REFINITY to Model and Prove Program Transformation Rules*

Dominic Steinhöfel[0000−0003−4439−7129]

TU Darmstadt, Dept. of Computer Science, Darmstadt, Germany
steinhoefel@cs.tu-darmstadt.de

Abstract. REFINITY is a workbench for modeling statement-level transformation rules on JAVA programs with the aim to formally verify their correctness. It is based on Abstract Execution, a verification framework for abstract programs with a high degree of proof automation, and interfaces with the KeY program prover. We describe the user interface and functionality of REFINITY, and illustrate its capabilities along the application to proving conditional correctness of a code refactoring rule.

1 Introduction

Systematic program transformations are ubiquitous in modern program development. Which programmer has never used a refactoring technique like method extraction, not to mention a compiler? Further, less mundane transformation-based approaches comprise optimization, incremental program development which is "correct-by-construction" [9] or program synthesis from a high-level specification. The latter two are examples for domains where correctness is built into the problem statement; yet, the question of correctness is also relevant, and has been approached, in other areas [5, 10, 12–14, 17]. Mechanized correctness arguments about code transformations are frequently conducted in interactive environments like Isabelle or Coq. An example is the work on verified compilers [12, 17]. While this approach permits expressing a wide range of properties, substantial effort has to be invested to prove them *manually* by writing proof scripts. Existing approaches to prove transformations *automatically*, on the other hand, are tailored to *specific* applications (such as regression verification [6], "peephole" optimizations [13] or symbolic execution rules [2]) and lack expressiveness.

Proving the correctness of program transformation rules is a *second-order* property involving quantification over programs. It can be understood as a *relational verification* [3] problem over *schematic* programs. For example, the schematic programs "$p\ \ q$" and "$q\ \ p$" (where p, q represent arbitrary statements) describe a transformation swapping two statements. It is correct if we can prove, as usual under additional assumptions, that all instances of those two programs behave equivalently. Recently, *Abstract Execution (AE)* has been proposed [15, 16], a technique for proving properties of *abstract* (i.e., schematic)

---

* This work was funded by the Hessian LOEWE initiative within the Software-Factory 4.0 project. **The final authenticated version is available online at https://doi.org/10.1007/978-3-030-64437-6_16.**

programs by symbolic *execution*. AE bridges the gap between expressiveness and automation by restricting the class of addressable problems to a (reasonably big) subset—*universal* properties of program *behavior*—while at the same time offering a versatile specification framework. Many transformations, even loop transformations, can be proven *fully automatically* using AE, including the example regarded in this paper and the complete refactoring case study of [15].[1]

AE is implemented on top of KeY [1], a deductive verification framework for Java programs based on symbolic execution. AE extends the Java language by *Abstract Statements (ASs)* "**\abstract_statement** `P;`", and *Abstract Expressions (AExps)* "**\abstract_expression** T `e;`", where `P` and `e` are the *identifiers* of the abstract statement / expression, and T is the type of the abstract expression `e`. Programs containing ASs or AExps are called *abstract programs*.

In this paper, we present REFINITY, a graphical tool for modeling statement-level program transformation rules based on AE. REFINITY supports the specification of abstract programs representing inputs and outputs of transformation rules and of relational pre- and postconditions defining the proof objective. It automatically generates non-trivial proof obligations for the KeY prover and initiates an automatic proof attempt. Thus, it significantly eases the workflow of specifying, proving and refining transformation models. We describe how to use REFINITY to model and prove statement-based refactoring techniques.

Related Work REFINITY is, to our knowledge, the only existing relational verification tool for *abstract* programs, and, thus, for general source-to-source program transformations. Therefore, we can only compare our work to existing tools for verification of *concrete* programs. LLRêve [8], for instance, is a tool for automatically proving the equivalence of two C programs. SymDiff [11] is a "differential program verifier." Both operate on intermediate languages (LLVM IR and Boogie) and use advanced techniques for automatically relating loops and recursive procedures. REFINITY relies on manually specified loop invariants and method contracts, and therefore requires more interaction for concrete code. However, loop invariants in *abstract contexts* can frequently be specified *generically* [15].

Organization Subsequently, we describe REFINITY's specification language for abstract programs along an illustrating example. Sect. 3 shows how to model and prove this example transformation in REFINITY. Sect. 4 concludes the paper.

REFINITY can be downloaded at `key-project.org/REFINITY/`, where we also publish continuously updated documentation material. Additional support can be obtained via email to the author of this paper, or via the channels mentioned at `key-project.org/getting-started/`. Furthermore, most GUI elements of the tool provide tooltips with brief help texts.

2 Specifying Abstract Programs

We explain the most relevant elements of REFINITY's specification language for abstract programs along a code refactoring rule. REFINITY is a frontend for AE

[1] Generally, proofs may require user interaction, especially when relying on incomplete theories like first-order arithmetic.

Listing 1: Input Program

```
try {
  TryStmt // throws exc. if cond holds
} catch (Throwable t) {
  CatchStmt
}
```

Listing 2: Output Program

```
if (!cond) {
  TryStmt
} else {
  CatchStmt
}
```

Fig. 1: The *Replace Exception with Test* Refactoring Schema

Listing 3: Input Program

```
try {
  z = 42;
  y = z / x;
} catch (Throwable t) {
  y = z; // <- rollback
  flag = true;
}
```

Listing 4: Output Program

```
if (x++ != 0) { // side effect in condition
  z = 42;
  y = z / x;
} else {
  y = z; // rollback incomplete + depends on TryStmt
  flag = true;
}
```

Fig. 2: Examples for Violated Constraints (*Replace Exception with Test*)

and uses its specification framework. Our aim here is not to provide a *complete* introduction to the AE framework, for which we refer to [15].

Refactoring is the process of changing code in a way that does not alter its external behavior, yet improves its internal structure [4]. The *Replace Exception with Test (REwT)* refactoring proposes to introduce a check for a condition causing an exception when it is reasonable to expect that the condition can be checked. A good example is a division of two numbers put into a **`try`**–**`catch`** block since an `ArithmeticException` is raised if the divisor is zero. Figure 1 visualizes this schema. *REwT* is a good example since it is generally *unsafe* due to a subtlety: If `TryStmt` changes relevant parts of the state before throwing an exception, the programs before and after the refactoring behave differently. Consider, e.g., an instantiation of `TryStmt` with "`z = 42; y = z / x;`": If `x` is 0, and the value of `z` is not changed by `CatchStmt`, the final value of `z` is 42 before the transformation, but equals the original value after.

One can create a provably correct model of *REwT* by demanding a statement `Rollback` before `CatchStmt` "resetting" locations changed by `TryStmt`. For the example, we could choose "`x=0; z=0;`" for `Rollback`. Note that the assigned rollback values must not depend on locations changed by `TryStmt`.

In the following, we call the locations that may be changed by abstract statements or expressions their *frame*, and the locations they may read their *footprint*. We have to encode the following constraints into the refactoring model: (1) `TryStmt` throws an exception iff *cond* holds, (2) *cond* has no side effects, (3) `Rollback` *must* assign the whole frame of `TryStmt`, and (4) the frame of `TryStmt` and the footprint of `Rollback` must be disjoint. Figure 2 shows a "non-legal" example instantiation where Constraints (2) to (4) are violated.

Listing 5: Input Model

```
1  /*@ ae_constraint \disjoint(
2    @   footprintRollback, frameTry);
3    @*/
4
5  try {
6
7
8
9
10
11
12
13
14
15   /*@ assignable frameTry;
16     @ accessible footprintTry;
17     @ exceptional_behavior requires
18     @   throwsExcTryStmt(
19     @     \value(footprintTry)); */
20   \abstract_statement TryStmt;
21 } catch (Throwable t) {
22   /*@ assignable \hasTo(frameTry);
23     @ accessible footprintRollback;
24     @*/
25   \abstract_statement Rollback;
26
27   /*@ assignable frameCatch;
28     @ accessible footprintCatch;
29     @*/
30   \abstract_statement CatchStmt;
31 }
```

Listing 6: Output Model

```
/*@ ae_constraint \disjoint(
  @   footprintRollback, frameTry);
  @*/

if (
  /*@ assignable \nothing;
    @ accessible footprintTry;
    @ normal_behavior ensures \result
    @   <==> !throwsExcTryStmt(
    @          \value(footprintTry));
    @ exceptional_behavior
    @   requires false; @*/
  \abstract_expression boolean cond
) {
  /*@ assignable frameTry;
    @ accessible footprintTry;
    @ exceptional_behavior requires
    @   throwsExcTryStmt(
    @     \value(footprintTry)); */
  \abstract_statement TryStmt;
} else {
  /*@ assignable \hasTo(frameTry);
    @ accessible footprintRollback;
    @*/
  \abstract_statement Rollback;

  /*@ assignable frameCatch;
    @ accessible footprintCatch;
    @*/
  \abstract_statement CatchStmt;
}
```

Fig. 3: Abstract Program Model for *Replace Exception with Test*

To impose constraints on the frames and footprints of abstract elements, we have to define which locations ASs and AExps may write and read. However, no additional constraints than the mentioned ones should be enforced: Frames and footprints should match to *all* programs satisfying Constraints (1) to (4). We achieve this by using abstract, set-valued specification variables inspired by the theory of *dynamic frames* [7]. Concretely, we introduce constants *frameTry*/*footprintTry*, *footprintRollback*, and *frameCatch*/*footprintCatch*, each representing an unknown set of concrete program variables or heap locations.

The complete abstract program model for *Replace Exception with Test* is shown in Fig. 3. Constraints on ASs and AExps are imposed using specification comments starting with "`@`". In lines 6/7, 15/16, 22/23, and 27/28, we assign the newly introduced abstract location set variables to the abstract program elements, where the keyword **`assignable`** specifies a frame, and **`accessible`** a footprint of an AS or an AExp. To realize constraint (3), we put a "**`\hasTo`**" specifier around the frame specification of `Rollback`. Without **`\hasTo`**, frame and footprint specifications are only upper bounds.

Constraint (1) is implemented by specifying a precondition on abrupt completion due to a thrown exception for `TryStmt` in lines 17–19. The specification language keyphrase used is "**`exceptional_behavior requires`**". There are two things to explain: i) The symbol *throwsExcTryStmt* is a new abstract predicate introduced for specification purposes, and ii) the term "**`\value`**`(`*footprintTry*`)`" represents the value of the location set *footprintTry* at this point in the program: The *locations* represented by *footprintTry* do not change during program execution, while their *values* can change. It remains to specify that *cond* evaluates according to the negated value of the predicate *throwsExcTryStmt*. In lines 8–10, we constrain the expression's value (represented by **`\result`**) accordingly. The specification keyphrase "**`normal_behavior ensures`**" is used to declare a *functional postcondition* on the normal completion behavior of `cond`.

For constraint (2) (*cond* is side effect-free), it suffices to specify that the frame of *cond* is empty ("**`assignable \nothing`**", line 6) and that it throws an exception iff "**`false`**" holds (lines 11&12)—i.e., never.

Finally, the disjointness of the frame of `TryStmt` and footprint of `Rollback` (Constraint (4)) is imposed on instantiations of the model by lines 1–3. The keyword "**`ae_constraint`**" initiates the declaration of a constraint. Apart from **`\disjoint`**, also other relations, like **`\subset`**, are supported.

This example covers all essential specification language features. We did not cover advanced features like abstract functions (similar to abstract predicates, but non-boolean), indexed abstract location set families (useful for involved loop transformations), and mutual exclusion of abrupt completion behavior (using the "**`\mutex`**" keyword in **`ae_constraint`**s). See [15] for a full account.

Expressiveness REFINITY addresses *statement-level* transformation rules and is additionally limited to *universal*, *behavioral* properties supported by AE. Transformations *above* statement level, e.g., moving a field, cannot be expressed. The same holds for *structural* properties which cannot be written using a fixed abstract program scaffold with only "behavioral holes." An example is a property addressing all statements with *at most* three loops: This is not expressible, since any AS with non-empty semantics represents statements with an arbitrary number of loops. Statements with *at least* three loops *are* in scope, since one can write an abstract program with three loops of arbitrary guards and bodies.

In the following section, we demonstrate how REFINITY can be used to model and prove program transformation rules such as *Replace Exception with Test*.

3 REFINITY in Action

Fig. 4 shows the abstract program model for *Replace Exception with Test* in the REFINITY GUI. The abstract program fragments representing input and output of the transformation are written to the two text fields at marker (1). Field (2) contains free program variables which can be referred to in the input and output model without declarations; we do not need this feature in our example. In the compartment labeled (3), we define abstract location set specification variables used in the model, i.e., *frameTry*/*footprintTry*, *footprintRollback*, and *frameCatch*/*footprintCatch*. REFINITY models include as default an additional

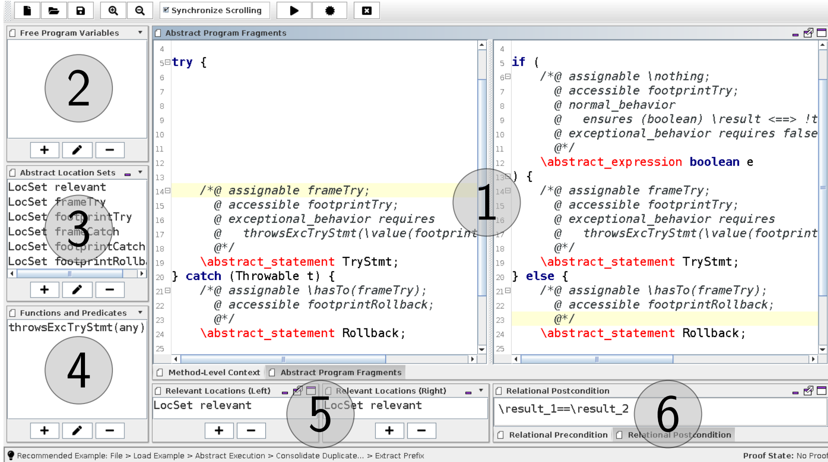

Fig. 4: The REFINITY Window

location set "*relevant*" representing all relevant locations. If we do not impose further constraints, e.g., exclude some locations from *relevant*, correctness has to be proven under the assumption that all locations are in this set. The sort for abstract location sets is "LocSet". The abstract predicate *throwsExcTryStmt* is declared in input field ④. The argument sort "any" in the declaration is a super type of all types. We use "any" since we pass the value of an abstract location set to the predicate which can be instantiated to any type.

Fields ⑤ and ⑥ specify global assumptions and proof objectives for the model. The effects of the abstract program fragments specified in field ① are recorded in two *sequences* `\result_1` and `\result_2` for the input and output model. Their elements can be accessed using standard array syntax, e.g., `\result_1[0]`. If an abstract program completed due to a **return** of a value, position 0 in the sequence contains the returned value. Likewise, if it completed due to a thrown exception, the exception object is stored at position 1. Starting from position 2, the final values of "relevant locations" declared in fields ⑤ (in the order defined there) are stored. In the example, the abstract set *relevant* is the only relevant location set, which is also the REFINITY default. The standard postcondition, which we see in field ⑥, is `\result_1==\result_2`. Without constraints about *relevant*, this specifies that returned values, thrown exceptions, and the whole memory after termination have to be identical. More fine-grained postconditions can also be specified: e.g., when an integer variable is registered as first relevant location, "`\result_1[2]>2*\result_2[2]`" is admissible.

The global "Relational Precondition" (⑥) has access to the initial values of free program variables (field ②) and abstract location sets (field ③). For the example, we did not specify a global precondition.

A model can be saved in REFINITY's XML-based format using 💾. Pressing ▶ transforms the model into a KeY proof obligation and starts the automatic

proof search. If KeY reports success, the specified model is correct. Saved proof certificates can be validated against the loaded model using the ✹ button. Proofs of correctly specified refactorings *without loops* usually take between 30 seconds and two minutes; for loop transformations, three minutes and more are possible. During development of a new model, KeY will usually finish unsuccessfully, leaving one or more proof goals open. In rare cases and for highly complicated models, the reason could be that KeY needs more time or is not able to close the proof although the model is valid—we hit a *prover incapacity*. In the latter case, one can try to close the proof by interacting with the prover. More likely, though, are problems in the model. Inspecting the open goals provides information on how to refine the model to make it sound. Possible refinements include

(1) declaring the disjointness of abstract location sets,
(2) imposing mutual exclusion on abrupt completion behavior,
(3) declaring a functional postcondition for ASs or AExps, and
(4) refining the *relational* postcondition or
(5) adding a relational *pre*condition.

The proof obligation REFINITY generates for KeY consists of a Java class with two methods `left(...)` and `right(...)` containing the abstract program fragments, and of a problem description file containing proof strategy settings, declarations of variable, function, predicate, and abstract location set symbols and the proof goal (expressed in KeY's program logic "Java Dynamic Logic" [1]). The proof goal for *Replace Exception with Test* in concrete syntax spans 36 lines. In a condensed representation, it has the form

$$
\begin{array}{ll}
 & \{\texttt{_result} := \mathit{null} \,||\, \texttt{_exc} := \mathit{null}\} \\
 & \quad \neg\langle \textbf{try}\ \texttt{\{ _result=obj.left()@Problem; \}} \\
 & \qquad \textbf{catch}\ \texttt{(Throwable t) \{ exc=t; \}}\rangle \\
 & \qquad\quad \neg P(\texttt{_result}, \texttt{_exc}, \mathit{value}(\mathit{relevant})) \\
 & \wedge\ \{\texttt{_result} := \mathit{null} \,||\, \texttt{_exc} := \mathit{null}\} \\
 & \quad \neg\langle \textbf{try}\ \texttt{\{ _result=obj.right()@Problem; \}} \\
 & \qquad \textbf{catch}\ \texttt{(Throwable t) \{ exc=t; \}}\rangle \\
 & \qquad\quad \neg Q(\texttt{_result}, \texttt{_exc}, \mathit{value}(\mathit{relevant})) \wedge \mathit{Pre} \wedge \cdots \\
\vdash & \exists\ \mathit{Seq}\ s_1, s_2;\ (P(s_1) \wedge Q(s_2) \wedge \mathit{Post}(s_1, s_2))
\end{array}
$$

where `obj` is the object under test, *Pre* and *Post* are the global precondition and relational postcondition, and P and Q are fresh predicates.

REFINITY spares the user from having to deal with such technicalities, simplifying the modeling process. It automatically creates the mentioned files, starts a KeY proof with reasonable presets, and displays proof status information in its status bar. Additionally, it supports syntactic extensions unsupported by KeY.

4 Conclusion

In this paper, we presented REFINITY, a graphical workbench for modeling and proving Java program transformation rules based on Abstract Execution,

a verification framework for abstract programs. We demonstrated how to use REFINITY by showing how to specify and prove correct a refactoring rule with a subtle snag. This builds on previous work, where "vanilla" AE has been used to prove the correctness of several statement-based refactoring rules [16]. REFINITY significantly eases the modeling process, making AE more accessible. For the future, we plan to further increase REFINITY's usability and apply it to different types of program transformations than refactoring rules.

References

1. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M. (eds.): Deductive Software Verification – The KeY Book, LNCS, vol. 10001. Springer (2016)
2. Ahrendt, W., Roth, A., Sasse, R.: Automatic Validation of Transformation Rules for Java Verification Against a Rewriting Semantics. In: Sutcliffe, G., Voronkov, A. (eds.) Proc. 12th LPAR. LNCS, vol. 3835, pp. 412–426. Springer (2005)
3. Beckert, B., Ulbrich, M.: Trends in Relational Program Verification. In: Principled Software Development - Essays Dedicated to Arnd Poetzsch-Heffter on the Occasion of his 60th Birthday. pp. 41–58 (2018)
4. Fowler, M.: Refactoring: Improving the Design of Existing Code. Object Technology Series, Addison-Wesley (Jun 1999)
5. Garrido, A., Meseguer, J.: Formal Specification and Verification of Java Refactorings. In: Proc. 6th SCAM. pp. 165–174. IEEE Computer Society (2006)
6. Godlin, B., Strichman, O.: Regression Verification: Proving the Equivalence of Similar Programs. Softw. Test., Verif. Reliab. **23**(3), 241–258 (2013)
7. Kassios, I.T.: The Dynamic Frames Theory. Formal Asp. Comput. **23**(3) (2011)
8. Kiefer, M., Klebanov, V., Ulbrich, M.: Relational Program Reasoning Using Compiler IR - Combining Static Verification and Dynamic Analysis. J. Autom. Reasoning **60**(3), 337–363 (2018)
9. Kourie, D.G., Watson, B.W.: The Correctness-by-Construction Approach to Programming. Springer (2012)
10. Kundu, S., Tatlock, Z., Lerner, S.: Proving Optimizations Correct Using Parameterized Program Equivalence. In: Proc. PLDI 2009. pp. 327–337 (2009)
11. Lahiri, S.K., Hawblitzel, C., Kawaguchi, M., Rebêlo, H.: SYMDIFF: A Language-Agnostic Semantic Diff Tool for Imperative Programs. In: Madhusudan, P., Seshia, S.A. (eds.) Proc. 24th CAV. LNCS, vol. 7358, pp. 712–717. Springer (2012)
12. Leroy, X.: Formal Verification of a Realistic Compiler. Communications of the ACM **52**(7), 107–115 (2009)
13. Lopes, N.P., Menendez, D., Nagarakatte, S., Regehr, J.: Practical Verification of Peephole Optimizations with Alive. Commun. ACM **61**(2), 84–91 (2018)
14. Srivastava, S., Gulwani, S., Foster, J.S.: From Program Verification to Program Synthesis. In: Proc. 37th POPL. pp. 313–326 (2010)
15. Steinhöfel, D.: Abstract Execution: Automatically Proving Infinitely Many Programs. Ph.D. thesis, TU Darmstadt, Dept. of Computer Science, Darmstadt, Germany (2020), `http://tuprints.ulb.tu-darmstadt.de/8540/`
16. Steinhöfel, D., Hähnle, R.: Abstract Execution. In: Proc. Third World Congress on Formal Methods - The Next 30 Years, (FM). pp. 319–336 (2019)
17. Tan, Y.K., Myreen, M.O., Kumar, R., Fox, A., Owens, S., Norrish, M.: A New Verified Compiler Backend for CakeML. In: Proc. 21st ICFP. ACM (2016)