Modular Transformation of Java Exceptions Modulo Errors

Robert Rubbens, Sophie Lathouwers, Marieke Huisman

Formal Methods and Tools group
University of Twente

August 12, 2021

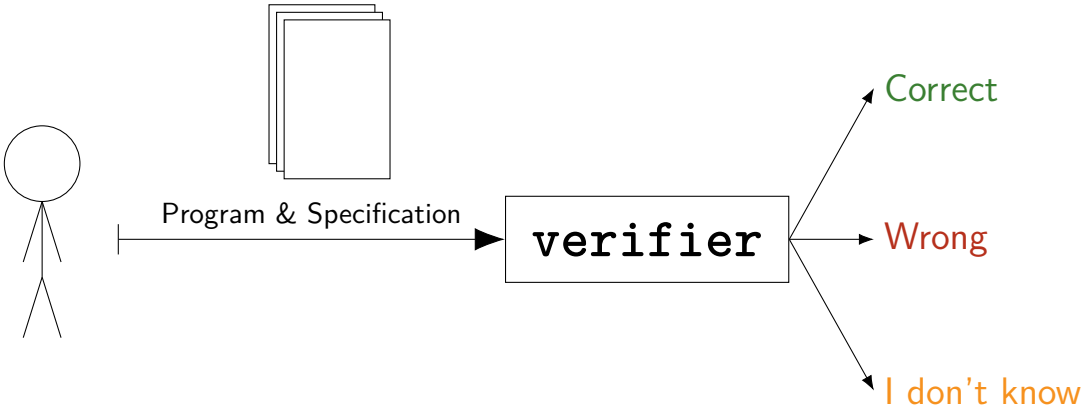
Outline

Outline

Deductive verification

- Deductive verification ensures a system follows a specification
- Correct w.r.t. a deductive system of rules
- User only reads & writes specification

Deductive verification: workflow



Deductive verification in practice

```
1  //@ ensures (a > b ? a : b) == \result;
2  //@ signals (ArithmeticException e) a < 0 || b < 0;
3  int max(int a, int b) {
4    if (a < 0 || b < 0) {
5      throw new ArithmeticException();
6    }
7    return a > b ? a : b;
8  }
```

Listing 1: An implementation of a method computing the maximum of two integers.

VerCors

- Deductive verifier for concurrent languages
- Separation logic
- Data races & functional correctness

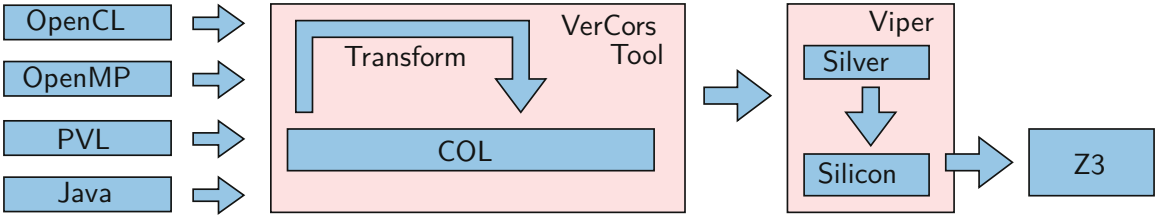


Figure 1: Architecture of VerCors

Verifier usage in practice?

Why are deductive verifiers not used *everywhere*?

Verifier usage in practice?

Why are deductive verifiers not used *everywhere*?

Possible answers:

- Upfront investment

Verifier usage in practice?

Why are deductive verifiers not used *everywhere*?

Possible answers:

- Upfront investment
- Capacity of tools

Verifier usage in practice?

Why are deductive verifiers not used *everywhere*?

Possible answers:

- Upfront investment
- Capacity of tools
- Language support

Verifier usage in practice?

Why are deductive verifiers not used *everywhere*?

Possible answers:

- Upfront investment
- Capacity of tools
- Language support

Deductive verifier capabilities

Name	Language	Separation logic	Exceptions
Nagini	Python	✓ Yes	✓ Yes
Gillian-JS	JavaScript	✓ Yes	✓ Yes
Verifast	Java	✓ Yes	✗ Up to `finally`
jStar	Java	✓ Yes	✗ Trivial `finally`
KeY	Java	✗ No	✓ Yes
OpenJML	Java	✗ No	✓ Yes
Krakatoa	Java	✗ No	✗ Up to `finally`

Table 1: Deductive verifiers and their capabilities

Deductive verifier capabilities

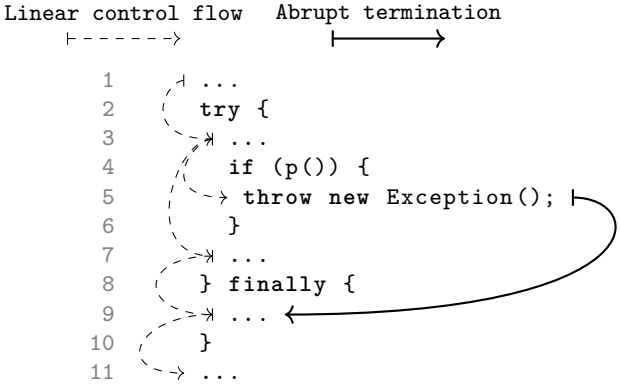
Name	Language	Separation logic	Exceptions
Nagini	Python	✓ Yes	✓ Yes
Gillian-JS	JavaScript	✓ Yes	✓ Yes
Verifast	Java	✓ Yes	✗ Up to `finally`
jStar	Java	✓ Yes	✗ Trivial `finally`
KeY	Java	✗ No	✓ Yes
OpenJML	Java	✗ No	✓ Yes
Krakatoa	Java	✗ No	✗ Up to `finally`
???	Java	✓ Yes	✓ Yes

Table 1: Deductive verifiers and their capabilities

Abrupt termination

- Abrupt termination: statement terminates earlier that normally expected
- Control flow redirected to any program point $\implies$ not linear/structured
- Examples:
  - `throw`
  - `return`, `break`, `continue`
  - Labeled `break`, `continue`

Abrupt termination: `throw` & `finally`



Listing 2: Example usage of `throw` and `finally`

Outline

Outline

Errors vs. sources of errors

Error types:

- `InternalError`
- `ClassFormatError`
- `OutOfMemoryError`
- ...

Errors vs. sources of errors

Error types:

- `InternalError`
- `ClassFormatError`
- `OutOfMemoryError`
- ...

Error sources:

- A bug occurring in the VM
- Loaded class is malformed
- Running out of memory on allocation
- ...

Errors vs. sources of errors

Error types:

- `InternalError`
- `ClassFormatError`
- `OutOfMemoryError`
- ...

Error sources:

- A bug occurring in the VM
- Loaded class is malformed
- Running out of memory on allocation
- ...

When error sources occur, error types are thrown:

A bug occurring in the VM	$\Longrightarrow$	`InternalError`
Loaded class is malformed	$\Longrightarrow$	`ClassFormatError`
Out of memory on allocation	$\Longrightarrow$	`OutOfMemoryError`
Out of memory when loading a class	$\Longrightarrow$	`OutOfMemoryError`

Errors vs. sources of errors

Error types:

- `InternalError`
- `ClassFormatError`
- `OutOfMemoryError`
- ...

Error sources:

- A bug occurring in the VM
- Loaded class is malformed
- Running out of memory on allocation
- ...

When error sources occur, error types are thrown:

A bug occurring in the VM	$\Longrightarrow$	`InternalError`
Loaded class is malformed	$\Longrightarrow$	`ClassFormatError`
Out of memory on allocation	$\Longrightarrow$	`OutOfMemoryError`
Out of memory when loading a class	$\Longrightarrow$	`OutOfMemoryError`

Errors and error sources are not one-to-one!

Annotation overhead: out of memory error source

```
1  String concatenateWithSemicolon(String a, String b) {
2    StringBuilder sb = new StringBuilder();
3
4    sb.append(a +   ";");
5    sb.append(b +   ";");
6
7    return sb.toString();
8  }
```

Listing 3: A method which concatenates two strings

Annotation overhead: out of memory error source

```
1  String concatenateWithSemicolon(String a, String b) {
2    StringBuilder sb = new StringBuilder();①
3
4    sb.append(a +② ";");③
5    sb.append(b +   ";");
6
7    return sb.toString();④
8  }
```

Listing 3: A method which concatenates two strings

Outline

The “ideal” semantics

- Ideal static analysis tool: implements Java Language Specification (JLS)
- Maximizes chance of capturing bug behaviour

Why the ideal semantics is less useful

1. Induced annotation overhead in user code
2. Run-time dependency

Annotation overhead: implications

It is hard to write Java methods that do not allocate

Annotation overhead: implications

It is hard to write Java methods that do not allocate

$$\Longleftrightarrow$$

Most Java methods allocate indirectly

Annotation overhead: implications

It is hard to write Java methods that do not allocate

$\Longleftrightarrow$

Most Java methods allocate indirectly

$\Longleftrightarrow$

Most Java methods will need the following contract in the ideal semantics:

```
//@ signals (OutOfMemoryError) true;
```

Annotation overhead: implications

Note: problem is not implementing all error sources in VerCors/verifier, but meaningless annotations that user will have to write!

Run-time dependency

- Error sources may depend on the runtime environment:
  - `ClassFormatError`: loading improperly formatted code
  - `InternalError`: JVM bug occurs
- Avoid flooding user with unworkable run-time requirements
  - Classpath, VM version, ...
  - **Non-standard**

Why the ideal semantics is less useful: conclusion

1. Induced annotation overhead in user code
2. Run-time dependency

$\Longrightarrow$ The ideal semantics is not useful if implemented

Outline

An optimal and usable semantics

Definition (Exceptions modulo errors)

A subset of Java semantics where exceptions can only come from the `throw` statement and method calls.

An optimal and usable semantics

Definition (Exceptions modulo errors)

A subset of Java semantics where exceptions can only come from the `throw` statement and method calls.

Definition (VerCors exception guarantee)

All thrown exceptions are either:

- Handled in a catch
- Indicated in a throws clause

Additionally, the following errors will not occur:

- `NullPointerException` when a `null` reference is dereferenced.
- `ArithmeticException` when division by zero or modulo zero takes place.
- `ArrayIndexOutOfBoundsException` for out of bounds array accesses.

An optimal and usable semantics: motivation

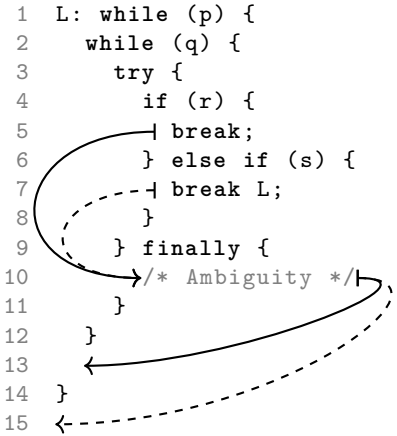
- Reduces annotation burden
  - No annotations for `InternalError`, `OutOfMemoryError`, ...
- In practice, `throw` and `throws` are the only considered sources.
- Users can "opt in" to certain error sources by adding `throws`/`signals` manually to user code.

Outline

Encoding exceptions without `finally`: ok

- `break` exits a `while`
- `continue` jumps back to beginning
- `return` exits method
- `throw` jumps to `catch`

Encoding exceptions with `finally`: ambiguity



Listing 4: `finally` can introduce ambiguous code paths

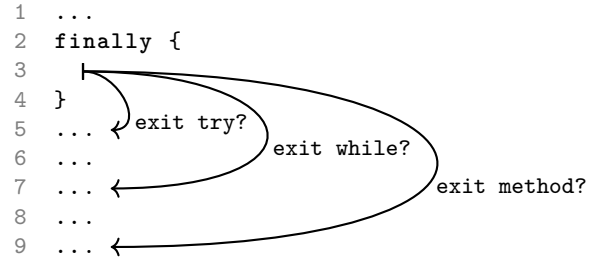
Three implementation strategies

- Inlining: intuitive, but risks blowup
- Control flow flags: avoids blowup, harder to implement
- Via exceptions

Three implementation strategies

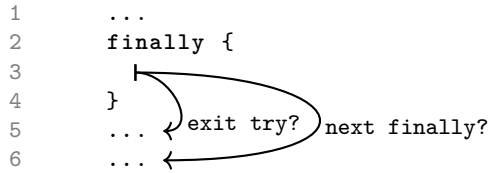
- Inlining: intuitive, but risks blowup
- Control flow flags: avoids blowup, harder to implement
- Via exceptions

What to do after a finally?



Listing 5: Where to go after a `finally`?

What to do after a finally, simplified



Listing 6: Two paths remain when considering only exceptions

Implementation strategy: via exceptions

- Insight: for exceptions, `finally` is straightforward
- Strategy:
  1. Abrupt termination to exceptions
  2. Exceptions to `goto`
- Transformation becomes more modular
- `break`/`return`/`continue` $\implies$ `throw` is straightforward

Transforming `break` into `throw`

```
1 l: while (p) {
2   ...
3   break l;
4   ...
5 }
```

Listing 7: Before transformation

```
1 try {
2   while (p) {
3     ...
4     throw new BreakLException();
5     ...
6   }
7 } catch (BreakLException e) { }
```

Listing 8: After transformation

Outline

Evaluating exceptions modulo errors

How to evaluate exceptions modulo errors?

Evaluating exceptions modulo errors

How to evaluate exceptions modulo errors?

1. What are common exception patterns from commercial software?
2. Can VerCors verify the patterns?

Evaluation table

Used in `catch`	**Used in `finally`**
Empty	Empty
Log, stack trace	Log
`if`, `while`, `switch`, `continue`, `break`, `return`	`continue`, `return`
`throw e`, `throw new E()`, `throw new E(e)`	`throw new E()`
Nested `try`	Nested `try`

Table 2: Exception pattern overview.

Example program

```
1  class CatchStackTrace {
2      void m () {
3          try {
4              throw new Exception();
5          } catch (Exception e) {
6              e.printStackTrace();
7          } } }
```

Listing 9: Example program `CatchStackTrace.java`.

Discussion

- Exceptions modulo errors *might* miss bugs
  - Failing allocations, stack overflow, ...
  - **Beneficial tradeoff**
- Backend requirements
  - `goto`
  - Conditional permissions
    - `ensures b ==> Perm(x, write);`
    - *`RuntimeException is thrown`* `==> Perm(x, write);`
- Performance
  - Bigger encoded output, complex control flow, more conditional permissions
  - Inherent to exceptions
  - Not problematic in practice

Outline

Conclusion

- An approach for supporting exceptions in a deductive verifier
  - First convert control flow into exceptions
  - Then convert exceptions into `goto`
- A simplified exception semantics, "exceptions modulo errors"
  - Reduces annotation burden for the user
  - At the cost of less precision
- Evaluated exceptions modulo errors against common exception patterns
  - It can handle common exception patterns