On the Interplay between the Semantics of Java's Finally Clauses and the JML Run-Time Checker

M. Huisman*
University of Twente Netherlands

ABSTRACT

This paper discusses how a subtle interaction between the semantics of Java and the implementation of the JML run-time checker can cause the latter to fail to report errors. This problem is due to the well-known capability of finally clauses to implicitly override exceptions. We give some simple examples of annotation violations that are not reported by the run-time checker because the errors are caught within the program text; even without any explicit reference to them. We explain this behaviour, based on the official Java Language Specification. We also discuss what are the consequences of this problem, and we sketch different solutions to the problem (by adapting the implementation of the JML run-time checker, or by adopting a slightly different semantics for Java).

1. INTRODUCTION

It is common folklore that if one could analyse the whole state space of a program using a run-time checker, then any violation of the specification would be found (assuming that the specification is executable). Thus, provided that one has a correct specification of the intended program behaviour, if run-time checking does not signal errors for any program execution, this gives a 100 % correctness guarantee for the program. And only because in general it is impossible to consider all possible execution paths of a program, it makes sense to apply static verification and other static analysis methods.

However, this point of view is actually too optimistic: we show how the Java semantics (*cf.* [7]) can cause the JML run-time checker to fail to report some annotation violations to the user – even though they can be detected using static verification. The source of the problem, the semantics of `finally` in Java, is well-known to be problematic [6, 10]; programmers are usually advised to adhere to particular programming patterns when using `finally`, to avoid unexpected program behaviour. This paper discusses how the semantics of `finally` also can have consequences for the use of a run-time checker. This paper does not intend to say that run-time checking does not work; it merely points out a peculiarity that is caused by the interaction between the semantics of finally clauses, and the implementation of the JML run-time checker. Standard solutions cannot avoid the problem: the programming patterns that are proposed for finally clauses are useless here, because one cannot expect the programmer to add code in the implementation of a `finally` block that checks whether run-time checking might have produced an error. Instead run-time checking tool builders should be aware of the problem (since it seems unlikely to expect a change to the Java semantics), and adapt their tools appropriately.

JML is an expressive annotation language for Java, that allows one to encode many useful safety and security properties. It is supported by a wide variety of tools, for run-time and static verification, annotation generation *etc.*, see [3]. The standard JML run-time checker tool set compiles the specification into run-time checks (using `jmlc`), by adding appropriate instructions to the program. When the actual run-time checker `jmlrac` is executed, appropriate errors are thrown if the annotations are violated [4]. However, since the way to signal specification violations is encoded in the programming language itself, the JML errors are treated as any other kind of exception. And thus, in particular, it is possible to catch them, and return to a "normal" program state again. Moreover, because of the subtle semantics of Java for `try-catch-finally` statements, it is even possible to overwrite a JML error with another exception, and thus to catch it without explicitly mentioning it in the program text.

This paper emerged as an unexpected difficulty in a larger project on inlining security monitors [8] (in collaboration with A. Tamalet, University of Nijmegen, Netherlands). Ultimate goal of the project is to develop a static verification method that makes monitoring of an application unnecessary. The inlining is defined in two steps: first the monitor is translated into appropriate annotations of the methods directly involved in the security property that the monitor encodes, next the annotations are propagated, so that a verification condition generator can generate provable proof obligations. Correctness of the procedure is also proven in two steps: (1) a monitored program only gets stuck if run-time checking of the basic annotations would raise an excep-

*This work is partially funded by the IST FET programme of the European Commission, under the IST-2005-015905 Mobius project. Part of the research done while the author was at INRIA Sophia Antipolis.

FTfJP '09, July 6 2009, Genova, Italy

tion; (2) propagation of annotations would never result in an incorrect program being accepted. One of the initial assumptions of the project was that any incorrect application would always be rejected by run-time annotation checking. However, when formalising the correctness proof, we realised that this assumption was actually incorrect.

Section 2 presents some examples where run-time annotation violations are not reported to the user. Section 3 explains this behaviour, based on the Java Language Semantics. Section 4 discusses consequences for run-time checkers, while Section 5 sketches possible solutions, and Section 6 concludes.

2. EXAMPLES

This section illustrates how a program can be manipulated, so that it does not signal specification violations. Suppose that we have a method `decrypt`, whose first argument is a `key`, and its second a special `access_code` that determines whether one is allowed to use decryption. Thus, the access code should be in a particular range; for simplicity let us assume that it should be below 10. In JML, this method could be specified as follows.

```
//@ requires access_code < 10;
public void decrypt(int [] key, int access_code){
  ...
}
```

Now suppose that somebody who does not have the access code nevertheless attempts to decrypt a message. A naive approach would be to just guess an access code (say 36), and to call the decrypt message anyway:

```
// Example 1
public void sneakyMethod(){
  int r = 36;
  decrypt(key, r);
}
```

However, any attempt with a guess that is out-of-range would be signalled by the JML run-time checker `jmlrac`:

```
> jmlrac TryCatchExample
Exception in thread "main"
org.jmlspecs.jmlrac.run-time.JMLInternalPreconditionError:
by method TryCatch.decrypt
at TryCatch.sneakyMethod(TryCatchExample.java:432)
at TryCatch.internal$cheat(TryCatchExample.java:11)
at TryCatch.cheat(TryCatchExample.java:292)
at TryCatchExample.internal$main(TryCatchExample.java:1101)
at TryCatchExample.main(TryCatchExample.java:1373)
```

A possible way to hide this would be to put the code inside a try-block, and to catch the JML error.

```
// Example 2
public void sneakyMethod(){
  try{
    int r = 36;
    decrypt(key, r);
  }
  catch (Error e){
  }
}
```

Run-time checking with `jmlrac` would not signal an error anymore, even though one attempts to call `decrypt` with illegal arguments. However, an attentive code inspector might get suspicious by the attempt to catch an `Error`, since "the class `Error` and its subclasses are exceptions from which ordinary programs are not ordinarily expected to recover." [7, §11.5] (JML errors are a subclass of the class `Error`, which inherits directly from `Throwable`, and is thus incompatible with the class `Exception`). Therefore, errors can be distinguished from exceptions, from which recovery might be possible.

A smarter way to hide this attempt is by using a `finally` block to override the possible JML error.

```
// Example 3
public void sneakyMethod() throws ArbitraryException{
  try{
    int r = 36;
    decrypt(key, r);
  }
  finally{
    throw new ArbitraryException();
  }
}
```

In this case, the program will throw an `ArbitraryException` (or terminate normally if this `ArbitraryException` is caught by the method that triggered `sneakyMethod`). For this program, manual code inspection to reveal whether a JML error was hidden might not be straightforward (remember that the code for the `decrypt` method and the invocation of `sneakyMethod` might be stored in different classes). Naturally, tools for static validation of JML annotations (such as ESC/Java [5] or Jack [2]) would detect this attempt to call `decrypt` with illegal arguments.

3. A SEMANTICS-BASED EXPLANATION

This failure of run-time checking with `jmlrac` can be explained directly on the basis of the Java semantics. The Java Language Specification gives a detailed description of the behaviour of the `try-catch` and the `try-catch-finally` statements [7, §14.20]. Here, we repeat only those fragments of the specification that are relevant for this paper.

> A `try-catch-finally` statement with a `finally` block is executed by first executing the `try` block. [..] If execution of the `try` block completes abruptly because of a throw of a value V, then there is a choice:
>
> - If the run-time type of V is assignable to the parameter of any `catch` clause of the `try-catch-finally` statement, then the first (leftmost) such `catch` clause is selected. The value V is assigned to the parameter of the selected `catch` clause, and the block of that `catch` clause is executed. [..]
> - If the run-time type of V is not assignable to the parameter of any `catch` clause of the `try-catch-finally` statement, then the `finally` block is executed. Then there is a choice:
>   - If the `finally` block completes normally, then the `try` statement completes ab-

ruptly because of a throw of the value V.

– If the `finally` block completes abruptly for reason S, then the `try-catch-finally` statement completes abruptly for reason S (and the throw of value V is discarded and forgotten).

[...]

Notice that if execution of a `catch` block completes abruptly because of a throw of an exception, similar rules apply for the execution of the `finally` block. Also, if the `try` or `catch` block completes abruptly for any other reason R (*e.g.*, execution of a `return` statement), abrupt completion of the `finally` block always discards reason R.

It is not too complicated to give a formal specification (in a formal language) of the behaviour of the `try-catch-finally` statement, based on the description from the Java Language Specification, see for example the formalisation that was used within the LOOP project [9].

Thus, in Example 2 above, the JML error is caught explicitly by the catch clause, since its run-time type is assignable to the parameter `Error e`.

In Example 3, there is no matching `catch` clause (as there are none). Therefore, the `finally` block will be executed directly, and as this will throw an (arbitrary) exception, the JML error will be discarded. Thus, this behaviour of the `finally` block in fact allows implicit catching of all exceptions, including errors from which recovery should not be possible.

As mentioned above, this behaviour of finally clauses is well-known, and requires use of special programming patterns to avoid problems [6, 10]. Finally clauses are typically used to perform clean-up on objects that deal with an external resource, *e.g.*, to close a file. Correct use of finally clauses ensures that even though a program might terminate because of an exception, it will not corrupt external resources. Notice however that finally clauses are not essential; algorithms exist to eliminate their bytecode counterparts, *i.e.*, `jsr` instructions, see *e.g.,* [1].

Notice further that the semantics of the `try-catch-finally` statement prescribes that any abrupt completion of the `finally` block discards the throw of value V. Thus, any `return`, `break` or `continue` statement would have a similar effect. In particular, the following variant of `sneakyMethod`:

```
// Example 4
public void sneakyMethod(){
  try{
    int r = 36;
    decrypt(key, r);
  }
  finally{
    return;
  }
}
```

would have the same effect as Example 3 above, *i.e.*, it would discard the JML error, but instead of throwing a new exception, `sneakyMethod` would terminate normally.

Finally, we would like to remark that IDEs like Eclipse try to help the user to avoid writing a `finally` block that completes abruptly. From the Eclipse 3.0 release notes:

> The Java compiler can now find and flag `finally` blocks which cannot complete normally (as defined in the Java Language Specification). `Finally` blocks which cannot complete normally can be confusing and are considered bad practice.

However, this feature has limited value, as it does not give a warning when in a `finally` block a method is called that might throw an exception – even when this method has an explicit throws clause as part of its declaration.

4. CONSEQUENCES FOR RUN-TIME CHECKING

The main consequence of the behaviour described above is that it causes run-time checking with `jmlrac` not to be completely transparent, where *transparency* means that if no annotation violations are reported (to the user), running a program with run-time checking results in the same behaviour as running the program without run-time checking [3].

In particular, both the second and the third example program in Section 2 will behave differently when executed with run-time checking enabled or disabled even though no annotation violations are reported (to the user): run-time checking will ensure that the method `decrypt` is not executed, while it will be executed when run-time checking is disabled. In fact, the annotation violation is detected by the JML runtime checker, but the underlying program semantics causes the violation not be reported to the user.

The loss of transparency can in particular be exploited to call a method in an inconsistent state. Suppose we have a method `m` that requires data to be consistent, as specified by a class invariant (for example: the value of a variable is within a certain range). If we call method `m` within a `try-catch-finally` statement in a state that does not respect this invariant, the violation of the invariant will not be reported to the user – provided the state is set back to a consistent one before completing the caller of method `m`. This might make him/her believe that the invariant is never violated, and thus that it is safe to run the program without run-time checking enabled (which is more efficient). But if this is done, the method `m` will be called in this inconsistent state, which can cause highly unexpected behaviour, and can easily violate security.

However, without adding additional `try-catch-finally` statements, it is not possible to disguise all the annotation violations – provided the program is sufficiently annotated. In particular, JML run-time checking will check whether the postcondition of the method containing the `try-catch-finally` statement holds upon termination of this method, and whether all class invariants have been reestablished. Thus, this prevents the inconsistent state to be propagated. However, this has the consequence that the error is reported at a different point, than that where it occurred, and thus it breaks *isolation*, *i.e.,* the ability of the run-time checker to identify the source of a problem [3].

5. POSSIBLE SOLUTIONS

5.1 Changing the JML Run-Time Checker

The problem discussed above is caused by the semantics of the finally clause, and its ability to discard exceptions.

Therefore, a simple way to avoid the problem is by forbidding all uses of finally clauses; but of course this solution is a bit drastic, as it would mean that the JML run-time checker cannot be used for all programs written in Java. Moreover, as mentioned above, finally clauses are useful to ensure clean-up of external resources.

As also mentioned above, algorithms exist to eliminate the `jsr` instruction, the bytecode equivalent of the `finally` clause [1]. Even though applying such an algorithm would make it more explicit where potentially a JML error might be discarded, it would not avoid it happening.

In fact, to guarantee that run-time checking with `jmlrac` will report all annotation violations to the user, it is sufficient to restrict the behaviour of `try-catch-finally` statements where the `try` or the `catch` clause could throw a JML error. To ensure that the JML error is not discarded, either the finally clause should terminate normally, or it should throw a JML error itself. This ensures that the whole `try-catch-finally` statement will terminate in an exceptional state, because of a JML error. This is expressed by the following conditions:

- If execution of the `try` block completes abruptly because of a throw of a JML error J, then execution of the whole `try-catch-finally` statement completes abruptly because of a throw of a JML error J'.
- If execution of the `try` block completes abruptly because of a throw of a value V, that is not assignable to a JML error, and if execution of the (leftmost) matching `catch` clause completes abruptly because of a throw of a JML error J, then execution of the whole `try-catch-finally` statement completes abruptly because of a throw of a JML error J'.

Notice that here we leave it unspecified whether the JML error that the whole statement terminates with (J') is the same as the JML error that the `try` or `catch` block terminates with (J). Requiring that these JML errors are identical would ensure even better isolation of the problem.

However, it is not always straightforward to determine statically whether a program satisfies this property. For example, consider the following code fragment, where we suppose that method `m` throws a JML error, and does not change the variable `b`.

```
try {
  if (b) {m();}
  }
finally {
  if (!b) {
     throw new ArbitraryException();
  }
}
```

This satisfies the conditions above, because if `m` is called, and the JML error is thrown, the finally clause will not throw an `ArbitraryException`, thus the JML error will not be discarded. However, it requires advanced analysis techniques to determine this statically.

An alternative solution would be to have `jmlc` instrument the `finally` block by additional `try-catch-finally` constructs, that ensure that JML errors are not discarded – even when the `finally` block throws another exception. Suppose we have a statement `try` T `catch` C `finally` F. To ensure that JML errors that occur in T or C cannot be discarded by F, the statement could be transformed according to the following pattern:

```
try {
  try {
    try T
    catch C
      }
  catch (JMLError j) {
    ... // set flag, store j
    }
  finally F
  }
finally {
  ... // if flag set, rethrow j
}
```

This ensures that whatever happens, F will be executed. However, if execution of T or C gave rise to a JML error `j`, this error will be stored, and the last `finally` clause ensures that the complete statement ends with abruptly because of the JML error `j`.

Finally, another possibility is to change the instrumentation approach of JML: instead of (or in addition to) throwing a special error, any annotation violation is stored into a special log file. However, this means that one gives up on the idea to stop the execution as soon as an annotation violation is detected (to avoid any further damage).

5.2 Changes to the Java Semantics

In fact, we believe that the real source of the problem is that a finally clause can (implicitly) catch exceptions *and* errors and discard them, even though recovery from errors should not be possible, see the Java Language Specification [7, §11.5]. This problem is not only related to errors thrown by the run-time checker, it applies to *all* errors. Consider for example the program in Figure 1. The call to the constructor of class `A` will recursively invoke itself, to initialise the field `a` in class `A`. This will quickly result in a `java.lang.StackOverflowError`. However, the finally clause will discard this error, and throw some other, arbitrary exception. This exception is caught in the surrounding `try-catch` statement, and thus the program will terminate by printing `still alive` – and the occurrence of the `StackOverflowError` has been forgotten. Notice that in a similar way other errors, such as `NoClassDefFoundError`, `OutOfMemoryError` and internal JVM errors, can be discarded.

To avoid the discarding of errors, the semantics of the `try-catch-finally` statement could be adapted such that a finally clause is only executed if the `try` or `catch` clause terminates normally, or with an exception V that is assignable to a variable of type `Exception`. Otherwise, if V is not assignable to an `Exception`, the whole `try-catch-finally` statement completes abruptly because of a throw of value V. This means that the only way to recover from an error is by an explicit catch. In particular, to ensure that run-time checking does report all annotation violations, it is sufficient to check that there are no `catch` clauses for class `Throwable`, `Error` or any JML-specific error class. However, this would mean that upon occurrence of an error, the JVM would have to ensure that external resources are handled appropriately.

```
class A {
    A a = new A();
}


class FinallyCatchesAll {

   public static void main (String [] args) {
     try {
       try {
         A a = new A();
       }
       finally {
         throw new ArbitraryException();
       }
     }
     catch (Exception e) {
     };
     System.out.println ("still alive");
   }
}
```

Figure 1: Recovering from Unrecoverable Errors

An alternative solution would be to prioritise throwables, in such a way that errors have a higher priority than exceptions, while exceptions have a higher priority than other reasons for abrupt completion. One then requires that finally clauses cannot discard errors (unless maybe, they throw an error themselves). Thus, for example, if a `try` (or `catch`) block throws an *error* E, the finally `block` is executed. If execution of the `finally` block results in *exception* E', then exception E' is discarded, and the whole `try-catch-finally` statement terminates abruptly because of a throw of error E.

6. CONCLUSIONS

This paper discusses a case where run-time checking with JML does not report an annotation violation, even though it properly detects it. This is due to an interaction between the run-time checker, and the implicit recovery of exceptions, as specified by the semantics of finally clauses in Java. We argue that in this case, behaviour of the JML run-time checker is not transparent, *i.e.*, applications for which no run-time errors are reported to the user can have different behaviours depending on whether run-time checking is enabled or not. In many cases, the annotation violation will still be signalled later, because other, related annotations are violated, but this has the consequence that the run-time checker looses its ability to isolate problems at the point where they occur.

This paper concentrates on the case of Java applications, annotated with JML, but the observations presented here are partly applicable in any context where the run-time checker is inlined into the code of the application that is being monitored. If the programming language contains a mechanism to recover from the errors that are used to signal run-time annotation violations, then it is possible to recover from them – and thus not to report them to the user. However, what is particular about Java is that finally clauses *implicitly* can discard errors, so that recovery is possible without even explicitly referring to the error found by the run-time checker.

In addition, we showed that the semantics of finally clauses also allows to recover (implicitly) from other errors, that are in fact supposed to be unrecoverable, such as stack overflows.

Above, we discuss two possible changes to the Java semantics that would fix this problem. Of course, given the wide-spread use of Java, and the constraints on backwards compatibility of Java bytecode, such a change to the Java semantics is possibly not a viable solution, and the concrete problem at hand would better be solved by changing the implementation of the JML run-time checker. However, we believe that for future language designs, it is important to consider the issues discussed in this paper. And of course, if the Java semantics would be changed as described above, one would better be suspicious of applications whose behaviour is affected by this change to the language semantics.

Acknowledgements

I would like to thank Alejandro Tamalet, who gave useful feedback on an earlier version of this paper, and pointed out several places where related problems were signalled. Moreover, he also suggested the idea to prioritise errors over exceptions. Also thanks to the anonymous reviewers, who suggested the solution to log the JML errors.

7. REFERENCES

[1] C. Artho and A. Biere. Subroutine inlining and bytecode abstraction to simplify static and dynamic analysis. In *First Workshop on Bytecode Semantics, Verification, Analysis and Transformation (Bytecode 2005)*, volume 141 of *Electronic Notes in Theoretical Computer Science*, pages 109–128, 2005.

[2] G. Barthe, L. Burdy, J. Charles, B. Grégoire, M. Huisman, J.-L. Lanet, M. Pavlova, and A. Requet. JACK: A tool for validation of security and behaviour of Java applications. In *Formal Methods for Components and Objects (FMCO 2006)*, volume 4709 of *Lecture Notes in Computer Science*, pages 152–174. Springer, 2007.

[3] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G.T. Leavens, K.R.M. Leino, and E. Poll. An overview of JML tools and applications. In T. Arts and W. Fokkink, editors, *Formal Methods for Industrial Critical Systems (FMICS 2003)*, volume 80 of *ENTCS*. Elsevier, 2003.

[4] Y. Cheon. *A Runtime Assertion Checker for the Java Modeling Language*. PhD thesis, 2003. Technical Report 03-09.

[5] D. Cok and J. R. Kiniry. ESC/Java2: Uniting ESC/Java and JML: Progress and issues in building and using ESC/Java2 and a report on a case study involving the use of ESC/Java2 to verify portions of an internet voting tally system. In *Construction and Analysis of Safe, Secure and Interoperable Smart Devices (CASSIS 2004)*, volume 3362 of *Lecture Notes in Computer Science*, pages 108–128. Springer, 2005.

[6] B. Eckel. *Thinking in Java*. Prentice-Hall Inc., `http://www.mindview.net/Books/TIJ/`, 2002.

[7] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, third edition*. The Java Series. Addison-Wesley, 2005.

[8] M. Huisman and A. Tamalet. A formal connection

between security automata and JML annotations. In *Fundamental Approaches to Software Engineering (FASE 2009)*, 2009. To appear.

[9] B. Jacobs. A formalisation of Java's exception mechanism. In D. Sands, editor, *Programming Languages and Systems (ESOP)*, volume 2028 of *Lecture Notes in Computer Science*, pages 284–301. Springer, 2001.

[10] J. Jenkov. Java exception handling, 2005 - 2007. Online tutorial, `http://tutorials.jenkov.com/java-exception-handling/index.html`.