

Formal Correctness Proofs of Refactorings

Volker Stolz¹

Ole Jørgen Abusdal¹ Eduard Kamburjan² Violet Ka I Pun¹

¹Western Norway University of Applied Sciences

²University of Oslo



← Our KeY updates — for the paper, see ISoLA (2)!

Refactoring and relational verification

Refactoring:

Improve structure of code, preserve behavior of executions

`if(E) {S1;} else {S2;} return; ~ if(!E) {S2;return;} S1; return;`

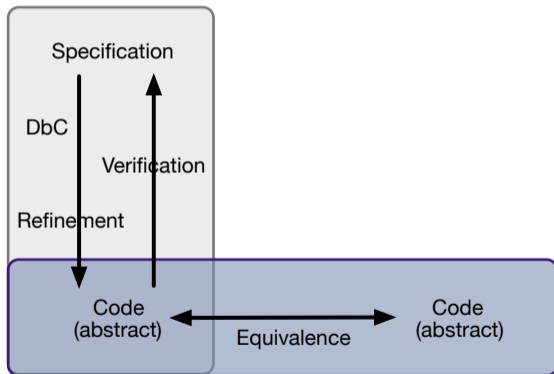
placeholders

Relational verification:

Relate pairs of executions, given initial state satisfy Φ then final state satisfy Ψ

`original ~ refactored : $\Phi \implies \Psi$`

Applications to Security



- KeY framework: information flow, non-interference for Java programs
- REFINITY/abstract execution: proofs on code-fragments (as specifications)

A fundamental relational property

Program equivalence

Two programs are equivalent iff they produce the same output when executed on the same input.

Here: let's look at Java fragments that we consider equivalent.

- How far can current tool support take us?
- Other definitions of equivalence?

Relational verification in practice

REFINITY

- Built on top of the KeY automated theorem prover
- Enables relational verification of “Java” with placeholders
- Placeholders are subject to Abstract Execution
- Has been sufficiently powerful to verify statement level refactorings¹

¹See Dominic Steinhöfel's PhD thesis: <https://tuprints.ulb.tu-darmstadt.de/8540/>

Free Program Variables

Abstract Program Fragments

```

1 | /*@ ae_constraint
2 | @ \disjoint(frameA, frameB) &&
3 | @ \disjoint(frameA, footprintB) &&
4 | @ \disjoint(frameB, footprintA) &&
5 | @
6 | @ \mutex(returnsA(\value(footprintA)), returnsB(\value(footprintB))) &
7 | @ \mutex(returnsA(\value(footprintA)), throwsExcB(\value(footprintB))) &
8 | @ \mutex(throwsExcA(\value(footprintA)), throwsExcB(\value(footprintB))) &
9 | @ \mutex(throwsExcA(\value(footprintA)), returnsB(\value(footprintB))) &
10 | @
11 | @ (throwsExcA(\value(footprintA)) || returnsA(\value(footprintA))) ==>
12 | @ (throwsExcB(\value(footprintB)) || returnsB(\value(footprintB))) ==>
13 | @*/
14 |
15 | /*@ assignable frameA;
16 | /*@ accessible footprintA;
17 | /*@ exceptional_behavior requires throwsExcA(\value(footprintA));
18 | /*@ return_behavior requires returnsA(\value(footprintA));
19 | \abstract_statement A;
20 |
21 | /*@ assignable frameB;
22 | /*@ accessible footprintB;
23 | /*@ exceptional_behavior requires throwsExcB(\value(footprintB));
24 | /*@ return_behavior requires returnsB(\value(footprintB));
25 | \abstract_statement B;

```

Abstract Location Sets

- LocSet relevant
- LocSet frameA
- LocSet footprintA
- LocSet frameB
- LocSet footprintB

Functions and Predicates

- throwsExcA(any)
- throwsExcB(any)
- returnsA(any)
- returnsB(any)

Method-Level Context

Abstract Program Fragments

```

1 | /*@ ae_constraint
2 | @ \disjoint(frameA, frameB) &&
3 | @ \disjoint(frameA, footprintB) &&
4 | @ \disjoint(frameB, footprintA) &&
5 | @
6 | @ \mutex(returnsA(\value(footprintA)), returnsB(\value(footprintB))) &
7 | @ \mutex(returnsA(\value(footprintA)), throwsExcB(\value(footprintB))) &
8 | @ \mutex(throwsExcA(\value(footprintA)), throwsExcB(\value(footprintB))) &
9 | @ \mutex(throwsExcA(\value(footprintA)), returnsB(\value(footprintB))) &
10 | @
11 | @ (throwsExcA(\value(footprintA)) || returnsA(\value(footprintA))) ==>
12 | @ (throwsExcB(\value(footprintB)) || returnsB(\value(footprintB))) ==>
13 | @*/
14 |
15 | /*@ assignable frameB;
16 | /*@ accessible footprintB;
17 | /*@ exceptional_behavior requires throwsExcB(\value(footprintB));
18 | /*@ return_behavior requires returnsB(\value(footprintB));
19 | \abstract_statement B;
20 |
21 | /*@ assignable frameA;
22 | /*@ accessible footprintA;
23 | /*@ exceptional_behavior requires throwsExcA(\value(footprintA));
24 | /*@ return_behavior requires returnsA(\value(footprintA));
25 | \abstract_statement A;

```

Relevant Locations (Left)

- LocSet relevant

Relevant Locations (Right)

- LocSet relevant

Relational Postcondition

```
\result_1==\result_2
```

Relational Precondition

Relational Postcondition

Try to use tooltips if feeling unsure about the functionality of an element.

Proof State: No Proof

Loaded Proofs

Proofs

Env. with model src@4:34:35 PM

✓ [var/folders/bq/rypr1v9j2p9b2wzs2gr44sm0000gn/T/KeYunzip13]

Proof

Info

Proof Search Strategy

Proof Tree

- if returns true
- if returns false

Show taclet info (inner nodes only)

Strategy: Applied 3895 rules (6.7 sec), closed 27 goals, 0 remaining

Sequent

Inner Node

```
====>
  !_objUnderTest = null
  & Problem::exactInstance(_objUnderTest) = TRUE
  & wellFormed(heap)
  & measuredByEmpty
  & !_objUnderTest.<created> = TRUE
```

Proof Statistics

Proved.

Nodes	3,922
Branches	27
Interactive steps	0
Symbolic execution steps	117
Automode time	6747ms
Avg. time per step	1.720ms

Rule applications

Quantifier instantiations	17
One-step Simplifier apps	0
SMT solver apps	0
Dependency Contract apps	0
Operation Contract apps	0
Block/Loop Contract apps	2
Loop invariant apps	0
Abstract Execution apps	7
Merge Rule apps	0
Total rule apps	3,921

Close

Export as CSV

Export as HTML

Source

Problem.java

```
23
24  // @ assignable \dl_frameB;
25  // @ accessible \dl_footprintB;
26  // @ exceptional_behavior requires \dl_thr
27  // @ return_behavior requires \dl_returnsB
28  \abstract_statement B;
29  return null;
30
31
32
33  public Object right() {
34    /* @ ae_constraint
35     @ \disjoint(\dl_frameA, \dl_frameB) &
36     @ \disjoint(\dl_frameA, \dl_footprint
37     @ \disjoint(\dl_frameB, \dl_footprint
38     @ \mutex(\dl_returnsA(\value(\dl_foot
39     @ \mutex(\dl_returnsA(\value(\dl_foot
40     @ \mutex(\dl_throwsExcA(\value(\dl_fo
41     @ \mutex(\dl_throwsExcA(\value(\dl_fo
42     @
43     @ (\dl_throwsExcA(\value(\dl_footprin
44     @ (\dl_throwsExcB(\value(\dl_footprin
45     @*/
46    { ; }
47
48  // @ assignable \dl_frameB;
49  // @ accessible \dl_footprintB;
50  // @ exceptional_behavior requires \dl_thr
51  // @ return_behavior requires \dl_returnsB
52  \abstract_statement B;
53
54  // @ assignable \dl_frameA;
55  // @ accessible \dl_footprintA;
56  // @ exceptional_behavior requires \dl_thr
57  // @ return_behavior requires \dl_returnsA
58  \abstract_statement A;
59  return null;
60
61
62
63 }
```

Show Postcondition/Assignable

The screenshot displays the REFINITY IDE interface with the following components:

- Top Bar:** Includes icons for file operations (save, search), a "Synchronize Scrolling" checkbox, and playback controls (play, stop, refresh).
- Left Panel:**
 - Free Program Variables:** Lists `java.lang.Object x` and `java.lang.Object y`.
 - Abstract Location Sets:** Lists `LocSet relevant`.
 - Functions and Predicates:** Currently empty.
- Main Area:** Split into two panes under the "Abstract Program Fragments" header.
 - Left Pane:** Contains code:

```
1 x = new C();  
2 y = new D();
```
 - Right Pane:** Contains code:

```
1 y = new D();  
2 x = new C();
```
- Bottom Panel:** Titled "Method-Level Context" and "Abstract Program Fragments", it contains three sub-panels:
 - Relevant Locations (Left):** Lists `java.lang.Object x` and `java.lang.Object y`.
 - Relevant Locations (Right):** Lists `java.lang.Object x` and `java.lang.Object y`.
 - Relational Postcondition:** Contains the expression `\result_1==\result_2`.

Recommended Example: File > Load Example > Abstract Execution > Consolidate Duplicate... > Extract Prefix

Proof State: No Proof




REFINITY

○○○○●○○○○○○○

Object Creation

○○○○○○○○○

Go to page 8

Synchronize Scrolling   

Abstract Program Fragments

1 x = new C();	1 y = new D();
2 y = new D();	2 x = new C();

Run Z3

Loaded Proofs

Proofs

Env. with model src@4:37:47 PM

`[var/folders/bq/rypr1v9j2p9b2wzs2gr44sm00000gn/T/KeYunzip728`

Proof

Info

Proof Search Strategy

```

134: return this;
135: return __NEW__;
136: o_2 = __NEW__;
137: sequentialToParallel2
138: simplifyUpdate2
139: simplifyUpdate3
140: applyOnElementary
141: pushHeapUpdateToEnd
142: applyOnPV
143: {}
144: {}
145: o = o_2;
146: sequentialToParallel2
147: simplifyUpdate2
148: simplifyUpdate3
149: applyOnElementary
150: pushHeapUpdateToEnd
151: applyOnPV
152: Problem var#0 = _objUnderTest;
153: Problem var#0;
154: var = _objUnderTest;
155: sequentialToParallel2

```

Show taclet info (inner nodes only)

Strategy: Applied 155 rules (0.4 sec), closed 0 goals, 1 remaining

Sequent

Current Goal

```

Problem.C: exactInstance(o_5) = TRUE,
Problem.D: exactInstance(o_4) = TRUE,
forall Seq_res1; (!P(_res1) | !Q(_res1)),
Problem: exactInstance(_objUnderTest) = TRUE,
wellFormed(heap),
measuredByEmpty,
_objUnderTest.<created> = TRUE
==>
o_5.<created> = TRUE,
o_5 = null,
o_4.<created> = TRUE,
o_4 = null,
pvElementOf(PV(result), relevant),
pvElementOf(PV(exc), relevant),
_objUnderTest = null,
{result:null
 || exc:null
 || x:=x
 || y:=y
 || __NEW__1:=o_5
 || heap:=heap[create(o_5)][o_5.<initialized> := FALSE]}
\<{try {method-frame(result->result, source=left(java.
  Problem,this = _objUnderTest)
  : { {method-frame(result->o_3, source=<createObjec
    : {method-frame(source=<create>())@Problem.C,th
      : { {method-frame(source=<prepareEnter>())@Prob
        : {
          super.<prepare>();
        }
      }
    }
  }
  return this;
}
}
}
o_1=o_3; {
  Problem var#1 = _objUnderTest;
  o_1.<init>(<var#1>@Problem.C;
}
o_1.<initialized>=true;
x=o_1;
y=new D ();
return null;
}
} catch (Throwable t) {

```

Source

Problem.java

```

1 public class Problem {
2   public Object left(java.lang.Object x,java.lang
3     x = new C();
4     y = new D();
5     return null;
6   }
7
8   public Object right(java.lang.Object x,java.lar
9     y = new D();
10    x = new C();
11    return null;
12  }
13
14  class C{}
15    class D{}
16 }

```

Normal Execution

Equivalence

REFINITY checks that the following are identical by default:

- return values
- exceptions
- objects in the so-called relevant location set

Equivalent?

Refactoring tools often get this wrong:

```
x.n();  
x.n();
```

(a) Before

```
X temp = x;  
temp.n();  
temp.n(); //change?
```

(b) After

REFINITY won't close the proof unless you can show the required side-conditions on method `n()`.

The screenshot displays the REFINITY IDE interface. At the top, there is a toolbar with icons for file operations, search, and synchronization. Below the toolbar, the main workspace is divided into several panels:

- Free Program Variables:** Contains the text `java.lang.Object x`.
- Abstract Location Sets:** Lists `LocSet frN` and `LocSet fpN`.
- Functions and Predicates:** Currently empty.
- Abstract Program Fragments:** Two panels showing code snippets. The left panel contains:

```
1 //@ ae_constraint \disjoint(x, frN);
2 assert x instanceof X;
3 ((X)x).n();
4 ((X)x).n();
5 return x;
6
```

The right panel contains:

```
1 //@ ae_constraint \disjoint(x, frN);
2 assert x instanceof X;
3 X temp = (X)x;
4 temp.n();
5 temp.n();
6 return temp;
7
```
- Method-Level Context:** A pop-up window showing the context for the `n()` method:

```
1 public class X {
2
3   public void n() {
4
5     /*@ assignable frN;
6      @ accessible fpN;
7      @ exceptional_behavior requires false;
8      @*/
9     \abstract_statement N;
10  }
11
12 }
```
- Relevant Locations (Left) and (Right):** Empty panels.
- Relational Postcondition:** Contains the text `\result_1 == \result_2`.
- Relational Precondition and Relational Postcondition:** Empty panels.

A black arrow points from the `n()` method call in the left abstract program fragment to the `n()` method definition in the Method-Level Context window.

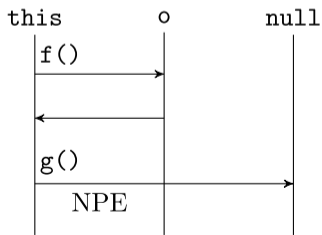
There are code templates for abstract statements, expressions, and constraints! Type "aexp" / "as" / "aec" / "mut" / "disj" followed by Ctrl+Shift+Space (Mac: Command+Shift+Space).

Proof State: No Proof

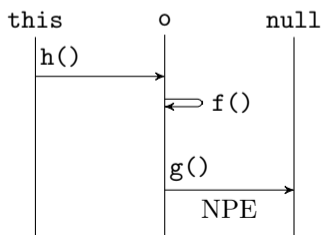
Different output, but equivalent?

Exception origin moved, no additional capture in `h()`

```
o.f().g();  
→ o.h();    with h(){ this.f().g();}
```



original



refactored

Free Program Variables

```
java.lang.Object in
```

Abstract Location Sets

```
LocSet fpG
LocSet frF
LocSet frG
LocSet fpF
```

Functions and Predicates

Abstract Program Fragments

```
1 assert in instanceof Resource;
2 return ((Resource)in
3     .getOwner()
4     .getResource());
5
6
```

Method-Level Context

```
1 public class Resource {
2
3     Owner owner;
4
5     public Owner getOwner() {
6
7         /*@ assignable frF;
8         @ accessible fpF;
9         @*/
10        \abstract_statement F;
11        return owner;
12    }
13
14    public Resource hiddenDelegate() {
15        return this.getOwner().getResource();
16    }
17
18 }
19
20
21 public class Owner {
22
23     Resource resource;
24
25     public Resource getResource() {
26
27
28         /*@ assignable frG;
29         @ accessible fpG;
30         @*/
31        \abstract_statement G;
32        return resource;
33    }
34
35 }
```

Method-Level Context

```
1 assert in instanceof Resource;
2 return ((Resource)in
3     .hiddenDelegate());
```

Method-Level Context

Relevant Locations (Left)

Relational Postcondition

```
\result_1[0] == \result_2[0] &&
(
    (
        \result_1[1] instanceof java.lang.NullPointerException &&
        \result_2[1] instanceof java.lang.NullPointerException
    ) ||
    (\result_1[1] == \result_2[1])
)
```

Relational Precondition

Relational Postcondition

Try to use tooltips if feeling unsure about the functionality of an element.

Proof State: No Proof

Challenges in complex refactorings

Successfully verified variants of *Extract Local Variable* and *Hide Delegate* and investigated how to approach others.

We discuss

Simplifying postcondition specifications

Unresolved

Making the proofs useful artefacts:
what about *instantiations*?

Object equality

REFINITY lacked rules for object equality over multiple modalities:

- can verify `SLIDE STATEMENT` with abstract statements

Free Program Variables

Abstract Program Fragments

```

1 | /*@ ae_constraint
2 | @ disjoint(frameA, frameB) &&
3 | @ disjoint(frameA, footprintB) &&
4 | @ disjoint(frameB, footprintA) &&
5 | @
6 | @ !mutex(returnsA(\value(footprintA)), returnsB(\value(footprintB))) &
7 | @ !mutex(returnsA(\value(footprintA)), throwsExcB(\value(footprintB))) &
8 | @ !mutex(throwsExcA(\value(footprintA)), throwsExcB(\value(footprintB))) &
9 | @ !mutex(throwsExcA(\value(footprintA)), returnsB(\value(footprintB))) &
10 | @
11 | @ (throwsExcA(\value(footprintA)) || returnsA(\value(footprintA))) ==>
12 | @ (throwsExcB(\value(footprintB)) || returnsB(\value(footprintB))) ==>
13 | @*/
14 |
15 | /*@ assignable frameA;
16 | /*@ accessible footprintA;
17 | /*@ exceptional_behavior requires throwsExcA(\value(footprintA));
18 | /*@ return_behavior requires returnsA(\value(footprintA));
19 | \abstract_statement A;
20 |
21 | /*@ assignable frameB;
22 | /*@ accessible footprintB;
23 | /*@ exceptional_behavior requires throwsExcB(\value(footprintB));
24 | /*@ return_behavior requires returnsB(\value(footprintB));
25 | \abstract_statement B;

```

Abstract Location Sets

- LocSet relevant
- LocSet frameA
- LocSet footprintA
- LocSet frameB
- LocSet footprintB

Functions and Predicates

- throwsExcA(any)
- throwsExcB(any)
- returnsA(any)
- returnsB(any)

Method-Level Context

Abstract Program Fragments

```

1 | /*@ ae_constraint
2 | @ disjoint(frameA, frameB) &&
3 | @ disjoint(frameA, footprintB) &&
4 | @ disjoint(frameB, footprintA) &&
5 | @
6 | @ !mutex(returnsA(\value(footprintA)), returnsB(\value(footprintB))) &
7 | @ !mutex(returnsA(\value(footprintA)), throwsExcB(\value(footprintB))) &
8 | @ !mutex(throwsExcA(\value(footprintA)), throwsExcB(\value(footprintB))) &
9 | @ !mutex(throwsExcA(\value(footprintA)), returnsB(\value(footprintB))) &
10 | @
11 | @ (throwsExcA(\value(footprintA)) || returnsA(\value(footprintA))) ==>
12 | @ (throwsExcB(\value(footprintB)) || returnsB(\value(footprintB))) ==>
13 | @*/
14 |
15 | /*@ assignable frameB;
16 | /*@ accessible footprintB;
17 | /*@ exceptional_behavior requires throwsExcB(\value(footprintB));
18 | /*@ return_behavior requires returnsB(\value(footprintB));
19 | \abstract_statement B;
20 |
21 | /*@ assignable frameA;
22 | /*@ accessible footprintA;
23 | /*@ exceptional_behavior requires throwsExcA(\value(footprintA));
24 | /*@ return_behavior requires returnsA(\value(footprintA));
25 | \abstract_statement A;

```

Relevant Locations (Left)

- LocSet relevant

Relevant Locations (Right)

- LocSet relevant

Relational Postcondition

```

\result_1==\result_2

```

Relational Precondition

Relational Postcondition

Try to use tooltips if feeling unsure about the functionality of an element.

Proof State: No Proof

Object equality

REFINITY lacked rules for object equality over multiple modalities:

- can verify Slide Statement with abstract statements
- can't verify Slide Statement with statements involving concrete objects

The screenshot displays the REFINITY IDE interface with the following components:

- Top Bar:** Includes icons for file operations (save, search), a "Synchronize Scrolling" checkbox, and playback controls (play, stop, refresh).
- Left Panel:**
 - Free Program Variables:** Lists `java.lang.Object x` and `java.lang.Object y`.
 - Abstract Location Sets:** Shows `LocSet relevant`.
 - Functions and Predicates:** Currently empty.
- Main Editor:** Split into two panes under the "Abstract Program Fragments" tab.
 - Left Pane:** Contains code:

```
1 x = new C();
2 y = new D();
```
 - Right Pane:** Contains code:

```
1 y = new D();
2 x = new C();
```
- Bottom Panel:** Titled "Method-Level Context" and "Abstract Program Fragments", it contains:
 - Relevant Locations (Left):** `java.lang.Object x`, `java.lang.Object y`
 - Relevant Locations (Right):** `java.lang.Object x`, `java.lang.Object y`
 - Relational Postcondition:** `\result_1==\result_2`
 - Relational Precondition:** (tabbed but empty)
 - Relational Postcondition:** (tabbed but empty)

At the bottom, a status bar shows: **Recommended Example:** File > Load Example > Abstract Execution > Consolidate Duplicate... > Extract Prefix. On the right, it says **Proof State: No Proof**.

REFINITY Internals

Core issue:

- Objects are placed in a symbolic heap during SE
- Before and After program executed in same proof

Not sufficient for two new objects to be equal:

- the allocation must, additionally, be deterministic

Schematic sequent rules in KeY are specified as *taclets*:

- we add rules to make objects indistinguishable under under certain conditions

New *taclet* for object creation

$$\frac{\Gamma, \{U\} (v \neq \text{null} \wedge v \doteq C :: \text{allocate}(\text{heap}) \wedge C :: \text{exactInstance}(v) \doteq \text{TRUE}) \Rightarrow \{U\} \{ \text{heap} := \text{create}(\text{heap}, v) \} [s] \phi, \Delta}{\Gamma \Rightarrow \{U\} [v = C.\text{allocate}(); s] \phi, \Delta}$$

Additionally, we give two simplification rules for heaps within any allocate function application. Let \sqsubseteq be the subtype relation and $T(t)$ the type of a term.

$$\begin{array}{ll} C :: \text{allocate}(\text{store}(h, o, f, v)) \rightsquigarrow C :: \text{allocate}(h) & \text{if } f \neq \langle \text{allocated} \rangle \\ C :: \text{allocate}(\text{create}(h, o)) \rightsquigarrow C :: \text{allocate}(h) & \text{if } C \not\sqsubseteq T(o) \end{array}$$

Postcondition simplification

In `HIDE DELEGATE` exception objects now equivalent

- we need no special postcondition to handle exceptions...
- ...although we should because in practice exceptions capture state! (Not *our* problem, though 🤖)

Free Program Variables

```
java.lang.Object in
```

Abstract Location Sets

```
LocSet fpG
LocSet frF
LocSet frG
LocSet fpF
```

Functions and Predicates

Abstract Program Fragments

```
1 assert in instanceof Resource;
2 return ((Resource)in)
3   .getOwner()
4   .getResource();
5
6
```

```
1 assert in instanceof Resource;
2 return ((Resource)in)
3   .hiddenDelegate();
```

Method-Level Context

Relevant Locations (Left)

Relevant Locations (Right)

Relational Postcondition

```
\result_1 == \result_2
```

Relational Precondition

Relational Postcondition

⚠ There are code templates for abstract statements, expressions, and constraints! Type "aexp" / "as" / "aec" / "mut" / "disj" followed by Ctrl+Shift+Space (Mac: Command+Shift+Space).

Proof State: Closed

The screenshot shows the REFINITY IDE interface. At the top, there are two tabs: "REFINITY" (with 10 empty circles) and "Object Creation" (with 7 empty circles and 1 filled circle). Below the tabs is a toolbar with a "Synchronize Scrolling" checkbox (checked), a play button, a gear icon, and a close button. The main area displays two code snippets side-by-side under the heading "Abstract Program Fragments".

```
1 assert in instanceof Resource;  
2 return ((Resource)in  
3     .getOwner()  
4     .getResource());
```

```
1 assert in instanceof Resource;  
2 return ((Resource)in  
3     .hiddenDelegate());
```

Future Challenge

Trace based notions of equivalence

```

File f = new File();
String s = "";
/*@ ensures finite ** call(f.open) ** finite; */
\abstract_statement A;
s = f.read();
f.write(s);
/*@ ensures finite ** call(f.close) ** finite; */
\abstract_statement B;
  
```

(a) Before

```

File f = new File();
String s = "";
/*@ ensures finite ** call(f.open) ** finite; */
\abstract_statement A;
f.write(s);
s = f.read();
/*@ ensures finite ** call(f.close) ** finite; */
\abstract_statement B;
  
```

(b) After

$$\theta ::= [\phi]$$

| call(m)

| finite

| $\theta ** \theta$

Summary

- REFINITY/KeY excellent foundation for reasoning about OO in general
- abstract code + side conditions
- initial application area: checking refactorings via symbolic execution
- next: application-specific?

SILM Workshop

SILM Workshop

[Dates](#)[Call for Papers](#)[Submission](#)[Program](#)[Registration](#)[Past Events](#)

SILM 2024

Welcome to the 6th edition of our workshop on the **Security of Software/Hardware Interfaces**. SILM 2024* will take place on **Friday, July 12 2024, in Vienna (Austria)**, co-located with the [9th IEEE European Symposium on Security and Privacy \(EuroS&P 2024\)](#)

SUBMISSIONS

Submission deadline is **March 29, 2024 -- 11:59pm AoE** (was: ~~March 15, 2024 -- 11:59pm AoE~~); check our [Call for Papers](#) for details.