A Notion of Equivalence for Refactorings with Abstract Execution

Ole Jørgen Abusdal[1], Eduard Kamburjan[2], Violet Ka I Pun[1], and Volker Stolz[1]

[1] Western Norway University of Applied Sciences, Norway
{ojab,vpu,vsto}@hvl.no
[2] University of Oslo, Norway
eduard@ifi.uio.no

Abstract. Relational verification through dynamic logic is a promising approach for verifying object oriented programs. Recent advances from symbolic to abstract executions have enabled reasoning about incomplete/abstract versions of such programs. This has proven fruitful in the exploration of correctness of refactorings primarily related to code blocks in Java. In this paper we explore further types of equivalent transformations and refactorings and discuss the challenges that still need to be overcome for full round-trip correctness of refactorings in object-oriented languages.

1 Introduction

Refactoring is a fundamental activity in software engineering to reorganize code to improve its structure, e.g., to simplify maintenance, while preserving its observable behavior of the program. A refactoring can be defined as a pattern it matches on, and a subsequent program transformation on the matched part. To ensure that the transformed program indeed has the same observable behavior one can either compare the transformed program with the original or reason about the program transformation itself.

Relational verification through dynamic logic is a promising approach to verification of refactoring patterns in object oriented programs. Recent advances from symbolic to Abstract Execution (AE) [30] have enabled reasoning about incomplete/abstract versions of such programs. This has proven fruitful in the exploration of correctness of refactorings primarily related to code blocks in Java: AE introduces abstract statements (and expressions), which act as named and specified placeholders for statement-sequences of the host language. A refactoring proof is a relational verification proof that compares two programs which can have abstract program elements. Consider relating

```
if (E_boolean) {S1;} else {S2;} return;
```

and

```
if (!E_boolean) {S2; return;} S1; return;
```

where $\texttt{E}_{boolean}$ is an arbitrary boolean expression, $\texttt{S}_1$ and $\texttt{S}_2$ correspond to arbitrary statements. One of the programs represents the schema of the code before the refactoring, and the other one the code afterwards, with respect to some relational post-condition that defines the notion of *program equivalence*. This reasoning about abstract programs is exactly the aforementioned reasoning about the program transformation behind the refactoring.

Despite the promising results due to AE, it has only been applied to *statement-level* refactorings that change the body of a single method. However, refactorings are not limited to single methods or statement-blocks, but may also restructure classes and data structures [14]. In this paper we investigate the role of AE for verification of refactorings beyond statement-level.

Challenges. More complex refactorings require more elaborate specification and verification techniques for relational verification. The reason is that the programs surrounding the abstract statements, as well as the notion of equivalence, become more involved. This holds for both structure and behaviour. Following the general approach of AE, REFINITY [30], the tool supporting verification on top of the KeY system, an automated theorem prover for Java [29], has been primarily designed to verify the correctness of refactorings that are based on moving code within a method, or on extracting some statements into their own method (i.e., the EXTRACT METHOD refactoring [15, p. 106]).

To investigate the use of AE for refactorings on the class-level, we investigate the HIDE DELEGATE refactoring [15, p. 189] that moves code between classes and show how it can be encoded in REFINITY. As for specification, we discuss the interpretation of *equivalence* from the perspective of the user and how to encode this — for example, if the surrounding programs throw exceptions, under which conditions are the exceptions considered equivalent? The equivalence interpretation goes beyond exceptions, but touches on a fundamental problem: there are several possible choices for when newly created objects (and exceptions, which are objects in Java) are considered to be equal in relational verification using dynamic logic. This was first investigated by Beckert et al. [7] and we discuss alternatives here.

For exceptions and object creation, we describe several possibilities when newly created objects (resp. thrown exceptions) are considered equal and how this information can be used in dynamic logic proofs. The different possibilities are implemented as multiple (sets of) rules, from which the developer chooses the one corresponding to his assumptions on object allocations – this choice does not have to be encoded explicitly in the relational post-condition. This reduces the size of the required specification, which is an advantage, since it is a notorious bottleneck in formal verification [6,17].

Furthermore, we discuss the necessary extensions needed to prove equivalent behaviour where one data structure is replaced by another, e.g., an array or any primitive type by a class. Here, the main challenge lies in the encoding that the structures are used correctly throughout the execution. This could, for example, be handled by coupled invariants [8], whose connection to AE is yet unclear. Lastly, we discuss the challenge to apply AE to novel specification approaches

for *traces* which aim to simplify the specification of temporal properties for expressive properties, but whose use for relational verification is unexplored.

Contributions and Structure. Our contributions include an investigation into the necessary side conditions to be able to proof of the correctness of the Extract Local Variable- and Hide Delegate refactorings, extending the collection of proven refactorings. The latter is a refactoring beyond code motion within a method and highlights the interaction of AE with general relational verification challenges. Then, we discuss possible extensions that would be required to address further refactorings with AE.

We first describe AE and relational verification using the Extract Local Variable refactoring in Section 2, before we show necessary conditions for the Hide Delegate refactoring to be correct and then discuss the challenges for AE in Section 3. Section 4 proposes future directions for improvements in REFINITY and AE which most likely require major development effort. In Section 5, we discuss our results. We discuss the related work in Section 6, and lastly conclude in Section 7.

2 Preliminaries

In this section, we will first briefly describe AE and how it extends symbolic execution. Then, we will show how AE can be used to prove refactoring correctness with an example.

2.1 Abstract Execution

Succinctly, as stated by Steinhöfel in their Ph.D. thesis, "Abstract Execution" denotes the idea to process abstract programs by symbolic execution (SE) [28]. SE [2,33] abstracts concrete execution by means of symbolic representations of language runtime state in place of concrete machine representations of such artefacts. Thus, a store, a program counter, values, and so on, all have a symbolic representation in SE. Branching points, such as encountered when symbolically executing e.g., an if-then-else statement that splits an execution path into new paths for each possible branch arm. For each of these paths, for instance, the symbolic store may be preserved in the new paths, but different conditions may also be carried through such that in the path where the symbolic program counter refers to the then-branch the evaluation of the boolean expression in the if statement must be valid, whereas it is not valid in the path where the symbolic program counter refers to the else-branch. Possible executions are not just captured through branching paths, but along a path itself through the symbolic store; a symbolic value represents any valid concrete substitution.

The SE found in KeY and REFINITY operates on a dynamic logic, JavaDL, for a restricted subset of Java. Syntactically, JavaDL is an extension of first-order logic with program variables and program modalities. Semantically, JavaDL formulas are evaluated in a Kripke structure, which is a collection of first-order structures [1, Sec. 3.3].

```
void n() {
/*@ assignable frN;
  @ accessible fpN;
  @ exceptional_behavior requires false;
  @*/
  \abstract_statement N;
 }
```

Listing 1: Method in REFINITY

The use of SE to potentially explore every possible execution a program can have is a popular program analysis technique. AE extends SE by introducing abstract program elements (APEs) to the base language that is symbolically executed. For statements and expressions, a corresponding abstract statement and abstract expression are introduced. APEs represent any possible substitution with concrete program elements, which can be statements or expressions, from the base language being symbolically executed. Execution of APEs is then the leap taken in AE over traditional SE. It requires the introduction of abstract state changes, SE branching for any abrupt completion an APE may have (e.g. exceptions thrown), over-approximation of returned values and thrown exceptions by symbols created "dependently fresh" for identifiers of abstract program placeholders, and a way to specify the behavior of APEs [28].

We summarise how AE is implemented in REFINITY by showing how to specify a method **`void`** `n()` with a mostly unknown method body. Listing 1 shows a specification in REFINITY, in which a method `n()` is specified and its only content is an abstract statement **`\abstract_statement`** `N` preceded by a Java Modelling Language (JML) like[3] specification [18]. The specification indicates that the method can possibly assign to some abstract locations (its frame) and can access some abstract locations (its footprint), and that no exceptions will be thrown by the method body. The specification is straightforward: abstract statement `N` may assign to the abstract location set `frN`, access the abstract location set `fpN` and may not throw any exceptions.

An abstract location set represents a fixed set of memory locations that (a part of) a program may read from or write to. The set is fixed through a program's duration but the values at these locations may change. When an abstract location set occurs in an assignable or accessible specification, it is to be understood as an upper bound; the locations may possibly all be accessed or assigned to, not at all or anything in between.

2.2 Proving Refactoring Correctness with Abstract Execution

In the following, we use an example to show how AE can be used to prove a refactoring correct with REFINITY.

[3] We say "like" as JML does not deal with abstract Java programs.

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

(a) Before

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

(b) After

Listing 2: Extract Local Variable refactoring

```
assert x instanceof X;
((X)x).n();
((X)x).n();
return x;
```

(a) Before

```
assert x instanceof X;
X temp = (X)x;
temp.n();
temp.n();
return temp;
```

(b) After

Listing 3: Extract Local Variable refactoring in REFINITY

Let us consider the Extract Local Variable refactoring seen in Listing 2. The example is an instance of a more general case, where preserving the behaviour of the program depends on other parts of the code. The behaviour of the program changes if the method call `n()` has access to the attribute `x` and overwrites it.

Before we would have potentially method calls to different objects o_1`.n()` and then o_2`.n()`, whereas after applying the refactoring we would have o_1`.n()` followed by o_1`.n()`. In this case if, e.g., `n()` simply prints **`this`**`.toString()`, we will observe a difference in the two programs.

The dynamic check for such a change detailed in other work [13] codified the necessity that the reference `x` remains unchanged through the introduction of an assertion `assert temp == x;` to uncover violations after the fact, which is useful, e.g., when checking the refactored code against its suite of unit tests.

In REFINITY one proceeds to verify a refactoring as correct by supplying the code before and after refactoring, and supplying a desired precondition and postcondition to relate the two programs. We essentially ask REFINITY: *given these preconditions does the postcondition hold after abstract execution of these two abstract programs?* A proof of correctness in REFINITY is a proof for any concrete Java programs that can be *instantiated* to adhere to the abstract specification given in REFINITY.

We describe the refactoring to REFINITY as shown with a left side (Before) in Listing 3a, a right side (After) in Listing 3b and a method level context which contains the method we have already shown in Listing 1.

Additionally, some information is declared in parts of REFINITY's interface that are not shown here: Free program variables, here `x`; abstract location sets, here `frN` and `fpN`; relevant locations for the before and after code, here empty; the desired precondition, here empty, and the desired postcondition.

The pre- and postcondition are specified in terms of equations that may relate to the effects of the before and after side, such as return values, exceptions thrown and any of the relevant locations declared. Here we use the default postcondition REFINITY provides which is simply `\result_1 == \result_2`, where `\result_1` and `\result_2` are each respectively a sequence of results for the abstract execution of the Before and After program. Each sequence contain in this order: 1) The return value if any, otherwise null; 2) exceptions thrown if any, otherwise null; and 3) the values at specified abstract location sets (that the user selects as relevant).

The default postcondition in our case will be that `x` returned in Before must be identical to `temp` returned in After, and that any exception thrown must be identical for the Before and After program. When left empty, as done here and so trivially true, equality with respect to the last component of the result sequences is an assertion that the values at all abstract location sets selected as relevant are identical.

We use an assertion to ensure that we consider only the refactoring when `x` is an instance of its intended type. A current limitation of the implementation of REFINITY necessitate the casting `(X)x` as free program variables may only be declared to be of type Object.

With only the specification shown in Listing 3, REFINITY is unable to prove that `x` will be identical to `temp` after AE of both sides. We get several instances of SE resulting in the reference `x` being changed by the abstract statement `N` in `n()`.

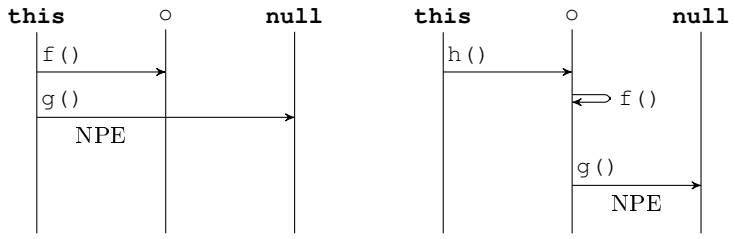
To prohibit this, we specify that the execution of the abstract statement `N` cannot interfere with `x`. A constraint on the frame `frN` of abstract statement `N`, namely that it is disjoint from `x` must be introduced, which is achieved by putting an abstract execution constraint `@ ae_constraint \disjoint(x,frN)` on each side which ensures `x` will be assumed not to be in `frN`.

After the aforementioned change, REFINITY manages to automatically prove that the postcondition holds after abstract execution of both sides; `x` and `temp` will hold identical references and any exceptions thrown will be identical. Note that it remains for any concrete application of this refactoring to prove that any code matching abstract statements fulfills their annotated constraints.

Although we successfully prove the refactoring we have encoded here, the required scaffolding of return statements leaves something to be desired: Finding intended concrete instances of the abstract programs now involve, potentially, ignoring the return statements.

3 Challenges in Complex Refactorings

In this section we explore the possibilities of applying REFINITY beyond its original vision. In particular, we are interested in moving away from statement-based refactorings to more complex changes that also affect the structure of the code. The first new refactoring, Hide Delegate, expresses the desired behaviour of



(a) Before (b) After

Fig. 1: Null Pointer Exceptions (NPEs)

REFINITY in a straight-forward manner and KeY automatically completes the proof once the right preconditions are identified.

After that, we present a refactoring that is related to Dead Code Elimination. This requires introducing additional constructs for object-creation within the underlying KeY system, which are then harnessed in *taclets* expressing that certain non-identical heaps guarantee equivalent behaviour.

We continue our wishlist for further flexibility in expressing equivalent program behaviour based on the execution history of user-defined observable actions, and finally discuss challenges related to expressing equivalence in the face of different data types.

3.1 Encoding the Hide Delegate refactoring

The Hide Delegate refactoring can be described as an Extract Method refactoring on a call chain. Consider the statement `Y y = o.f().g()`, where the call chain is extracted to a new method on `o`, say `h()`, which contains the extraction `Y h() { return this.f().g(); }`, such that we can replace the chain above with `Y y = o.h()`. The refactoring can enable less coupling as the class that contained the call chain afterwards does not need to know the return type of `f()`.

Note that in the case of the more general pattern `X x = o.f(); Y y = x.g()` with non-interfering intermediate statements between the two statements, we can reach the considered pattern through applications of Slide Statement and finally Inline Variable on `x`.

The scenarios shown in the sequence diagrams in Fig. 1 will both result in `NullPointerException` (NPE) when the call to `f()` returns **`null`**. In the strictest sense of behavioral preservation, we will observe a difference in the behaviour before and after the refactoring. Concretely, the NPE thrown in Fig. 1a (before) will show a different stacktrace than the one thrown in Fig. 1b (after). Thus we consider a behavioral equivalence that allows for disagreement in stacktraces for such matching exceptions. In fact one is unable to make any other distinction in REFINITY as it does not consider such effects.

```
assert in instanceof Resource;
return ((Resource)in)
       .getOwner()
       .getResource();
```

(a) Before

```
assert in instanceof Resource;
return ((Resource)in)
       .hDelegate();
```

(b) After

Listing 4: Program fragments for HIDE DELEGATE refactoring in REFINITY

```
class Resource {
 Owner owner;
 Owner getOwner() {
 /*@ assignable frF;
   @ accessible fpF;
   @*/
   \abstract_statement F;
   return owner;
  }
  Resource hDelegate() {
   return this.getOwner()
          .getResource();
  }
}
```

(a) Before

```
class Owner {
 Resource resource;
 Resource getResource() {
/*@ assignable frG;
  @ accessible fpG;
  @*/
  \abstract_statement G;
  return resource;
 }
}
```

(b) After

Listing 5: Classes in HIDE DELEGATE refactoring in REFINITY

We specify the before- and after-program fragment for a HIDE DELEGATE refactoring in Listing 4 which faithfully captures the previously sketched out refactoring. The classes and methods used in the refactoring are presented in Listing 5 and show that we minimally specify the contents of the involved methods by using abstract statements in their bodies. Note that we allow abrupt completion in the abstract statements `F` and `G` in the methods `getOwner()` and `getResource`. That means the abstract statements may for instance throw exceptions. For instance, the sketched out scenario considered in Fig. 1, where `getResource()` will return **`null`** and cause the following call to throw a NPE, is a possibility.

To prove the specified HIDE DELEGATE refactoring in an original published version (v0.9.7) of REFINITY, we need a postcondition that consists of a conjunction of return values of the before- and after-programs being identical and that any thrown exceptions are both instances of NPE or otherwise equal. This is owing to the fact that REFINITY does not consider occurrences of **`new`** `NullPointerException()`, or any other newly created objects, to be equal. In particular we remark that although strictly speaking certain exceptions before

and after a refactoring may be distinguished by different stacktraces, we may want to consider them to be equal even so.

We have improved REFINITY[4] to resolve this issue; we may keep the default postcondition that simply matches return results and exceptions, and REFINITY automatically manages to prove the shown HIDE DELEGATE refactoring to be correct. In the following section we will detail the changes needed to accomplish this.

3.2 Object Creation

As we have seen in the previous section, REFINITY encodes a rather harsh regimen on program equivalence: in the absence of a more fine-grained (application-specific) post-condition, it encodes that return values or exceptions must be identical on both sides, as well as the objects in the relevant location set (and the observables in this location set must be adequately specified).

This, in combination with the symbolic execution of both programs, creates a hurdle for programs that contain object creations (and, subsequently exceptions). An object allocation in JavaDL is, roughly sketched, symbolically executed by creating a fresh function symbol for the allocated object and storing it on the symbolic heap.

The question of equivalence for created objects is not specific to abstract execution, yet important for its practicability: abstract statements are embedded in concrete programs which affect state as well, and more complex and application-specific refactorings must take all language features of the host language into account.

At its core, the challenge lies in the fact that, as both programs/versions are executed in the same proof, objects created within them are not equal to each other: it is not possible to prove that the program `return new C();` is equivalent to itself. Indeed, it is not obvious whether the program should be considered equivalent to itself in the first place. The program is executed twice from the same state, but this does not suffice for the two created objects to be equal – the allocation must, additionally, be deterministic. In the following, we make the assumption that this is indeed the case and use this information in the symbolic execution.

Approach. To formalize this assumption, we first need to express the determinism of object creation.

We assume that the program semantics is expressed as a Kripke structure, where the domain of each Kripke state is constant. This means that all objects always *exist*. To *allocate* an object, it must be marked as allocated. To do so, each object has a field `<allocated>` that is set to **`true`** for all allocated objects, and to **`false`** for all others.

To express deterministic allocation, we use a total order $<$ on all objects, which is pivoted on some object o. All objects before o are allocated, and all

[4] Available at `https://github.com/selabhvl/REFINITY-abstractallocate`

objects after o are not allocated. Allocation then uses o as the next object to create. As the order is fixed for the Kripke structure, two allocations in different states, but with the same pivot object o, will allocate the same object next: o.

Formalisation. As the next step of formalization, we express this as a sequent calculus rule in JavaDL. We give the basic concepts behind JavaDL next, for a formal treatment we refer to an introduction to KeY and JavaDL [1].

A sequent has the form $\Gamma \Rightarrow \Delta$, where the antecedent Γ and the succedent Δ are sets of JavaDL formulas. JavaDL is a typed first-order dynamic logic with program variables and updates. Its operators are the usual first-order operators, a modality $[\mathtt{s}]\phi$ expressing that formula ϕ holds after executing statement $\mathtt{s}$, and updates. An update is a syntactic representation of a substitution on a program variable. A simple update, which is the only form of update we require here, has the form $\mathtt{v} := \mathtt{t}$ and expresses that the value of program variable $\mathtt{v}$ is set to the value of term $\mathtt{t}$. The truth value of a formula ϕ after the substitution expressed by update U is expressed by applying the update to the formula, denoted by $U\phi$.

Semantically, JavaDL is evaluated over a Kripke structure and an interpretation I. The interpretation assigns values to predicates and function symbols, while the Kripke structure is a set of states: assignments for the program variables. The semantics of a modality is the transition from one state to another, according to the semantics of the program. The semantics of an update is the transition from one state to another, according to the substitution expressed by it. The sequent calculus for JavaDL realizes symbolic execution by reducing modalities to updates (under certain side-conditions, added to the premise/path-condition).

To handle objects, JavaDL uses a special program variable `heap`, which maps from objects and fields to their value. The heap is written and read with the usual theory of arrays [23], where fields are used as indices, following the approach by Weiß [32]. The special function `create` sets the `<allocate>` field to true. It cannot be set otherwise (as one cannot de-allocate an object within JavaDL). The function `C::exactInstance` maps each term to true, that is a member of class `C` and none of its subclasses.

This is formalised in the following definition, where we model our assumption directly in the model and extend the `allocateInstance` taclet in KeY.

Definition 1. *We assume that in every state of the Kripke structure, for every heap h, all objects are ordered by some total order $<_h$, such that there is some object o_h, such that (1) for all $o' <_h o_h$, the object o' is allocated (i.e., its `<allocated>` field is set to true), and (2) for all $o_h \leq_h o''$, the object o'' is not allocated. We introduce a unary function symbol* `allocate` *with the signature* $\mathtt{Heap} \rightarrow \mathtt{Object}$*, whose interpretation must adhere to* $\mathcal{I}(\mathtt{allocate})(h) = o_h$*. The (slightly prettified) rule is as follows:*

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

The modification is the addition of $v = \texttt{allocate}(\texttt{heap})$ to the antecedent. The modified rule suffices to show the simple equivalence of **`return new`** `C()` to itself from above.

Continuing our investigation of when objects are considered equal, where the two allocations are independent from each other, i.e., any side-effects of the constructors are not visible to each other, `C` is not a subtype of `D` and vice versa. Again the question where the two `C` (resp. `D`) objects are equal arises. They are not equal in the sense that, if there is a implicit[5] global counter that counts all allocations, they get the same number from this counter. They are equal in the sense that there is no explicit way to distinguish them in the program.

They are, however, distinguishable in the proof system due to the term-representation of the heap. If we choose to consider them equal, we must adapt our `allocate` mechanism: firstly, it must be able to distinguish between the allocation of different classes and, secondly, it must be able to simplify the heap to ignore irrelevant operations on it. We adapt the previous definition to take into account the types of objects, by having a *set* of orders $<_h$, one for each class.

Definition 2. *We assume that in every state of the Kripke structure, for every heap h and every class C, all objects of type C are ordered by some total order $<_h^C$, such that there is some object o_h^C, such that (1) for all $o' <_h^C o_h^C$, the object o' is allocated (i.e., its `<allocated>` field is set to true), and (2) for all $o_h^C \leq_h^C o''$, the object o'' is not allocated. Additionally, if D is a subtype of C, then $<_h^D$ must be a suborder of $<_h^C$. For each class C, we introduce a unary function symbol* $\texttt{C} :: \texttt{allocate}$ *with the signature* $\texttt{Heap} \to \texttt{Object}$*, whose interpretation must adhere to* $\mathcal{I}(C :: \texttt{allocate})(h) = o_h^C$*. We obtain:*

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

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.

$$\texttt{C} :: \texttt{allocate}(\texttt{store}(\texttt{h}, \texttt{o}, \texttt{f}, \texttt{v})) \rightsquigarrow \texttt{C} :: \texttt{allocate}(\texttt{h}) \quad \textit{if } f \neq \texttt{<allocated>}$$
$$\texttt{C} :: \texttt{allocate}(\texttt{create}(\texttt{h}, \texttt{o})) \rightsquigarrow \texttt{C} :: \texttt{allocate}(\texttt{h}) \quad \textit{if } C \not\sqsubseteq T(o)$$

The first rule uses the assumption that the orders are only sensitive to the `<allocated>` field, and the second one that they are independent, besides the subtyping relation. Intuitively, this implements a different counter for each class in the type hierarchy and two objects are considered equal if the counters of their classes are equal. Creating an object of a class increases the counter of

[5] The counter is indeed implicit, as the order $<_h$ cannot be accessed by the proof system.

```
x = new C();
y = new D();
```

(a) Before

```
y = new D();
x = new C();
```

(b) After

Listing 6: Object creation

```
class C {
  private int j;
  public C(int j){
    this.j = j;
  }
}
```

(a) Class

```
return new C(1);
```

(b) Before

```
return new C(2);
```

(c) After

Listing 7: Object creation and fields

this class and all its superclasses. If `C` is a subtype of `D`, then the proof fails. This suffices to prove the programs in Listing 6 to be equivalent. Interestingly, Steinhöfel already proved the general Slide Statement refactoring correct for abstract statements under the right constraints ([28], p.231), yet KeY needs additional rules for these concrete statements. The second rule is able to remove some intermittent operations that are symbolically represented on the heap, e.g., a call to an imaginary setter in `x = new C(); x.set(1); y = new D()` versus `y = new D(); x = new C(); x.set(1)`, if they are not relevant to object creation.

Note that we ignore the state of fields in the heap, meaning that the equivalence of two objects is determined only by their counters. This requires to be careful with specification: it does not suffice for each notion of equivalence to specify that two objects are equal, but also *their fields* must be equal. Consider the class and programs in Listing 7. The returned objects are identical, but in the post states the heap assigns different values to their field.

Let us close with three remarks. Firstly, the solutions we described here are enabling the programmer (or refactoring designer) to fine tune their notion of equivalence, which must be specified in addition to the refactoring itself. Realising the choice, however, is rather simple by enabling and disabling rules, resp. taclets. Thus, we avoid to put even more burden on the first-order specification of the relational post-condition, and merely require the programmer to select from a number of options. We emphasise that the taclets for the different options are not *unsound*, but merely switch between different version of assumptions about the (in this case underspecified) Java object model. Of course, the default version is not using the new taclets and stick with the originals, i.e., not assuming deterministic allocation.

```
C x;
x = new C();
x = new C();
```

(a) Before

```
C x;
x = new C();
```

(b) After

Listing 8: Dead Object

Secondly, while we only discussed object creation here, the same solution also handles *exceptions*, which must be created before being thrown. Exceptions are special in the sense that they have access to the program beyond the parts that are exposed to (non-reflective) programs. For example, they can access the line number of the throwing statement for their stack trace. As the stack trace is not modelled in JavaDL, its treatment for verification is an open research question in itself, and because we consider exhibiting the stack trace within the program a dubious practice in the first place, we chose to ignore it in this paper.

Thirdly, one could argue that we have not so much proven the refactoring to be correct, but rather moved this decision further down the chain: unlike in a full formalisation of the Java object model from ground up, we do not have a way of proving the taclets correct (i.e., derive them as lemmas) within KeY, as they model our assumptions about object identity when running two Java programs in the exact same state, which is not described by the Java semantics.

Excursus: Dead Code Elimination. Similar considerations of equivalent heap manipulations need to be considered also in the are of optimizations, whose soundness proofs rely on relation verification as well. For example, consider the program in Listing 8.

Again, we assume that the constructor of `C` has no side-effects except object creation on the heap. In this case, the first object creation is of no consequence. The additional rules above however will not yet be sufficient to prove that this version is equivalent to the version without the redundant object creation and assignment. The location-set mechanism would still insist that the second object created in the redundant version is a different object from the first (and only) object created in the optimised version. On the one hand this can be addressed through a relaxed post-condition where we accept that we only need *some* object of the right type and arguments, but it relies on the side-condition of the constructor not having side-effects, which requires a very restrictive method contract for the constructor. Alternatively, the notion of equivalence becomes even more specification-heavy for optimisations, as it may be more fine-grained. We foresee that such and other instances will give rise to various further taclets in the future.

4 Potential Future Improvements

In the previous section, we have established that relational verification using abstract execution must take care of the general effects of language semantics outside the abstract statements. In this section, we illustrate a different obstacle to practical abstract execution, which touches on its core specification principles: sequences of side effects and events.

4.1 Trace Properties

Following Fowler's persuasion of what constitutes correct refactorings, developers are content if refactored code gives the same observable behavior [15].

This behavior is first and foremost encoded through unit-tests, but also on tests through side-effects and their order (e.g. output via `print`-statements). Here we then have a much more relaxed setting where equivalence is decoupled from the fine-grained program semantics. More realistically, one may want to establish that after refactoring, certain operations happened in the same order.

Abstract execution supports the specification of read and write events, via dynamic frames, but does not specify their order or give a general possibility to specify the order of side-effects/events. The most straightforward approach is to use a special model variable to keep track of the events explicitly in a trace, and specify properties using the surrounding logic, in our case, JavaDL. This approach has been taken for dynamics logics (without investigating relation verification) in, e.g., ABSDL [11] and for relational verification (albeit without using a dynamic logic) by, e.g., Barthe et al. [5]. This explicit encoding of user-defined execution histories has the advantage that it is not only useful for proofs, but can also directly be harnessed in concrete unit-tests where we can explicitly compare the recorded history of an earlier execution of a test with the history of the same test but on the refactored code base.

A main advantage of encoding the trace in the post state is that relational verification (either based on self-composition or its variants [10,4,3] or using the proof obligation described above), can use the state logic to describe both states. However, first-order specifications of temporal properties have been proven to be unwieldy, large and hard to understand. This led to the development of other dynamic logics which interact with novel trace specifications, such as BPL [19] and symbolic traces [9], where the post-condition of a modality is a *trace formula* without quantifiers over indices, whose models are single traces. It has neither been investigated how such logics can be used for relational verification, nor how abstract statements can be specified with respect to such trace properties.

We concentrate on, simplified, symbolic traces here, which are defined by the grammar

$$\theta ::= \lceil \phi \rceil \mid \mathsf{call}(\mathtt{m}) \mid \mathbf{finite} \mid \theta * * \theta$$

where $\lceil \phi \rceil$ denotes the trace where the state formula ϕ holds, $\mathsf{call}(\mathtt{m})$ a call event on method `m`, **finite** an arbitrary finite trace without any events[6] and $**$ is the

[6] We deviate here from the original definition for example's sake.

```
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

Listing 9: Refactoring for Trace-Based-Notions of Equivalence.

chop [9], a special concatenation. Models for such formulas are traces: sequences of states and events.

Consider the two programs in Listing 9. It shows a refactoring of some program using a `File`, where we only give the trace specification. The first program specifies that some initialisation happens that is guaranteed to call `f.open`, then the file is read and written, and then some finalisation happens that calls `f.close`. The second program switches the order of read and write. The programs are, obviously, not equivalent in a strict sense, but if the trace property we are interested in is only concerned with the order of operations on the file (`open`, `write`, `read` and `open`), for example to express that only opened files are read and written to, then we need to specify a notion of equivalences. It remains to be seen in how far abstract executions can be expanded with such a mechanism in the future.

4.2 Relational Invariants

Another area of interest for equivalence is replacing one data structure with another, e.g. Fowler's Replace Array with Object [14, p.186] or Replace Primitive with Object [15]. As an example, in the following we look at a piece of code where an array is replaced with an object (or vice versa). Again, from the strict default perspective of "equal return values, equal heaps", any two programs using the data structures are obviously not equal. Encoding observability

```
String[] p = new String[2];

p[0] = "36.452999";
p[1] = "28.226376";
```

(a) Before

```
Pos p = new Pos();

p.setLat("36.452999");
p.setLon("28.226376");
```

(b) After

Listing 10: Replace array with object refactoring

```
String[] p = new String[2];
/* M only uses the static
method API to modify `p` and
does not assign a new value
to `p`. */
\abstract_statement M
setX(p,value);
```

(a) Before

```
Pos p = new Pos();
/* ditto */
\abstract_statement M
p.setX(value);
```

(b) After

Listing 11: Replace array with object refactoring

through traces as per the previous section will obviously solve this issue. A new challenge arises when both programs use different or disjoint sets of operations, i.e., we have different alphabets for their trace languages.

Let us consider a refactoring that replaces an array containing a geographical position given by a latitude and longitude as in Listing 10a with an object that gives read or write access to the same values through setters and getters that make it immediately clear what is being accessed as seen in Listing 10b. In either direction of this refactoring, we must be certain that indexing can only occur within the bounds of the original program as there will be no corresponding out of bounds failure for method calls. We note the added (syntactical) complication that in the direction from array to object, that if array offsets are computed, there is no direct correspondence to a setter/getter, and the refactored code needs to dispatch on the corresponding component. In the following, for simplification, we assume that the array is only used with constants.

Having established that the two programs are not equal, but should be considered equivalent, we need to establish a correspondence in the specific case. To avoid having to establish the correspondence in all uses (of either setter/getter or the array), we can assume that accesses in either case are wrapped in a method — if we assume that Extract Method/Inline Method are already proven as correct. This reliance on other refactorings allows us to compartmentalise the reasoning, and mostly focus on contracts for the involved methods. While abstract execution is good at reasoning about placeholders for abstract behaviour, a similar mechanism for abstract structures is missing. Through some intermediate steps, we can offload most of the reasoning to a history-based mechanism

with some static assumptions on the code that can be easily checked. The first is to encapsulate all accesses in methods; setters/getters on the object-side, and static helpers on the array-side. We can then formulate the abstract programs for REFINITY as given in Listing. 11 for each of the operations in the API, here the matching pairs of setters/getters. It remains to show that a) neither abstract statement overwrites p, that b) the histories of API operations with arguments called on either side match pairwise, and that c) in the post-condition the values in their respective components match.

5 Discussion

In the following, we shortly discuss some of the raised issues and in how far they can be adressed in the future and possibly in the short-term.

The first challenge is due to the way REFINITY prepares the environment and the top-level proof obligation for KeY, in fact both sides in a refactoring share the same Java namespace. This means for example that the Extract Method refactoring no so much proves the original correct, but rather a version where the extracted method is already present. Care must be taken to set this up correctly, and e.g. in the Extract Method example make sure that the methods is not used on the *Before*-side. The same holds for refactorings that remove code.

Addressing this would require choosing unique names in either schemata, since they go into a single KeY-proof, and hence would require some form of mapping classes/objects of distinct types (due to the nominal type system of Java) between both sides. This issue is closely related to the challenge we discussed before when trying to relate unrelated yet semantically equivalent data types (see Section 4.2). In general, we observe that currently REFINITY requires some scaffolding that makes the actual refactoring less obvious, such as our use of assertions and casts.

A more general problem is capturing the most general instance of a refactoring. Currently, REFINITY's lack of placeholders for names, means that e.g. in the Extract Local Variable refactoring we would have to instantiate the *Before*-schema for every concrete instance with the corresponding identifier names for variables. The otherwise straight-forward refactoring Rename Temporary already exposes the issue faced by REFINITY due to renaming on the block-level. Likewise, the related Hide Delegate example uses concrete method- and class names. During our development of this refactoring, we have found ourselves revising the encoding and use of placeholders with their annotations repeatedly. Conversely, due to the challenges in checking instantiation of schemata against concrete programs already pointed out by Steinhöfel [28, 119,137], one has to take care not to write too restrictive programs that rule out useful working instances.

Another area for placeholders would be a generalisation of storage locations: We foresee that there exist refactorings that may need to be specified twice, once with using local variables and another time using attributes in their schemata.

For the time being we are limited to checking schemata against each other. In the future, when we move on to checking instantiations, we feel that often necessary pre-conditions on a refactoring can easily be discharged by simple syntactical or static analysis (e.g. "code does never read attribute `x` of objects of type `C`"). Yet unlike in other formal work where the program is encoded as part of the proof-term, we cannot implement such analyses within KeY, and can only informally document and require such side-conditions on refactorings. Correspondingly, we will also not be able to use KeY to formulate and prove a lemma that such a static property entails the necessary consequences.

6 Related Work

Similar or other approaches to formal verification of refactorings can be found in work by Garrido et al. [16] who formalize Push Down Method, Pull Up Field and Rename Temporary using an executable Java formal semantics in Maude and give partially mechanised proofs for the two former.

Long Quan et al. [25] formulate refactorings as refinement laws in the calculus of refinement of component and object-oriented systems (rCOS), focusing on correctness proofs of refactoring rules themselves. They did not have the benefit of any tool support, but similarly were able to describe refactorings on schematic programs. Statement level refactorings as well as refactorings that transform class hierarchies are considered.

While KeY and REFINITY are unique for their relational verification capacity for schematic programs (or abstract programs) they are limited in power for verification of concrete programs relying much more on manual specification or interaction [29] than tools like LLRêve [21] or SymDiff [22] which offer more automation for concrete programs.

Peter Müller et al. [24] present a verification infrastructure whose intermediate language supports an expressive permission model natively, with tool support including two back-end verifiers: one based on symbolic execution and one on verification condition generation, an inference tool based on abstract interpretation reportedly under development.

Stolz, Pun and Gheyi investigate how well-known refactorings interact with concurrency in Active Object languages [31]. Findings show that refactorings that are straight-forward in Java are not necessarily so under the concurrency model considered and identify key program transformations that may cause interactions. In contrast to their work, REFINITY and its foundation KeY strictly consider sequential Java programs, but they already explore the notion of equivalent executions in their formal considerations of syntactically different, but overlapping, programs.

Eilertsen, Stolz and Bagge demonstrate a technique of introducing runtime checks in Java for two refactorings Extract And Move Method and Extract Local Variable [13]. The technique in combination with testing can detect changed behavior and allow identification of which refactoring step intro-

duced the change the deviant behavior. Our proof of correctness of Extract Local Variable in REFINITY is inspired by their technique.

Schaefer et al. develop microrefactorings that can be composed to specify several refactorings in a concise manner [26]. They use an infrastructure to preserve correct name binding in refactorings.

Soares et al. [27] describe and evaluate SafeRefactor - a tool that given a program input and a refactoring to apply can automatically generate testcases to detect behavioural changes. It would be interesting to adapt SafeRefactor to do deal with REFINITY's abstract programs, and generate test cases for the instances where REFINITY fails to prove a refactoring. These could then be run against concrete refactored programs.

Dovland et al. [12] propose a proof system that allows incrementally reasoning about adaptable class hierarchies, based on lazy behavioural subtyping, for an object-oriented kernel language similar to Featherweight Java. The proof system avoids reverifying methods that are not modified explicitly by the class adaptation. We are unaware whether the incremental reasoning broached for the proof system has a counterpart in KeY and REFINITY, but the latter allows for modularity in proof and verification which may achieve a similar result.

7 Conclusion

We have presented two new encodings of refactorings (Extract Local Variable and Hide Delegate) and their necessary preconditions (constraints) for them to be behaviour-preserving for REFINITY. REFINITY has syntactical constructs that capture abstract program executions, for which the KeY system, an automated theorem prover for JavaDL, succeeds in proving the refactorings as correct (equal) wrt. to the Java semantics without user interaction.

The Hide Delegate refactoring departs from statement-based refactorings and considers changes involving multiple classes and requires us to consider the first subtle difference between equivalent-yet-not-equal objects in the form of equivalent exceptions and how we need to explicitly address this in the post-condition of the proof-obligation. This also allows us to make a contribution through further taclets that capture some indistinguishable programs that only differ in placement of objects on the heap.

We discuss in how far REFINITY could be used to capture other refactorings, broadening our investigation into the underlying notion of (sometimes use-case specific) *equivalent behaviour*. For example, while traces over observable behaviour could be explicitly encoded and checked against each other as return values in REFINITY, a more general process-algebra inspired approach of execution histories for abstract executions would avoid distorting the original program logic with scaffolding to achieve an encoding without having to extend the tool. We also point out the difficulties due to a naming issue in the current encoding from Refinity to KeY proof-obligations for refactorings that change the class hierarchies or in general attempt to relate behaviour across different types.

Future Work. To investigate the discussed problems with refactoring using AE wrt. trace properties, we are currently implementing AE for BPL in the Crowbar tool [20] as a starting point, a symbolic execution engine to prototype behavioural symbolic execution.

We are also working on contributing further encodings of common refactorings that can already now be handled by REFINITY. In addition, we are particularly interested in additional taclets for KeY that would enable automation of proofs that currently are stuck on the explicit symbolic encoding of program state although it would be indistinguishable from equivalent states in pratice. The latter is of relevance e.g. for proving common optimisations as in our DEAD CODE ELIMINATION example.

Acknowledgements This work was partially supported by the Research Council of Norway via SIRIUS (237898), PeTWIN (294600) and CROFLOW (326249).

References

1. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M. (eds.): Deductive Software Verification - The KeY Book - From Theory to Practice, Lecture Notes in Computer Science, vol. 10001. Springer (2016). https://doi.org/10.1007/978-3-319-49812-6
2. Baldoni, R., Coppa, E., D'Elia, D.C., Demetrescu, C., Finocchi, I.: A survey of symbolic execution techniques. ACM Comput. Surv. **51**(3), 50:1–50:39 (2018). https://doi.org/10.1145/3182657
3. Barthe, G., Crespo, J.M., Kunz, C.: Relational verification using product programs. In: FM. Lecture Notes in Computer Science, vol. 6664, pp. 200–214. Springer (2011)
4. Barthe, G., D'Argenio, P.R., Rezk, T.: Secure information flow by self-composition. In: 17th IEEE Computer Security Foundations Workshop, (CSFW-17 2004), 28-30 June 2004, Pacific Grove, CA, USA. pp. 100–114. IEEE Computer Society (2004). https://doi.org/10.1109/CSFW.2004.17
5. Barthe, G., Eilers, R., Georgiou, P., Gleiss, B., Kovács, L., Maffei, M.: Verifying relational properties using trace logic. In: Barrett, C.W., Yang, J. (eds.) 2019 Formal Methods in Computer Aided Design, FMCAD 2019, San Jose, CA, USA, October 22-25, 2019. pp. 170–178. IEEE (2019). https://doi.org/10.23919/FMCAD.2019.8894277
6. Baumann, C., Beckert, B., Blasum, H., Bormer, T.: Lessons learned from microkernel verification – specification is the new bottleneck. In: SSV. EPTCS, vol. 102, pp. 18–32 (2012)
7. Beckert, B., Bruns, D., Klebanov, V., Scheben, C., Schmitt, P.H., Ulbrich, M.: Information flow in object-oriented software. In: LOPSTR. Lecture Notes in Computer Science, vol. 8901, pp. 19–37. Springer (2013)
8. Beckert, B., Ulbrich, M.: Trends in relational program verification. In: Principled Software Development. pp. 41–58. Springer (2018)
9. Bubel, R., Din, C.C., Hähnle, R., Nakata, K.: A dynamic logic with traces and coinduction. In: TABLEAUX. Lecture Notes in Computer Science, vol. 9323, pp. 307–322. Springer (2015)

10. Darvas, Á., Hähnle, R., Sands, D.: A theorem proving approach to analysis of secure information flow. In: SPC. Lecture Notes in Computer Science, vol. 3450, pp. 193–209. Springer (2005)
11. Din, C.C., Owe, O.: A sound and complete reasoning system for asynchronous communication with shared futures. J. Log. Algebraic Methods Program. **83**(5-6), 360–383 (2014). https://doi.org/10.1016/j.jlamp.2014.03.003
12. Dovland, J., Johnsen, E.B., Owe, O., Yu, I.C.: A proof system for adaptable class hierarchies. J. Log. Algebraic Methods Program. **84**(1), 37–53 (2015). https://doi.org/10.1016/j.jlamp.2014.09.001
13. Eilertsen, A.M., Bagge, A.H., Stolz, V.: Safer refactorings. In: Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques. LNCS, vol. 9952, pp. 517–531. Springer (2016). https://doi.org/10.1007/978-3-319-47166-2_36
14. Fowler, M.: Refactoring - Improving the Design of Existing Code. Addison Wesley object technology series, Addison-Wesley (1999)
15. Fowler, M.: Refactoring: Improving the Design of Existing Code, 2nd Edition. Addison-Wesley Signature Series (Fowler), Addison-Wesley (2018)
16. Garrido, A., Meseguer, J.: Formal specification and verification of java refactorings. In: 2006 Sixth IEEE International Workshop on Source Code Analysis and Manipulation. pp. 165–174. IEEE (2006)
17. Hähnle, R., Huisman, M.: Deductive software verification: From pen-and-paper proofs to industrial tools. In: Computing and Software Science, Lecture Notes in Computer Science, vol. 10000, pp. 345–373. Springer (2019)
18. Huisman, M., Ahrendt, W., Bruns, D., Hentschel, M.: Formal specification with jml. Tech. Rep. 10, Karlsruher Institut für Technologie (KIT) (2014). https://doi.org/10.5445/IR/1000041881
19. Kamburjan, E.: Behavioral program logic. In: TABLEAUX. Lecture Notes in Computer Science, vol. 11714, pp. 391–408. Springer (2019)
20. Kamburjan, E., Wasser, N.: Deductive verification of programs with underspecified semantics by model extraction. CoRR **abs/2110.01964** (2021)
21. Kiefer, M., Klebanov, V., Ulbrich, M.: Relational program reasoning using compiler IR - combining static verification and dynamic analysis. J. Autom. Reason. **60**(3), 337–363 (2018). https://doi.org/10.1007/s10817-017-9433-5
22. 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.) Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings. Lecture Notes in Computer Science, vol. 7358, pp. 712–717. Springer (2012). https://doi.org/10.1007/978-3-642-31424-7_54
23. McCarthy, J.: Towards a mathematical science of computation. In: Information Processing, Proceedings of the 2nd IFIP Congress 1962, Munich, Germany, August 27 - September 1, 1962. pp. 21–28. North-Holland (1962)
24. Müller, P., Schwerhoff, M., Summers, A.J.: Viper: A verification infrastructure for permission-based reasoning. In: Pretschner, A., Peled, D., Hutzelmann, T. (eds.) Dependable Software Systems Engineering, NATO Science for Peace and Security Series - D: Information and Communication Security, vol. 50, pp. 104–125. IOS Press (2017). https://doi.org/10.3233/978-1-61499-810-5-104
25. Quan, L., Qiu, Z., Liu, Z.: Formal use of design patterns and refactoring. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation, Third International Symposium, ISoLA 2008, Porto Sani,

Greece, October 13-15, 2008. Proceedings. Communications in Computer and Information Science, vol. 17, pp. 323–338. Springer (2008). https://doi.org/10.1007/978-3-540-88479-8_23
26. Schäfer, M., de Moor, O.: Specifying and implementing refactorings. In: Object-Oriented Programming, Systems, Languages, and Applications
27. Soares, G., Gheyi, R., Serey, D., Massoni, T.: Making Program Refactoring Safer. IEEE Software **27**(4), 52–57 (2010)
28. Steinhöfel, D.: Abstract Execution: Automatically Proving Infinitely Many Programs. Ph.D. thesis, TU Darmstadt, Dept. of Computer Science (May 2020), https://tuprints.ulb.tu-darmstadt.de/id/eprint/8540
29. Steinhöfel, D.: REFINITY to model and prove program transformation rules. In: d. S. Oliveira, B.C. (ed.) Programming Languages and Systems - 18th Asian Symposium, APLAS 2020, Fukuoka, Japan, November 30 - December 2, 2020, Proceedings. Lecture Notes in Computer Science, vol. 12470, pp. 311–319. Springer (2020). https://doi.org/10.1007/978-3-030-64437-6_16
30. Steinhöfel, D., Hähnle, R.: Abstract execution. In: ter Beek, M.H., McIver, A., Oliveira, J.N. (eds.) Proc. of Formal Methods - The Next 30 Years - Third World Congress. LNCS, vol. 11800, pp. 319–336. Springer (2019). https://doi.org/10.1007/978-3-030-30942-8_20
31. Stolz, V., Pun, V.K.I., Gheyi, R.: Refactoring and active object languages. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation: Engineering Principles - 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Rhodes, Greece, October 20-30, 2020, Proceedings, Part II. Lecture Notes in Computer Science, vol. 12477, pp. 138–158. Springer (2020). https://doi.org/10.1007/978-3-030-61470-6_9
32. Weiß, B.: Deductive verification of object-oriented software: dynamic frames, dynamic logic and predicate abstraction. Ph.D. thesis, Karlsruhe Institute of Technology (2011), https://d-nb.info/1010034960
33. Yang, G., Filieri, A., Borges, M., Clun, D., Wen, J.: Chapter five - advances in symbolic execution. Adv. Comput. **113**, 225–287 (2019). https://doi.org/10.1016/bs.adcom.2018.10.002