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 2\textsuperscript{nd}, 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 2\textsuperscript{nd}, 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

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

```java
y += x;
if (x >= 0) {
    x = -x;
} else {
    y = -y;
}
```
Goal: *Equivalence* of Programs Before and After Refactoring

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

**UNSAFETY:**

```java
if (x >= 0) {
    y += --x;
    x = -x;
} else {
    y += --x;
    y = -y;
}
```
Automatic, Statement-Level Relational Program Verification Tool

key-project.org/REFINITY/
1st DEMO
Going \textit{Abstract} – Properties of Many Programs

\begin{verbatim}
if (expr) {
  \textcolor{green}{P}
  \textcolor{green}{Q_1}
} else { 
  \textcolor{green}{P}
  \textcolor{green}{Q_2}
}
\end{verbatim}

\begin{verbatim}
\textcolor{green}{P}
\textcolor{green}{Q_1}
\textcolor{green}{Q_2}
\end{verbatim}
## Hierarchy of Verification Approaches

<table>
<thead>
<tr>
<th>Testing</th>
<th>Show correctness of <em>one</em> program for <em>one</em> set of inputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program Proving</td>
<td>Show correctness of <em>one</em> program for <em>all</em> possible inputs</td>
</tr>
<tr>
<td><em>Abstract</em> Program Proving</td>
<td>Show correctness of <em>all</em> programs (matching a pattern) for <em>all</em> possible inputs</td>
</tr>
</tbody>
</table>

**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** \(\text{evaluation\_deterministic:}\)

\[ \forall \ c \ s t \ s t1 \ s t2, \]
\[ c \ / \ s t \ \ \ \ \ \ \ \ \ \ s t1 \rightarrow c \ / \ s t \ \ \ \ \ \ \ \ \ \ s t2 \rightarrow st1 = st2. \]

**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*!
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*
Abstract Execution

Specification of Abstract Programs +
Symbolic Execution +
Abstract State Changes

Abstract Execution
//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**

```plaintext
//@ 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*

```c
//@ 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;
if (  
  /*@ normal_behavior ensures */  
  @ \result <=>  
  @ throwsP(\value(footprintP));  
  @*/  
  \abstract_expression boolean e
 ){  
  println(“P threw Exception!”);  
} else {  
  println(“P did not throw Exception!”);  
}
2nd DEMO
Abstract Execution

*Specification* of Abstract Programs +
Symbolic *Execution* +
Abstract State Changes
Symbolic Execution of an Assignment (in JavaDL)

assignment

$$
\Gamma \vdash \{U\}\{x := e\} [\pi\omega] \varphi, \Delta \\
\Gamma \vdash \{U\}[\pi x=e; \omega] \varphi, \Delta
$$
Symbolic Execution of a Conditional Statement (in JavaDL)

IfElseSplit

\[ \Gamma, e \doteq \text{TRUE} \vdash \{\mathcal{U}\}[\pi\ p_1\ \omega]\phi, \Delta \]
\[ \Gamma, e \doteq \text{FALSE} \vdash \{\mathcal{U}\}[\pi\ p_2\ \omega]\phi, \Delta \]
\[ \Gamma \vdash \{\mathcal{U}\}[\pi\ \text{if}\ (e)\ p_1\ \text{else}\ p_2\ \omega]\phi, \Delta \]
Abstract Execution of an Abstract Statement

abstractStatement
\[ \Gamma \vdash \{ \mathcal{U} \}\{ \text{throwsExc} := \text{throwsExc}(\text{footprint}) \mid \text{exc} := \text{exc}(\text{footprint}) \mid \text{returns} := \text{returns}(\text{footprint}) \mid \text{res} := \text{res}(\text{footprint}) \} \]

\[ \left( (\text{returns} = \text{TRUE} \leftrightarrow \neg \text{throwsExc} = \text{TRUE}) \right) \land \]

\[ (\text{normal} = \text{TRUE} \leftrightarrow \neg \text{returns} = \text{TRUE} \land \neg \text{throwsExc} = \text{TRUE}) \land \]

\[ (\text{throwsExc} = \text{TRUE} \rightarrow \neg \text{exc} = \text{null}) \land \]

\[ (\text{throwsExc} = \text{TRUE} \leftrightarrow \text{excPre}) \land \]

\[ (\text{returns} = \text{TRUE} \leftrightarrow \text{returnsPre}) \} \rightarrow \]

\[ \{ \mathcal{U}_P(\text{frame} : \approx \text{footprint}) \} \]

\[ \left( (\text{throwsExc} = \text{TRUE} \rightarrow \text{excPost}) \right) \land \]

\[ (\text{returns} = \text{TRUE} \rightarrow \text{returnsPost}) \land \]

\[ (\text{normal} = \text{TRUE} \rightarrow \text{normalPost}) \} \rightarrow \]

\[ [\pi \text{ if (returns) return res;} \]

\[ \text{if (throwsExc) throw exc; } \omega \phi, \Delta \]

\[ \Gamma \vdash \{ \mathcal{U} \}[\pi \backslash \text{abstract_statement } P; \omega] \phi, \Delta \]
Abstract Execution of an Abstract Expression

\[
\begin{align*}
\text{abstractExpression} & \quad \Gamma \vdash \{\mathcal{U}\} \{\text{throwsExc} := \text{throwsExc(footprint)} \mid \\ & \quad \text{exc} := \text{exc(footprint)} \mid \text{res} := \text{res(footprint)}\} \\
& \quad ((\text{normal} \doteq \text{TRUE} \iff \neg \text{throwsExc} \doteq \text{TRUE}) \land \\
& \quad (\text{throwsExc} \doteq \text{TRUE} \Rightarrow \neg \text{exc} \doteq \text{null}) \land \\
& \quad (\text{throwsExc} \doteq \text{TRUE} \iff \text{excPre}) \} \Rightarrow \\
& \quad (\text{throwsExc} \doteq \text{TRUE} \Rightarrow \\
& \quad \{\mathcal{U}_e(\text{frame} : \approx \text{footprint})\} \\
& \quad (\text{excPost} \Rightarrow [\pi \ \text{throw} \ \text{exc} ; \ \omega] \phi) \land \\
& \quad (\neg \text{throwsExc} \doteq \text{TRUE} \Rightarrow \\
& \quad \{\mathcal{U}_e(\text{frame} : \approx \text{footprint}) \mid \text{v} := \text{res}\} \\
& \quad (\text{normalPost} \Rightarrow [\pi \ \omega] \phi), \Delta \\
\end{align*}
\]

\[
\Gamma \vdash \{\mathcal{U}\} [\pi \ \text{v=} \text{abstract_expression} \ T \ e ; \ \omega] \phi, \Delta
\]
Abstract Execution

Specification of Abstract Programs + Symbolic Execution + Abstract State Changes
We use all the existing rules for (concrete) updates...

\[
\{U\}(a := t) \rightsquigarrow a := \{U\}t
\]

\[
\{U\}f(t_1, \ldots, t_n) \rightsquigarrow f(\{U\}t_1, \ldots, \{U\}t_n)
\]

\[
\{U\}{U'}t \rightsquigarrow \{U \parallel \{U\}U'\}t
\]

\[
\vdots
\]
...and adapt the “drop update” rules.

\[
\{ \ldots \ || \ x := t' \ || \ \ldots \}t \leadsto \{ \ldots \ || \ \text{skip} \ || \ \ldots \}t
\]

if \( x \notin \text{fpv}(t) \) and \( \forall "\text{value(loc)} \in t", \text{disjoint(loc, x)} \)
...and adapt the “drop update” rules.

\[ \{ \ldots || \mathcal{U}_p(\ldots, frame_k, \ldots : \sim footprint) || \ldots \}^t \]

\[ \sim \{ \ldots || \mathcal{U}_p(\ldots, _, \ldots : \sim footprint) || \ldots \}^t \]

if "value(loc) \in t", \text{disjoint}(loc, frame_k)
...and adapt the "drop update" rules.

\[ \{ \ldots \parallel U_P(\_ , \ldots , \_ : \approx \text{footprint}) \parallel \ldots \}t \]

\[ \sim \rightarrow \{ \ldots \parallel \text{skip} \parallel \ldots \}t \]
Special Case: \( \text{"\textbackslash hasTo" Locations} \)

(1) “Outsourcing” \textit{Concrete Updates} 

\[
\{ \ldots \parallel U_P(\ldots, \text{\textbackslash hasTo}(x), \ldots : \approx fp) \parallel \ldots \} t
\]

\[
\leadsto \{ \ldots \parallel U_P(\ldots, x, \ldots : \approx fp) \parallel x := \text{anon}_P^k(fp) \parallel \ldots \} t
\]
Special Case: \(\texttt{\textbackslash hasTo}\) Locations

(2) **Stronger Simplifications**

\[
\begin{align*}
\{ x := t' \} \{ U_P(\ldots, x, \ldots :\simeq fp) \} t \\
\sim \rightarrow \{ x := t' \parallel U_P(\ldots, x, \ldots :\simeq \{ x := t' \} fp) \} t
\end{align*}
\]

\[
\begin{align*}
\{ x := t' \} \{ U_P(\ldots, \texttt{\textbackslash hasTo}(x), \ldots :\simeq fp) \} t \\
\sim \rightarrow \{ x \neq t' \parallel U_P(\ldots, \texttt{\textbackslash hasTo}(x), \ldots :\simeq \{ x := t' \} fp) \} t
\end{align*}
\]
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 ⇔ after refactoring”) generated by REFINITY into KeY

3. Start **automatic proof**
   1. Proof closed ⇒ Modeled **refactoring correct**
   2. Open goals ⇒ Inspect proof, maybe **adapt model**
Generation of **Proof Obligations**

```plaintext
// ...
\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(\langle value(singletonPV(_result))\rangle ∘
  \langle value(singletonPV(_result))\rangle ∘
  \langle value(singletonPV(_exc))\rangle ∘
  \langle value(relevant)\rangle )
& {_result:=null||_exc:=null}
  \try {
    _result = _objUnderTest.right()@Problem;
  } catch (Throwable t) { _exc = t; }
}\> !_Q(\langle value(singletonPV(_result))\rangle ∘
  \langle value(singletonPV(_result))\rangle ∘
  \langle value(singletonPV(_exc))\rangle ∘
  \langle value(relevant)\rangle )
  \Rightarrow (\exists Seq _res1;
    (\exists Seq _res2; ( _P(_res1) & _Q(_res2) & (_res1=_res2))))
```
Proof Inspection: *Imprecise I/O Specifications*
Proof Inspection: **Missing Abrupt Completion Specifications**

```java
throwsExc_0(var:=var || U_P(frameP:=(var:=var) value(footprintP))) = TRUE

throwsExc_P(var:=var) value(footprintP)) = TRUE
```

```java
public Object right(java.lang.Object var) {
    /* @ae_constraint */
    disjoint(\dl_footprintR, \dl_frameR);
    
    /* @assignable \dl_frameP; */
    accessible \dl_footprintP;
    
    \abstract_statement_P;
    
    var = extracted(var);  
}
```
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;
}
```
doSomethingInteresting(arg);
\textbf{int} \ boringVar = 17;

\textbf{int} \ boringVar = 17;
doSomethingInteresting(arg);
int x = 17;
int y = 1;
int z = x + 1;
while (z --> 1) {
  y *= z;
}
int result = y/2;

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

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

Binding Abrupt Completion of APE:
`//@ return_behavior requires false;`  
`/*@ exceptional_behavior`  
`//@ requires throwsExcP(\value(footprintP)); */`

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

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:** Automatic proofs of abstract programs

• **Relational verification** tool for AE

• Specification of frame/footprint based on *Dynamic Frames*

• Core Idea: **2nd-order Skolemization**

• **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**