Appendix

We provide additional information and explanations that had to be left out of the paper for space reasons. Numbered (sub-)sections relate to those of the paper.

2 Specifying Abstract Programs

Remark. The specification constructs listed in Table 1 include some syntactic sugar that is removed before loading a problem into our tool; details about that are provided in the implementation.

3.1 Principles of JavaDL

To help understanding the JavaDL sequent calculus a little better, we briefly discuss three exemplary calculus rules. Figure 6 shows the rules allRight, orLeft and ifElseSplit. The first two are logical first-order rules, whereas the last one is a (splitting) Symbolic Execution rule. Rule allRight processes a universal quantifier in the succedent of a JavaDL sequent by replacing the quantified variable by a fresh Skolem constant. Because a sequent $\Gamma \vdash \Delta$ is equivalent to the formula $\bigwedge \Gamma \rightarrow \bigvee \Delta$, the logical rule orLeft splits the proof into two branches for a disjunction in the antecedent of a sequent. A disjunction in the succedent (or a *con*junction in the antecedent) would be handled by simply adding the constituents as individual formulas to the succedent (antecedent). Finally, ifElseSplit also splits the proof into two branches. It is a "classical" Symbolic Execution rule: because the expression *simpleExpr* (which stands for "simple expression", an expression without side effects) might contain symbolic values, we have to evaluate the *then* and *else* branches separately. For each branch, we add as a new precondition that *simpleExpr* evaluates to TRUE resp. FALSE.

$$\textsf{allRight}\ \frac{\Gamma \vdash [x/c](\varphi), \Delta}{\Gamma \vdash \forall x; \varphi, \Delta}\ c \text{ is a fresh constant of suitable type}$$

$$\textsf{orLeft}\ \frac{\Gamma, \varphi \vdash \Delta \quad \Gamma, \psi \vdash \Delta}{\Gamma, \varphi \vee \psi \vdash \Delta}$$

ifElseSplit

$$\frac{\begin{array}{c}\Gamma, \mathit{simpleExpr} \doteq \text{TRUE} \vdash \{\mathcal{U}\}[\pi\ p_1\ \omega]\varphi, \Delta \\ \Gamma, \mathit{simpleExpr} \doteq \text{FALSE} \vdash \{\mathcal{U}\}[\pi\ p_2\ \omega]\varphi, \Delta\end{array}}{\Gamma \vdash \{\mathcal{U}\}[\pi\ \texttt{if}\ (\mathit{simpleExpr})\ p_1\ \texttt{else}\ p_2\ \omega]\varphi, \Delta}$$

Fig. 6: Some example JavaDL calculus rules

3.2 Formalization of Abstract Execution

Proof (Thm. 2). Let $\mathtt{P}^0$ be any legal instantiation of P. We create a program $\mathtt{P}^0_\dagger$, which is equivalent to $\mathtt{P}^0$, as follows: First, we transform $\mathtt{P}^0$ to a program $\mathtt{P}^{0\prime}$ without irregular termination. We define the program transformation operator transf which replaces in the input program each occurrence of

- "**return** *expr*;" with "returns=true; result=*expr*; **break** outer;",
- "**break** l_i;" with "breaksToLbl_i=**true**; **break** outer;"

and, for **break** and **continue** statements on the top level (not in the scope of a loop within the input program), each occurrence of

- "**break**;" with "breaks=true; **break** outer;",
- "**continue**;" with "continues=true; **break** outer;".

The program $\mathtt{P}^{0\prime}$ then is defined as

$$\mathtt{P}^{0\prime} = \begin{pmatrix} \texttt{outer: \{} \\ \quad \textbf{try} \texttt{ \{ transf(P}^0\texttt{) \}} \\ \quad \textbf{catch} \texttt{ (Throwable t) \{ exc=t; \}} \\ \texttt{\}} \end{pmatrix}$$

The label outer is freshly introduced. The other flags introduced by transf coincide with those in the premise of nonVoidLoopAERule. All boolean variables are initialized with **false**, and exc with null. Since the considered JAVA fragment is sequential, does not support reflection, and we additionally do not consider errors (only exceptions), $\mathtt{P}^{0\prime}$ can only terminate regularly (if it does *not* terminate, the conclusion of rule nonVoidLoopAERule is trivially valid). Note that the rule excludes labeled **continue** statements, which is why we also do not consider them here. Then, we define $\mathtt{P}^0_\dagger$ as:

$$\mathtt{P}^0_\dagger = \begin{pmatrix} \mathtt{P}^{0\prime} \\ \textbf{if} \texttt{ (returns) } \textbf{return} \texttt{ result;} \\ \textbf{if} \texttt{ (exc != } \textbf{null}\texttt{) } \textbf{throw} \texttt{ exc;} \\ \textbf{if} \texttt{ (breaks) } \textbf{break}\texttt{;} \\ \textbf{if} \texttt{ (continues) } \textbf{continue}\texttt{;} \\ \textbf{if} \texttt{ (breaksToLbl\_1) } \textbf{break}\ l_1\texttt{;} \\ \ldots \\ \textbf{if} \texttt{ (breaksToLbl\_n) } \textbf{break}\ l_n\texttt{;} \end{pmatrix}$$

The program $\mathtt{P}^0_\dagger$ is equivalent to $\mathtt{P}^0$ since, if $\mathtt{P}^0$ terminates regularly, the behavior of $\mathtt{P}^{0\prime}$ equals that of $\mathtt{P}^0$ (the **try** statement has no effect) and the added **if** statements are not entered. If irregular termination occurs, it is captured and deferred equivalently to the outside. In the following, we consider, without loss of generality, the instantiation of P in the conclusion with $\mathtt{P}^0_\dagger$ instead of $\mathtt{P}^0$.

Assume that $\texttt{P}^0$ terminates normally. Since we can assume that it respects the contract of APS `P`, all the behavior specification formulas like *returnsSpec* are equivalent to false, and the **`if`** statements in the premise are not entered. The premise is therefore, in that case, logically equivalent to the one of rule simpleAERule and the soundness argument similar to Thm. 1, except that the abstract update and path condition do not range over all locations, but exactly over the assignable and accessible locations of $\texttt{P}^0$. Again, since $\texttt{P}^0$ respects the contract of `P`, we can find a suitable legal instantiation of the premise which implies the conclusion.

If $\texttt{P}^0$ terminates irregularly, we easily find a suitable legal instantiation of the premise implying the conclusion for the breaking and continuing cases. For the returning and exceptional cases, the validity of the premise implies that the conclusion is valid for every possible returned result and thrown exception, since the variables `result` and `exc` are set to fresh Skolem constants $returns_0$ and exc_0. In particular, it is therefore valid for the concretely returned result or thrown exception. □

4 Proving the Correctness of Refactoring Techniques

We created a variant of the existing loop invariant rule based on loop scopes in JavaDL [30,32]. The new rule, depicted in Fig. 7, only applies to formulas with a special type of post condition, namely those containing the uninterpreted *Post* predicate used in AE equivalence proofs. In the part of the post condition of the "preserved & use case" where in the case that the loop continues (i.e., the loop scope index `x` is FALSE), normally only the invariant has to be shown, we additionally include the post condition (highlighted in gray), but with the second component of the *Post* predicate set to FALSE. This provides us with the possibility to only assume and show a simple invariant *Inv* containing, e.g., information necessary for showing termination, and to otherwise continue with abstract relational reasoning, thereby relating runs continuing loop execution separately from those leaving the loop.

Two other variants (Figures 8 and 9) are useful for situations where one loop has a bigger amount of iterations than another. The first one implements a general unrolling pattern as described, e.g., in [16], for harmonizing the iteration structure of two loops. The second variant realizes the unrolling pattern as displayed in Listings 6 and 7 along the *Remove Control Flag* example. It is specialized to loops where the guards consist of two conjuncts and only the first should trigger a direct break out of the loop. Both rules are parametric in a number i determining how often the body should be unrolled. Thus, they spare the harmonization of the loop iteration structure by manual code transformation.

Performance

Figures 10 and 11 visualize proof sizes and needed time for proof completion for the studied refactorings. Figure 10 also shows the numbers of APSs for all models. Proof size roughly corresponds to auto mode time; higher numbers of

loopScopeInvariantAE

$$\frac{\begin{array}{ll}\Gamma \vdash \{\mathcal{U}\}Inv, \Delta & \text{(initially valid)}\\ \Gamma \vdash \{\mathcal{U}\}\,\{\mathcal{U}_{havoc}\}\Big(Inv \rightarrow [\pi & \text{(preserved \& use case)}\\ \quad \texttt{boolean x = true;}\ \circlearrowleft_{\texttt{x}} & \\ \quad \texttt{if(}expr\texttt{)}\ \{ & \\ \quad\quad body & \\ \quad\quad \texttt{x = false;} & \\ \quad \}\ {}_{\texttt{x}}\circlearrowleft\ \omega]\big((\texttt{x} \doteq TRUE \rightarrow \varphi[Post(\texttt{result}, \text{TRUE})]) \wedge & \\ \qquad\qquad (\texttt{x} \doteq FALSE \rightarrow (Inv \wedge \varphi[Post(\texttt{result}, \text{FALSE})]\ ))\big)\Big), \Delta & \end{array}}{\Gamma \vdash \{\mathcal{U}\}[\pi\ \texttt{while(}expr\texttt{)}\ body\ \omega](\varphi[Post(\texttt{result}, \text{TRUE})]), \Delta}$$

Fig. 7: Loop Scope Invariant Rule for Abstract Execution

loopScopeInvariantAEUnrolling

$$\frac{\begin{array}{ll}\Gamma \vdash \{\mathcal{U}\}Inv, \Delta & \text{(initially valid)}\\ \Gamma \vdash \{\mathcal{U}\}\,\{\mathcal{U}_{havoc}\}\Big(Inv \rightarrow [\pi & \text{(preserved \& use case)}\\ \quad \texttt{boolean x = true;}\ \circlearrowleft_{\texttt{x}} & \\ \quad \texttt{if(}expr\texttt{)}\ \{ & \\ \quad\quad \{\ (\ \texttt{if(}expr\texttt{)}\ body\ \texttt{else break;}\ )^{i}\ \} & \\ \quad\quad \texttt{x = false;} & \\ \quad \}\ {}_{\texttt{x}}\circlearrowleft\ \omega]\big((\texttt{x} \doteq TRUE \rightarrow \varphi[Post(\texttt{result}, \text{TRUE})]) \wedge & \\ \qquad\qquad (\texttt{x} \doteq FALSE \rightarrow (Inv \wedge \varphi[Post(\texttt{result}, \text{FALSE})]))\big)\Big), \Delta & \end{array}}{\Gamma \vdash \{\mathcal{U}\}[\pi\ \texttt{while(}expr\texttt{)}\ body\ \omega](\varphi[Post(\texttt{result}, \text{TRUE})]), \Delta}\ i \geq 1$$

Fig. 8: "Unrolling" Loop Scope Invariant Rule for Abstract Execution

APSs usually lead to bigger proofs, although due to the influence of other factors (unrelated proof tree branching, loops etc.), there is no direct correspondence.

Source Code Samples

The source code of most refactoring models is shown in Figs. 12 to 17. For space reasons, we usually omit showing the JML specifications of the APSs.

Additional Implementation Remarks

In the KeY system, rules can either be programmed in JAVA as "built-in rules", or added as so-called *taclet*s, schematic, theory-specific rules written in a domain-specific language for SE rules [1, Chapter 4]. Our AE rules are written as taclets, because this helps to modularize soundness arguments. Taclets are also easier to read and to maintain than JAVA code. A typical AE taclet needs ca. 50–60 lines of text (excluding comments and empty lines). For the implementation, we had to extend the taclet language by various features like new variable conditions

loopScopeInvariantAEUnrollingSplitCond

$$\frac{\begin{array}{ll}\Gamma \vdash \{\mathcal{U}\} Inv, \Delta & \text{(initially valid)}\\ \Gamma \vdash \{\mathcal{U}\}\,\{\mathcal{U}_{havoc}\}\Big(Inv \rightarrow [\pi & \text{(preserved \& use case)}\\ \quad \texttt{boolean x = true;}\ \circlearrowleft_{\texttt{x}} & \\ \quad \texttt{if}(e_1 \texttt{\&\&} e_2)\ \{ & \\ \quad\quad \{\ (\ \texttt{if}(e_1)\ \{\ \texttt{if}(e_2)\ body\ \}\ \texttt{else break;}\ )^i\ \} & \\ \quad\quad \texttt{x = false;} & \\ \quad \}\ {}_{\texttt{x}}\circlearrowleft\ \omega]\big((\texttt{x} \doteq TRUE \rightarrow \varphi[Post(\texttt{result}, \text{TRUE})]) \wedge & \\ \qquad\qquad (\texttt{x} \doteq FALSE \rightarrow (Inv \wedge \varphi[Post(\texttt{result}, \text{FALSE})]))\big)\Big), \Delta & \end{array}}{\Gamma \vdash \{\mathcal{U}\}[\pi\ \texttt{while}(e_1 \texttt{\&\&} e_2)\ body\ \omega](\varphi[Post(\texttt{result}, \text{TRUE})]), \Delta}\ i \geq 1$$

Fig. 9: Special "Unrolling" Loop Scope Invariant Rule for Abstract Execution, Harmonizing Iteration Structure for a Compound Loop Condition

and term transformers (both allow to inline small portions of JAVA code) and a new loop construct for realizing the "$\cdots$" in rules like nonVoidLoopAERule.

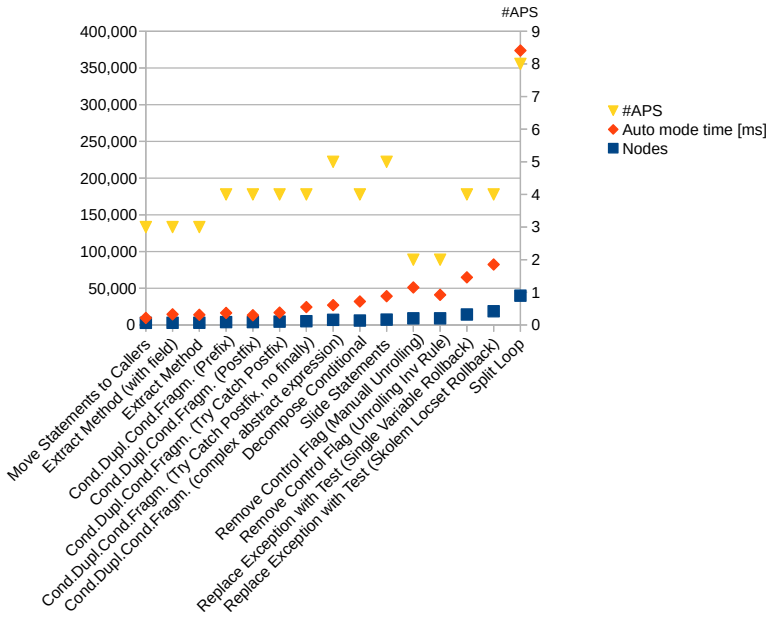

Fig. 10: Performance: Numbers of nodes and APSs and auto mode time

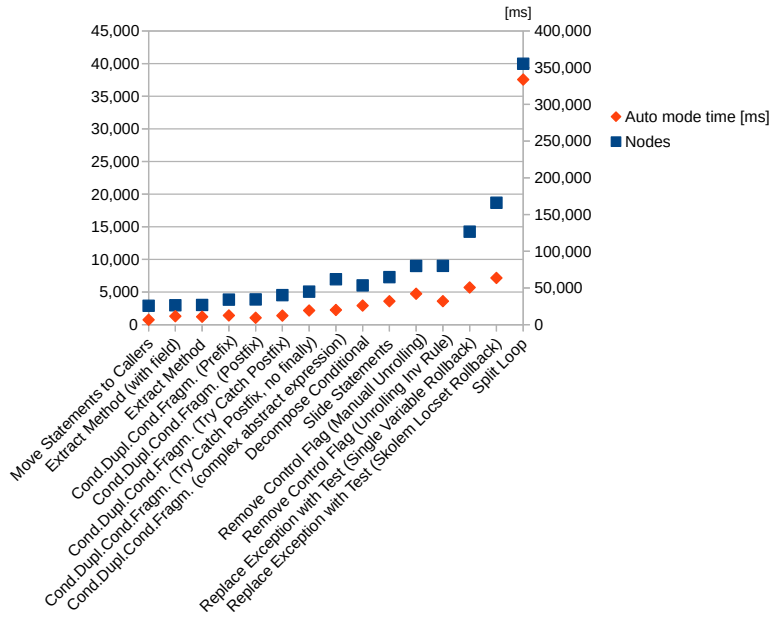

Fig. 11: Performance: Numbers of nodes and auto mode time

Listing 6: Remove Control Flag, Original

```
while (!done && i < threshold) {


  if (condition) {
    abstract_statement Body;
    done = true;
  }

  i++;









}
```

Listing 7: Remove Control Flag, After Unrolling

```
while (!done && i < threshold) {
  if (!done) {
    if (i < threshold) {
      if (condition) {
        abstract_statement Body;
        done = true;
      }
    }
    i++;
  } else break;
  if (!done) {
    if (i < threshold) {
      if (condition) {
        abstract_statement Body;
        done = true;
      }
    }
    i++;
  } else break;
}
```

Listing 8: Before

```
abstract_statement Guard;
if (guard) {
  abstract_statement Then;
} else {
  abstract_statement Else;
}

abstract_statement Post;
return result;
```

Listing 9: After

```
guard = mGuard(result);
if (guard) {
  tmp = mThen(result, guard, tmp);
} else {
  tmp = mElse(result, guard, tmp);
}

abstract_statement Post;
return result;

//...

//@ declares final(args);
private Object mThen(Object result,
    boolean guard, Object tmp) {
  abstract_statement Then;
  return tmp;
}
```

Fig. 12: The *Decompose Conditional* Refactoring Technique

Listing 10: Before

```
abstract_statement P;
abstract_statement Q;
abstract_statement R;
return result;
```

Listing 11: After

```
abstract_statement P;
tmp = extracted(res, tmp);
abstract_statement R;
return result;

// ...

//@ declares final(locals(P)), final(args);
public Object extracted(Object res,
    Object tmp) {
  abstract_statement Q;
  return tmp;
}
```

Fig. 13: The *Extract Method* Refactoring Technique

Listing 12: Before

```
result = called();

abstract_statement B;

return result;

// ...

//@ declares final(args);
public Object called() {
  abstract_statement C;
  abstract_statement A;
  return result;
}
```

Listing 13: After

```
result = called();

abstract_statement A;
abstract_statement B;

return result;

// ...

//@ declares final(args);
public Object called() {
  abstract_statement C;
  return result;
}
```

Fig. 14: The *Move Statements to Callers* Refactoring Technique

Listing 14: Before

```
abstract_statement Init;

try {
  //@ exceptional_behavior
  //@     requires throwsExc;
  abstract_statement Normal;
} catch (Throwable t) {
  abstract_statement Rollback;
  abstract_statement Exceptional;
}

abstract_statement After;
return result;
```

Listing 15: After

```
abstract_statement Init;

if (!throwsExc) {
  //@ exceptional_behavior
  //@     requires throwsExc;
  abstract_statement Normal;
} else {
  abstract_statement Rollback;
  abstract_statement Exceptional;
}

abstract_statement After;
return result;
```

Fig. 15: The *Replace Exception With Test* Refactoring Technique

Listing 16: Before

```
abstract_statement A;
abstract_statement B;
abstract_statement C;
abstract_statement D;
abstract_statement E;
```

Listing 17: After

```
abstract_statement A;
abstract_statement D;
abstract_statement C;
abstract_statement B;
abstract_statement E;
```

Fig. 16: The *Slide Statements* Refactoring Technique

Listing 18: Before

```
abstract_statement PreProc;

abstract_statement InitA;
abstract_statement InitB;

for (int i=0;
    i < loopArgs.length; i++) {
  o = loopArgs[i];
  //@ assignable outA;
  abstract_statement LoopBodyA;
  //@ assignable outB;
  abstract_statement LoopBodyB;
}

//@ assignable result;
//@ accessible outA;
abstract_statement PostProcA;
//@ assignable result;
//@ accessible outB;
abstract_statement PostProcB;




abstract_statement PostProc;
return result;
```

Listing 19: After

```
abstract_statement PreProc;

abstract_statement InitA;
for (int i=0;
    i < loopArgs.length; i++) {
  o = loopArgs[i];
  //@ assignable outB;
  abstract_statement LoopBodyB;
}
//@ assignable result;
//@ accessible outA;
abstract_statement PostProcA;

abstract_statement InitB;
for (int i=0;
    i < loopArgs.length; i++) {
  o = loopArgs[i];
  //@ assignable outB;
  abstract_statement LoopBodyB;
}
//@ assignable result;
//@ accessible outB;
abstract_statement PostProcB;

abstract_statement PostProc;
return result;
```

Fig. 17: The *Split Loop* Refactoring