How to Prove the Correctness of Refactoring Rules (with Abstract Execution)

15th International Conference on integrated Formal Methods—Refactoring Tutorials, Bergen, Norway

---

Dominic Steinhöfel and Reiner Hähnle

steinhoefel@cs.tu-darmstadt.de

December 2nd, 2019

Software Engineering Group, Computer Science Department, TU Darmstadt

This work was funded by the Hessian LOEWE initiative within the Software-Factory 4.0 project.

How to Get Away with Refactoring (with Abstract Execution)

15th International Conference on integrated Formal Methods—Refactoring Tutorials, Bergen, Norway

Dominic Steinhöfel and Reiner Hähnle

steinhoefel@cs.tu-darmstadt.de

December 2nd, 2019

Software Engineering Group, Computer Science Department, TU Darmstadt

This work was funded by the Hessian LOEWE initiative within the Software-Factory 4.0 project.

Our research is motivated by the observation that common refactorings can easily, and accidentally, *change a program's behaviour*.

A.M. Eilertsen, A.H. Bagge, and V. Stolz: *Safer Refactorings*. ISoLA 2017

For each refactoring we characterize the preconditions that make it semantics-preserving. *Most preconditions are not mentioned in the literature*.

D. Steinhöfel and R. Hähnle: *Abstract Execution*. FM 2019

Goal: *Equivalence* of Programs Before and After Refactoring

```
if (x >= 0) {
  y += x;
  x = -x;
} else {
  y += x;
  y = -y;
}
```

$\xleftrightarrow{\text{equivalent}}$

```
y += x;

if (x >= 0) {
  x = -x;
} else {
  y = -y;
}
```

Goal: *Equivalence* of Programs Before and After Refactoring

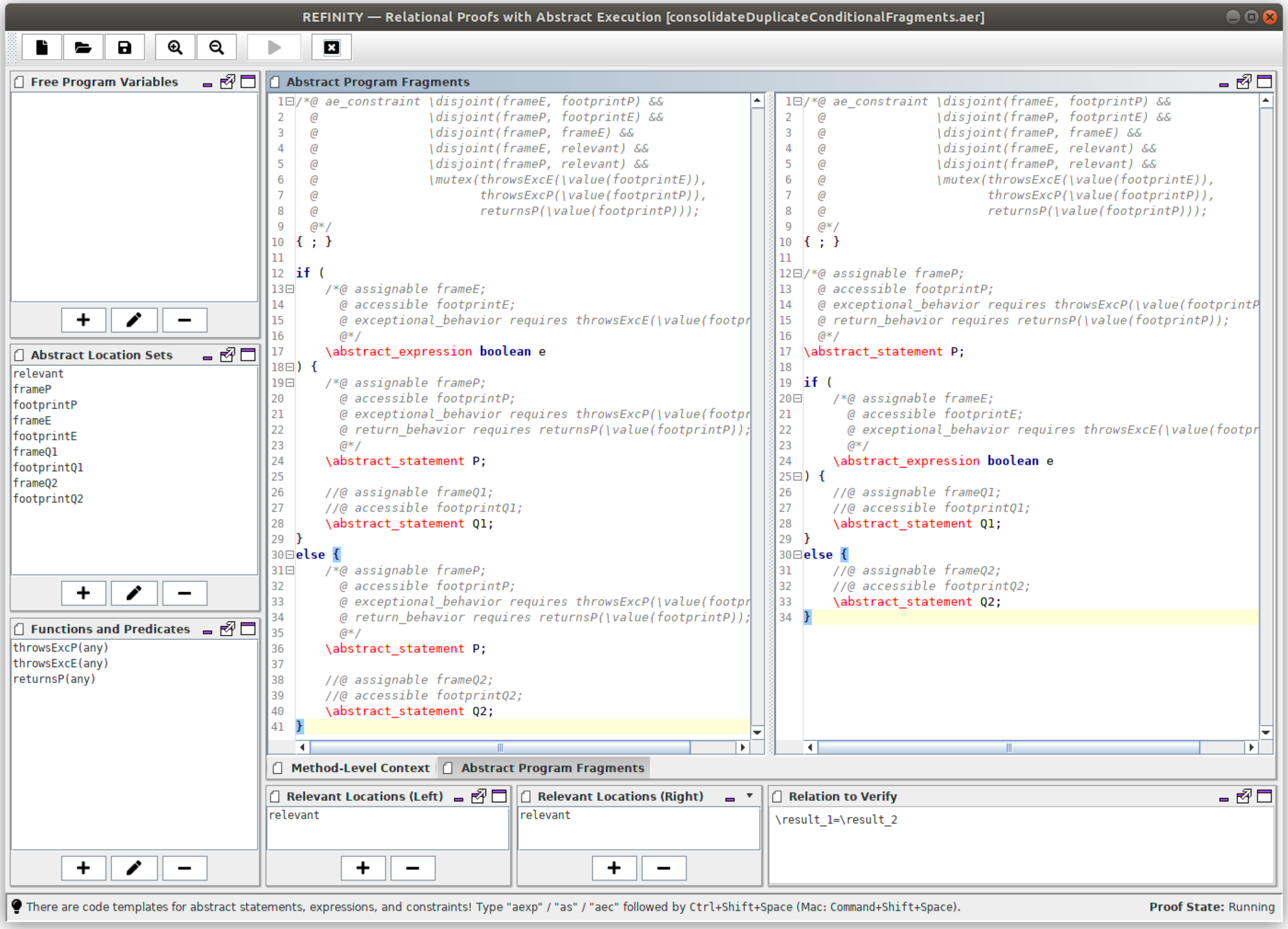
```
if (x >= 0) {
  y += --x;
  x = -x;
} else {
  y += --x;
  y = -y;
}
```

~~equivalent~~ $\longleftrightarrow$

```
y += --x;
if (x >= 0) {
  x = -x;
} else {
  y = -y;
}
```

UNSOUND

REFINITY Automatic, Statement-Level Relational Program Verification Tool



key-project.org/REFINITY/

1st DEMO

Going *Abstract* – Properties of Many Programs

```
if (expr) {
  P
  Q1
} else {
  P
  Q2
}
```

$$\xleftrightarrow{\text{equivalent}}\ ?$$

```
P
if (expr) {
  Q1
} else {
  Q2
}
```

Hierarchy of Verification Approaches

Testing	Show correctness of *one* program for *one* set of inputs
Program Proving	Show correctness of *one* program for *all* possible inputs
Abstract Program Proving	Show correctness of all programs (matching a pattern) for all possible inputs.

Abstract Programs =
Programs with *Abstract Program Elements (APEs)*
(Abstract Statements & Abstract Expressions)

How to show the correctness of an *abstract* program?

```
Inductive com : Type :=
  | CSkip : com
  | CAss : pvs → aexp → com
  | CSeq : com → com → com
  | CIf : bexp → com → com → com
  | CWhile : bexp → com → com.

Inductive ceval : com → state → state → Prop :=
  | E_Skip : forall st,
      SKIP / st \\ st
  | E_Ass : forall st a1 n x,
      aeval st a1 = n →
        ( x =! a1) / st \\ ( s_update st x n)
  (* ... *)
```

How to show the correctness of an *abstract* program?

```
Theorem evaluation_deterministic:
  ∀ c st st1 st2,
    c / st \\ st1 → c / st \\ st2 → st1 = st2.
Proof.
  intros c st st1 st2 H1 H2.
  generalize dependent st2.
  induction H1.
    - (* E_Skip *) reflexivity.
    - (* E_Ass *) reflexivity.
    - (* ... *)
Qed.
```

Abstract Program Proofs by *Structural Induction*

- Frequently practiced in
  - *pen-and-paper proofs* and
  - *interactive theorem provers* like Isabelle and Coq (e.g., *CompCert* and *CakeML*)
- Precise *second-order reasoning* over program properties
- ...but very *hard to automate*!

Goal

Automatic Reasoning about *Universal* Properties of *Abstract* Programs in an *Industrial Programming Language*

Our Solution

Abstract Execution

Abstract Execution

Specification of Abstract Programs +
Symbolic *Execution* +
Abstract State Changes

***Specification* of Abstract Programs** +

Symbolic *Execution* +

Abstract State Changes

Specification of Abstract *Statements*, *Expressions* & *Constraints*

```
//@ specs
\abstract_statement Ident;


//@ specs
\abstract_expression Type Ident


/*@ ae_constraint
  @   formula;
  @*/
{;}
```

Dynamic Frames

frame = what the program *may change*

footprint = what the program *may read*

Dynamic *Frames* are *abstract sets of locations* (or set-valued specification variables).

Specification of *Frames & Footprints*

```
//@ assignable x, someAbstrFrame, \hasTo(y);
\abstract_statement P;
```

```
if (
  //@ accessible z, someAbstrFootprint;
  \abstract_expression boolean e
) { … }
```

```
/*@ ae_constraint
  @   \disjoint(someAbstrFrame, z) && …; */
{;}
```

Specification of *Abrupt Completion Conditions*

```
//@ exceptional_behavior requires condition;
//@ return_behavior      requires condition;
//@ break_behavior       requires condition;
//@ continue_behavior    requires condition;
\abstract_statement Ident;
```

Mutually Exclusive Abrupt Completion Behavior

```
/*@ ae_constraint
  @   \mutex(throwsP(\value(footprintP)),
  @          throwsQ(\value(footprintQ)));
  @*/
{;}

/*@ accessible footprintP;
  @ exceptional_behavior requires
  @   throwsP(\value(footprintP)); */
\abstract_statement P;

/*@ accessible footprintQ;
  @ exceptional_behavior requires
  @   throwsQ(\value(footprintQ)); */
\abstract_statement Q;
```

Specifying the Post Condition

```
if (
  /*@ normal_behavior ensures
    @   \result <==>
    @      throwsP(\value(footprintP));
    @*/
  \abstract_expression boolean e
) {
  println("P threw Exception!");
} else {
  println("P did not throw Exception!");
}
```

2^{nd} DEMO

Abstract Execution

Specification of Abstract Programs +
Symbolic *Execution* +
Abstract State Changes

Symbolic Execution of an Assignment (in JavaDL)

$$\text{assignment}\ \frac{\Gamma \vdash \{\mathcal{U}\}\{\mathtt{x} := e\}[\pi\ \omega]\varphi, \Delta}{\Gamma \vdash \{\mathcal{U}\}[\pi\ \mathtt{x=}e\mathtt{;}\ \omega]\varphi, \Delta}$$

Symbolic Execution of a Conditional Statement (in JavaDL)

IfElseSplit

$$\frac{\Gamma, e \doteq \mathsf{TRUE} \vdash \{\mathcal{U}\}[\pi\ p_1\ \omega]\phi, \Delta \qquad \Gamma, e \doteq \mathsf{FALSE} \vdash \{\mathcal{U}\}[\pi\ p_2\ \omega]\phi, \Delta}{\Gamma \vdash \{\mathcal{U}\}[\pi\ \texttt{if (}e\texttt{)}\ p_1\ \texttt{else}\ p_2\ \omega]\phi, \Delta}$$

Abstract Execution of an Abstract *Statement*

abstractStatement

$$
\frac{
\begin{array}{l}
\Gamma \vdash \{\mathcal{U}\}\{\mathtt{throwsExc} := \mathit{throwsExc}(\mathit{footprint}) \,||\, \mathtt{exc} := \mathit{exc}(\mathit{footprint}) \,||\\
\qquad \mathtt{returns} := \mathit{returns}(\mathit{footprint}) \,||\, \mathtt{res} := \mathit{res}(\mathit{footprint})\}\\
\qquad\quad \big((\mathtt{returns} \doteq \mathsf{TRUE} \leftrightarrow \neg\mathtt{throwsExc} \doteq \mathsf{TRUE})\land\\
\qquad\quad\; (\mathtt{normal} \doteq \mathsf{TRUE} \leftrightarrow \neg\mathtt{returns} \doteq \mathsf{TRUE} \land \neg\mathtt{throwsExc} \doteq \mathsf{TRUE})\land\\
\qquad\quad\; (\mathtt{throwsExc} \doteq \mathsf{TRUE} \to \neg\mathtt{exc} \doteq \mathtt{null})\land\\
\qquad\quad\; (\mathtt{throwsExc} \doteq \mathsf{TRUE} \leftrightarrow \mathit{excPre})\land\\
\qquad\quad\; (\mathtt{returns} \doteq \mathsf{TRUE} \leftrightarrow \mathit{returnsPre})\big) \to\\
\qquad\qquad \{\mathcal{U}_P(\mathit{frame} :\approx \mathit{footprint})\}\\
\qquad\qquad\quad \big((\mathtt{throwsExc} \doteq \mathsf{TRUE} \to \mathit{excPost})\land\\
\qquad\qquad\quad\; (\mathtt{returns} \doteq \mathsf{TRUE} \to \mathit{returnsPost})\land\\
\qquad\qquad\quad\; (\mathtt{normal} \doteq \mathsf{TRUE} \to \mathit{normalPost})\big) \to\\
\qquad\qquad\qquad [\pi\ \mathtt{if\ (returns)\ return\ res;}\\
\qquad\qquad\qquad\quad \mathtt{if\ (throwsExc)\ throw\ exc;}\ \ \omega]\phi, \Delta
\end{array}
}{
\Gamma \vdash \{\mathcal{U}\}[\pi\ \mathtt{\backslash abstract\_statement}\ P\mathtt{;}\ \ \omega]\phi, \Delta
}
$$

Abstract Execution of an Abstract *Expression*

abstractExpression

$$\frac{\begin{array}{l}\Gamma \vdash \{\mathcal{U}\}\{\texttt{throwsExc} := \mathit{throwsExc}(\mathit{footprint}) \,||\, \\ \quad \texttt{exc} := \mathit{exc}(\mathit{footprint}) \,||\, \texttt{res} := \mathit{res}(\mathit{footprint})\} \\ \quad\big((\texttt{normal} \doteq \mathsf{TRUE} \leftrightarrow \neg\texttt{throwsExc} \doteq \mathsf{TRUE})\wedge \\ \quad\ (\texttt{throwsExc} \doteq \mathsf{TRUE} \rightarrow \neg\texttt{exc} \doteq \texttt{null})\wedge \\ \quad\ (\texttt{throwsExc} \doteq \mathsf{TRUE} \leftrightarrow \mathit{excPre})\big) \rightarrow \\ \quad\quad (\texttt{throwsExc} \doteq \mathsf{TRUE} \rightarrow \\ \quad\quad\quad \{\mathcal{U}_e(\mathit{frame} :\approx \mathit{footprint})\} \\ \quad\quad\quad\quad (\mathit{excPost} \rightarrow [\pi\ \ \texttt{throw exc;}\ \ \omega]\phi))\wedge \\ \quad\quad (\neg\texttt{throwsExc} \doteq \mathsf{TRUE} \rightarrow \\ \quad\quad\quad \{\mathcal{U}_e(\mathit{frame} :\approx \mathit{footprint}) \,||\, \texttt{v} := \texttt{res}\} \\ \quad\quad\quad\quad (\mathit{normalPost} \rightarrow [\pi\ \ \omega]\phi)), \Delta\end{array}}{\Gamma \vdash \{\mathcal{U}\}[\pi\ \ \texttt{v=\textbackslash abstract\_expression}\ \mathit{T}\ \mathit{e}\texttt{;}\ \ \omega]\phi, \Delta}$$

Specification of Abstract Programs +

Symbolic *Execution* +

Abstract State Changes

We use all the existing rules for (concrete) updates...

$$\{\mathcal{U}\}(\mathtt{a} := t) \rightsquigarrow \mathtt{a} := \{\mathcal{U}\}t$$

$$\{\mathcal{U}\}f(t_1, \ldots, t_n) \rightsquigarrow f(\{\mathcal{U}\}t_1, \ldots, \{\mathcal{U}\}t_n)$$

$$\{\mathcal{U}\}\{\mathcal{U}'\}t \rightsquigarrow \{\mathcal{U} \mid\mid \{\mathcal{U}\}\mathcal{U}'\}t$$

$$\vdots$$

...and adapt the “drop update” rules.

$$\{\ldots \mid\mid \mathtt{x} := t' \mid\mid \ldots\}t \rightsquigarrow \{\ldots \mid\mid \mathit{skip} \mid\mid \ldots\}t$$

$$\text{if } \mathtt{x} \notin \mathit{fpv}(t) \ \mathit{and}\ \forall \text{“}\backslash \mathit{value}(\mathit{loc}) \in t\text{”}, \backslash \mathit{disjoint}(\mathit{loc}, \mathtt{x})$$

...and adapt the “drop update” rules.

$$\{\ldots \,||\, \mathcal{U}_{\mathrm{P}}(\ldots, \mathit{frame}_k, \ldots :\approx \mathit{footprint}) \,||\, \ldots\}t$$

$$\rightsquigarrow \{\ldots \,||\, \mathcal{U}_{\mathrm{P}}(\ldots, \_, \ldots :\approx \mathit{footprint}) \,||\, \ldots\}t$$

if “$\backslash\mathit{value}(\mathit{loc}) \in t$”, $\backslash\mathit{disjoint}(\mathit{loc}, \mathit{frame}_k)$

...and adapt the “drop update” rules.

$$\{\ldots \parallel \mathcal{U}_{\mathrm{P}}(\_, \ldots, \_ :\approx \mathit{footprint}) \parallel \ldots\}t$$

$$\rightsquigarrow \{\ldots \parallel \mathit{skip} \parallel \ldots\}t$$

Special Case: “\hasTo” Locations (1) “Outsourcing” *Concrete Updates*

$$\{\ldots \,||\, \mathcal{U}_{\mathtt{P}}(\ldots, \backslash \mathit{hasTo}(\mathtt{x}), \ldots :\approx \mathit{fp}) \,||\, \ldots\}t$$

$$\leadsto \{\ldots \,||\, \mathcal{U}_{\mathtt{P}}(\ldots, \mathtt{x}, \ldots :\approx \mathit{fp}) \,||\, \mathtt{x} := \mathit{anon}^{k}_{\mathtt{P}}(\mathit{fp}) \,||\, \ldots\}t$$

Special Case: "\hasTo" Locations
(2) *Stronger Simplifications*

$$\{\mathtt{x} := t'\}\{\mathcal{U}_{\mathtt{P}}(\ldots, \mathtt{x}, \ldots :\approx fp)\}t$$

$$\rightsquigarrow \{\mathtt{x} := t' \,||\, \mathcal{U}_{\mathtt{P}}(\ldots, x, \ldots :\approx \{\mathtt{x} := t'\}fp)\}t$$

$$\{\mathtt{x} := t'\}\{\mathcal{U}_{\mathtt{P}}(\ldots, \backslash hasTo(\mathtt{x}), \ldots :\approx fp)\}t$$

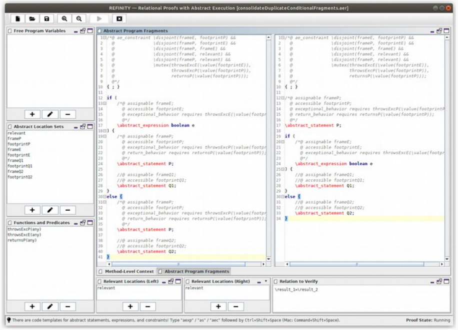
$$\rightsquigarrow \{\cancel{\mathtt{x} := t'} \,||\, \mathcal{U}_{\mathtt{P}}(\ldots, \backslash hasTo(\mathtt{x}), \ldots :\approx \{\mathtt{x} := t'\}fp)\}t$$

REFINITY and How to Prove the Correctness of Refactoring Rules

Analyzing and *Proving Refactoring Techniques* with REFINITY and Abstract Execution: Methodology

1. Create *refactoring models:* Two *abstract programs* (before / after refactoring) with minimal specification
2. Load *proof obligation* (“before refactoring $\Leftrightarrow$ after refactoring”) generated by REFINITY into KeY
3. Start *automatic proof*
    1. Proof closed $\Rightarrow$ Modeled *refactoring correct*
    2. Open goals $\Rightarrow$ Inspect proof, maybe *adapt model*

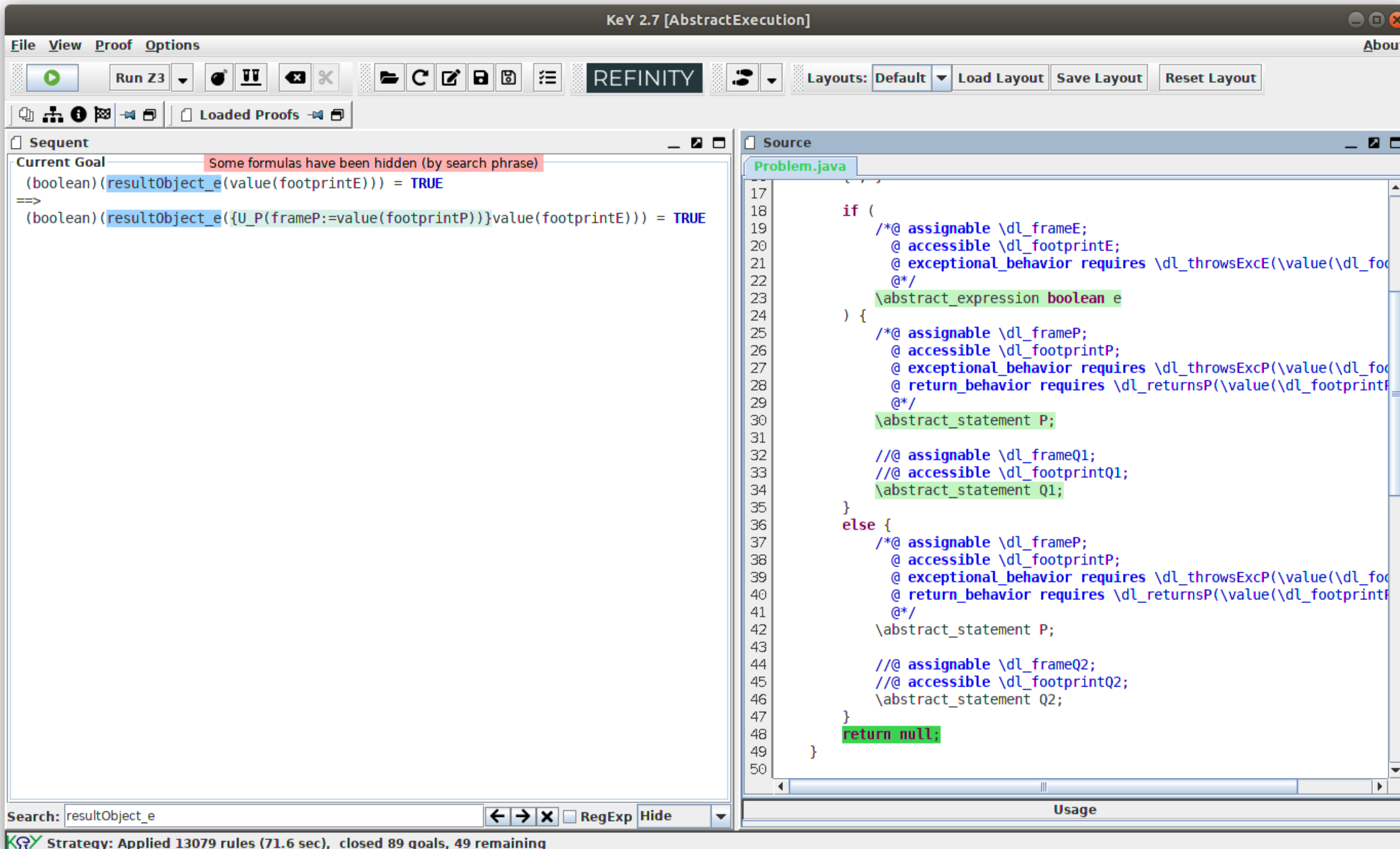
Generation of *Proof Obligations*



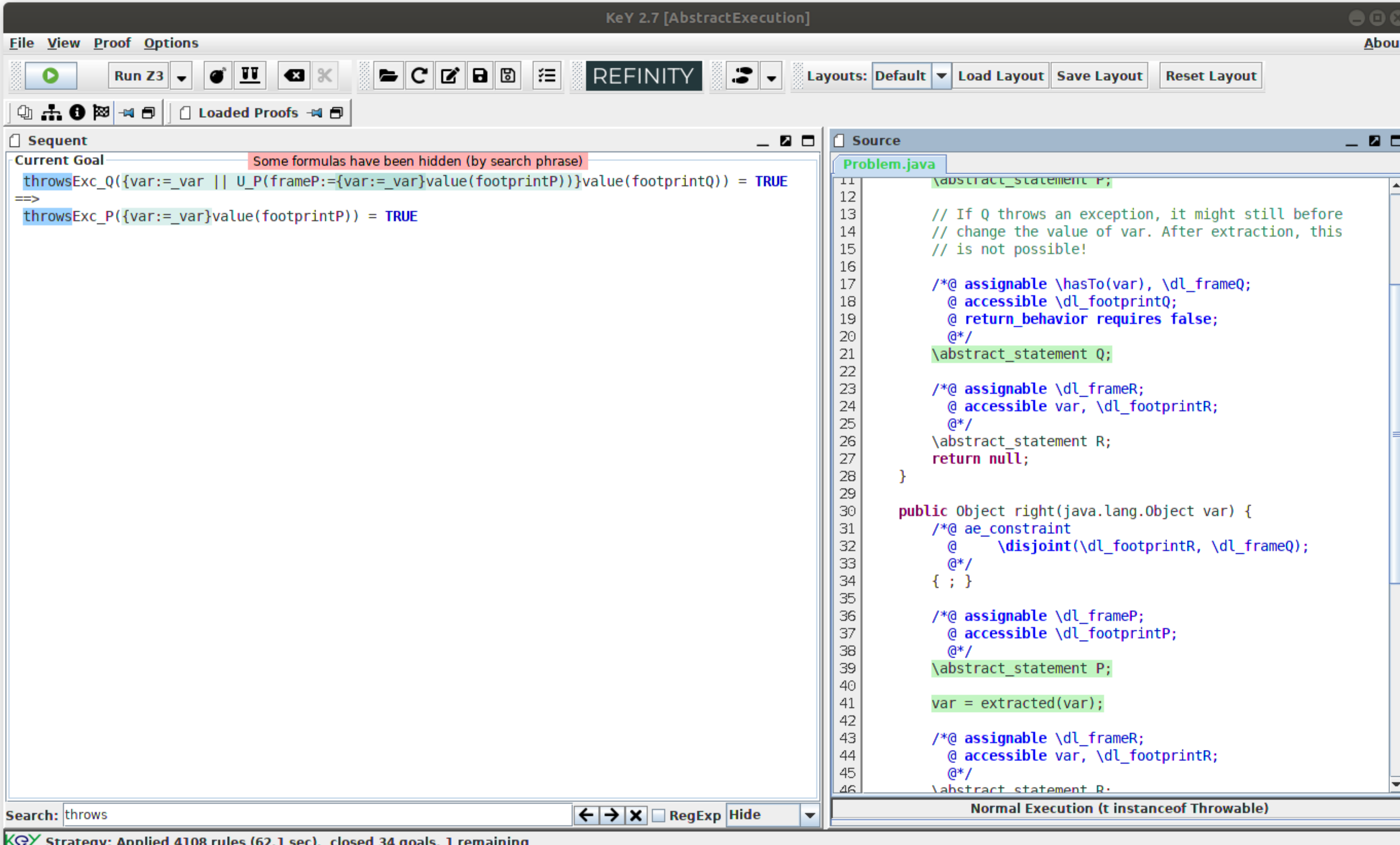
$\rightsquigarrow$

```
// ...
\problem {
     !_objUnderTest = null
   & disjoint(singletonPV(_result),relevant)
   & disjoint(singletonPV(_exc),relevant)
   & ...
   & {_result:=null||_exc:=null}
        !\<{ try {
                _result = _objUnderTest.left()@Problem;
             } catch (Throwable t) { _exc = t; }
          }\> !_P(⟨value(singletonPV(_result))⟩∘
                  ⟨value(singletonPV(_exc))⟩∘
                  ⟨value(relevant)⟩)
   & {_result:=null||_exc:=null}
       !\<{ try {
               _result = _objUnderTest.right()@Problem;
            } catch (Throwable t) { _exc = t; }
         }\> !_Q(⟨value(singletonPV(_result))⟩∘
                 ⟨value(singletonPV(_exc))⟩∘
                 ⟨value(relevant)⟩)
   -> (\exists Seq _res1;
         (\exists Seq _res2; (
             _P(_res1) & _Q(_res2) & (_res1=_res2))))
}
```

Proof Inspection: *Imprecise I/O Specifications*



Proof Inspection: *Missing Abrupt Completion Specifications*



Proving Refactoring Techniques: *Results*

- Proved correctness of *models for 8 refactorings*:
  (1) Consolidate Duplicate Conditional Fragments (four variants), (2) Decompose Conditional, (3) Extract Method, (4) Replace Exception with Test, (5) Move Statements to Callers, (6) Slide Statements, (7) Split Loop, (8) Remove Control Flag
- Elicitation of *non-trivial behavioral restrictions* not mentioned in literature for 10 of 11 studied models

It's your turn!

3rd DEMO

Your Task

Find constraints under which the a given *refactoring technique* is *correct*.

Prove correctness of an abstract program model for the refactoring, which should be *as general as possible*.

…using REFINITY

1st Suggestion: *Replace Exception with Test*

```java
z = 0;

try {
    z = 42;
    x = x / y;
} catch (ArithmeticException e) {
    x = Integer.MAX_VALUE;
}
```

```java
z = 0;

if (y != 0) {
    z = 42;
    x = x / y;
} else {
    x = Integer.MAX_VALUE;
}
```

2nd Suggestion: *Slide Statements*

```
doSomethingInteresting(arg);
int boringVar = 17;
```

```
int boringVar = 17;
doSomethingInteresting(arg);
```

If-Bored-Suggestion: *Extract Method*

```java
int x = 17;

int y = 1;
int z = x + 1;
while (z --> 1) {
    y *= z;
}

int result = y/2;
```

```java
int x = 17;
int y = factorial(x);
int result = y/2;

// …

private int factorial(int x) {
    int y = 1;
    int z = x + 1;
    while (z --> 1) {
        y *= z;
    }
    return y;
}
```

“as” + Ctrl+Shift+Space

“aexp” + Ctrl+Shift+Space

Abstract Statement: `\abstract_statement P;`
Abstract Expression: `\abstract_expression int e;`

Abstract Location Set (“footprintP”)

Frame: `//@ assignable frameP;`
Footprint: `//@ accessible footprintP;`
HasTo: `//@ assignable \hasTo(x), …;`
Frame Constraints: `//@ ae_constraint \disjoint(…, …);`

Free Program Variable (“int x”)

“aec” + Ctrl+Shift+Space

“disj” + Ctrl+Shift+Space

Abstract Predicate (“throwsExcP(any)”)

Binding Abrupt Completion of APE:

```
//@ return_behavior requires false;
/*@ exceptional_behavior
  @   requires throwsExcP(\value(footprintP)); */
```

Behavior Constraints:

“mut” + Ctrl+Shift+Space

```
/*@ ae_constraint \mutex(
  @     throwsExcE(\value(footprintE)),
  @     returnsP(\value(footprintP));
```

Attention: *No JML*

Specifying Post Condition for Abstract Statements and Expressions:

```
//@ normal_behavior ensures \result == true <==> …;
```

Specifying Result Relation:

JavaDL Syntax: `!\result_1[1] = null | \result_1[2] = 2*\result_2[3]`
Returned Results: `\result_1[0], \result_2[0]`
Thrown Exceptions: `\result_1[1], \result_2[1]`
Relevant Locations: `\result_1[2], \result_2[2], \result_1[3], …`

- *Abstract Execution:* `\abstract_statement P;`
  Automatic proofs of *abstract* programs
- *Relational verification* tool for AE REFINITY
- Specification of frame/footprint `//@ assignable frameP;`
  based on *Dynamic Frames*
- Core Idea: *2nd-order Skolemization* $\mathcal{U}_{\mathrm{P}}(\backslash \mathit{hasTo}(x), \mathit{frameP} :\approx \mathit{footprintP})$
- *Implemented* for Java in the KeY framework
- Suitable for *analyzing and proving soundness* ✓
  of *statement-level refactoring techniques*
- Several *collaborations* based on Abstract Execution
  planned or already started