

# Sound reasoning about unchecked exceptions

Bart Jacobs<sup>1</sup> Peter Müller<sup>2</sup> Frank Piessens<sup>1</sup>

<sup>1</sup>Katholieke Universiteit Leuven  
Belgium  
{bartj,frank}@cs.kuleuven.be

<sup>2</sup>Microsoft Research, Redmond  
USA  
mueller@microsoft.com

## Abstract

*In most software development projects, it is not feasible for developers to handle explicitly all possible unusual events which may occur during program execution, such as arithmetic overflow, highly unusual environment conditions, heap memory or call stack exhaustion, or asynchronous thread cancellation. Modern programming languages provide unchecked exceptions to deal with these circumstances safely and with minimal programming overhead. However, reasoning about programs in the presence of unchecked exceptions is difficult, especially in a multithreaded setting where the system should survive the failure of a subsystem.*

*We propose a static verification approach for multithreaded programs with unchecked exceptions. Our approach is an extension of the Spec# verification methodology for object-oriented programs. It verifies that objects encapsulating shared resources are always ready to be disposed of, by allowing ownership transfers to other threads only through well-nested parallel execution operations. Also, the approach prevents developers from relying on invariants that may have been broken by a failure. We believe the programming style enforced by our approach leads to better programs, even in the absence of formal verification. The proposed approach enables developers using mainstream languages to gain some of the benefits of approaches based on isolated sub-processes. We believe this is the first verification approach that soundly verifies common exception handling and locking patterns in the presence of unchecked exceptions.*

## 1 Introduction

In most software development projects, it is not feasible for developers to handle explicitly all possible unusual events which may occur during program execution. To deal with such events, modern programming languages provide the mechanism of *exceptions*. This mechanism allows pro-

grammers to write code for the normal case and to centralize the handling of many unusual events in one place, namely the exception handler. Specifically, most programming languages provide *try-catch statements* or equivalent. If execution of the try block terminates due to an exception, the catch block is executed and the overall try-catch statement terminates normally. Some languages, including Java and C#, also provide *try-finally statements*. The finally block is executed after the execution of the try block, no matter whether the try block terminates normally or exceptionally. This facilitates for instance the disposal of resources. The overall try-finally statement terminates normally if the try block and the finally block terminate normally.

Some programming languages, particularly Java and Spec# [18], distinguish between *checked* and *unchecked* exceptions. The compiler checks that for each checked exception that may occur in a given method, the method either catches the exception or declares the exception in its header (using a throws clause). Methods need not catch or declare unchecked exceptions. Checked exceptions are typically used for expected, but rare errors, for instance, failure to open a file. Unchecked exceptions are typically used in the following situations:

1. in abnormal situations caused by bugs in the program (for instance, null-pointer dereferencing or assertion violation);
2. in unexpected situations that may occur at a large number of program points and that usually cannot be handled locally by a method (for instance, out of memory errors and asynchronous thread cancellation);

Making these exceptions checked would clutter up the code with throws clauses since virtually any method potentially contains a bug, runs out of memory, etc. In some programming languages, in particular C#, all exceptions are effectively unchecked.

Checked exceptions do not cause major problems for program verification. Since they can be thrown only at certain known program points, verifiers can take the excep-

tional control flow into account [14]. In particular, exceptional postconditions [15] permit callers of a method to reason about the exceptional termination of a call, and methods can be required to preserve object invariants even in case a checked exception occurs.

Even though program verifiers typically show the absence of unchecked exceptions of kind 1, verified programs may still throw unchecked exceptions of kind 2. Proving the absence of the latter kind is either too cumbersome (for instance, proving the absence of out-of-memory errors entails reasoning about resource properties and the garbage collector) or not possible during modular verification (for instance, asynchronous thread cancellation is triggered by code that is outside the current module). Therefore, our goal is to verify that a program does not throw unchecked exceptions of kind 1, and that programmer-specified safety properties, such as method contracts and in-line assertions, hold. This verification should be sound, even if the program throws unchecked exceptions of kind 2. It is crucial for the verification to be *modular*, that is, to allow reasoning about a class independent of its clients and subclasses. Modularity is important to verify class libraries and for scalability.

We illustrate this challenge using the example in Fig. 1. Objects of class *Processor* perform computations. Whenever a processor allocates a resource, it is stored in a *ResourceTable* together with the processor and a time stamp. After the processing has terminated normally or exceptionally, the finally block in method *Process* calls method *DisposeAll* of class *ResourceTable* to dispose all resources allocated by the terminating processor.

The third invariant of class *ResourceTable* expresses that an element of array *r* is non-null if the corresponding element of *p* is non-null. This invariant is used by method *DisposeAll* to guarantee that the call *r[i].Dispose()* does not throw a null-pointer exception.

Our example should not verify because it potentially throws a null-pointer exception. Consider an execution of method *Process* that calls *Add*. In method *Add*, the call of the static property *DateTime.Now* may cause a *StackOverflowException*. In this case, method *Add* terminates exceptionally in a state where the resource table does not satisfy its third invariant: *p[i]* is already non-null, but *r[i]* may still contain null. Consequently, when the finally block of *Process* calls *rt.DisposeAll*, this method may throw a null-pointer exception when it tries to dispose *r[i]*.

Separately dealing with each control flow path caused by an unchecked exception is typically infeasible, since an unchecked exception may occur at almost every program point. For the same reason, it is in general not possible to establish exceptional postconditions or to re-establish object invariants when an unchecked exception is thrown. Consequently, our example verifies in existing program verifiers,

```

class Resource {
    public void Dispose() { /* ... */ }
}

class ResourceTable {
    Processor[]! p := new Processor[16];
    DateTime[]! t := new DateTime[16];
    Resource[]! r := new Resource[16];

    invariant p.Length = t.Length;
    invariant p.Length = r.Length;
    invariant  $\forall i : 0 \leq i < p.Length \wedge p[i] \neq \text{null} \Rightarrow r[i] \neq \text{null}$ ;

    public void Add(Processor! pp, Resource! rp) {
        for(int i := 0; i < p.Length; i := i + 1)
            if(p[i] == null) { // found empty slot
                p[i] := pp;
                t[i] := DateTime.Now;
                r[i] := rp;
                return;
            }
        /* resize arrays and add data */
    }

    public void DisposeAll(Processor! pp) {
        for(int i := 0; i < p.Length; i := i + 1)
            if(p[i] == pp) {
                p[i] := null;
                r[i].Dispose();
            }
    }
}

class Processor {
    ResourceTable! rt := new ResourceTable();

    public void Process() {
        try {
            // do some processing
            Resource! r := new Resource();
            rt.Add(this, r);
            // do some processing
        } finally {
            rt.DisposeAll(this);
        }
    }
}

```

**Figure 1. Implementation of a resource table and its client in a notation similar to Spec#. Exclamation points mark non-null types. For instance, *Processor!* is the type of non-null references to *Processor* objects. A reference of type *Processor[]!* points to a non-null array, but the array elements may be null. We omitted Spec#'s unpack and pack statements, but assume that objects are unpacked before their fields are updated. All methods require implicitly that their receiver is consistent, that is, satisfies its invariant.**

which is unsound. For instance, ESC/Java [14] only considers unchecked exceptions that are propagated by a method, and the Spec# verifier Boogie [1] ignores exceptional control flow altogether.

A naive solution to this soundness problem is not to assume anything about the program state upon entry of a catch or finally block. With this solution, a program verifier would not assume the invariant of  $rt$  to hold upon entry of the finally block in method *Process*. Consequently, the call to *DisposeAll* would not verify because the method (implicitly) requires the invariant of its receiver to hold.

Whereas this naive solution is sound, it does not permit verification of common programming patterns, in particular, two idioms that the C# designers found important enough to provide special syntax for: the using block and the lock block. The statement

```
using (o) { S }
```

is equivalent to

```
try { S } finally { o.Dispose(); }
```

In general, *o.Dispose()* relies on *o* being in a consistent state. However, this may not be assumed with the naive solution. The statement

```
lock (o) { S }
```

is equivalent to

```
Monitor.Enter(o);  
try { S } finally { Monitor.Exit(o); }
```

(assuming *o* is side-effect-free). Our verification methodology for programs that use locks [11] requires that when a thread releases the lock of an object *o*, *o* is accessible by the thread and *o* is consistent. Again, both requirements cannot be shown with the naive solution.

To be able to support common programming idioms, a verification methodology must be able to prove that certain objects that are manipulated in the try block are accessible and consistent upon entry to a catch or finally block. In this paper, we propose a static verification methodology for multithreaded programs that is sound in the presence of unchecked exceptions and handles the above idioms.

**Outline.** This paper is structured as follows. Sec. 2 provides the background on the Spec# verification methodology needed in the rest of the paper. Sec. 3 introduces block invariants, which enable the verification of simply try statements, but cannot guarantee that objects are consistent and accessible to the current thread. We address consistency in Sec. 4 and accessibility in Secs. 5 and 6. We discuss strengths and weaknesses of our methodology in Sec. 7 and review related work in Sec. 8.

This paper provides an informal overview of the approach. A formalization and soundness proof is provided in an accompanying technical report [13].

## 2 Verifying multithreaded programs

Our approach is based on an existing verification approach for multithreaded object-oriented programs [10, 11, 12]. To focus on the most important aspects of our methodology, we ignore inheritance in the following. However, an extension is straightforward.

The Spec# methodology verifies the absence of data races, deadlocks, and operation precondition violations such as null dereferences, as well as compliance of the program with programmer-specified correctness criteria such as method specifications and object invariants.

The absence of data races is proved using additional state variables, in particular a per-thread access set, and a global shared set. Each of these state variables holds a set of object references. An access of a field *o.f* by a thread *t* is illegal if the target object *o* is not in *t*'s access set at the time of the operation. The methodology ensures that two threads' access sets are always disjoint; it follows that verified programs do not contain data races.

The state variables evolve as follows. Initially, each thread's access set is empty. When a thread creates a new object, the object is added to the thread's access set. Also, when a thread acquires an object *o*'s lock, by entering a **synchronized** (*o*) statement (in Java) or a **lock** (*o*) statement (in C#), the object is added to the thread's access set. When the thread releases *o*'s lock, *o* is removed from the thread's access set.

To avoid a race between the thread that creates an object and a thread that acquires the object's lock, the programming model imposes the rule that a thread may attempt to acquire an object's lock only if the object is in the shared set. The shared set is initially empty. A thread may transfer an object from its access set to the shared set using the special command **share** *o*, which is important for our verification methodology, but does not influence the observable behavior of the program. After a thread shares an object, it no longer has access to it, unless and until it acquires the object's lock, at which point the programming language's locking semantics ensures mutual exclusion with other threads that acquire the lock.

The programming model deals with thread creation in Java as follows. When a thread *t* starts a new thread *t'* using a call *t'.start()*, the *Thread* object *t'* is transferred from the access set of *t* to the access set of *t'*. In C#, when a thread *t* starts a new thread *t'* using a call

```
new Thread(o.M).Start();
```

object  $o$  is transferred from the access set of  $t$  to the access set of  $t'$ .

The data race prevention approach is integrated with the Spec# (a.k.a. Boogie) object invariant approach [2, 16], as follows. Each object is implicitly extended with a boolean ghost field called  $inv$ . Each class  $C$  may declare an object invariant  $Inv_C$ . The approach ensures that in each program state, for each object  $o$  of class  $C$ , we have

$$o.inv \Rightarrow Inv_C[o/\mathbf{this}]$$

Object invariants are verified as follows. Each object's  $inv$  bit is initially *false*. The programmer may set  $o.inv$  to *true* using the special command `pack o`, and to *false* using the special command `unpack o`. `pack o` asserts  $o$ 's invariant; an assignment to a field of  $o$  requires  $o.inv$  to be *false*.

Object invariants interact with locking as follows. The approach enforces the property that in each program state, if a shared object  $o$  is not locked by any thread, then its  $inv$  bit is *true*. It does so by imposing the proof obligation  $o.inv = true$  at each `share o` operation and at each exit from a `synchronized (o)`, resp. `lock (o)`, block.

The approach prevents both lock re-entry and deadlocks (defined as a group of threads where each thread is waiting for another thread in the group to release a lock). It does so by allowing the programmer to define a partial order among shared objects, and considering a program legal only if it acquires locks according to this partial order.

### 3 Block invariants

The basic observation underlying our methodology is that upon entry to a finally block, we can assume those properties that are true at every point during the try block. Consider the two code snippets in Fig. 2. The code on the left should verify because  $t$  is non-null throughout the try block. Therefore, it can safely be assumed to be non-null upon entry to the finally block. However, the code on the right might throw a null-pointer exception in case method `Foo` terminates exceptionally.

To support this kind of reasoning, we equip each try block with a block invariant. This invariant must hold throughout the try block. Consequently, it may be assumed upon entry to corresponding catch and finally blocks. Adding the block invariant  $t \neq \mathbf{null}$  to the left try block in Fig. 2 allows us to verify the finally block.

The block invariant of a try block must be preserved by each operation performed during the execution of the block, including operations performed by methods called from the try block. For instance in our example, the block invariant must also hold throughout the execution of `Foo`, which is trivially the case because `Foo` cannot change the local variable  $t$ . In general, block invariants that depend on heap lo-

<pre> T t := new T(); try {   Foo();   t := new T();   /* more code not      changing t */ } finally {   t.Dispose(); } </pre>	<pre> T t := null; try {   Foo();   t := new T();   /* more code not      changing t */ } finally {   t.Dispose(); } </pre>
--	---

**Figure 2. Two examples illustrating block invariants. The example on the left verifies with the block invariant  $t \neq \mathbf{null}$ . The example on the right may throw a null-pointer exception.**

cations lead to proof obligations for all method implementations that are potentially called from a try block. In the presence of dynamic method binding, such proof obligations are difficult to show modularly. However, modular solutions are possible for certain block invariants:

First, block invariants that do not depend on heap locations cannot be violated by method calls. Therefore, they can be verified modularly. We verify such a block invariant  $I$  by asserting  $I$  immediately before the try block and after each statement of the try block. At the beginning of the catch or finally block, we assume  $I$ . This solution allows us to verify the left example in Fig. 2.

Second, if the frame properties of a method were required to specify even temporary modifications of fields, one could use this information to show that block invariants are preserved. Such frame specifications are supported by JML's assignable clauses [15]. For brevity, we do not present the details of this approach here.

While block invariants allow one to verify many simple try statements, they are not powerful enough for the resource table example in Fig. 1. The finally block of method `Process` relies on  $rt$ 's invariant. However,  $rt.inv$  is not a block invariant, since  $rt.inv$  is temporarily modified during the execution of  $rt.Add$  since `Add` unpacks its receiver in order to modify its state (not shown in Fig. 1). Furthermore, using a block invariant to verify accessibility of an object does not support temporary transfer of an object's accessibility to another thread. In the following sections, we present solutions specific to object consistency and accessibility.

### 4 Dynamic checks for object consistency

The resource table example shows that object consistency in general cannot be expressed as a block invariant because methods called from try blocks unpack their receiver, thereby modifying its  $inv$  field. Consequently, when

such a method throws an exception before the receiver is repacked, the object invariant is violated upon entry to the finally block. Consequently, verifying modularly that certain objects are always valid on entry to the finally block does not seem possible. Therefore, in our proposal, we make it possible to test consistency efficiently at runtime.

This is achieved by making the *inv* field explicit in each object’s state such that programs can read its value. Updates of the *inv* field are still permitted only through the **unpack** and **pack** statements. The explicit *inv* field allows catch and finally blocks to test whether they can rely on an object’s invariant. For instance, the call *rt.DisposeAll* in *Process*’s finally block (Fig. 1), can be guarded by an **if**(*rt.inv*) statement.

As described in Sec. 2, the Spec# methodology for multi-threaded programs enforces that if a shared object *o* is not locked by any thread, then its *inv* bit is *true*. However, this requirement cannot be maintained in the presence of unchecked exceptions. Consider the statement **synchronized** (*o*) { *S* } (in Java) or **lock** (*o*) { *S* } (in C#). Acknowledging that it is not feasible to guarantee the consistency of *o* on exit from *S*, we cannot guarantee that *o* is consistent after its lock has been released. Therefore, in our methodology, releasing an object’s lock no longer requires that the object is consistent, and acquiring an object’s lock no longer provides the guarantee that the object is consistent. Therefore, after acquiring an object’s lock, consistency of the object has to be checked at runtime before any methods are called on the object that require its consistency.

Checking the explicit *inv* is more efficient than checking the whole object invariant. The space overhead can be reduced by introducing explicit *inv* fields only for those objects that are used in catch and finally blocks, in particular the objects used as arguments in using and lock statements.

If the runtime test for object consistency succeeds, the object invariant may be safely assumed in subsequent code inside and outside the catch or finally block. If the test fails, the invariant may not be assumed. In many such situations, it is possible to program conservatively, that is, without relying on object invariants to hold. For instance, since *DisposeAll* is used in a finally block, one might consider rewriting the method such that it checks all conditions it relies on at runtime. In particular, if *r[i]* is found to be null, there is no resource to be disposed, and the method may simply skip this array element. In other cases, the only appropriate reaction is to throw another unchecked exception. Most likely, subsequent code will not be able to use the inconsistent object. Nevertheless, we believe it is important that our methodology makes this potential problem of inconsistent objects explicit and forces programmers to think about appropriate reactions to failed checks. Programmers not using Spec# can benefit from our methodology by using it as a style guide in their programming language.

```
class Client {
    public void main() {
        Socket s := ...;
        InputStream i := s.getInputStream();
        try {
            foo(i);
        } finally {
            s.close(); // Requires access to i
        }
    }
}
```

**Figure 3. A Java example. Verification of this example fails unless a `make.threadlocal` clause is added.**

## 5 Thread-local objects

In order to prevent data races and object invariant violations, the verification methodology described in Section 2 considers an access of a field *o.f* by a thread *t* to be legal only if *o* is in *t*’s access set.

If a finally block accesses an object *o*’s fields, verifying the block requires proof that *o* is in the current thread’s access set at the time of the access. However, the corresponding block invariant is not supported by the two solutions described in Sec. 3: The first solution does not apply because methods called in the try block can change the accessibility of *o* by transferring *o* to another thread; the second solution does not apply because accessibility, like the *inv* field, is often modified temporarily. In this section, we extend our methodology to support block invariants about accessibility. In the next section, we further extend the methodology to support temporarily transferring accessibility of an object to another thread.

The problem with accessibility is illustrated by the example (in Java) in Fig. 3. Method *main* obtains a *Socket* object. At this point, the *Socket* object’s input stream object is accessible. However, method *foo* might change the accessibility of *i*, which is, thus, not a block invariant. Since the call to *s.close* requires *s*’s input stream *i* to be accessible, verification of the finally block fails.

To support block invariants about accessibility, we extend our methodology to prevent certain objects from becoming inaccessible. We add a boolean ghost field *threadlocal* to each object. We use this field to prevent thread-local objects from becoming transferred to another thread. This is achieved by imposing proof obligations on all operations that transfer object ownership.

To manipulate the *threadlocal* field, we add a **make.threadlocal** clause to try blocks, specifying the objects that should be thread-local for the duration of the

try block:

```
try make_threadlocal o; { S1 } finally { S2 }
```

For such a try statement, we first assert that  $o$  is actually accessible. Before executing the try block, we set  $o.threadlocal$  to true in order to prevent  $o$  from becoming inaccessible. When the try block terminates (normally or exceptionally),  $o.threadlocal$  is set back to its original value. Since  $o$  cannot become inaccessible during the execution of the try block, the accessibility of  $o$  is now a block invariant, which may be assumed upon entry to the finally block.

In our example, the clause `make_threadlocal s, i` allows us to verify the finally block. Note that if  $Foo$  attempted to transfer accessibility of  $i$  to another thread, it would require  $\neg i.threadlocal$  in its precondition. Therefore, this attempt would become apparent as precondition violation.

## 6 Ownership transfer between threads: Structured concurrent programming

The extension of the previous section permits block invariants that enforce an object to be accessible throughout a try block. However, it is often useful to temporarily pass accessibility of an object to another thread. Such transfers happen when threads are started or joined. In this section, we further extend our methodology to enable this.

The example in Fig. 4 is an extension of Fig. 3. In this extension, method `main` starts a new thread  $p$  inside the try block. When  $p$  is started, accessibility of  $i$  is implicitly transferred to the new thread, and later transferred back when  $p$  and the current thread join. Because of this ownership transfer, accessibility of  $i$  is not a block invariant. Adding the clause `make_threadlocal s, i` would fail because the `Start` method requires  $p.i$  to be non-threadlocal.

Our methodology enables temporary ownership transfer to other threads by introducing support for structured concurrent programming. We could introduce a classical parallel composition statement:

```
par { SA } { SB }
```

However, it is easier to introduce this functionality in the form of a library method. For Java:

```
public class ParallelExecutionException {
    public Throwable getBranch1Exception();
    public Throwable getBranch2Exception();
    ...
}

public static void
    executeInParallel(Runnable b1, Runnable b2)
{ ... }
```

```
class IncomingPump extends Thread {
    InputStream i;
    public void run() {
        pumpIncoming(i);
    }
    ...
}

class Client {
    public void main() {
        Socket s := ...;
        InputStream i := s.getInputStream();
        try {
            IncomingPump p := new IncomingPump();
            p.i := i;
            p.start(); // Transfers i to new thread
            /* ... */
            p.join(); // Transfers i back
        } finally {
            s.close(); // Requires access to i
        }
    }
}
```

**Figure 4. A Java example. Because of the ownership transfer, accessibility of  $i$  is not a block invariant. Consequently, verification of the finally block fails.**

For C#:

```
public delegate void Branch();

public class ParallelExecutionException {
    public Exception Branch1Exception { get { ... } }
    public Exception Branch2Exception { get { ... } }
    ...
}

public static void
    ExecuteInParallel(Branch b1, Branch b2)
{ ... }
```

The semantics of this method is that it executes *b1* and *b2* in different threads (one of which is likely to be the current thread, but this is unimportant). Control exits the method only after control has exited both branches. Control exits normally only if control has exited both branches normally; if either branch exits with an exception, the *ExecuteInParallel* or *executeInParallel* call exits with a *ParallelExecutionException* (but it first waits for the other branch to exit).

Note that this method can be implemented easily using the existing threading support, such as methods *Thread.start* and *Thread.join* in Java. However, some care must be taken to ensure that exceptions and thread interruptions are handled correctly. Note also that the implementation of this method cannot itself be verified under this approach; it is part of the “trusted base”.

Note also that the semantics of this method differs from the Java Concurrency API’s

```
java.util.concurrent.ExecutorService.invokeAll
```

method in that the latter returns normally after all tasks have completed, even if some of the tasks completed because of an exception.

Verification of a call

```
ExecuteInParallel( $o_1.M_1, o_2.M_2$ );
```

in C#, where the relevant method contracts are

```
void  $M_1()$  requires  $P_1$ ; ensures  $Q_1$ ;
```

and

```
void  $M_2()$  requires  $P_2$ ; ensures  $Q_2$ ;
```

proceeds by translating it to:

```
assert  $P_1[o_1/\mathbf{this}]$ ;
assert  $P_2[o_2/\mathbf{this}]$ ;
assert  $\mathcal{A}(P_1[o_1/\mathbf{this}]) \cap \mathcal{A}(P_2[o_2/\mathbf{this}]) = \emptyset$ ;
havoc  $\bar{A} \cup \mathcal{A}(P_1[o_1/\mathbf{this}]) \cup \mathcal{A}(P_2[o_2/\mathbf{this}])$ ;
assume  $Q_1[o_1/\mathbf{this}]$ ;
assume  $Q_2[o_2/\mathbf{this}]$ ;
```

where  $\mathcal{A}(P)$  denotes the *required access set* of  $P$ , defined as

$$\mathcal{A}(P) = \{o \mid (\forall A \bullet P(A) \Rightarrow o \in A)\}$$

that is, the set of objects that are in all access sets that satisfy predicate  $P$  under the current heap. In other words, the parallel execution of two calls is valid if both calls’ preconditions hold and their required access sets are disjoint. The effect of the parallel execution is encoded by havocing the (fields of) the objects that are not in the current access set or that are in either call’s required access set, and then assuming both calls’ postconditions. Furthermore, neither  $P_1$  nor  $P_2$  are allowed to mention the current thread identifier, since one or both of the calls may be executed in a different thread.

Verification of a call

```
executeInParallel( $r_1, r_2$ );
```

in Java is analogous, except that verification is based on the contracts of  $r_1$  and  $r_2$ ’s *run* methods, which must be identifiable statically.

Note that this verification approach is analogous to the proof rule for parallel composition in separation logic [22].

Figure 5 shows a Java code fragment that uses an *executeInParallel* call. It is part of a chat server. When a client connection is received, incoming messages are received in one thread and outgoing messages sent in another. Closing the socket requires both the input stream and the output stream to be accessible. This is proven thanks to the **make\_threadlocal** clause. Note that contrary to the *Thread.start* method, *executeInParallel* allows the objects whose accessibility is temporarily transferred to other threads to be thread-local.

## 7 Discussion

Our methodology enables sound reasoning in the presence of unchecked exceptions using the following ingredients:

1. Simple block invariants allow us to verify many try statements, where the block invariant does not depend on heap locations. Such try statements are common and can be verified with little overhead.
2. Since object consistency in general cannot be enforced using block invariants, we provide a way to check consistency efficiently at runtime. Although this is not entirely satisfactory, it suffices to ensure soundness. It shows that programmers need to insert code to track object consistency manually.
3. Thread-local objects and structured concurrent programming allow us to prove accessibility of objects in catch and finally blocks.

```

class IncomingPump implements Runnable {
  InputStream i;
  public void run()
    requires i is accessible and thread-local;
    ensures i is accessible and thread-local;
  {
    ChatServer.pumpIncoming(i);
  }
}

class OutgoingPump implements Runnable {
  OutputStream o;
  public void run()
    requires o is accessible and thread-local;
    ensures o is accessible and thread-local;
  {
    ChatServer.pumpOutgoing(o);
  }
}

class Client {
  public void main() {
    Socket s := ...;
    InputStream i := s.getInputStream();
    OutputStream o := s.getOutputStream();
    try make_threadlocal i, o; {
      IncomingPump ip := new IncomingPump();
      ip.i := i;
      OutgoingPump op := new OutgoingPump();
      op.o := o;
      ThreadingUtils.executeInParallel(ip, op);
    } finally {
      s.close(); // Requires access to i and o
    }
  }
}

```

**Figure 5. Example of parallel execution, in Java.**

In this paper, we focus on making program verification sound, but do not address other important issues such as resource management and fault tolerance. In fact, our verification methodology does not guarantee that all allocated resources are freed. In particular, when the explicit consistency check in a finally block fails, typically an unchecked exception is thrown without freeing resources first. To achieve fault tolerance, it would be helpful to support selective cancellation of high-level computations while protecting lower layers in a complex program.

A tentative suggestion for dealing with these issues is to extend the programming language with the notion of *subsystems* that fail independently of each other. When a failure occurs (that is, an unchecked exception is thrown) in a

certain thread, that thread’s subsystem fails. At that point, all threads currently running in that subsystem are aborted up to the statement where the thread originally entered the subsystem.

The resource leakage problem is dealt with as follows. When a provider subsystem hands a handle to a resource allocated in the provider subsystem to a client subsystem, the handle is registered with the client subsystem. The provider may associate a cleanup routine with a handle. When a subsystem fails or otherwise finishes, all handles associated with it are executed in the corresponding provider subsystems. If the cleanup action fails, the provider subsystem fails. This prevents resource leaks where the client subsystem finishes but the resource remains allocated in the provider subsystem and the provider subsystem outlives the client subsystem for an extended period of time. We plan to investigate this idea as future work.

## 8 Related work

Our methodology builds on the verification methodology of Spec# [3, 17]. Although the Spec# language supports checked and unchecked exceptions, the program verifier Boogie ignores them. Therefore, verification is not sound for executions where exceptions occur.

JML [15] provides elaborate support for specifying exceptional behavior of methods. JML requires a method to preserve object invariants even if the method terminates with an unchecked exception. Since unchecked exceptions can occur at almost every program point, this requirement essentially prevents temporary violations of object invariants, which is overly restrictive. Our methodology imposes weaker requirements at the price of runtime checks of object invariants.

To our knowledge, the existing verifiers for Java and JML do not fully support unchecked exceptions. ESC/Java [8] only considers unchecked exceptions that are propagated by a method, but ignores other sources of unchecked exceptions such as object creation, which may cause out-of-memory errors. This can lead to unsoundness. Jive’s [20] handling of unchecked exceptions is not faithful to the JML semantics. Whereas Jive proves the absence of exceptions of kind 1 (bugs, see Sec. 1), exceptions of kind 2 are ignored. To our knowledge, the same is true for KeY [4], Jack [7], LOOP [5], and Krakatoa [19]. Although our methodology is based on Spec#, the main ideas could be adopted by Java/JML verifiers, especially, block invariants.

Recent work applies separation logic [25] to verification of object-oriented programs [23, 24] and concurrent programs [22, 6]. However, we are not aware of any work that applies separation logic to a language that includes exceptions.

We propose structured concurrent programming as a part

of our solution. While other approaches such as CSP [9] and SCOOP [21] abandon threads altogether, we want to support Java-style programs with threads and explicit locking. Therefore, we resort to structured concurrent programming only in cases where try blocks need to transfer object ownership between threads.

## 9 Conclusion

We propose a methodology for modular static verification of programmer-specified safety properties in multithreaded object-oriented programs. Our methodology is sound in the presence of unchecked exceptions and soundly verifies common exception handling patterns. The approach enforces that programmers use structured concurrent programming if they pass objects that are accessed in catch blocks or finally blocks to other threads. Furthermore, it enforces that programmers track the consistency of an object at runtime, using an explicit *inv* bit, if they need to rely on the object's consistency after a failure or after acquiring its lock.

As future work, we plan to investigate the subsystem idea sketched in Sec. 7 and to implement our methodology in the program verifier SpecLeuven [10].

**Acknowledgments.** The authors would like to thank the members of the Formal Methods club at ETH Zurich for a very entertaining and helpful discussion on this topic. Bart Jacobs is a Research Assistant of the Fund for Scientific Research - Flanders (F.W.O.-Vlaanderen) (Belgium). Peter Müller's work was carried out at ETH Zurich. It was funded in part by the Information Society Technologies program of the European Commission, Future and Emerging Technologies under the IST-2005-015905 MOBIUS project.

## References

- [1] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. 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.
- [2] M. Barnett, R. DeLine, M. Fähndrich, K. R. M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004.
- [3] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *CASSIS 2004*, volume 3362 of *LNCS*. Springer, 2004.
- [4] B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*. LNCS 4334. Springer-Verlag, 2007.
- [5] J. v. d. Berg and B. Jacobs. The LOOP compiler for Java and JML. In T. Margaria and W. Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2031 of *Lecture Notes in Computer Science*, pages 299–312. Springer, 2001.
- [6] S. Brookes. A semantics for concurrent separation logic, 2004. Invited paper, in *Proceedings of CONCUR*.
- [7] L. Burdy, A. Requet, and J.-L. Lanet. Java applet correctness: A developer-oriented approach. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *FME 2003: Formal Methods: International Symposium of Formal Methods Europe*, volume 2805 of *Lecture Notes in Computer Science*, pages 422–439. Springer-Verlag, 2003.
- [8] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *PLDI 2002*, volume 37 of *SIGPLAN Notices*, pages 234–245. ACM, May 2002.
- [9] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.
- [10] B. Jacobs. *A Statically Verifiable Programming Model for Concurrent Object-Oriented Programs*. PhD thesis, Katholieke Universiteit Leuven, Department of Computer Science, 2007.
- [11] B. Jacobs, K. R. M. Leino, F. Piessens, and W. Schulte. Safe concurrency for aggregate objects with invariants. In *Proc. Int. Conf. Software Engineering and Formal Methods (SEFM 2005)*, pages 137–146. IEEE Computer Society, sep 2005.
- [12] B. Jacobs, K. R. M. Leino, and W. Schulte. Verification of multithreaded object-oriented programs with invariants. In M. Barnett, S. H. Edwards, D. Giannakopoulou, G. T. Leavens, and N. Sharygina, editors, *SAVCBS 2004 Workshop Proceedings*, 2004. Technical Report 04-09, Computer Science, Iowa State University.
- [13] B. Jacobs, P. Müller, and F. Piessens. Sound reasoning about unchecked exceptions: Soundness proof. Technical Report CW494, Katholieke Universiteit Leuven, 2007.
- [14] J. R. Kiniry and D. R. Cok. ESC/Java2: Uniting ESC/Java and JML: Progress and issues in building and using ESC/Java2, including a case study involving the use of the tool to verify portions of an Internet voting tally system. In *Construction and Analysis of Safe, Secure, and Interoperable Smart devices (CASSIS)*, volume 3362 of *Lecture Notes in Computer Science*, pages 108–128. Springer, 2004.
- [15] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Müller, and J. Kiniry. JML reference manual. Department of Computer Science, Iowa State University. Available from [www.jmlspecs.org](http://www.jmlspecs.org), 2006.
- [16] K. R. M. Leino and P. Müller. Object invariants in dynamic contexts. In M. Odersky, editor, *ECOOP 2004*, volume 3086 of *LNCS*, pages 491–516. Springer-Verlag, 2004.
- [17] K. R. M. Leino and P. Müller. Object invariants in dynamic contexts. In M. Odersky, editor, *European Conference on Object-Oriented Programming (ECOOP)*, volume 3086 of *Lecture Notes in Computer Science*, pages 491–516. Springer-Verlag, 2004.
- [18] K. R. M. Leino and W. Schulte. Exception safety for c#. In *Software Engineering and Formal Methods (SEFM)*, pages 218–227. IEEE Computer Society, 2004.

- [19] C. Marché, C. Paulin-Mohring, and X. Urbain. The Krakatoa tool for certification of Java/JavaCard programs annotated in JML. *Journal of Logic and Algebraic Programming*, 58(1–2):89–106, 2004.
- [20] P. Müller, J. Meyer, and A. Poetzsch-Heffter. Programming and interface specification language of JIVE—specification and design rationale. Technical Report 223, Fernuniversität Hagen, 1997.
- [21] P. Nienaltowski. *Practical framework for contract-based concurrent object-oriented programming*. PhD thesis, Department of Computer Science, ETH Zurich, 2007.
- [22] P. W. O’Hearn. Resources, concurrency, and local reasoning. To appear in *Theoretical Computer Science*; preliminary version in CONCUR’04.
- [23] M. J. Parkinson. *Local reasoning for Java*. PhD thesis, Computer Laboratory, Cambridge University, 2005.
- [24] M. J. Parkinson and G. M. Bierman. Separation logic and abstraction. In *Proc. POPL*, 2005.
- [25] J. C. Reynolds. Separation logic: a logic for shared mutable data structures, 2002. Invited Paper, *Proceedings of the 17th IEEE Symposium on Logic in Computer Science*, pages 55–74.