UNIVERSITY OF TWENTE.

Improving Support for Java Exceptions and Inheritance in VerCors

Bob Rubbens, BSc.

Committee: Prof. Dr. Marieke Huisman, Dr. Luís Ferreira Pires, Sophie Lathouwers, MSc.

Overview

Verification of concurrent software

Deductive verification

Exceptions

Inheritance

Conclusion

Overview

- Verification of concurrent software
- Deductive verification
- Exceptions
- Inheritance
- Conclusion

Verification of concurrent software

Verification of concurrent software?

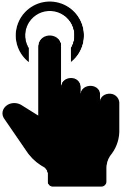
Verification of concurrent software

Verification of concurrent software?

"Verification"

- Verification:
  1. Verify that something works
  2. In relation to a specification
- Specification of a coffee machine

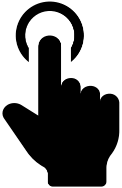
Coffee machine specification



`Action:`
When button is pushed

`Result:`
Coffee must be produced

Coffee machine specification



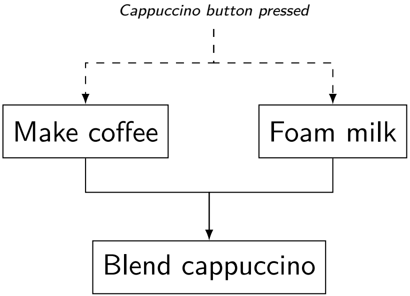
`Action:`
When button is pushed

`Result:`
Coffee must be produced

The challenge: verify an implementation against a specification
<u>automatically</u>, <u>statically</u>

“Concurrent”

- Concurrency means: interleaving of processes
- For cappuccino, need to foam milk & make coffee



“Software”

- Anything that regulates daily life through a computer or electronic device
  1. WhatsApp
  2. PowerPoint
  3. Internet bankieren

Verification of concurrent software

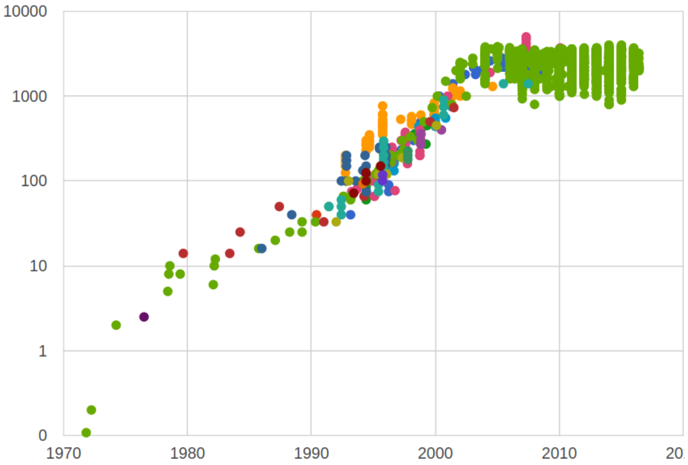
Verification of concurrent software!

Verification	of	concurrent	software!
Ensure it works		Interleaved	Software

Why verification?

- Design is trial and error
- Prevent bugs
- Automation

Why concurrency?

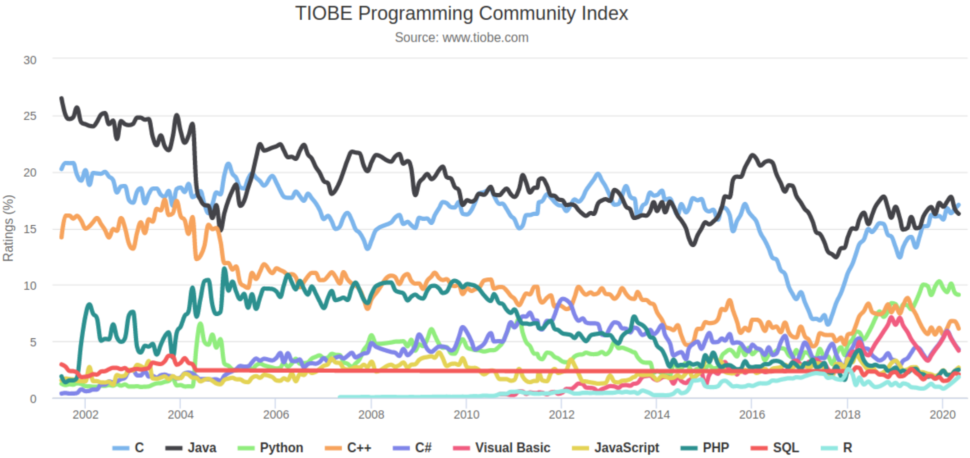


(from Stanford CPUDB)

Why software?

- Society has become dependant on software
- Therefore, we want to verify it

Java

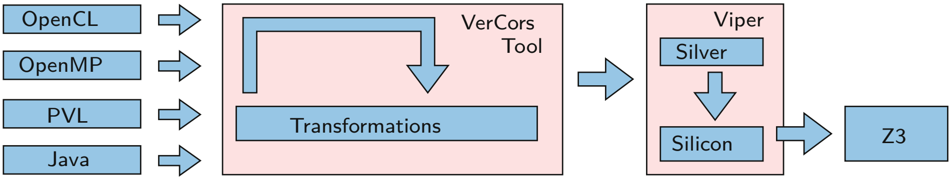

TIOBE Programming Community Index
Source: www.tiobe.com
Ratings (%)
30
25
20
15
10
5
0
2002
2004
2006
2008
2010
2012
2014
2016
2018
2020
C
Java
Python
C++
C#
Visual Basic
JavaScript
PHP
SQL
R

Verify Java?

- Yes, with static verifiers!
- One example, topic of this presentation: VerCors
- Others:
  - Verifast
  - jStar
  - OpenJML
  - KeY
  - And more...

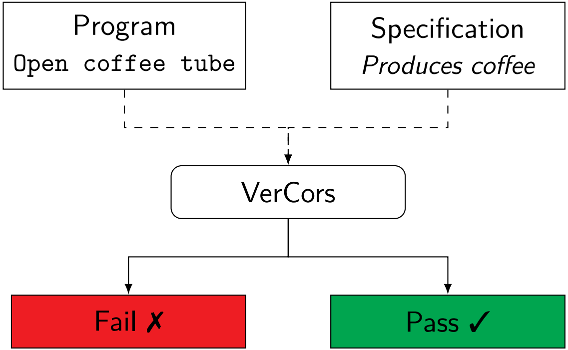
VerCors

- Static deductive verifier for concurrent software
- Developed at the FMT group, University of Twente
- Java, C, OpenCL, PVL
- Data race freedom, memory safety, functional correctness



VerCors architecture

VerCors usage



So, why are there bugs?

So, why are there bugs?

- Problem: support for commercial programming languages
  - Verifast, Nagini
- "Advanced" language features
- exceptions, inheritance, lambdas, streams

Overview

Verification of concurrent software

Deductive verification

Exceptions

Inheritance

Conclusion

Deductive verification of Java

Using JML annotations in comments:

```
//@ requires a >= 0 && b >= 0;
//@ ensures a > b ? \result == a : \result == b;
int positive_max (int a, int b) {
  //@ assert a >= 0;
  if (a > b) {
    return a;
  } else {
    return b;
  }
}

positive_max(-1, 5);           // Fail
int x = positive_max(4, 10); // Pass
//@ assert x == 10;            // Pass
```

Separation logic

- Developed by John C. Reynolds, Peter O'Hearn, Samin Ishtiaq, and Hongseok Yang.
- Intended to describe ownership in programs with references.
- Turns out to also work surprisingly well for concurrent programs! (with some extensions)

Permissions

- `Perm`(x, f)
- Means:
  - Given heap location x...
  - $f = 1 \implies$ Read/write x
  - $0 < f < 1 \implies$ Read x
- Examples:
  - `Perm(x, 1/1)`
  - `Perm(this.y, 1/2)`
  - `Perm(obj.field, 3/6)`

Permissions

- A permission is a resource
- Finite: split/merge, but not duplicate
- Examples:

```
assert Perm(x, 1/1);
assert Perm(x, 1/2) ** Perm(x, 1/2);
assert Perm(x, 1/1);
assert Perm(x, 1/1) ** Perm(x, 1/1); // Fails!
```

VerCors backend: Viper

- Developed at ETH Zürich
- Verifies simple language with permissions
- “Silver”

Java to Silver

1: Java

```
//@ ensures \result == 3;
void m() {
    int x = 2;
    return x + 1;
}
```

2: Silver

```
method m()
  returns (res: Int)
  ensures res == 3
{
  var x: Int;
  x := 2;
  res := x + 1;
}
```

Overview

Verification of concurrent software

Deductive verification

Exceptions

Inheritance

Conclusion

Exceptions

```
void close() throws Exception {
  if (f == null) {
    throw new Exception("f is null");
  } else {
    f.close();
  }
}

void doWork() {
  try {
    close();
  } catch (Exception e) {
    print("Something went wrong!");
  }
}
```

Exception specification

```
//@ signals (Exception e) f == null;
void close() throws Exception {
  if (f == null) {
    throw new Exception("f is null");
  } else {
    f.close();
  }
}
```

Abrupt termination

```
void m() {
  l: while(p()) {
    if (p()) {
      throw new RuntimeException();
    } else {
      break l;
    }
  }
}
```

Abrupt termination

```
void m() {
  l: while(p()) {
    if (p()) {
      throw new RuntimeException();
    } else {
      break l;
    }
  }
}
```

No problem, right?

Abrupt termination to goto

```
void close() {
  l: while(p()) {
    if (p()) {
      goto close_end;
    } else {
      goto l_end;
    }
  } l_end:
close_end: }
```

Abrupt termination & `finally`

```
void close() {
  while(p()) {
    try {
      if (p()) {
        throw new RuntimeException();
      } else {
        break;
      }
    } finally {

    }
  }
}
```

Abrupt termination & `finally`

```
void close() {
  while(p()) {
    try {
      if (p()) {
        throw new RuntimeException();
      } else {
        break;
      }
    } finally {

    }
  }
}
```

Abrupt termination & `finally`

```
void close() {
  while(p()) {
    try {
      if (p()) {
        throw new RuntimeException();
      } else {
        break;
      }
    } finally {

    }
  }
}
```

"`finally` encoding problem"

Approaches to the `finally` encoding problem

1. Inlining
   - Inflates AST
   - Duplicates proof obligations

Approaches to the `finally` encoding problem

1. Inlining
2. Auxiliary state

```
if (p()) {
  break;
} else {
  return;
}
```

$\Rightarrow$

```
if (p()) {
  mode = BREAK;
  goto finally;
} else {
  mode = RETURN;
  goto finally;
}
```

Approaches to the `finally` encoding problem

1. Inlining
2. Auxiliary state
   - Creates constants to keep track of in the presence of labeled `break`
   - Leads to non-modular `finally`

Approaches to the `finally` encoding problem

1. Inlining
2. Auxiliary state
3. Via exceptions

Encode `finally` via exceptions

Consider `finally` with *only* exceptions:

```
try {
  ...
} catch (Exception e) {
  ...
} finally {
  ...
  if (exception) {
    goto next_handler;
  } else {
    goto after;
  }
}
after:
```

Encode `finally` via exceptions

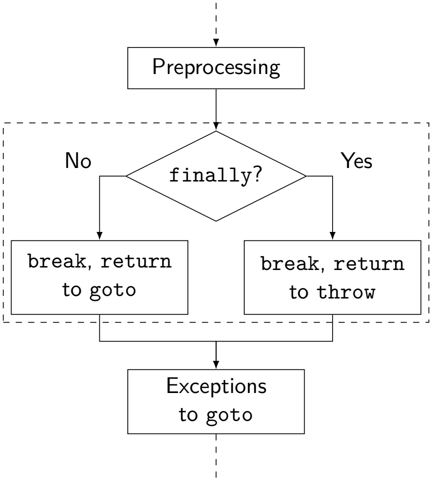
- This only works if there is only exceptional control flow
- That is possible:

```
l: while(p()) {
  ...
  break l;
  ...
}
```

⇒

```
try {
  while(p()) {
    ...
    throw new L();
    ...
  }
} catch (L e) {};
```

Implemented abrupt termination transformation



Overview

Coffee machine inheritance

Coffee machine inheritance

Parent, super

Child, sub

Inheritance example

```
class Cell {
  int val;
  void set(int newVal) {
    val = newVal;
  }
}

class ReCell extends Cell {
  int bak;
  void set(int newVal) {
    bak = super.get();
    super.set(newVal);
  }
}
```

Example with plain contracts

```
//@ requires true;
//@ ensures val == newVal;
void Cell.set(int newVal) {
  val = newVal;
}

//@ requires true;
//@ ensures bak == \old(val) && val == newVal;
void ReCell.set(int newVal) {
  bak = super.get();
  super.set(newVal);
}
```

Behavioural subtyping, informally

Wherever a parent method is used, a child method should also be usable.

Behavioural subtyping, informally

Wherever a parent method is used, a child method should also be usable.

In terms of contracts:

Definition (Method Subtyping)

Given a method *requires* P; *ensures* Q; $f()$ and f' that overrides it, f' is a behavioral subtype of f if:

- $P \implies P'$
- $Q' \implies Q$

Plain contracts subtype

```
//@ requires true;
//@ ensures val == newVal;
void Cell.set(int newVal);

//@ requires true;
//@ ensures bak == \old(val) && val == newVal;
void ReCell.set(int newVal);
```

```
true ==> true

(bak == \old(val) && val == newVal)
           ==> (val == newVal)
```

Example with separation logic contracts

```
//@ requires Perm(val, 1/1);
//@ ensures Perm(val, 1/1) ** val == newVal;
void Cell.set(int newVal);

//@ requires Perm(val, 1/1) ** Perm(bak, 1/1);
/*@ ensures Perm(val, 1/1) ** Perm(bak, 1/1)
            ** bak == \old(val)
            ** val == newVal; @*/
void ReCell.set(int newVal);
```

Example with separation logic contracts

```
//@ requires Perm(val, 1/1);
//@ ensures Perm(val, 1/1) ** val == newVal;
void Cell.set(int newVal);

//@ requires Perm(val, 1/1) ** Perm(bak, 1/1);
/*@ ensures Perm(val, 1/1) ** Perm(bak, 1/1)
            ** bak == \old(val)
            ** val == newVal; @*/
void ReCell.set(int newVal);
```

```
pre-condition Cell ==> pre-condition ReCell

Perm(val, 1/1) ==> Perm(val, 1/1) ** Perm(bak, 1/1)
```

Example with separation logic contracts

```
//@ requires Perm(val, 1/1);
//@ ensures Perm(val, 1/1) ** val == newVal;
void Cell.set(int newVal);

//@ requires Perm(val, 1/1) ** Perm(bak, 1/1);
/*@ ensures Perm(val, 1/1) ** Perm(bak, 1/1)
            ** bak == \old(val)
            ** val == newVal; @*/
void ReCell.set(int newVal);
```

```
pre-condition Cell ==> pre-condition ReCell

Perm(val, 1/1) ==> Perm(val, 1/1) ** Perm(bak, 1/1)
```

✗ Not subtype

Solution: abstract predicate families

- Abbreviated: APF
- Defines "name" shared between classes
- Class can choose "contents"
- Two forms:
  - "Generic": `state()`
  - "Specific": `state@Cell()`
- "Generic" $**$ dynamic type $\iff$ "specific"
- "Specific" $\iff$ "contents"

APF: Cell

```
/*@ resource state(int x) = Perm(val, 1/1)
        ** val == x; @*/

//@ requires state(oldVal);
//@ ensures state(newVal);
void Cell.set(int newVal) {
  //@ unfold state(oldVal);
  //@ unfold state@Cell(oldVal);
  //@ assert Perm(val, 1/1);
}

//@ requires state(oldVal, oldBak);
//@ ensures state(newVal, oldVal);
void ReCell.set(int newVal);
```

APF: Behavioural subtype?

```
state(oldVal) ==> state(oldVal, oldBak)

state(newVal, oldVal) ==> state(newVal)
```

APF: Behavioural subtype?

```
state(oldVal) ==> state(oldVal, oldBak)

state(newVal, oldVal) ==> state(newVal)
```

✓ APFs allow behavioural subtyping

APF exchange problem

- “Generic” ** dynamic type $\Longleftrightarrow$ “specific”
- Dynamic type is not known: only subtype
  - `super`

APF exchange problem

```
//@ requires state(oldVal);
void Cell.set(int newVal) {
  //@ assert this == Cell; // Maybe...?
  //@ assert this == ReCell; // Maybe...?
  //@ assert this instanceof Cell; // True
  //@ unfold state(oldVal); // Not allowed
```

For example:

```
void ReCell.set(int x) {
    // Dynamic type != Cell
    super.set(x);
}
```

Resolving the APF exchange problem

1. “Non-modular”
2. “Extension”
3. “Static/dynamic”

Suggested for VerCors: combine extension & static/dynamic

Static/dynamic

- Insight: dynamic dispatch $\implies$ dynamic type
- “Generic” ** dynamic type $\iff$ “specific”

```
Cell c = ...;
c.set(3);

void Cell.set(int x);

void ReCell.set(int x);
```

Static/dynamic example

```
//@ requires state(oldVal);
void Cell.set(int newVal) {
  //@ assert state@Cell(oldVal);
```

Static/dynamic trade-offs

- Benefit: modular, allows modelling parameters
- Drawback: complicated, no side-calling

Extension

- Insight: APFs the parent APF
- `extract` statement

extract example

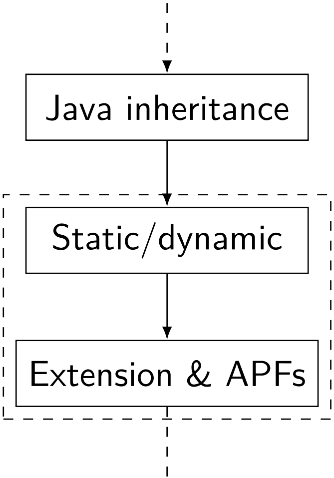
- “Generic” ** `instanceof` $\iff$ “partial specific”
- “Partial specific” ** `instanceof` $\iff$ “generic”

```
Cell c = ...;
//@ assert c.state(oldVal);
//@ extract c.state@Cell(oldVal);
//@ assert c.state@Cell(oldVal);
```

Extension trade-offs

- Benefits:
  - Straightforward to explain.
  - Integrates well with VerCors.
- Drawbacks:
  - Parent APF inclusion is mandatory.
  - `extract` is read-only.

Suggested transformation



Overview

Verification of concurrent software

Deductive verification

Exceptions

Inheritance

Conclusion

Future work

- Formal proof of correctness
- Further improving language support
- Standard library specification
- Improve theory of inheritance

Conclusion

- Static verifiers do not support commercial languages enough
- Abrupt termination can be encoded in exceptions
- VerCors could support inheritance through combined approaches
- Concluding:
  - Full exception support is achievable
  - Basic inheritance is possible with trade-offs.

Bonus slides

Name	Development	Viper	Concurrency	Exceptions	Inheritance
Nagini	Current	Yes	Full	Yes	Yes
Prusti	Current	Yes	Implicit	No	No
Soothsharp	Prototype	Yes	Implicit	No	No
Rust2Viper	Prototype	Yes	Implicit	No	No
Scala2Sil	Prototype	Yes	Implicit	No	No
Frama-C	Current	No	Full	No	No
Verifast	Current	No	Full	Up to `finally`	Yes
KeY	Current	No	No	Yes	Yes
OpenJML	Current	No	No	Yes	Yes
JaVerT	No	No	No	Yes	No
K	Current	No	Full	—	—
Spec#	No	No	No	Yes	Yes
jStar	No	No	Implicit	No	Yes
LOOP	No	No	No	Yes	Yes
Krakatoa	No	No	No	Yes	No
VCC	No	No	Full	—	—
Caper	Unclear	No	Implicit	—	—
Why3	Current	No	No	Yes	No

continue to break

```
l: while(p()) {
  ...
  continue l;
  ...
}
```

$\Rightarrow$

```
l: while(p()) {
  inner_l: {
    ...
    break inner_l;
    ...
  }
}
```

return to throw

```
void m() {
  ...
  return v;
  ...
}
```

⇒

```
void m() {
  try {
    ...
    throw new R_m(v);
    ...
  } catch (R_m e) {
    return e.value;
  }
}
```

```
try {
  loopA: while (p) {
    while (q) {
      try {
        if (r) {
          break;
        } else if (s) {
          break loopA;
        } else {
          return;
        }
      } finally {
        /* Ambiguity */
      }
    }
  }
} finally {
}
}
```

Extension & locking

```
class Cell {
  int val;
  //@ resource lock_invariant() = Perm(val, 1\1);
}
void doWork(Cell c) {
  synchronized (c) {
    //@ assert c.lock_invariant();
    //@ extract c.lock_invariant@Cell();
    //@ unfold c.lock_invariant@Cell();
    c.val = c.val + 2;
    //@ fold c.lock_invariant@Cell();
    //@ apply c.lock_invariant@Cell() -* c.lock_invariant();
    //@ assert c.lock_invariant();
  }
}
```

extract read-only

```
//@ resource state(int x) = Perm(val, 1\1) ** val == x;

//@ requires state(oldVal);
void set(Cell c, int newVal) {
  //@ extract c.state@Cell(oldVal);
  //@ unfold c.state@Cell(oldVal);
  c.val = newVal;
  //@ fold c.state@Cell(newVal);
  //@ assert c.state@Cell(oldVal) -* c.state(oldVal);
  // Impossible:
  //@ apply c.state@Cell(newVal) -* c.state(newVal);
}
```