prevent safe and leak-free resource cleanup. By associating a subsystem with the resource provider and executing the allocation, manipulation, and de-allocation of resources in the provider’s subsystem, the inconsistency issues are solved. Note that, during allocation, it is crucial to push the cleanup action onto the cleanup stack before leaving the provider subsystem. This ensures that any failure that intervenes between the actual allocation and the registration of the cleanup action causes the provider subsystem to fail and does not cause a leak.

However, issue 5 remains: there is still a risk that a failure intervenes between entry into the finally block and entry into the provider subsystem during a cleanup action. To solve this issue, we make two modifications to the cleanup stacks approach.

- Firstly, when a cleanup action is pushed onto a cleanup stack, the cleanup stack records not only the cleanup action but the current subsystem as well.
- Secondly, we introduce *using statements*, to be used instead of try-finally statements for running cleanup stacks.

Our using statements are similar to C#’s using statements, but they are specifically for using cleanup stacks and they have special semantics as follows: after control exits the using statement’s body, the cleanup actions are popped from the cleanup stack in LIFO order and each cleanup action is executed *in its associated subsystem*. Our proposed semantics does not allow any failures to intervene in this process. (It is up to the language implementation to guarantee this; we discuss a possible implementation in Section 8.) However, failures during execution of the cleanup actions are allowed as usual. When a cleanup action fails, its associated subsystem fails and is marked as such, but cleanup continues with the next action. As in the case of nested try-finally blocks, if one or more cleanup actions fail, the outermost (i.e., last) action’s exception is propagated outside the using statement.

The approach is illustrated in Figure 10.

A soundness statement for our client-controlled cleanup approach can be given as follows. Consider given a *compensation dependency relation*, which, in a given execution, relates a computation  $A$ , a *Disposable* object  $o$  created by  $A$ , and a computation  $B$  such that  $A$  happens-before  $B$ . We say that  $B$  depends on  $A$  being compensated with  $o$ . An execution is *compensation-dependency-safe* if for each such  $A$ ,  $o$ , and  $B$ , exactly one call of  $o.dispose()$  happens-between  $A$  and  $B$ , and neither  $A$  nor this call fail. An execution *uses subsystems correctly* with respect to a given compensation dependency relation, if for each  $A$ ,  $o$ , and  $B$  such that  $B$  depends on  $A$  being compensated with  $o$ , all of the following hold:

- $A$  executes in some subsystem  $s$
- $A$  pushes  $o$  onto some cleanup stack  $cs$  exactly once

```

class Widget implements Disposable {
    WidgetManager m;
    Widget(WidgetManager m) { this.m := m; }
    public void dispose() { m.removeWidget(this); }
    ...
}
class WidgetManager {
    Subsystem s := Subsystem.getCurrent();
    List<Widget> widgets := new ArrayList<Widget>();
    Widget allocWidget(CleanupStack cs) {
        enter(s) {
            Widget w := new Widget();
            widgets.add(w);
            cs.push(w);
            return w;
        }
    }
    void removeWidget(Widget w)
    { enter(s) { widgets.remove(w); } }
}
class Program {
    static void allocGreenWidget(
        WidgetManager m, CleanupStack cs) {
        Widget w := m.allocWidget(cs);
        w.setColor(Color.green);
        return w;
    }
    public static void main(String[] args) {
        WidgetManager m := new WidgetManager();
        while (true) {
            String cmd := readCommand();
            try {
                ...
                using (CleanupStack cs := new CleanupStack()) {
                    Widget w := allocGreenWidget(m, cs);
                    ...
                }
                ...
            } catch (Throwable t) { showErrorMessage(t); }
        }
    }
    ...
}

```

**Figure 10.** The program of Figure 8, corrected to perform safe cleanup using subsystems, using client-controlled cleanup. If the `setColor` call fails, the widget is still cleaned up correctly, since the widget was already pushed onto the cleanup stack. The `using` statement calls each cleanup routine on the cleanup stack in the subsystem that registered it, so if the `dispose` call fails, subsystem  $s$  (i.e., the global root subsystem) is marked as failed and the program terminates, preventing a resource leak.

- $o$  is not pushed onto any other cleanup stack in the execution, and the execution does not perform any direct calls of  $o.dispose()$
- $cs$  is used as the operand of exactly one using statement execution
- This using statement execution encloses  $A$
- The end of the using statement happens-before  $B$ , or else  $B$  is part of some call  $o'.dispose()$  and  $o'$  is pushed onto  $cs$  prior to  $o$  (and  $o'$  is not otherwise pushed onto any cleanup stack and there are no direct calls of  $o'.dispose()$ )
- The using statement executes in some subsystem  $s'$  other than  $s$
- $B$  executes in  $s$  or a descendant of  $s$ , but not in  $s'$  or a descendant of  $s'$

We then have that if an execution uses subsystems correctly, then it is compensation-dependency-safe. We call this property the soundness of the client-controlled cleanup approach.

## 7. Safety Proof

We have formalized the semantics of our proposed language extension, as well as the statement and a proof outline for the soundness of the approach. The full formalization is available in the appendix; here, we sketch the proof of the main theorem.

**Theorem 1 (Soundness).** *If a computation  $A$  happens-before a computation  $B$  and  $A$  and  $B$  are well-guarded, and  $A$  executes in some subsystem  $s$  and  $B$  executes in  $s$  or some descendant of  $s$ , then  $A$  is not a failed computation.*

*Proof.* By contradiction. Assume  $A$  fails. It suffices to prove for every prefix of some path of atomic happens-before edges (i.e. where the derivation of the edges does not use transitivity) from  $A$  to  $B$ , that at the thread execution point  $C$  at the end of the prefix, one or more of the following hold:

- the thread is failing and the current subsystem is  $s$  and the immediately enclosing block (if any) is not a lock block
- subsystem  $s$  and its descendants have been marked as failed and one or more of the following hold:
  - the current subsystem is not  $s$  or a descendant of  $s$ , or
  - $C$  is enclosed immediately by a lock block, or
  - the thread is failing.

This can be proved easily by induction on the length of the prefix and case analysis on the last edge. The contradiction follows from the fact that at  $B$  the thread is not failing, the current subsystem is  $s$  or a descendant of  $s$ , and  $B$  is not enclosed immediately by a lock block.  $\square$

## 8. Implementation Issues

We created a prototype implementation of the approach on the .NET Framework as a C# 3.0 library. C# 3.0's lambda

expression syntax can be used to write reasonably concise enter statements.

A major complication for achieving a fully correct implementation of the approach in the form of a library, is the fact that the .NET Framework Common Language Runtime may throw an exception at any program point, due to an internal resource limit being reached or an internal error being discovered within the execution engine (Toub 2005). (The same holds for the Java Virtual Machine. See the Java Virtual Machine Specification, Second Edition (Lindholm and Yellin 1999), Section 2.16.2.) To illustrate the difficulty, consider the implementation sketch in Figure 3. The expansion of an enter statement is not correct in the presence of internal exceptions; indeed, if an internal exception occurs immediately prior to the assignment  $s.failed := true$ ;, the enter statement completes without marking the subsystem as failed, breaking dependency safety.

Version 2.0 of the .NET Framework introduced constructs specifically for writing code that must execute reliably in the presence of internal exceptions (Toub 2005). We used these constructs in our prototype implementation to ensure that on abrupt completion of the body of an enter statement, the subsystem and its descendants are marked as failed and stop signals are sent to other threads executing in the subsystem or its descendants. Specifically, we used the following API:

*ExecuteCodeWithGuaranteedCleanup*( $t, c, u$ )

where  $t$  and  $c$  are delegates (similar to function pointers in C) and  $u$  is arbitrary user data that is passed to  $t$  and  $c$ . The API first executes  $t$ . When  $t$  completes, either normally or abruptly, the cleanup delegate  $c$  is executed. The API guarantees that no internal exceptions occur during the execution of  $c$ , provided that  $c$  satisfies certain constraints, such as: no heap memory allocation, and no unbounded call stack memory allocation. Unfortunately, these constraints have not been spelled out very precisely anywhere; we had to make some assumptions as to what can reasonably be executed without the risk of internal exceptions.

We also used this construct to implement using statements, as follows. In  $t$ , we first execute the body of the using statement, and then we attempt to execute all cleanup actions. We cannot execute the cleanup actions in  $c$  since they might use unbounded amounts of resources. If an internal exception occurs between the body and the cleanup actions or between two cleanup actions, we simply mark the subsystem of each remaining cleanup action as failed in  $c$ .

We have performed a few microbenchmark performance tests. These indicate the following approximate timings for the following statements:

Statement	Timing	Timing*
<code>try {} catch {}</code>	13 $\mu$ s	1.7 $\mu$ s
<code>try { enter (s) {} } catch {}</code>	23 $\mu$ s	2.0 $\mu$ s

To measure the impact of the *ExecuteCodeWithGuaranteedCleanup* construct, we replaced it with a dummy that

uses a simple try-finally statement. The resulting timings are shown in the third column. It turns out that the overhead of this construct dominates the run time.

Even though the current performance is probably acceptable for most real-world applications, we believe it can still be improved significantly, in particular if the constructs are implemented directly in the virtual machine rather than as a library. Performing such an implementation is future work.

We have also prepared a prototype implementation of subsystems as a library on the Java virtual machine. The absence of constructs to prevent internal exceptions on this platform is not a complete showstopper. The workaround we use is to record each enter statement entry in a global data structure. In order to check if a subsystem has failed, in addition to checking the failed flag, we inspect this data structure to look for enter statement entries that should have been removed, but were not due to an internal exception. One limitation with this approach, which seems inevitable, is that to make this work in a multithreaded setting, user code must use a modified implementation of synchronized blocks, provided by the library, rather than the built-in Java one.

Another limitation of the Java platform is that the Fail Fast feature cannot be implemented fully on it. The problem is that a *Thread.stop* signal cannot be cancelled. This is necessary to prevent a stop signal intended for a subsystem  $s$  from arriving when the target thread has transitioned into an unrelated subsystem  $s'$ . (In the .NET Framework, a stop signal can be cancelled using the *ResetAbort* API method.) We partially worked around this by using *Thread.interrupt()* instead. This signal can be cancelled using the *Thread.interrupted()* API method. A limitation of this approach is that this signal stops threads that are waiting on a condition variable, but not long-running computations, unless they explicitly poll the interrupt flag.

We used the Scala language for the Java platform implementation, because of its support for very concise syntax.

## 9. Related Work

To the best of our knowledge, our proposed subsystems approach is the first fully compositional minimal extension of the imperative object-oriented programming paradigm that addresses dependency safety in exception handling, the conflict between locking and exceptions, the problem of asynchronous cancellation, and the resource cleanup problem. It is fully compositional in that subsystems may be nested arbitrarily. As a result, the approach supports failure and cancellation of any subsystem and at the same time allows any subsystem to protect itself from failure or cancellation of descendant subsystems.

*Non-compositional approaches* Starting with version 2, the .NET Framework includes reliability features that make it possible to write cleanup routines that are guaranteed to execute even in the presence of failure or cancellation (Toub 2005). However, the approach is not compositional:

these cleanup routines cannot be cancelled; furthermore, they must be carefully coded to rule out failures within the cleanup routines themselves since those are not dealt with safely. The mechanism is intended only for manipulation of execution environment resources; it is not for general application use.

Three further reliability-related features in .NET Framework version 2 are the following. Firstly, cancellation is disabled during finally blocks. This enables safe cleanup in the presence of cancellation (but not failure). Secondly, an unhandled exception in one thread kills all other threads, without executing catch or finally blocks. However, in the thread that throws the unhandled exception, finally blocks are executed normally and locks are released, leaving a time window between the release of the lock and the time the exception reaches the toplevel (possibly after executing other finally blocks) where other threads can see inconsistent state. Thirdly, a method *Environment.FailFast* was added, which terminates the program immediately.

Buhr and Mok (2000) address the problem of asynchronous cancellation; they call it the *non-reentrant problem*. They suggest the use of *protected blocks*, which protect critical regions by temporarily disabling cancellation.

Rudys et al. (2001) proposes weaving code into an untrusted plugin (such as an applet) that polls a cancellation request flag to enable forcibly cancelling the plugin. The flag is also checked whenever the host system calls into the plugin. In our approach, a thread running in one subsystem may protect itself from cancellation of its subsystem by entering an ancestor subsystem to which it has a reference; however, separate techniques (e.g., perhaps by associating permissions with subsystems) could be used to prevent this in case the thread is running untrusted code.

**Message-passing-based approaches** DrScheme (Flatt et al. 1999; Flatt and Findler 2004) is a Scheme implementation that has extensive support for writing programs that serve as execution environments for other programs. Specifically, it supports safe cancellation of threads by running shared data structures in separate threads. Cleanup of operating system resources after a thread is cancelled is guaranteed using *custodians*. The authors do not discuss failures and their associated safety risks.

Erlang (Armstrong 2003) is a language focused on reliability. Inconsistent data structures within a process are ruled out because the language has no destructive update. Fail-fast is achieved by linking processes: when a process dies, an exit signal is sent to linked processes, causing those to die as well by default.

The SCOOP multithreading approach for Eiffel (Meyer 1992) has a notion of subsystems, but a subsystem in SCOOP is a thread and a set of objects handled by that thread. Brooke and Paige (2007) suggest marking an object as “dead” when the processing of an asynchronous incom-

ing call fails, causing subsequent calls to fail immediately. SCOOP subsystems cannot be nested.

**Other related work** Garcia et al. (2001) provide a survey of exception mechanisms. However, the authors do not discuss the dependency safety issue. In fact, most modern imperative and/or object-oriented languages have inherited the exception mechanism of CLU (Liskov and Snyder 1979) and therefore all of these suffer from the problems addressed by our approach.

Weimer and Necula (2004) propose *compensation stacks* to make it easier to write effective cleanup code. As discussed in Section 6, compensation stacks by themselves do not provide any safety guarantees; however, when combined with subsystems, they provide a basis for our safe client-controlled cleanup approach.

Fetzer et al. (2003) assume the viewpoint that “exception handling is only effective if the premature termination of a method due to an exception does not leave an object in an inconsistent state”. The paper proposes techniques to detect and “mask” *non-atomic exception handling*, i.e. violations against *failure atomicity*. This paper assumes that after catching an exception, the entire application should be in a consistent state, whereas we allow *failed subsystems* to remain in an inconsistent state, while preventing control from entering a failed subsystem. The authors find a large number of Java methods that are not failure atomic. This would strengthen the case for subsystems, because it indicates that exceptions do indeed commonly leave objects in an inconsistent state.

There has been much work on extending programming languages with mechanisms similar to operating system processes. Java Isolates (Palacz et al. 2006) are an example of this. One of the goals of Java Isolates and similar proposals is to prevent erratic or malicious behavior of one computation from affecting the safety of another computation. Therefore, these proposals enforce strong isolation between computations. This way, they can guarantee safety properties in the presence of arbitrary code, at the expense of additional programming and/or performance overhead imposed on communication between isolated computations. For example, in Java Isolates, computations can communicate only through message passing or through conventional inter-process communication mechanisms, such as sockets. In contrast, our subsystems approach does not provide any guarantees for arbitrary code; the approach guarantees dependency safety only for programs that use subsystems correctly. In return, subsystems impose a much lower programming and runtime overhead: communication between computations can occur through ordinary shared objects and method calls.

An alternative way to deal with failures is to roll the state of the objects involved back to a consistent state, through the use of transactions (e.g. Shavit and Touitou (1995); Welc et al. (2004); Fetzer et al. (2003)). However, this has a greater performance overhead; also, it presents problems when the

computation that failed performed I/O. Our subsystems approach is more conservative from a semantic and performance point of view.

This work was inspired by our research in program verification for Java-like languages that is sound in the presence of failures. To the best of our knowledge, no existing program verifiers for Java-like languages (including ESC/Java (Flanagan et al. 2002) and Spec# (Barnett et al. 2006)) have this property. In Jacobs et al. (2007), we propose a verification approach for Java programs where the programmer manually guards dependent code using flag variables that track an object's consistency. This work proposes language support for programs that are verifiably safe in the presence of failures.

## 10. Conclusion

We propose a language extension, called *subsystems*, that facilitates writing sequential or multithreaded programs that provably preserve intended safety properties and that do not leak resources, even in the presence of failure, and that perform safe cancellation of computations. To the best of our knowledge, it is the first fully compositional minimal extension of a Java-like language that does so.

Future work includes gaining experience with our prototype implementation, mainly to assess the applicability and the usability of the approach. We anticipate the possible need to facilitate the placement of enter blocks, perhaps through annotations on methods, classes, or packages, or through some inference scheme. Other work includes applying the subsystems idea to the problem of exception handling in asynchronous and callback patterns.

## Acknowledgments

The authors would like to thank Jan Smans and Marko van Dooren for their helpful comments.

## References

Joe Armstrong. *Making reliable distributed systems in the presence of software errors*. PhD thesis, Royal Institute of Technology, Stockholm, Sweden, 2003.

Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Proceedings of the Fourth International Symposium on Formal Methods for Components and Objects (FMCO 2005)*, volume 4111 of *LNCS*. Springer, 2006.

Phillip J. Brooke and Richard F. Paige. Exceptions in Concurrent Eiffel. *Journal of Object Technology*, 6(10):111–126, nov 2007.

Peter A. Buhr and W. Y. Russell Mok. Advanced exception handling mechanisms. *IEEE Trans. Software Eng.*, 26(9):820–836, 2000.

Christof Fetzer, Karin Högstedt, and Pascal Felber. Automatic detection and masking of non-atomic exception handling. In

*Proc. Intl. Conf. Dependable Systems and Networks (DSN'03)*, 2003.

Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *PLDI*, pages 234–245, 2002.

Matthew Flatt and Robert Bruce Findler. Kill-safe synchronization abstractions. In *Proc. PLDI'04*, 2004.

Matthew Flatt, Robert Bruce Findler, Shriram Krishnamurthi, and Matthias Felleisen. Programming languages as operating systems (or Revenge of the son of the Lisp machine). In *Proc. Intl. Conf. on Functional Programming (ICFP 1999)*, 1999.

Alessandro F. Garcia, Cecília M. F. Rubira, Alexander B. Romanovsky, and Jie Xu. A comparative study of exception handling mechanisms for building dependable object-oriented software. *Journal of Systems and Software*, 59(2):197–222, 2001.

Bart Jacobs, Peter Müller, and Frank Piessens. Sound reasoning about unchecked exceptions. In *Proceedings of the Fifth International Conference on Software Engineering and Formal Methods*, 2007.

Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification, Second Edition*. Addison-Wesley, 1999. Online at <http://java.sun.com/docs/books/jvms/>.

Barbara Liskov and Alan Snyder. Exception handling in CLU. *IEEE Trans. Software Eng.*, 5(6):546–558, 1979.

Bertrand Meyer. *Eiffel: The Language*. Prentice Hall, 1992.

Krzysztof Palacz et al. JSR 121: Application Isolation API Specification. <http://jcp.org/en/jsr/detail?id=121>, 2006.

Algis Rudys, John Clements, and Dan S. Wallach. Termination in language-based systems. In *Network and Distributed System Security Symposium (NDSS)*, February 2001.

N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 204–213, 1995.

Jim Shore. Fail fast. *IEEE Software*, September 2004.

Stephen Toub. Keep your code running with the reliability features of the .NET Framework. *MSDN Magazine*, oct 2005.

Westley Weimer and George C. Necula. Finding and preventing run-time error handling mistakes. In *Proc. OOPSLA '04*, pages 419–431, October 2004.

Adam Welc, Suresh Jagannathan, and Antony L. Hosking. Transactional monitors for concurrent objects. In *Proc. ECOOP 2004*, 2004.

# Appendix: Formal treatment

In this appendix we provide a formal syntax and semantics of our proposed language extension. For conciseness, we show the semantics in the context of a tiny dynamically typed object-based language.

We provide a formal syntax in Section 1. Notational conventions are introduced in Section 2. The shape of the execution states of our small-step semantics is described in Section 3. The step rules are given in Section 4. A number of properties of the language extension are stated and proved in Section 5. Section 6 shows a number of example programs that illustrate the envisaged use of each of the language constructs.

## 1 Syntax

The set of variable names is  $\mathcal{X}$ . Meta-variable  $x$  ranges over  $\mathcal{X}$ .

```
program ::= s*
method  ::= m(x*) { s* }
s       ::= localCopy | objectCreation | call | read | write
          | fork | lock
          | tryCatch | throw
          | currentSubsystem | newSubsystem | enter | reenter
          | newCleanupStack | push | using

localCopy ::= x := x;
objectCreation ::= x := new { method* };
call ::= x := x.m(x*);
read ::= x := x.f;
write ::= x.f := x;
fork ::= fork { s* }
lock ::= lock (x) { s* }
tryCatch ::= try { s* } catch { s* }
throw ::= throw;
currentSubsystem ::= x := current_subsystem;
newSubsystem ::= x := new_subsystem(x);
enter ::= enter (x) { s* }
reenter ::= reenter (x) { s* }
newCleanupStack ::= x := new_cleanupstack;
push ::= x.push(x);
using ::= using (x) { s* }
```

The language is similar to an assembly language or a bytecode language in that the only expressions are local variable references. All computation occurs through statements. This simplifies the formalization of the dynamic semantics.

A program *program* is a sequence of statements  $s$ . There are no named classes; there are only anonymous class instance creation statements.

A method has a name, a parameter list, and a body consisting of a sequence of statements.

The statements are the general programming statements (local variable to local variable assignment statements, object creation statements, method call statements, field read statements, and field write statements), the concurrent programming statements (thread creation statements and synchronization statements), the conventional exception handling statements (try-catch statements and throw statements), the core subsystems statements (current subsystem query statements, subsystem creation statements, enter statements, and reenter statements), and the cleanup stacks-related statements (cleanup stack creation statements, push statements for pushing a new cleanup action onto a cleanup stack, and using statements for running the cleanup actions in a cleanup stack on completion of a block of code).

## 2 Notational Conventions

We denote the set of partial functions from a set  $A$  to a set  $B$  as  $A \hookrightarrow B$ .

We use  $T \triangleleft t$  as a shorthand for  $T \cup \{t\}$ , and if  $\Sigma$  is a function that maps a child to its parent, we use  $(\Sigma^{-1})^*(\sigma)$  to denote the set of descendants of  $\sigma$  (including  $\sigma$  itself). The empty list is denoted as  $\epsilon$ , and the list with head  $h$  and tail  $t$  as  $h \cdot t$ . If  $x$  ranges over  $X$ , then  $\bar{x}$  ranges over lists of elements of  $X$ . Function update is written as  $f(x := y)$ .

## 3 Execution States

Before going into the step rules for our language, we define here the execution states and their components.

The program values  $v \in \mathcal{V}$ , where  $\mathcal{V} = \{\text{null}\} \cup \mathcal{O} \cup \mathcal{S} \cup \mathcal{C}$ , are the null reference `null`, the object references  $o \in \mathcal{O}$ , the subsystem references  $\sigma \in \mathcal{S}$ , and the cleanup stack references  $c \in \mathcal{C}$ . (For the purpose of this formalization, subsystems and cleanup stacks are not considered to be objects.)

Each thread has a unique thread identifier  $tid \in \mathcal{T}$ .

An execution state is a tuple

$$(L, \Sigma, \Phi, C, T)$$

Its components are as follows:

- $L : \mathcal{O} \hookrightarrow \mathcal{T}$ , the lock map, is a partial function that maps object identifiers to thread identifiers. Specifically, its domain collects the objects whose lock is held by some thread, and if an object  $o$ 's lock is held by a thread  $tid$ ,  $L$  maps  $o$  to  $tid$ .
- $\Sigma : \mathcal{S} \hookrightarrow \mathcal{S}$  is a partial function that maps subsystem references to their parent subsystem references. Its domain collects the subsystems that have been created.
- $\Phi \subseteq \mathcal{S}$  is a set of subsystem references. Specifically, it is the set of failed subsystems.
- $C : \mathcal{C} \hookrightarrow (\mathcal{S} \times (\mathcal{X} \rightarrow \mathcal{V}) \times \mathcal{X})$  is a partial function that maps cleanup stack references to lists of cleanup actions. Its domain collects the cleanup stacks that have been created. A cleanup action is a  $(\sigma, V, x)$  tuple, where  $x$  is a local variable name, which, when evaluated in environment  $V$  should yield an object whose `dispose` method is to be called on behalf of subsystem  $\sigma$ .
- $T \subseteq \mathcal{T} \times \mathcal{S} \times (\mathcal{X} \rightarrow \mathcal{V}) \times s^* \times b^* \times F^*$  is a set of threads, which are  $(tid, \sigma, V, \bar{s}, \bar{b}, \bar{F})$  tuples.  $tid$  is the thread's thread identifier,  $\sigma$  is the thread's current subsystem,  $V$  is the current variable environment,  $\bar{s}$  is the statement list currently being executed,  $\bar{b}$  is

a list of enclosing blocks, and  $\overline{F}$  is a list of enclosing method activations. An enclosing method activation  $F = (V, x, \overline{s}, \overline{b})$  records the activation's variable environment  $V$ , the variable  $x$  into which the activation's callee's result is to be stored when the callee returns, the sequence of statements  $\overline{s}$  that followed the call, and the sequence of blocks  $\overline{b}$  that enclosed the call.

### 3.1 Enclosing blocks

The sequence-of-enclosing-blocks component of a thread keeps track of the blocks that the thread has entered but not left; specifically, it keeps track of the information needed when exiting the block.

Enclosing blocks have the following syntax:

$$b ::= \text{tryBlock} \mid \text{lockBlock} \mid \text{enterBlock} \mid \text{reenterBlock} \\ \mid \text{usingBlock} \mid \text{cleanupActionBlock}$$

$$\text{tryBlock} ::= \mathbf{try} (\sigma) \mathbf{catch} \{ \overline{s} \} s^*$$

An enclosing try block  $\mathbf{try} (\sigma) \mathbf{catch} \{ \overline{s} \} s^*$  records the subsystem  $\sigma$  that was current prior to the try-catch statement, as well as the catch block  $\overline{s}$  and the sequence of statements  $s^*$  that followed the try-catch statement.

$$\text{lockBlock} ::= \mathbf{lock} (o); s^*$$

An enclosing lock block  $\mathbf{lock} (o); s^*$  records the object  $o$  that was locked on entry to the block, as well as the sequence of statements  $s^*$  that followed the lock statement.

$$\text{enterBlock} ::= \mathbf{enter} (\sigma); s^*$$

An enclosing enter block  $\mathbf{enter} (\sigma); s^*$  records the subsystem  $\sigma$  that was current prior to the enter statement ( $\mathbf{not}$  the subsystem that was entered), as well as the sequence of statements  $s^*$  that followed the enter statement.

$$\text{reenterBlock} ::= \mathbf{reenter} (\sigma); s^*$$

Analogously to an enclosing enter block, an enclosing reenter block  $\mathbf{reenter} (\sigma); s^*$  records the subsystem  $\sigma$  that was current prior to the reenter statement, as well as the sequence of statements  $s^*$  that followed the reenter statement.

$$\text{usingBlock} ::= \mathbf{using} (c); s^*$$

An enclosing using block  $\mathbf{using} (c); s^*$  records the cleanup stack  $c$  specified as an operand to the using statement, as well as the sequence of statements  $s^*$  that followed the using statement.

$$\text{cleanupActionBlock} ::= \mathbf{cleanup} (c, \sigma, V, s^*); s^*$$

An enclosing cleanup action block  $\mathbf{cleanup} (c, \sigma, V, s^*); s^*$  is in the sequence of enclosing blocks if the thread has finished executing the body of a using statement and is executing a cleanup action. The block records the cleanup stack  $c$  being processed, the subsystem  $\sigma$  that was current prior to the using statement, the variable environment  $V$  that was current on exit from the using statement's body, and sequence of statements  $s^*$  to be resumed after processing the cleanup action, and the sequence of statements  $s^*$  that followed the using statement. The sequence of statements  $s^*$  is either the empty sequence or a  $\mathbf{throw}$  statement. It indicates whether the using statement's body completed normally or abruptly.

## 3.2 Memory model

In this formalization, we attempt to be sound with respect to Java’s weak memory consistency model, which specifies that the executions of multithreaded Java programs that contain data races are not necessarily sequentially consistent. In other words, there is not necessarily a total order on all field accesses in an execution such that each read yields the value written by the most recent preceding write of the same field. Therefore, an execution state in our formalization does not include a heap. The step relation of our formalization does not determine the values yielded by field read statements. Rather, we define the set of valid executions as being the set of sequences of execution states that obey both the step relation and the Java Memory Model. We do not formalize the Java Memory Model in this paper.

## 3.3 Initial execution state

In an execution’s initial state, there are no allocated objects. There is a single subsystem, called the global root subsystem. Its parent is itself. There are no other allocated subsystems, no failed subsystems, and no allocated cleanup stacks. Furthermore, there is a single thread, called the main thread. Its current subsystem is the global root subsystem, its variable environment maps each variable name to the null reference, its current sequence of statements is the program being executed, its list of enclosing blocks is empty, and its list of enclosing method executions is empty as well.

Formally, the initial state for a program  $\bar{s}$  is  $(\emptyset, \{(\sigma, \sigma)\}, \emptyset, \emptyset, \{(\sigma, (\lambda x. \text{null}), \bar{s}, \epsilon, \epsilon)\})$ , for some global root subsystem  $\sigma$ .

# 4 Step Rules

This section introduces the step rules for the general programming statements (Section 4.1), the concurrent programming statements (Section 4.2), the conventional exception handling statements (Section 4.3), the subsystems statements (Section 4.4), and the cleanup statements (Section 4.5).

## 4.1 General Programming

In this section, we specify the general programming statements: the local variable to local variable assignment statements, the object creation statements, the method call statements, the field read statements, and the field write statements.

### 4.1.1 Local Variable to Local Variable Assignment Statements

$$\text{localCopy} ::= x := x;$$

A statement  $x := x'$ ; assigns the value of local variable  $x'$  to local variable  $x$ .

Formally:

$$\begin{array}{l} \text{LOCALCOPY} \\ (L, \Sigma, \Phi, C, T \triangleleft (tid, \sigma, V, x := x'; \bar{s}, \bar{b}, \bar{F})) \rightarrow \\ (L, \Sigma, \Phi, C, T \triangleleft (tid, \sigma, V(x := V(x')), \bar{s}, \bar{b}, \bar{F})) \end{array}$$

### 4.1.2 Object Creation Statements

$$\text{objectCreation} ::= x := \text{new}\{ \text{method}^* \};$$

An object creation statement  $x := \mathbf{new}\{ \mathit{body} \}$ ; picks an object reference  $o$  whose body is  $\mathit{body}$  and assigns  $o$  to  $x$ .

The step rule is as follows:

$$\frac{\text{OBJECTCREATION} \quad \mathbf{body}(o) = \mathit{body}}{(L, \Sigma, \Phi, C, T \triangleleft (tid, \sigma, V, x := \mathbf{new}\{ \mathit{body} \}; \bar{s}, \bar{b}, \bar{F})) \rightarrow (L, \Sigma, \Phi, C, T \triangleleft (tid, \sigma, V(x := o), \bar{s}, \bar{b}, \bar{F}))}$$

Note on the formalization: A function  $\mathbf{body}$  is assumed to exist which maps each object reference to a sequence of methods. It is assumed that for each sequence of methods  $\mathit{body}$ , there are infinitely many object references  $o$  such that  $\mathbf{body}(o) = \mathit{body}$ .

**Execution Validity Rule 1.** *There are no two object creation steps that pick the same object.*

An object's body does not specify its fields. Rather, all fields conceptually exist in all objects; reading a field before it is written yields a null reference. What happens in the case of data races (i.e. unsynchronized access by different threads to the same field of the same object) is as specified by the Java Memory Model.

#### 4.1.3 Method Call Statements

$$\mathit{call} ::= x := x.m(x^*);$$

A method call  $x := x'.m(\bar{x})$ ; looks up the value of the target variable  $x'$  and the argument variables  $\bar{x}$  in the local variable environment and looks up the parameter list and the body of  $m$  in the target object's body. If the target variable is not bound to an object, or if method  $m$  does not exist in the body of the target object, the below step rule does not apply and the only rule that applies is rule FAIL, i.e., an exception is thrown.

If the lookups succeed, the current activation record is pushed onto the call stack, along with the name of the variable  $x$  into which the result value should be copied when the call returns. In the new current activation record, the local variable environment maps the local variable name  $\mathit{this}$  to the target object, the parameter names to the values of the argument variables, and all other variable names to the null reference. The new current sequence of statements is the body of the method being called.

$$\frac{\text{CALL} \quad V(x') = o \quad \mathbf{body}(o) = \mathit{body} \quad m \in \text{dom}(\mathit{body}) \quad \mathit{body}(m) = (\bar{p}) \{ \bar{s}' \}}{(L, \Sigma, \Phi, C, T \triangleleft (tid, \sigma, V, x := x'.m(\bar{x}); \bar{s}, \bar{b}, \bar{F})) \rightarrow (L, \Sigma, \Phi, C, T \triangleleft (tid, \sigma, (\lambda x. \text{null})(\mathit{this} := V(x'), \bar{p} := V(\bar{x}), \bar{s}', \epsilon, (V, x, \bar{s}, \bar{b}) \cdot \bar{F}))}$$

When a thread has finished executing a method's body, it returns to the caller. Specifically, it pops the top activation record from the call stack and installs it as the current activation record. It then assigns the value of the variable named  $\mathit{result}$  in the callee's variable environment into the variable whose name was stored in the call stack, in the caller's variable environment.

$$\frac{\text{RETURN} \quad (L, \Sigma, \Phi, C, T \triangleleft (tid, \sigma, V, \epsilon, \epsilon, (V', x, \bar{s}, \bar{b}) \cdot \bar{F})) \rightarrow (L, \Sigma, \Phi, C, T \triangleleft (tid, \sigma, V'(x := V(\mathit{result})), \bar{s}, \bar{b}, \bar{F}))}$$

If the execution of a method body completed abruptly with an exception, the exception is propagated to the caller.

$$\begin{array}{c} \text{RETURN-ABRUPT} \\ (L, \Sigma, \Phi, C, T \triangleleft (tid, \sigma, V, \mathbf{throw}; , \epsilon, (V', x, \bar{s}, \bar{b}) \cdot \bar{F})) \rightarrow \\ (L, \Sigma, \Phi, C, T \triangleleft (tid, \sigma, V', \mathbf{throw}; , \bar{b}, \bar{F})) \end{array}$$

#### 4.1.4 Field Read Statements

$$\text{read} ::= x := x.f;$$

A field read statement  $x := x'.f$ ; reads field  $f$  of object  $x'$  and assigns the resulting value to variable  $x$ . If variable name  $x'$  is not bound to an object, the below step rule does not apply and only rule FAIL applies, i.e., an exception is thrown.

$$\begin{array}{c} \text{READ} \\ \frac{V(x') = o}{(L, \Sigma, \Phi, C, T \triangleleft (tid, \sigma, V, x := x'.f; \bar{s}, \bar{b}, \bar{F})) \rightarrow} \\ (L, \Sigma, \Phi, C, T \triangleleft (tid, \sigma, V(x := v), \bar{s}, \bar{b}, \bar{F})) \end{array}$$

**Execution Validity Rule 2.** *The values resulting from the field reads are determined as per the Java Memory Model.*

#### 4.1.5 Field Write Statements

$$\text{write} ::= x.f := x;$$

A field write statement  $x'.f := x$ ; writes the value of variable  $x$  into field  $f$  of the object referred to by the value of  $x'$ . If variable name  $x'$  is not bound to an object reference, the below step rule does not apply and only rule FAIL applies, i.e., an exception is thrown.

$$\begin{array}{c} \text{WRITE} \\ \frac{V(x') = o}{(L, \Sigma, \Phi, C, T \triangleleft (tid, \sigma, V, x'.f := x; \bar{s}, \bar{b}, \bar{F})) \rightarrow} \\ (L, \Sigma, \Phi, C, T \triangleleft (tid, \sigma, V, \bar{s}, \bar{b}, \bar{F})) \end{array}$$

The only effect of this statement is that it influences the values resulting from field reads as per the Java Memory Model.

## 4.2 Concurrent Programming

### 4.2.1 Thread creation statements

$$\text{fork} ::= \mathbf{fork} \{ s^* \}$$

A thread creation statement  $\mathbf{fork} \{ \bar{s}' \}$  starts a new thread and executes  $\bar{s}'$  in it. The new thread's initial current subsystem is a newly created child subsystem of the current thread's current subsystem.

Note: An intended property of the formalization is that in each execution state, the set of failed subsystems is closed under the child subsystem relation. That is, if a subsystem is marked as failed, then so are all of its descendants. Therefore, the intention is that no rule creates a new child of a subsystem that has failed. For this reason, the below rule applies

only if the current thread's current subsystem has not failed. If the current thread's current subsystem has failed, only rule FAIL applies, i.e. an exception is thrown.

$$\text{FORK} \frac{tid' \notin \text{dom}(T) \quad tid' \neq tid \quad \sigma' \notin \text{dom}(\Sigma) \quad \sigma \notin \Phi}{(H, L, \Sigma, \Phi, C, T \triangleleft (tid, \sigma, V, \mathbf{fork} \{ \bar{s}' \} \bar{s}, \bar{b}, \bar{F})) \rightarrow (H, L, \Sigma(\sigma' := \sigma), \Phi, C, T \triangleleft (tid', \sigma', V, \bar{s}', \epsilon, \epsilon) \triangleleft (tid, \sigma, V, \bar{s}, \bar{b}, \bar{F}))}$$

The premises of rule FORK state that the new thread's identifier  $tid'$  is distinct from the identifiers of the existing threads. They further state that the new thread's initial current subsystem is a fresh subsystem, and that the current thread's current subsystem has not yet failed.

If a thread completes abruptly, its subsystem and that subsystem's descendants are marked as failed.

$$\text{FORK-ABRUPT} \frac{\sigma \notin \Phi}{(H, L, \Sigma, \Phi, C, T \triangleleft (tid, \sigma, V, \mathbf{throw}; , \epsilon, \epsilon) \rightarrow (H, L, \Sigma, \Phi \cup (\Sigma^{-1})^*(\sigma), C, T \triangleleft (tid, \sigma, V, \mathbf{throw}; , \epsilon, \epsilon))}$$

The above rule applies only if the subsystem has not yet failed, to avoid spurious infinite executions.

#### 4.2.2 Synchronization statements

$$lock ::= \mathbf{lock} (x) \{ s^* \}$$

A lock statement  $\mathbf{lock} (x) \{ \bar{s}' \}$  waits until no thread holds the lock of the object bound to variable  $x$ , and then acquires the lock. It then executes the lock statement's body  $\bar{s}'$ .

$$\text{LOCK} \frac{V(x) = o \quad o \notin \text{dom}(L)}{(H, L, \Sigma, \Phi, C, T \triangleleft (tid, \sigma, V, \mathbf{lock} (x) \{ \bar{s}' \} \bar{s}, \bar{b}, \bar{F})) \rightarrow (H, L(o := tid), \Sigma, \Phi, C, T \triangleleft (tid, \sigma, V, \bar{s}', (\mathbf{lock} (o); \bar{s}) \cdot \bar{b}, \bar{F}))}$$

Notice that, in contrast to **synchronized** statements in Java and **lock** statements in C#, lock statements in our formalization are not re-entrant. This serves to simplify the formalization; lock re-entrancy does not cause significant difficulties for the subsystems approach.

When a lock statement's body completes normally, the lock is released and the sequence of statements that followed the lock statement is executed.

$$\text{UNLOCK} \frac{}{(H, L, \Sigma, \Phi, C, T \triangleleft (tid, \sigma, V, \epsilon, (\mathbf{lock} (o); \bar{s}) \cdot \bar{b}, \bar{F})) \rightarrow (H, L \setminus \{(o, tid)\}, \Sigma, \Phi, C, T \triangleleft (tid, \sigma, V, \bar{s}, \bar{b}, \bar{F}))}$$

Similarly, when a lock statement's body completes abruptly, the lock is released and the exception is propagated.

$$\text{UNLOCK-ABRUPT} \frac{}{(H, L, \Sigma, \Phi, C, T \triangleleft (tid, \sigma, V, \mathbf{throw}; , (\mathbf{unlock} (x), \bar{s}) \cdot \bar{b}, \bar{F})) \rightarrow (H, L \setminus \{(o, tid)\}, \Sigma, \Phi, C, T \triangleleft (tid, \sigma, V, \mathbf{throw}; , \bar{b}, \bar{F}))}$$

## 4.3 Conventional Exception Handling Statements

### 4.3.1 Try-catch statements

$$\text{tryCatch} ::= \mathbf{try} \{ s^* \} \mathbf{catch} \{ s^* \}$$

A try-catch statement  $\mathbf{try} \{ \bar{s}' \} \mathbf{catch} \{ \bar{s}'' \}$  starts by executing the try block  $\bar{s}'$ . It does so in a newly created child subsystem of the current subsystem.

$$\begin{array}{c} \text{TRY} \\ \hline \sigma' \notin \text{dom}(\Sigma) \\ \hline (H, L, \Sigma, \Phi, C, T \triangleleft (tid, \sigma, V, \mathbf{try} \{ \bar{s}' \} \mathbf{catch} \{ \bar{s}'' \} \bar{s}, \bar{b}, \bar{F})) \rightarrow \\ (H, L, \Sigma(\sigma' := \sigma), \Phi, C, T \triangleleft (tid, \sigma', V, \bar{s}', (\mathbf{try}(\sigma) \mathbf{catch} \{ \bar{s}'' \} \bar{s}) \cdot \bar{b}, \bar{F})) \end{array}$$

If a try-catch statement's try block completes normally, and the subsystem that was current prior to the try-catch statement has not failed, the subsystem that was current prior to the try-catch statement becomes again the current subsystem, the catch block is skipped, and the sequence of statements that followed the try-catch statement is executed.

$$\begin{array}{c} \text{TRY-COMPLETE-NORMAL} \\ \hline \sigma' \notin \Phi \\ \hline (H, L, \Sigma, \Phi, C, T \triangleleft (tid, \sigma, V, \epsilon, (\mathbf{try}(\sigma') \mathbf{catch} \{ \bar{s}' \} \bar{s}) \cdot \bar{b}, \bar{F})) \rightarrow \\ (H, L, \Sigma, \Phi, C, T \triangleleft (tid, \sigma', V, \bar{s}, \bar{b}, \bar{F})) \end{array}$$

If a try-catch statement's try block completes abruptly, and the subsystem that was current prior to the try-catch statement has not failed, the current subsystem is marked as failed, the subsystem that was current prior to the try-catch statement again becomes the current subsystem, and the catch block is executed.

$$\begin{array}{c} \text{CATCH} \\ \hline \sigma' \notin \Phi \\ \hline (H, L, \Sigma, \Phi, C, T \triangleleft (tid, \sigma, V, \mathbf{throw};, (\mathbf{try}(\sigma') \mathbf{catch} \{ \bar{s}' \} \bar{s}) \cdot \bar{b}, \bar{F})) \rightarrow \\ (H, L, \Sigma, \Phi \cup (\Sigma^{-1})^*(\sigma), C, T \triangleleft (tid, \sigma', V, \bar{s}', \bar{b}, \bar{F})) \end{array}$$

Notice that the above two rules apply only if the subsystem that was current prior to the try-catch block has not failed. If it has, only rule CATCH-FAIL applies, i.e., the current subsystem is marked as failed, the catch block is skipped, and an exception is thrown. This serves to ensure that if at some point  $A$  in an execution, a failure occurs in a thread  $tid$  with current subsystem  $\sigma$ , then if at any later point  $B$  in the execution, subsystem  $\sigma$  or a descendant of  $\sigma$  is the current subsystem of any thread  $tid'$ , then either  $A$  does not happen before  $B$  (i.e.,  $A$  and  $B$  are concurrent), or  $tid'$  is executing a throw statement.

$$\begin{array}{c} \text{CATCH-FAIL} \\ \hline (H, L, \Sigma, \Phi, C, T \triangleleft (tid, \sigma, V, \bar{s}'', (\mathbf{try}(\sigma') \mathbf{catch} \{ \bar{s}' \} \bar{s}) \cdot \bar{b}, \bar{F})) \rightarrow \\ (H, L, \Sigma, \Phi \cup (\Sigma^{-1})^*(\sigma), C, T \triangleleft (tid, \sigma', V, \mathbf{throw};, \bar{b}, \bar{F})) \end{array}$$

Note: The fact that the above rule applies not just to the case where  $\bar{s}'' = \epsilon$  and the case where  $\bar{s}'' = \mathbf{throw}$ ; does not matter, given rule FAIL.

### 4.3.2 Throw statements

Failures may occur at any point; this is reflected by rule FAIL. This also means that our Fail Fast feature does not appear specifically from the semantics; indeed, the Fail Fast feature merely increases the probability that an exception occurs in a thread, it does not enable

any executions that are not otherwise enabled. (Notice that this would be different if our formalization specified the type of exception thrown. This formalization does not track the type of exception being thrown because that is not relevant for dependency safety.)

$throw ::= \mathbf{throw};$

A throw statement cannot complete normally; it always fails.

$$\frac{\text{FAIL} \quad (\bar{s}) \neq (\mathbf{throw};)}{(H, L, \Sigma, \Phi, C, T \triangleleft (tid, \sigma, V, \bar{s}, \bar{b}, \bar{F})) \rightarrow (H, L, \Sigma, \Phi, C, T \triangleleft (tid, \sigma, V, \mathbf{throw};, \bar{b}, \bar{F}))}$$

The premise of the above rule prevents spurious infinite executions.

## 4.4 Subsystems Statements

### 4.4.1 Current subsystem query statements

$currentSubsystem ::= x := \mathbf{current\_subsystem};$

A current subsystem query statement assigns the current subsystem to variable  $x$ .

$$\frac{\text{CURRENTSUBSYSTEM}}{(H, L, \Sigma, \Phi, C, T \triangleleft (tid, \sigma, V, x := \mathbf{current\_subsystem}; \bar{s}, \bar{b}, \bar{F})) \rightarrow (H, L, \Sigma, \Phi, C, T \triangleleft (tid, \sigma, V(x := \sigma), \bar{s}, \bar{b}, \bar{F}))}$$

### 4.4.2 Subsystem creation statements

$newSubsystem ::= x := \mathbf{new\_subsystem}(x);$

New subsystems are created implicitly when entering a try block and when starting a new thread; however, new subsystems may also be created explicitly using a subsystem creation statement  $x := \mathbf{new\_subsystem}(x')$ . It creates a new subsystem whose parent is the subsystem bound to variable  $x'$ . It fails if  $x'$  is not bound to a subsystem, or if the subsystem bound to  $x'$  has failed. This ensures that in each execution state, if a subsystem is marked as failed, then so are all of its descendants.

$$\frac{\text{NEWSUBSYSTEM} \quad V(x') = \sigma' \quad \sigma' \notin \Phi \quad \sigma'' \notin \text{dom}(\Sigma)}{(H, L, \Sigma, \Phi, C, T \triangleleft (tid, \sigma, V, x := \mathbf{new\_subsystem}(x'); \bar{s}, \bar{b}, \bar{F})) \rightarrow (H, L, \Sigma(\sigma'' := \sigma'), \Phi, C, T \triangleleft (tid, \sigma, V(x := \sigma''), \bar{s}, \bar{b}, \bar{F}))}$$

### 4.4.3 Enter statements

$enter ::= \mathbf{enter}(x) \{ s^* \}$

An enter statement  $\mathbf{enter}(x) \{ \bar{s}' \}$  first checks that variable  $x$  is bound to a subsystem and that the subsystem bound to  $x$  has not failed. Otherwise, it fails. Then, it installs the subsystem bound to  $x$  as the current subsystem and executes its body.

$$\frac{\text{ENTER} \quad V(x) = \sigma' \quad \sigma' \notin \Phi}{(H, L, \Sigma, \Phi, C, T \triangleleft (tid, \sigma, V, \mathbf{enter}(x) \{ \bar{s}' \} \bar{s}, \bar{b}, \bar{F})) \rightarrow (H, L, \Sigma, \Phi, C, T \triangleleft (tid, \sigma', V, \bar{s}', (\mathbf{enter}(\sigma); \bar{s}) \cdot \bar{b}, \bar{F}))}$$

When an enter statement's body completes normally, and the subsystem that was current prior to the enter statement has not failed, the subsystem that was current prior to the enter statement is installed again as the current subsystem and the sequence of statements that followed the enter statement is executed.

$$\frac{\text{ENTER-COMPLETE-NORMAL} \quad \sigma' \notin \Phi}{(H, L, \Sigma, \Phi, C, T \triangleleft (tid, \sigma, V, \epsilon, (\mathbf{enter}(\sigma'); \bar{s}) \cdot \bar{b}, \bar{F})) \rightarrow (H, L, \Sigma, \Phi, C, T \triangleleft (tid, \sigma', V, \bar{s}, \bar{b}, \bar{F}))}$$

When an enter statement's body completes abruptly, the current subsystem is marked as failed, the subsystem that was current prior to the enter statement is installed again as the current subsystem, and the exception is propagated.

$$\frac{\text{ENTER-COMPLETE-ABRUPT} \quad (H, L, \Sigma, \Phi, C, T \triangleleft (tid, \sigma, V, \mathbf{throw};, (\mathbf{enter}(\sigma'); \bar{s}) \cdot \bar{b}, \bar{F})) \rightarrow (H, L, \Sigma, \Phi \cup (\Sigma^{-1})^*(\sigma), C, T \triangleleft (tid, \sigma', V, \mathbf{throw};, \bar{b}, \bar{F}))}$$

#### 4.4.4 Reenter statements

$$\mathit{reenter} ::= \mathbf{reenter}(x) \{ s^* \}$$

A reenter statement is like an enter statement, except that it does not propagate exceptions that occur in its body. However, it may fail if the subsystem that was current prior to the reenter statement has failed.

A reenter statement  $\mathbf{reenter}(x) \{ \bar{s}' \}$  first checks that variable  $x$  is bound to a subsystem and that this subsystem has not failed. Otherwise, it fails. Then, it installs this subsystem as the current subsystem and it executes its body  $\bar{s}'$ .

$$\frac{\text{REENTER} \quad V(x) = \sigma' \quad \sigma' \notin \Phi}{(H, L, \Sigma, \Phi, C, T \triangleleft (tid, \sigma, V, \mathbf{reenter}(x) \{ \bar{s}' \} \bar{s}, \bar{b}, \bar{F})) \rightarrow (H, L, \Sigma, \Phi, C, T \triangleleft (tid, \sigma', V, \bar{s}', (\mathbf{reenter}(\sigma), \bar{s}) \cdot \bar{b}, \bar{F}))}$$

When the body of a reenter statement completes normally, and the subsystem that was current prior to the reenter statement has not failed, this subsystem is again installed as the current subsystem and the sequence of statements that followed the reenter statement is executed.

$$\frac{\text{REENTER-COMPLETE-NORMAL} \quad \sigma' \notin \Phi}{(H, L, \Sigma, \Phi, C, T \triangleleft (tid, \sigma, V, \epsilon, (\mathbf{reenter}(\sigma'), \bar{s}) \cdot \bar{b}, \bar{F})) \rightarrow (H, L, \Sigma, \Phi, C, T \triangleleft (tid, \sigma', V, \bar{s}, \bar{b}, \bar{F}))}$$

When the body of a reenter statement completes abruptly, and the subsystem that was current prior to the reenter statement has not failed and is not the current subsystem or a descendant of the current subsystem, then the current subsystem and its descendants are marked as failed, the subsystem that was current prior to the reenter statement is installed again as the current subsystem, and the sequence of statements that followed the reenter statement is executed.

$$\frac{\text{REENTER-COMPLETE-ABRUPT} \quad \sigma' \notin \Phi \cup (\Sigma^{-1})^*(\sigma)}{(H, L, \Sigma, \Phi, C, T \triangleleft (tid, \sigma, V, \mathbf{throw};, (\mathbf{reenter}(\sigma'), \bar{s}) \cdot \bar{b}, \bar{F})) \rightarrow (H, L, \Sigma, \Phi \cup (\Sigma^{-1})^*(\sigma), C, T \triangleleft (tid, \sigma', V, \bar{s}, \bar{b}, \bar{F}))}$$

When the body of a reenter statement completes normally, a failure may occur. (This is always possible, but it is the only rule that applies if the subsystem that was current prior to the reenter statement has failed.) In this case, an exception is thrown.

$$\begin{array}{c} \text{REENTER-COMplete-NORMAL-FAIL} \\ (H, L, \Sigma, \Phi, C, T \triangleleft (tid, \sigma, V, \epsilon, (\mathbf{reenter}(\sigma'), \bar{s}) \cdot \bar{b}, \bar{F})) \rightarrow \\ (H, L, \Sigma, \Phi, C, T \triangleleft (tid, \sigma', V, \mathbf{throw};, \bar{b}, \bar{F})) \end{array}$$

When the body of a reenter statement completes abruptly, a failure may occur. (This is always possible, but it is the only rule that applies if the subsystem that was current prior to the reenter statement has failed.) In this case, the current subsystem and its descendants are marked as failed and the exception is propagated.

$$\begin{array}{c} \text{REENTER-COMplete-ABRUPT-FAIL} \\ (H, L, \Sigma, \Phi, C, T \triangleleft (tid, \sigma, V, \mathbf{throw};, (\mathbf{reenter}(\sigma'), \bar{s}) \cdot \bar{b}, \bar{F})) \rightarrow \\ (H, L, \Sigma, \Phi \cup (\Sigma^{-1})^*(\sigma), C, T \triangleleft (tid, \sigma', V, \mathbf{throw};, \bar{b}, \bar{F})) \end{array}$$

## 4.5 Cleanup Statements

### 4.5.1 Cleanup stack creation statements

$newCleanupStack ::= x := \mathbf{new\_cleanupstack};$

The statement  $x := \mathbf{new\_cleanupstack};$  creates a new cleanup stack and assigns it to variable  $x$ . It is initially empty.

$$\begin{array}{c} \text{NEWCLEANUPSTACK} \\ c \notin \text{dom}(C) \\ \hline (H, L, \Sigma, \Phi, C, T \triangleleft (tid, \sigma, V, x := \mathbf{new\_cleanup\_stack}; \bar{s}, \bar{b}, \bar{F})) \rightarrow \\ (H, L, \Sigma, \Phi, C(c := \epsilon), T \triangleleft (tid, \sigma, V, \bar{s}, \bar{b}, \bar{F})) \end{array}$$

### 4.5.2 Push statements

$push ::= x.\mathbf{push}(x);$

The statement  $x.\mathbf{push}(x);$  first checks that variable  $x$  is bound to a cleanup stack. It then pushes a cleanup action onto the cleanup stack bound to variable  $x$ , which will look up the object bound to variable  $x$  in the current variable environment and execute method `dispose` on it in the current subsystem. If  $x$  is not bound to an object, or this object does not implement a `dispose` method, the cleanup action will fail at cleanup time.

$$\begin{array}{c} \text{PUSH} \\ V(x) = c \\ \hline (H, L, \Sigma, \Phi, C, T \triangleleft (tid, \sigma, V, x.\mathbf{push}(x); \bar{s}, \bar{b}, \bar{F})) \rightarrow \\ (H, L, \Sigma, \Phi, C(c := (\sigma, V, x) \cdot C(c)), T \triangleleft (tid, \sigma, V, \bar{s}, \bar{b}, \bar{F})) \end{array}$$

### 4.5.3 Using statements

$using ::= \mathbf{using}(x) \{ s^* \}$

A using statement  $\mathbf{using}(x) \{ s^* \}$  first checks that variable  $x$  is bound to a cleanup stack. Otherwise, it fails. It then executes its body  $\bar{s}'$ .

$$\begin{array}{c} \text{USING} \\ V(x) = c \\ \hline (H, L, \Sigma, \Phi, C, T \triangleleft (tid, \sigma, V, \mathbf{using}(x) \{ \bar{s}' \} \bar{s}, \bar{b}, \bar{F})) \rightarrow \\ (H, L, \Sigma, \Phi, C(c := (\sigma, V, x) \cdot C(c)), T \triangleleft (tid, \sigma, V, \bar{s}', (\mathbf{run}(c), \bar{s}) \cdot \bar{b}, \bar{F})) \end{array}$$

When a using statement's body completes (either normally or abruptly), there are two cases to be considered, depending on whether the cleanup stack specified as the using statement's operand is empty or non-empty.

If the cleanup stack is non-empty, there are two further cases to be considered

If the cleanup stack specified as the using statement's operand is non-empty, and the top cleanup action's subsystem has not failed, the current subsystem, variable environment, and sequence of statements are saved, and the top cleanup action is popped and executed. Specifically, the cleanup action's subsystem and variable environment are installed as the current subsystem and the current variable environment and a call of method `dispose` on the cleanup action's target variable is executed.

USING-COMPLETE-NONEMPTY

$$\frac{\bar{s}' \in \{\epsilon, \mathbf{throw};\} \quad \sigma' \notin \Phi}{(H, L, \Sigma, \Phi, C(c := (\sigma', V', x) \cdot \bar{a}), T \triangleleft (tid, \sigma, V, \bar{s}', (\mathbf{using} (c); \bar{s}) \cdot \bar{b}, \bar{F})) \rightarrow (H, L, \Sigma, \Phi, C(c := \bar{a}), T \triangleleft (tid, \sigma', V', \mathbf{result} := x.\mathbf{dispose}();, (\mathbf{cleanup} (c, \sigma, V, \bar{s}'); \bar{s}) \cdot \bar{b}, \bar{F}))}$$

If the cleanup stack specified as the using statement's operand is non-empty, and the top cleanup action's subsystem has failed, the cleanup action is popped and the current sequence of statements is replaced with a throw statement, indicating that after cleanup completes, an exception will be thrown.

USING-COMPLETE-NONEMPTY-FAIL

$$\frac{\bar{s}' \in \{\epsilon, \mathbf{throw};\} \quad \sigma' \in \Phi}{(H, L, \Sigma, \Phi, C(c := (\sigma', V', x) \cdot \bar{a}), T \triangleleft (tid, \sigma, V, \bar{s}', (\mathbf{using} (c); \bar{s}) \cdot \bar{b}, \bar{F})) \rightarrow (H, L, \Sigma, \Phi, C(c := \bar{a}), T \triangleleft (tid, \sigma, V, \mathbf{throw};, (\mathbf{using} (c); \bar{s}) \cdot \bar{b}, \bar{F}))}$$

If the cleanup stack specified as the using statement's operand is empty, there are two further cases to be considered, depending on whether the using statement's body (and any preceding cleanup actions) have completed successfully or not.

If the using statement's body and all cleanup actions have completed successfully, then the sequence of statements that followed the using statement is executed.

USING-COMPLETE-EMPTY-NORMAL

$$\frac{}{(H, L, \Sigma, \Phi, C(c := \epsilon), T \triangleleft (tid, \sigma, V, \epsilon, (\mathbf{using} (c); \bar{s}) \cdot \bar{b}, \bar{F})) \rightarrow (H, L, \Sigma, \Phi, C(c := \epsilon), T \triangleleft (tid, \sigma, V, \bar{s}, \bar{b}, \bar{F}))}$$

If either the using statement's body or any of the cleanup actions have not completed successfully, the exception is propagated.

USING-COMPETE-EMPTY-ABRUPT

$$\frac{}{(H, L, \Sigma, \Phi, C(c := \epsilon), T \triangleleft (tid, \sigma, V, \mathbf{throw};, (\mathbf{using} (c); \bar{s}) \cdot \bar{b}, \bar{F})) \rightarrow (H, L, \Sigma, \Phi, C(c := \epsilon), T \triangleleft (tid, \sigma, V, \mathbf{throw};, \bar{b}, \bar{F}))}$$

When a cleanup action completes normally, the saved subsystem, variable environment, and sequence of statements are restored. The effect is that the next cleanup action will be processed, or, if the cleanup stack is empty, the using statement will complete normally, as per the above rules.

CLEANUPACTION-COMPLETE-NORMAL

$$\frac{}{(H, L, \Sigma, \Phi, C, T \triangleleft (tid, \sigma', V', \epsilon, (\mathbf{cleanup} (c, \sigma, V, \bar{s}'); \bar{s}) \cdot \bar{b}, \bar{F})) \rightarrow (H, L, \Sigma, \Phi, C, T \triangleleft (tid, \sigma, V, \bar{s}', (\mathbf{using} (c); \bar{s}) \cdot \bar{b}, \bar{F}))}$$

When a cleanup action completes abruptly, the current subsystem and its descendants are marked as failed, and the saved subsystem and variable environment are restored. The

saved sequence of statements is not restored; rather, an exception is thrown. The effect is that the next cleanup action will be processed, or, if the cleanup stack is empty, the using statement will complete abruptly, as per the above rules.

$$\begin{array}{l} \text{CLEANUPACTION-COMplete-ABRUPT} \\ (H, L, \Sigma, \Phi, C, T \triangleleft (tid, \sigma', V', \mathbf{throw};, (\mathbf{cleanup} (c, \sigma, V, \bar{s}'); \bar{s}) \cdot \bar{b}, \bar{F})) \rightarrow \\ (H, L, \Sigma, \Phi \cup (\Sigma^{-1})^*(\sigma'), C, T \triangleleft (tid, \sigma, V, \mathbf{throw};, (\mathbf{using} (c); \bar{s}) \cdot \bar{b}, \bar{F})) \end{array}$$

## 5 Properties

In this section, we sketch a few properties of our language extension.

The current subsystem after the execution of a statement is the same as the current subsystem before the execution of the statement.

**Property 1** (Current subsystem is well-nested). *For each program  $P$  and for each execution  $E$  of  $P$ , and for each point  $j$  in  $E$ , and for each thread  $tid \in \text{dom}(T_i)$ , if  $j$  is a post-state for  $tid$  and for some pre-state  $i$  in  $E$ , then the current subsystem for  $tid$  at  $j$  in  $E$  equals the current subsystem for  $tid$  at  $i$  in  $E$ .*

The conservativity theorem states that enter statements do not add behavior. The soundness theorem states that enter statements remove non-dependency-safe behavior. (They may remove additional behavior as well, specifically if the programmer over-uses them.)

**Theorem 1** (Conservativity). *For each program  $P$  and for each execution  $E$  of  $P$ , the erasure of  $E$  is an execution of the erasure of  $P$ .*

*Proof.* By induction on the length of  $E$  and case analysis on the last step of  $E$ .

The proof relies on the property of our formalization that the semantics does not distinguish between different types of exceptions being thrown. As a result, if, for example, in  $E$  an enter block throws an exception because the subsystem being entered is marked as failed, this corresponds in the erasure of  $E$  with an asynchronous exception thrown by the JVM.  $\square$

**Definition 1.** *A sequence of execution states  $E = S_0, \dots, S_n$  is an execution of a program  $P$  if  $S_0$  is an initial state for  $P$  and each pair of consecutive states satisfies the step relation.*

$$\frac{S_0 \in \text{Initial}_P \quad (\forall k \in \{1, \dots, n\}. S_{k-1} \rightarrow S_k)}{S_0, \dots, S_n \in \text{Exec}_P}$$

**Definition 2.** *A sequence of execution states is an execution if it is an execution of some program.*

$$\frac{E \in \text{Exec}_P}{E \in \text{Exec}}$$

**Definition 3.** *Given an execution  $E = S_0, \dots, S_n$ , the thread execution points  $\text{Pts}(E)$  of  $E$  are the pairs  $(i, tid)$  where  $0 \leq i \leq n$  and state  $S_i$  contains a thread with thread identifier  $tid$ .*

$$\frac{0 \leq i \leq n \quad S_i = (L, \Sigma, \Phi, T) \quad (tid, \sigma, V, \bar{s}, \bar{b}, \bar{F}) \in T}{(i, tid) \in \text{Pts}(S_0, \dots, S_n)}$$

**Definition 4.** *Given an execution  $E = S_0, \dots, S_n$ , the steps of  $E$  are the integers  $k$  where  $1 \leq k \leq n$ .*

$$\frac{1 \leq k \leq n}{k \in \text{Steps}(S_0, \dots, S_n)}$$

**Definition 5.** Given an execution  $E = S_0, \dots, S_n$ , a step  $k$  of  $E$  is performed by a thread  $tid$  if  $tid$ 's thread state is the only thread state that changes.

$$(L, \Sigma, \Phi, T \triangleleft (tid, \sigma, V, \bar{s}, \bar{b}, Fs)) \xrightarrow{tid} (L', \Sigma', \Phi', (T \cup T') \triangleleft (tid, \sigma', V', \bar{s}', \bar{b}', \bar{F}'))$$

**Property 2.** Each step in an execution is performed by exactly one thread.

$$\frac{S \rightarrow S'}{\exists! tid. S \xrightarrow{tid} S'}$$

*Proof.* By case analysis on the step rule.  $\square$

**Definition 6.** The happens-before nodes of an execution are the thread execution points and the steps.

$$HBNodes(E) = Pts(E) \cup Steps(E)$$

**Definition 7.** Given an execution  $E$ , the happens-before relation  $\xrightarrow{hb}_E$  collects the pairs of happens-before nodes of  $E$  derivable by the following rules:

- Any thread execution point of thread  $E$  happens-before its successor in the same thread.

$$\begin{array}{c} \text{HB-PTS-PROGORDER} \\ (i, tid) \xrightarrow{hb}_E (i + 1, tid) \end{array}$$

- The start of a step happens-before the step.

$$\begin{array}{c} \text{HB-START} \\ \frac{S_{k-1} \xrightarrow{tid} S_k}{(k-1, tid) \xrightarrow{hb}_E k} \end{array}$$

- A step happens-before its end.

$$\begin{array}{c} \text{HB-END} \\ \frac{S_{k-1} \xrightarrow{tid} S_k}{k \xrightarrow{hb}_E (k, tid)} \end{array}$$

- The creation of a thread happens-before the start of the thread.

$$\begin{array}{c} \text{HB-THREADSTART} \\ \frac{S_{k-1} = (L, \Sigma, \Phi, T \triangleleft (tid, \sigma, V, \mathbf{fork} \{ \bar{s}' \} \bar{s}, \bar{b}, \bar{F}))}{S_k = (L, \Sigma(\sigma' := \sigma), \Phi, T \triangleleft (tid', \sigma', V, \bar{s}', \epsilon, \epsilon) \triangleleft (tid, \sigma, V, \bar{s}, \bar{b}, \bar{F}))} \\ k \xrightarrow{hb}_E (k, tid') \end{array}$$

- An exit from a **lock** (o) statement happens-before a subsequent entry into a **lock** (o) statement.

$$\begin{array}{c} \text{HB-LOCK} \\ \frac{S_{k-1} = (-, -, -, - \triangleleft (tid, -, -, -, (\mathbf{lock} (o); -) \cdot \bar{b}, \bar{F}))}{S_k = (-, -, -, - \triangleleft (tid, -, -, -, \bar{b}, \bar{F})) \quad S_{k'-1} = (-, -, -, - \triangleleft (tid', -, -, -, \bar{b}', \bar{F}'))} \\ \frac{S'_k = (-, -, -, - \triangleleft (tid', -, -, -, (\mathbf{lock} (o); -) \cdot \bar{b}', \bar{F}')) \quad k < k'}{k \xrightarrow{hb}_E k'} \end{array}$$

- The happens-before relation is transitive.

$$\text{HB-TRANS} \quad \frac{\nu_1 \xrightarrow{\text{hb}}_E \nu_2 \quad \nu_2 \xrightarrow{\text{hb}}_E \nu_3}{\nu_1 \xrightarrow{\text{hb}}_E \nu_3}$$

**Definition 8.** Thread  $tid$  is failing at thread execution point  $(i, tid)$  in execution  $E$  if the thread is propagating an exception.

$$\frac{S_i = (-, -, -, \triangleleft (tid, -, -, \mathbf{throw}; -, -, -))}{(i, tid) \in \text{Fail}_E}$$

**Definition 9.** Given a dependency relation  $D$ , an execution  $E$  is strongly dependency-safe if for each pair of computations  $A$  and  $B$  such that  $A$  happens-before  $B$  and  $B$  depends on  $A$ ,  $A$  does not fail in  $E$ .

$$\frac{\forall (A, B) \in D(E). A \xrightarrow{\text{hb}}_E B \Rightarrow A \notin \text{Fail}_E}{E \in \text{DepSafe}_D}$$

**Definition 10.** Given a dependency relation  $D$ , an execution  $E$  is weakly dependency-safe if for each pair of computations  $A$  and  $B$  such that  $A$  happens-before  $B$  and  $B$  depends on  $A$  and  $A$ 's immediately enclosing block (if any) is not a lock or using block, and  $B$ 's immediately enclosing block (if any) is not a lock or using block, in both cases disregarding blocks that enclose both  $A$  and  $B$ ,  $A$  does not fail in  $E$ .

**Definition 11.** The current subsystem of an execution  $E = S_0, \dots, S_n$  at a thread execution point  $(i, tid)$  is the current subsystem of thread  $tid$  in execution state  $S_i$ .

$$\frac{S_i = (-, -, -, T \triangleleft (tid, \sigma, -, -, -))}{\text{Subsystem}_E((i, tid)) = \sigma}$$

**Definition 12.** A subsystem  $\sigma'$  is a descendant of a subsystem  $\sigma$  in an execution  $E = S_0, \dots, S_n$  if  $\sigma$  is a transitive parent of  $\sigma'$  in the last execution state.

$$\frac{S_n = (-, \Sigma, -, -) \quad (\sigma', \sigma) \in \Sigma^*}{\text{DescendantOf}(\sigma', \sigma)}$$

**Definition 13.** Given a dependency relation  $D$ , an execution  $E$  uses subsystems correctly if for each pair of computations  $A$  and  $B$  such that  $A$  happens-before  $B$  and  $B$  depends on  $A$ ,  $A$  executes in some subsystem  $s$  and  $B$  executes in a descendant of  $s$ .

$$\frac{\forall (A, B) \in D(E). A \xrightarrow{\text{hb}}_E B \Rightarrow \text{DescendantOf}_E(\text{Subsystem}_E(B), \text{Subsystem}_E(A))}{E \in \text{Correct}_D}$$

**Lemma 1.** In any execution  $E = S_0, \dots, S_n$ , if in  $S_i$  a subsystem  $\sigma$  is marked as failed, then in each subsequent state,  $\sigma$  and its descendants are marked as failed.

$$\frac{S_i = (L, \Sigma, \Phi, T) \quad \sigma \in \Phi \quad i \leq j \quad S_j = (L', \Sigma', \Phi', T')}{(\Sigma^{-1})^*(\sigma) \subseteq \Phi}$$

*Proof.* By induction on the number of intervening steps and case analysis on the last step rule.  $\square$

**Theorem 2** (Soundness I). *Given a dependency relation  $D$ , if an execution uses subsystems correctly, then the execution is weakly dependency-safe.*

*Proof.* Consider a pair of computations  $A$  and  $B$  such that  $A$  happens-before  $B$  and  $B$  depends on  $A$  and  $A$ 's immediately enclosing block (if any) is not a lock or using block, and  $B$ 's immediately enclosing block (if any) is not a lock or using block, in both cases disregarding blocks that enclose both  $A$  and  $B$ . We need to prove that  $A$  does not fail in  $E$ . By contradiction. Assume  $A$  is failing. It suffices to prove for every prefix of some path of atomic happens-before edges (i.e. where the derivation of the edges does not use rule HB-TRANS) between  $A$  and  $B$ , that at the thread execution point  $C$  at the end of the prefix, one or more of the following hold:

- the thread is failing and the current subsystem is  $\sigma$  and one or more of the following hold:
  - there is no enclosing block, or
  - the immediately enclosing block is not a lock or using block, or
  - the immediately enclosing block encloses both  $A$  and  $B$
- subsystem  $\sigma$  and its descendants have been marked as failed and one or more of the following hold:
  - the current subsystem is not  $\sigma$  or a descendant of  $\sigma$ , or
  - $C$  is enclosed immediately by a lock block that does not enclose both  $A$  and  $B$ , or
  - the thread is failing.

This can be proved by induction on the length of the prefix and case analysis on the happens-before rule used to derive the last edge.  $\square$

**Theorem 3** (Soundness II). *For any program  $P$  and for any dependency relation  $D$ , if*

- $P$  uses subsystems correctly with respect to  $D$ , and
- the body of each lock statement in  $P$  is an enter statement, and
- the body of each using statement in  $P$  is an enter statement, and
- for any element  $(A, B)$  in  $D$ , neither  $A$  nor  $B$  are between the entry of a lock or using statement and the entry of the enter statement in its body, or between the exit from a lock or using statement and the exit from the enter statement in its body, or between two consecutive cleanup actions performed by the same using statement execution (i.e.,  $A$  and  $B$  are not internal points),

*then each execution  $E$  of  $P$  is strongly dependency-safe with respect to  $D$ .*

*Proof.* From Theorem 2, it follows that  $E$  is weakly dependency-safe. Therefore, it suffices to prove that  $A$ 's immediately enclosing block (if any) is not a lock or using block, and  $B$ 's immediately enclosing block (if any) is not a lock or using block. This follows immediately from the fact that  $A$  and  $B$  are not internal points.  $\square$

```

mainObject := new{
  createDatabase() { ... }
  getCommand() { ... }
  compute() { ... }
  showErrorMessage() { ... }
  main() {
    db := this.createDatabase();
    s := current_subsystem;
    _ := this.loop(db, s);
  }
  loop(db, s) {
    command := this.getCommand();
    try {
      ...
      this.compute();
      ...
      enter (s) {
        db.addEntry();
      }
      ...
    } catch {
      this.showErrorMessage();
    }
    _ := this.loop(db, s);
  }
};
_ := mainObject.main();

```

```

mainObject := new{
  createDatabase() { ... }
  getCommand() { ... }
  compute() { ... }
  showErrorMessage() { ... }
  main() {
    db := this.createDatabase();
    s := current_subsystem;
    _ := this.loop(db, s);
  }
  loop(db, s) {
    command := this.getCommand();
    fork {
      try {
        ...
        this.compute();
        ...
        lock (db) {
          enter (s) {
            db.addEntry();
          }
        }
        ...
      } catch {
        this.showErrorMessage();
      }
    }
    _ := this.loop(db, s);
  }
};
_ := mainObject.main();

```

Figure 1: (*left*) An example program illustrating the use of enter statements. Given the dependency relation where each *loop* execution depends on each preceding *addEntry* execution, this program is dependency safe. (*right*) An example program illustrating the use of enter statements in multithreaded programs. Given the dependency relation where each *addEntry* execution depends on each preceding *addEntry* execution, this program is dependency safe.

## 6 Example Programs

In this section we provide a few examples to illustrate our formal language. Specifically, we provide an example of the use of enter statements to achieve dependency safety in single-threaded programs (Figure 1, left) and in multithreaded programs (Figure 1, right), an example of the use of subsystems for achieving safe cancellation of computations (Figure 2), and two examples of the use of subsystems for safe and leak-free cleanup: one illustrating the provider-controlled cleanup pattern (Figure 3) and one illustrating the client-controlled cleanup pattern (Figure 4).

```

mainObject := new{
  createDatabase() { ... }
  createList() { ... }
  getCommand() { ... }
  isCancelAllCmd(cmd) { ... }
  compute() { ... }
  showErrorMessage() { ... }
  main() {
    db := this.createDatabase();
    tasks := this.createList();
    s := current_subsystem;
    _ := this.loop(db, tasks, s);
  }
  loop(db, tasks, s) {
    cmd := this.getCommand();
    b := this.isCancelAllCmd(cmd);
    cancelAll := new{
      run() {
        tasks := this.tasks;
        cancelTask := new{
          apply(t) {
            reenter (t) { throw; }
          }
        };
        _ := tasks.foreach(cancelTask);
        _ := tasks.clear();
      }
    };
    cancelAll.tasks := tasks;
    startTask := new{
      run() {
        mainObject := this.mainObject;
        db := this.db;
        tasks := this.tasks;
        s := this.s;
        cmd := this.cmd;
        task := new_subsystem(s);
        _ := tasks.add(task);
        fork {
          try {
            enter (task) {
              ...
              mainObject.compute();
              ...
              lock (db) {
                enter (s) {
                  db.addEntry();
                }
                ...
              }
            } catch {
              this.showErrorMessage();
            }
          }
        };
        startTask.mainObject := this;
        startTask.db := db;
        startTask.tasks := tasks;
        startTask.s := s;
        startTask.cmd := cmd;
        _ := b.ifThenElse(cancelAll, startTask);
        _ := this.loop(db, tasks, s);
      }
    };
    _ := mainObject.main();
  }
}

```

Figure 2: An example program illustrating the use of subsystems for safe cancellation. A subsystem can be cancelled by entering it using a `reenter` statement and throwing an exception in it. Given the dependency relation where each `addEntry` execution depends on each preceding `addEntry` execution, this program is dependency safe. If a task thread is executing an `addEntry` call at the time when the task subsystem is marked as failed, the thread continues executing; an exception is thrown in the task thread only when it exits the `enter (s)` statement.

```

mainObject := new{
  createWidgetManager() {
    m := new {
      allocWidget() { ... }
      deallocWidget(widget) { ... }
      usingWidget(clientCode) {
        s := current_subsystem;
        subsystem := this.subsystem;
        enter (subsystem) {
          widget := this.allocWidget();
          reenter (s) {
            _ := clientCode.run(widget);
          }
          _ := this.deallocWidget(widget);
        }
      }
    }
  };
  s := current_subsystem;
  m.subsystem := s;
  result := m;
}
getCommand() { ... }
compute() { ... }
showErrorMessage() { ... }
main() {
  m := this.createWidgetManager();
  _ := this.loop(m);
}
loop(m) {
  command := this.getCommand();
  try {
    ...
    this.compute();
    ...
    clientCode := new{
      run(widget) { ... }
    };
    m.usingWidget(clientCode);
    ...
  } catch {
    this.showErrorMessage();
  }
  _ := this.loop(m);
}
};
_ := mainObject.main();

```

Figure 3: An example program illustrating the use of enter statements and reenter statements to achieve safe and leak-free cleanup, using the provider-controlled cleanup pattern.

```

mainObject := new{
  createWidgetManager() {
    m := new {
      addWidget(w) { ... }
      removeWidget(w) { ... }
      allocWidget(cs) {
        s := this.subsystem;
        enter (s) {
          w := new{
            ...
            dispose() {
              m := this.m;
              _ := m.removeWidget(this);
            }
          };
          w.m := this;
          this.addWidget(w);
          cs.push(w);
        }
        result := w;
      }
    };
    s := current_subsystem;
    m.subsystem := s;
    result := m;
  }
}

getCommand() { ... }
compute() { ... }
showErrorMessage() { ... }
main() {
  m := this.createWidgetManager();
  _ := this.loop(m);
}
loop(m) {
  command := this.getCommand();
  try {
    ...
    this.compute();
    ...
    cs := new_cleanup_stack;
    using (cs) {
      w := m.allocWidget(cs);
      ...
    }
    ...
  } catch {
    this.showErrorMessage();
  }
  _ := this.loop(m);
}
};
_ := mainObject.main();

```

Figure 4: An example program illustrating the use of enter statements, cleanup stacks, push statements, and using statements to achieve safe and leak-free cleanup, using the client-controlled cleanup pattern.