Changing Programs Correctly: Refactoring with Specifications

Fabian Bannwart and Peter Müller

ETH Zürich, `peter.mueller@inf.ethz.ch`

Abstract. Refactorings change the internal structure of code without changing its external behavior. For non-trivial refactorings, the preservation of external behavior depends on semantic properties of the program that are difficult to check automatically before the refactoring is applied. Therefore, existing refactoring tools either do not support non-trivial refactorings at all or force programmers to rely on (typically incomplete) test suites to check their refactorings.
The technique presented in the paper allows one to show the preservation of external behavior even for complex refactorings. For a given refactoring, we prove once and for all that the refactoring is an equivalence transformation, provided that the refactored program satisfies certain semantic correctness conditions. These conditions can be added automatically as assertions to the refactored program and checked at runtime or verified statically. Our technique allows tools to apply even complex refactorings safely, and refactorings automatically improve program documentation by generating assertions.

1 Introduction

Refactorings are equivalence transformations on source code that change the internal structure of code without changing its external behavior [10]. They are applied to improve the understandability of code and to reduce its resistance to change.

The application of a refactoring is *correct* if it preserves the external behavior of the program. Whether an application of a refactoring is correct depends on certain correctness conditions. For instance, replacing the (side effect free) condition of a loop by a different condition generally preserves the external program behavior only if the old and the new condition yield the same value in all program executions.

Refactoring by hand is tedious and error-prone. Refactoring tools simplify the application of refactorings, but they guarantee the preservation of the program's external behavior only for refactorings with very simple correctness conditions such as "Rename Method" [10]. More complex refactorings such as "Move Field" are either not supported at all or not guaranteed to be applied correctly. That is, their application potentially alters the external behavior of the program.

Consequently, programmers rely on unit testing to determine whether a refactoring was applied correctly. This approach works well if the program has a complete set of unit tests that execute rapidly. However, in most practical applications, the set of unit tests is highly incomplete, for instance, because programmers have to make a trade-off between the completeness of the test suite and the time it takes to run all tests.

In this paper, we present a technique that guarantees that refactorings are applied correctly. For each refactoring, we perform the following three steps.

First, we determine the refactoring's *essential applicability conditions.* These syntactic conditions ensure that applying the refactoring results in a syntactically-correct program. In the "Move Field" example, a field f can be moved from a class S to a class T only if the field and both classes exist and if the target class does not already contain a field called f. Essential applicability conditions can easily be checked syntactically and are therefore not discussed in this paper.

Second, we determine the refactoring's *correctness conditions.* These semantic conditions ensure that each application of the refactoring preserves the external behavior of the program. For instance, an application of "Replace Expression" is correct only if the old and the new expression evaluate to the same value in all program executions. One of the novelties of our approach is that correctness conditions can be expressed in terms of the refactored program and added to the program as assertions such as JML [15] or Spec# [3] specifications. They can then be checked at runtime to make sure that the execution of a test actually covers the correctness condition. Alternatively, they can be checked statically by a program verifier such as Boogie [3] or ESC/Java2 [13]. Some correctness conditions can also be approximated and checked syntactically.

Third, we provide a formal proof that each application of the refactoring preserves the external behavior of the program, provided that the program satisfies the refactoring's essential applicability conditions and correctness conditions. We consider the original and the refactored program to have equivalent external behavior if they perform the same sequence of I/O operations. This notion of equivalence allows us to handle even complex changes of the program's internal structure. It is important to note that this correctness proof is done once and for all for each refactoring, whereas correctness conditions must be checked for each particular application of a refactoring.

Our technique improves the state of the art in three significant ways: (a) It handles refactorings with complex correctness conditions. Expressing these conditions as assertions improves test coverage and enables static verification. (b) It works on the source code level as opposed to more abstract models such as UML class diagrams. Working on the code level is important because refactorings are mainly applied during implementation rather than during design. Moreover, many correctness conditions depend on the intricacies of concrete code. For instance, the correctness conditions for "Move Field" fall on the field accesses, which are not present in an abstract model. (c) The specifications added to the transformed program convey the programmer's tacit knowledge why a particular refactoring is applicable. Therefore, they improve the documentation and facilitate the application of program verifiers.

In this paper, we present our technique and apply it to "Move Field", a prototypical example for a complex refactoring. Our technical report [2] describes several other successful applications of our technique as well as an implementation of "Move Field" in Eclipse and Visual Studio.

Overview. The rest of this paper is structured as follows. In the next section, we discuss correctness conditions. Section 3 formalizes the notion of external equivalence for a language with objects, references, and I/O operations. We apply our approach to "Move Field" in Sect. 4. Section 5 discusses related work.

2 Correctness Conditions

The correctness conditions for a refactoring can be split into *a-priori conditions* that are checked statically in the original program before the refactoring is applied and *a-posteriori conditions* that are checked in the refactored program. A-priori conditions are semantic conditions that can be easily checked or aptly approximated syntactically. For instance, splitting a loop is possible if there is no data dependency between the statements in the loop, which can be checked a-priori by data dependency analyses.

However, for many interesting refactorings, correctness conditions cannot be checked a-priori. Consider for instance the following code fragment, which calculates and prints the hypotenuse of a right triangle with legs `a` and `b`.

```
print(a/cos(atan(b/a)));
```

A possible refactoring is to replace the expression `a/cos(atan(b/a))` by the simpler expression `sqrt( a*a + b*b )`.

The correctness conditions for this refactoring are that both expressions (a) are side effect free and (b) evaluate to the same result (we ignore differences due to rounding here). While property (a) can be checked a-priori by a static analysis, property (b) must be checked at runtime or verified statically. Runtime assertion checking is not useful for a-priori conditions because it would be cumbersome to force programmers to run their test suite before applying a refactoring. Therefore, property (b) is better formulated as an a-posteriori condition and turned into a specification of the refactored program:

```
assert sqrt( a*a + b*b ) == a/cos(atan(b/a));
print(sqrt( a*a + b*b ));
```

The assertion can be checked at runtime when testing the refactored program or proved by a program verifier.

Assertions for a-posteriori conditions of refactorings document knowledge about the design of the code. Therefore, it is important that they stay in the program, even after a successful test or static verification. They are an important guideline for future modifications of the code.

To avoid cluttering up programs with assert statements, it is vital that assertions themselves can be refactored. In the above example, the assert statement could, for instance, be replaced by the stronger assertion `assert a > 0`. If `a` is a field, this assertion could then be further refactored into an object invariant.

We believe that the process of automatically generating assertions during refactoring and later refactoring the assertions into interface specifications greatly contributes to the documentation of the code by making design decisions explicit. This is an important side effect of our technique.

3 External Equivalence

Depending on the application domain, different aspects of external behavior are of interest such as functional behavior, timing, or resource consumption. In this paper, we focus on functional behavior. That is, we consider two programs to be externally-equivalent if they perform the same sequence of I/O operations.

In this section, we present the programming language that is used in the rest of the paper, formalize our notion of external equivalence, and explain how to prove the correctness of refactorings.

3.1 Programming Language

We present our results in terms of a small sequential class-based programming language. The language has the following statements: local variable update $l{:=}e$, field read $l{:=}o.f$, field update $o.f{:=}e$, object allocation $o{:=}\texttt{new}\ C$, I/O operation $l{:=}\texttt{io}(p)$, sequential composition $t_1; t_2$, and loop $\texttt{while}(e)\{t\}$, where l and o are local variables, e is an expression, p is a vector of expressions, f is a field name, and t is a statement. For simplicity, we ignore the declaration of local variables, but we assume that they are correctly typed and that the language is type-safe. We do not discuss methods, inheritance, and subtyping in this paper, but our technical report does [2].

Assertions are not part of the program representation and do therefore not have any operational meaning. For runtime assertion checking however, these assertions have to be compiled into executable code.

The program representation is denoted by Γ. $code_\Gamma$ is the executable program code of Γ. $fields_\Gamma(C)$ is the set of field names for a given class C in program Γ. After object creation, each field is initialized to the zero-equivalent value of its type T, denoted by $zero(T)$. $\mathrm{ctt}(name)$ returns the compile-time type of variable or field $name$.

States. A state s of a program execution consists of variables $vars$, object heap $heap$ and I/O interactions ext: $s = (vars, heap, ext)$. We refer to the components of a state s by using $s.vars$, $s.heap$, and $s.ext$, respectively. The variables map names to values: $vars(l) = v$. The heap maps references to objects, which in turn map field names to values: $heap(o)(f) = v$. The I/O interactions accumulate the executed I/O operations: $ext = [\ldots, v{:=}\texttt{io}(v_1, \ldots, v_n), \ldots]$. The result of an I/O operation is given by the exogenous input decision function inp. Program executions start in the initial state ini, where all local variables hold default values, and the heap as well as the I/O sequence is empty.

Operational Semantics. Expression evaluation $[\![e]\!]s$ is defined compositionally by mapping the operator symbols to the respective operations: $[\![f(e_1, \ldots, e_n)]\!]s = [\![f]\!]([\![e_1]\!]s, \ldots, [\![e_n]\!]s)$. Variable accesses are simply lookups in the state: $[\![l]\!]s = s.vars(l)$. Field accesses are not allowed in expressions. The big-step operational semantics is defined in Table 1. Updating the image of a value x in a map m

with a value v is denoted by $m[x \leftarrow v]$. The state obtained from a state s by updating a local variable l with a value v is denoted by $s[l \leftarrow v]$. We use an analogous notation for updates of $s.heap$ and $s.ext$. The function $\mathrm{rtt}(v)$ yields the (runtime) type of a value. The transition relation for a statement t from a state s to a state r in a program Γ is written as $\Gamma \vdash s \xrightarrow{t} r$.

Statement t	Terminal state	Condition
$l{:=}e$	$s[l \leftarrow [\![e]\!]s]$	–
$o.f{:=}e$	$s[heap(s.vars(o))(f) \leftarrow [\![e]\!]s]$	$s.vars(o) \neq \texttt{null}$
$l{:=}o.f$	$s[l \leftarrow s.heap(s.vars(o))(f)]$	$s.vars(o) \neq \texttt{null}$
$o{:=}\texttt{new}\ C$	$s[o \leftarrow p, heap(p) \leftarrow m]$	$p \notin \mathrm{dom}\, heap \wedge \mathrm{rtt}(p) = C \wedge\ m = \{f \leftarrow zero(\mathrm{ctt}(f)) \mid f \in fields_\Gamma(C)\}$
$l{:=}\texttt{io}(p)$	$s[ext \leftarrow ext', l \leftarrow i]$	$i = inp([\![p]\!]s, ext) \wedge ext' = ext \cdot [i{:=}\texttt{io}([\![p]\!]s)]$
$t_1; t_2$	r	$\Gamma \vdash s \xrightarrow{t_1} q \wedge \Gamma \vdash q \xrightarrow{t_2} r$
$\texttt{while}(e)\{t_1\}$	r	$[\![e]\!]s \wedge \Gamma \vdash s \xrightarrow{t_1} q \wedge \Gamma \vdash q \xrightarrow{t} r$
$\texttt{while}(e)\{t_1\}$	s	$\neg[\![e]\!]s$

Table 1. Operational semantics. Statement t of Γ (first column) performs a transition from state s to the terminal state (second column), provided that the antecedents (third column) hold.

3.2 Correspondence between Original and Transformed Program

A refactoring R is defined by a transformation function μ_R, which maps a program Γ to the refactored program $\Gamma' \equiv \mu_R(\Gamma)$ and occurrences t of statements in Γ to their respective counterparts: $t' \equiv \mu_R(t)$. For simplicity, we assume that μ_R is applied only to programs that satisfy the essential applicability conditions of R. Consequently, μ_R can be assumed to yield syntactically-correct programs.

Γ and Γ' are equivalent if they perform the same I/O operations, that is, if $r.ext = r'.ext$ for their respective terminal states r and r':

$$(\Gamma \vdash ini \xrightarrow{code_\Gamma} r) \Rightarrow \exists r' : (\Gamma' \vdash ini \xrightarrow{code_{\Gamma'}} r') \wedge r.ext = r'.ext \qquad (1)$$

For simplicity, we consider only terminating executions here. Non-terminating programs can be handled by considering prefixes of executions of Γ.

Simulation Method. Although Implication (1) expresses external equivalence of Γ and Γ', it is not useful for an equivalence proof because it is inaccessible to any inductive argument. This problem is avoided by using the simulation method for proving correspondence of two transition systems $\Gamma \vdash \_ \xrightarrow{\_} \_$ and $\Gamma' \vdash \_ \xrightarrow{\mu_R(\_)} \_$. To apply the simulation method, one defines a correspondence relation β_R on the states of the transition systems and proves that if both systems are in corresponding states and make a transition, they again reach corresponding states:

$$(\Gamma \vdash s \xrightarrow{t} r) \wedge \beta_R(s, s') \Rightarrow \exists r' : (\Gamma' \vdash s' \xrightarrow{\mu_R(t)} r') \wedge \beta_R(r, r') \tag{2}$$

For Implication (2) to imply the general condition (1), two sanity conditions must hold. First, β_R must imply identity of the *ext* parts of the states so that *external equivalence* is guaranteed: $\forall s, s' : \beta_R(s, s') \Rightarrow s.ext = s'.ext$. Second, the initial states of the program executions must correspond (*initial correspondence*): $\beta_R(ini, ini)$.

To prove that a refactoring R is correct, it suffices to devise a correspondence relation β_R and prove that it satisfies Implication (2) and the two sanity conditions. This method works for many refactorings, but cannot handle several interesting refactorings as we discuss next.

General Correspondence Requirement for Refactorings. Implication (2) is too strong for several important refactorings for the following reasons.

(a) Some refactorings do not define a statement $\mu(t)$ for each statement t of the original program[1]. Consider for instance the inlining of a local variable: $\mu(l{:=}f(p_1);\ l{:=}g(p_2)) \equiv l{:=}g(p_2[f(p_1)/l])$. In this case, $l{:=}f(p_1)$ and $l{:=}f(p_2)$ alone do not have corresponding statements in the refactored program. Therefore, $\mu(l{:=}f(p_1))$ and $\mu(l{:=}f(p_2))$ are undefined, and Implication (2) cannot be applied to these statements individually.

(b) Conversely, some refactorings transform a statement t of the original program into several non-consecutive statements in the transformed program. For instance, when unrolling a loop, the loop body t is duplicated and placed at different locations in the program, namely before and inside the loop. Again, $\mu(t)$ is not well-defined.

(c) Different statements in the original program may require different correspondence relations. Consider for instance the split of a variable l_0 into two fresh variables l_1 and l_2 in $t_0 \equiv t_1; t_2$:

$$\mu(t_1) \equiv l_1{:=}l_0;\ \underbrace{t_1[l_1/l_0]}_{t_b} \qquad\qquad \mu(t_2) \equiv l_2{:=}l_1;\ \underbrace{t_2[l_2/l_0]}_{t_d};\ l_0{:=}l_2$$

This refactoring requires a different correspondence relation for each active scope of the new variables. During the execution of statement t_b, l_0 and l_1 correspond, whereas l_0 and l_2 in general do not correspond since l_2 is not yet initialized. However, during the execution of statement t_d, l_0 and l_2 correspond, whereas l_1 is not used any more. Implication (2) does not permit such variations of the correspondence relation.

To address these problems, we allow the correspondence relation to be different at each program point: $\beta^{\text{before}}_{[t,t']}(s, s')$ is the correspondence between a state s before the execution of a statement t in the original program Γ and a state s' before the execution of a statement t' in the transformed program Γ'. $\beta^{\text{after}}_{[t,t']}(s, s')$ is the analogous correspondence after the execution of t and t'. External equivalence can then be proved using the following implication:

[1] We omit the subscript R when the refactoring is clear from the context.

$$(\Gamma \vdash s \xrightarrow{t} r) \wedge \beta^{\text{before}}_{[t,t']}(s, s') \Rightarrow \exists r' : (\Gamma' \vdash s' \xrightarrow{t'} r') \wedge \beta^{\text{after}}_{[t,t']}(r, r') \qquad (3)$$

Implication (3) implies the general condition (1) if the two sanity conditions hold, namely initial correspondence, $\beta^{\text{before}}_{[code_\Gamma, code_{\Gamma'}]}(ini, ini)$, and external equivalence, $\forall r, r' : \beta^{\text{after}}_{[code_\Gamma, code_{\Gamma'}]}(r, r') \Rightarrow r.ext = r'.ext$.

Besides these sanity conditions, the correspondence relation is not constrained. Therefore, Implication (3) provides enough flexibility to handle even complex program transformations. In particular, one is free to choose the statements t and t' to be related. For all statements t' in the refactored program Γ' that are not related to a statement t in the original program Γ, $\beta^{\text{before}}_{[t,t']}$ is simply defined to be empty.

For "Split Variable", we say that (a sub-statement of) t_1 in the original is comparable to (the corresponding sub-statement of) t_b in the transformed program and the same for t_2 and t_d. Moreover, a statement t outside t_0 that is not affected by the refactoring is comparable to its transformation $\mu(t)$. For statements t and t' that are not comparable, $\beta^{P}_{[t,t']}$ is empty. For comparable statements, corresponding states have an identical heap and identical I/O sequences. The requirement for local variables depends on whether t is (a sub-statement of) t_1, (a sub-statement of) t_2, or outside t_0.

In summary, we prove correctness of a refactoring R by devising a correspondence relation β_R that satisfies Implication (3) as well as initial correspondence and external equivalence. If β_R is simple enough, Implications (3) and (2) coincide. This will be the case for "Move Field", which we discuss in the next section. Examples that require the general correspondence, Implication (3), are presented in our technical report [2].

4 Example: "Move Field"

In this section, we apply our technique to "Move Field" [10]. This refactoring is interesting because it requires non-trivial correctness conditions and is therefore not supported by existing approaches. We describe "Move Field", present an appropriate correspondence relation as well as the required correctness conditions, and prove correctness of the refactoring.

4.1 The Refactoring

"Move Field" removes the declaration of a field f from a class *Source*, inserts it into a different class *Target*, and adjusts all accesses to f. To be able to access f after the refactoring, there must be a *Target* object for each *Source* object. For simplicity, we assume that there is a field *target* in *Source* that points to the corresponding *Target* of each *Source* object. A generalization to arbitrary associations between *Source* and *Target* objects is possible by using ghost fields to express the association.

Transformation Function. We assume that the classes *Source* and *Target* exist and that *Source* declares the distinct fields f and *target*. We assume further that *Target* does not declare a field f.

An original program $\mathit{\Gamma}$ and the transformed program $\mu(\mathit{\Gamma})$ are identical except for the following aspects: (a) The field f is moved from *Source* to *Target*:

$$\begin{aligned} \mathit{fields}_{\mu(\mathit{\Gamma})}(\mathit{Source}) &= \mathit{fields}_{\mathit{\Gamma}}(\mathit{Source}) - \{f\} \\ \mathit{fields}_{\mu(\mathit{\Gamma})}(\mathit{Target}) &= \mathit{fields}_{\mathit{\Gamma}}(\mathit{Target}) \cup \{f\} \end{aligned}$$

(b) Accesses to f are redirected via the *target* field:

$$\begin{aligned} \mu(l{:=}o.f) &\equiv \mathit{tmp}{:=}o.\mathit{target};\ l{:=}\mathit{tmp}.f \\ \mu(o.f{:=}e) &\equiv \mathit{tmp}{:=}o.\mathit{target};\ \mathit{tmp}.f{:=}e \end{aligned}$$

where $\mathrm{ctt}(o) = \mathit{Source}$ and *tmp* is a fresh variable.

Since “Move Field” either leaves statements of the original program unchanged or replaces them by simple sequential compositions, this refactoring does not require the general correspondence requirement, Implication (3). We can prove correctness using the simpler Implication (2). We present an appropriate correspondence relation in the next subsection.

4.2 Correspondence Relation

Intuitively, the correspondence relation for “Move Field” requires that $o.f$ in the original program and $o.\mathit{target}.f$ in the transformed program hold the same values. However, this correspondence requires $o.\mathit{target}$ to be different from `null`, which is not the case for a newly allocated object o before $o.\mathit{target}$ has been initialized. To handle object initialization, we use a more relaxed correspondence. If $\mathit{target} =$ `null`, f in the original program must hold its zero-equivalent default value, that is, be also uninitialized.

For two states s and s', the correspondence relation $\beta(s, s')$ yields true if and only if:

1. All local variables, except for temporary variables introduced by the refactoring, hold the same value in s and s': $s.\mathit{vars} = s'.\mathit{vars}|_{(\mathrm{dom}\, s.\mathit{vars})}$, where the domain restriction operator $F|_D$ restricts the domain of function F to D.
2. All references x that are not of type *Source* or *Target* are mapped to the same objects: $\mathrm{dom}\, s.\mathit{heap} = \mathrm{dom}\, s'.\mathit{heap}$ and $s.\mathit{heap}(x) = s'.\mathit{heap}(x)$. For a *Source* reference x, $s.\mathit{heap}(x)(f)$ corresponds to $s'.\mathit{heap}(s'.\mathit{heap}(x)(\mathit{target}))(f)$ or $\mathit{zero}(\mathrm{ctt}(f))$, depending on whether $s'.\mathit{heap}(x)(\mathit{target}) \neq$ `null` or not. All other field values are identical. We also have to encode that *Target* objects do not have an f field in the original program. That is, for all references $x \in \mathrm{dom}\, s.\mathit{heap}$ and object values $m \equiv s.\mathit{heap}(x)$ and $m' \equiv s'.\mathit{heap}(x)$, we require:

$$m = \begin{cases} m'[f \leftarrow s'.\mathit{heap}(m'(\mathit{target}))(f)] & \text{if } \mathrm{rtt}(x) = \mathit{Source} \wedge m'(\mathit{target}) \neq \texttt{null} \\ m'[f \leftarrow \mathit{zero}(\mathrm{ctt}(f))] & \text{if } \mathrm{rtt}(x) = \mathit{Source} \wedge m'(\mathit{target}) = \texttt{null} \\ m'[f \leftarrow \mathit{undef}] & \text{if } \mathrm{rtt}(x) = \mathit{Target} \\ m' & \text{otherwise} \end{cases} \tag{4}$$

3. The I/O part of both states is identical: $s.\mathit{ext} = s'.\mathit{ext}$.

This correspondence relation satisfies initial correspondence because there are no references in the initial heap *ini.heap*. External equivalence is trivially satisfies by the third requirement.

4.3 Correctness Conditions

The correctness conditions for "Move Field" have to guarantee that the refactoring preserves the external program behavior. In the following, we suggest sufficient correctness conditions and explain how they can be checked a-posteriori.

A-posteriori Conditions for "Move Field". Any conditions that allow one to prove Implication (2) for any statement t in the original program are possible correctness conditions. A pragmatic approach is to start with the proof of Implication (2) and to determine the weakest conditions that are necessary for the induction to go through. However, these weakest conditions are often difficult to express as assertions or difficult to check. Therefore, one typically has to devise stronger correctness conditions that are easier to express and to check.

In the "Move Field" example, Implication (2) holds trivially for most statements of the original program since $t \equiv \mu(t)$. The interesting cases are if t is (a) a read access to the moved field f, (b) an update of f, or (c) an update of *target*. For these cases, we present sufficient (but not weakest) a-posteriori conditions in the following (o is a variable of type *Source*):

(a) $t \equiv l{:=}o.f$: The statement t reads $o.f$ in the original program. It terminates normally if o is different from `null`. The correctness conditions must guarantee that the transformed statement $\mu(t) \equiv \mathit{tmp}{:=}o.\mathit{target};\ l{:=}\mathit{tmp}.f$ behaves correspondingly. This is the case if $o.\mathit{target}$ is different from `null`. Therefore, we introduce the following assertion before $\mu(t)$:

$$\texttt{assert}\ o.\mathit{target} \neq \texttt{null}; \tag{5}$$

(b) $t \equiv o.f{:=}e$: An update of $o.f$ in the original program changes the value of the f field of exactly one *Source* object, namely o. To achieve the corresponding behavior for the transformed statement $\mu(t) \equiv \mathit{tmp}{:=}o.\mathit{target};\ \mathit{tmp}.f{:=}e$, we require that the updated *Target* object is not shared by several *Source* objects. Moreover, we have to make sure the association exists, that is, $o.\mathit{target}$ is different from `null`. Therefore, we introduce the following assertion before $\mu(t)$:

$$\texttt{assert}\ o.\mathit{target} \neq \texttt{null} \wedge \forall \mathit{Source}\ p : p.\mathit{target} = o.\mathit{target} \Rightarrow p = o; \qquad (6)$$

Quantifiers over objects range over non-null allocated objects. Both t and $\mu(t)$ terminate normally if and only if o is different from `null`.

(c) $t \equiv o.\mathit{target}{:=}e$: An update of $o.\mathit{target}$ in the original program associates the *Source* object o with another *Target* object. For the transformed program, this means that o will potentially access a different f field. Updates of *target* are not transformed, that is, $t \equiv \mu(t)$. They lead to corresponding terminal states in the following cases:

– $o.\mathit{target}$ is set to `null` ($e =$ `null`), and the old $o.\mathit{target}$ either is already `null` (that is, the value remains unchanged), or $o.\mathit{target}.f$ holds the default value for f's type.

– $o.\mathit{target}$ is set from `null` to a proper *Target* object ($e \neq$ `null`) whose f field holds the default value.

– $o.\mathit{target}$ is set to a proper *Target* object whose f field holds the same value as the old $o.\mathit{target}.f$.

These conditions are expressed by the following assertion, which is inserted before $\mu(t)$:

$$\begin{array}{l} \texttt{assert}\ (e = \texttt{null} \wedge (o.\mathit{target} = \texttt{null} \vee o.\mathit{target}.f = \mathit{zero}(\mathrm{ctt}(f)))) \vee \\ \qquad (e \neq \texttt{null} \wedge (\ (o.\mathit{target} = \texttt{null} \wedge \mathit{zero}(\mathrm{ctt}(f)) = e.f) \vee \\ \qquad\qquad (o.\mathit{target} \neq \texttt{null} \wedge o.\mathit{target}.f = e.f)\ )\ ); \end{array} \qquad (7)$$

Both t and $\mu(t)$ terminate normally if and only if o is different from `null`.

Checking the Conditions. Assertion (5) is amenable to both runtime assertion checking and static verification. Alternatively, non-null types [9] can be used to check the condition syntactically.

The second conjunct of Assertion (6) is difficult to check at runtime. It requires code instrumentation to keep track of the number of *Source* objects that point to a *Target* object. Static verification is possible for this assertion, for instance, using the ownership discipline of the Boogie methodology [16]. A syntactic alternative is to use pointer confinement type systems such as ownership types and their descendants [5, 7], or linear types [8].

Assertion (7) is straightforward to check at runtime or by static verification. Nevertheless, it seems reasonable to impose additional restrictions to enforce this correctness condition. For instance in Java, the field *target* can be declared `final`, which ensures that *target* cannot be changed after its first initialization. With this restriction, the condition of Assertion (7) can be simplified to $e = \texttt{null} \vee \mathit{zero}(\mathrm{ctt}(f)) = e.f$.

4.4 Correctness Proof

In this subsection, we prove that "Move Field" is correct. That is, the transformed program $\mu(\Gamma)$ performs the same sequence of I/O operations as the original program Γ if the correctness conditions hold.

For the proof, we use the following *auxiliary lemma*: If the states s and s' correspond then the evaluation of an expression e yields the same value in both states: $\beta(s, s') \Rightarrow [\![e]\!]s = [\![e]\!]s'$. This lemma is a direct consequence of the definition of β ($\beta(s, s') \Rightarrow s.vars = s'.vars$) and the fact that expressions do not contain field accesses.

With this auxiliary lemma, the cases for local variable update, object allocation, I/O operation, sequential composition, and loop are straightforward and therefore omitted. In the following, we sketch the proof for the interesting cases: field read and field update. The other cases and further details of the proof are presented in our technical report [2].

We prove correctness by showing Implication (2) for any statement t in the original program Γ and any states s, s', and r. The proof runs by induction on the shape of the derivation tree for $\Gamma \vdash s \xrightarrow{t} r$. For each case, we present a terminal state r' and then show (a) that $\mu(\Gamma) \vdash s' \xrightarrow{\mu(t)} r'$ is a valid transition and (b) that r and r' correspond.

Field Read. Consider $t \equiv l{:=}o.\mathit{fld}$. If $\mathit{fld} \neq f$, the statement is not transformed and we have $\beta(s, s') \Rightarrow s.\mathit{heap}(x)(\mathit{fld}) = s'.\mathit{heap}(x)(\mathit{fld})$ for all x. Therefore, the field accesses yield the same value. Consequently $r' = r$ satisfies Implication (2).

For $\mathit{fld} = f$, the original and transformed statements are:

$$t \equiv l{:=}o.f \qquad \text{and} \qquad t' \equiv \mathit{tmp}{:=}o.\mathit{target};\ l{:=}\mathit{tmp}.f$$

They lead to the following terminal states r and r' from s and s' respectively.

$$\begin{aligned} r &= s[l \leftarrow s.\mathit{heap}(s.\mathit{vars}(o))(f)] \\ r' &= s'[l \leftarrow s'.\mathit{heap}(s'.\mathit{heap}(s'.\mathit{vars}(o))(\mathit{target}))(f), \\ &\quad\ \mathit{tmp} \leftarrow s'.\mathit{heap}(s'.\mathit{vars}(o))(\mathit{target})] \end{aligned}$$

The antecedents that have to hold for the transition in the transformed program, $\Gamma' \vdash s' \xrightarrow{\mathit{tmp}:=o.\mathit{target};\ l:=\mathit{tmp}.f} r'$, are:

$$s'.\mathit{vars}(o) \neq \texttt{null} \qquad \text{and} \qquad s'.\mathit{heap}(s'.\mathit{vars}(o))(\mathit{target}) \neq \texttt{null}$$

The first antecedent is implied by the antecedent $s.\mathit{vars}(o) \neq \texttt{null}$ of the corresponding transition in the original program because $\beta(s, s')$ implies $s.\mathit{vars} = s'.\mathit{vars}$. The second antecedent is directly guaranteed by the correctness condition preceding t', Assertion (5).

Next, we prove $\beta(r, r')$. r and r' have the same *heap* and *ext* components as s and s', respectively. Besides the temporary variable *tmp*, which is irrelevant according to the definition of β, their *vars* components differ from the variables of the initial states only for variable l. Therefore, it suffices to show $r.\mathit{vars}(l) = r'.\mathit{vars}(l)$, that is, we prove the following equation:

$$s.\mathit{heap}(s.\mathit{vars}(o))(f) = s'.\mathit{heap}(s'.\mathit{heap}(s'.\mathit{vars}(o))(\mathit{target}))(f)$$

This equation is directly implied by line 1 in Equation (4), which applies because type safety guarantees that $\mathrm{rtt}(o) = \mathit{Source}$ holds and Assertion (5) ensures $s'.\mathit{heap}(s'.\mathit{vars}(o))(\mathit{target}) \neq \mathtt{null}$.

Field Update. Consider $t \equiv o.\mathit{fld}{:=}e$. We present the proof for updates of f and of *target* in the following. For all other fields, the proof is trivial.

Updates of f. For $\mathit{fld} = f$, the original and transformed statements are:

$$t \equiv o.f{:=}e \qquad \text{and} \qquad t' \equiv \mathit{tmp}{:=}o.\mathit{target};\ \mathit{tmp}.f{:=}e$$

They lead to the following terminal states r and r' from s and s', respectively.

$$\begin{aligned} r &= s[\mathit{heap}(\mathit{vars}(o))(f) \leftarrow [\![e]\!]s] \\ r' &= s'[\mathit{heap}(s'.\mathit{heap}(s'.\mathit{vars}(o))(\mathit{target}))(f) \leftarrow [\![e]\!]s, \\ &\qquad \mathit{tmp} \leftarrow s'.\mathit{heap}(s'.\mathit{vars}(o))(\mathit{target})] \end{aligned}$$

where we used the auxiliary lemma to show that the evaluation of e is not affected by the transformation.

The antecedents that have to hold for the transition in the transformed program, $\Gamma' \vdash s' \xrightarrow{\mathit{tmp}:=o.\mathit{target};\ \mathit{tmp}.f:=e} r'$, are:

$$s'.\mathit{vars}(o) \neq \mathtt{null} \qquad \text{and} \qquad s'.\mathit{heap}(s'.\mathit{vars}(o))(\mathit{target}) \neq \mathtt{null}$$

The first antecedent is implied by the antecedent $s.\mathit{vars}(o) \neq \mathtt{null}$ of the corresponding transition in the original program and $\beta(s, s')$. The second antecedent is directly guaranteed by the correctness condition preceding t', Assertion (6).

Next, we prove $\beta(r, r')$. Besides the temporary variable *tmp*, which is irrelevant according to the definition of β, r and r' have the same *vars* and *ext* components as s and s', respectively. We get $\operatorname{dom} r.\mathit{heap} = \operatorname{dom} r'.\mathit{heap}$ and $r.\mathit{heap}(x) = r'.\mathit{heap}(x)$ for all references x from $\beta(s, s')$ and the definitions of r and r'. Therefore, it suffices to show that Equation (4) holds for all references $x \in \operatorname{dom} r.\mathit{heap}$. We show this by a case distinction on the value of x.

Case (i): $x = s'.\mathit{vars}(o)$. We have $\mathrm{rtt}(x) = \mathit{Source}$ (by type safety) and $s'.\mathit{heap}(x)(\mathit{target}) \neq \mathtt{null}$ by Assertion (6). Therefore, line 1 in Equation (4) applies. Since only the f field is updated, it suffices to prove:

$$r.\mathit{heap}(x)(f) = r'.\mathit{heap}(x)[f \leftarrow r'.\mathit{heap}(s'.\mathit{heap}(x)(\mathit{target}))(f)](f)$$

(We used $s'.\mathit{heap}(x)(\mathit{target}) = r'.\mathit{heap}(x)(\mathit{target})$, which holds because *target* is not updated.) The right-hand side of the above equation can be trivially simplified to $r'.\mathit{heap}(s'.\mathit{heap}(x)(\mathit{target}))(f)$. Using the definitions of r and r' above reveals that both sides of the equation evaluate to $[\![e]\!]s$, which concludes Case (i).

Case (ii): $x \neq s.vars(o)$. Since t updates a field of a *Source* object, this case is trivial if $\mathrm{rtt}(x) \neq \mathit{Source}$. For $\mathrm{rtt}(x) = \mathit{Source}$, we continue as follows. If $s'.heap(x)(target) = \texttt{null}$, line 2 of Equation (4) follows directly from $\beta(s, s')$. Otherwise, line 1 in Equation (4) applies, that is, we have to prove:

$$r.heap(x) = r'.heap(x)[f \leftarrow r'.heap(r'.heap(x)(target))(f)]$$

By the definition of r and the assumption of Case (ii), we get $r.heap(x) = s.heap(x)$. By the definition of r' and type safety, we get $r'.heap(x) = s'.heap(x)$ because x is a *Source* object and t' updates a field of a *Target* object. By using these two equalities, we can reduce our proof goal to:

$$s.heap(x) = s'.heap(x)[f \leftarrow r'.heap(s'.heap(x)(target))(f)]$$

Assertion (6) implies $s'.heap(x)(target) \neq s'.heap(s'.vars(o))(target)$. Therefore, we get $r'.heap(s'.heap(x)(target)) = s'.heap(s'.heap(x)(target))$ by the definition of r'. This condition together with $\beta(s, s')$ implies the above equation. This concludes Case (ii) and, thereby, the case for updates of f.

Updates of target. For $\mathit{fld} = target$, the original and transformed statements are identical. They lead to the following terminal states r and r' from s and s', respectively.

$$\begin{aligned} r &= s[heap(s.vars(o))(target) \leftarrow [\![e]\!]s] \\ r' &= s'[heap(s'.vars(o))(target) \leftarrow [\![e]\!]s] \end{aligned}$$

The proof of the antecedent for the transition $\Gamma' \vdash s' \xrightarrow{t} r'$ is analogous to the case for $\mathit{fld} = f$.

Next, we prove $\beta(r, r')$. This part is mostly analogous to the case for $\mathit{fld} = f$. The only new property we have to show is that $r.heap(x)$ and $r'.heap(x)$ satisfy Equation (4), where $x = s.vars(o) = s'.vars(o)$. From type safety, we know $\mathrm{rtt}(x) = \mathit{Source}$ because *target* is a field of class *Source*. Therefore, line 1 or line 2 of Equation (4) might apply. We continue by case distinction.

Case (i): $r'.heap(x)(target) = \texttt{null}$. From the definition of r' and the assumption of Case (i), we get $[\![e]\!]s = \texttt{null}$. Therefore, line 2 in Equation (4) applies and we have to prove:

$$r.heap(x) = r'.heap(x)[f \leftarrow zero(\mathrm{ctt}(f))]$$

Using the definitions of r and r' as well as $\beta(s, s')$ reveals that this is exactly the case if $r.heap(x)(f) = zero(\mathrm{ctt}(f))$ or, equivalently:

$$s.heap(x)(f) = zero(\mathrm{ctt}(f))$$

Due to the assumption of this case, Assertion (7) is known to guarantee:

$$s'.heap(x)(target) = \texttt{null} \lor s'.heap(s'.heap(x)(target))(f) = zero(\mathrm{ctt}(f))$$

If the first disjunct holds, $\beta(s, s')$ implies $s.heap(x)(f) = zero(\mathrm{ctt}(f))$ by line 2 in Equation (4). Otherwise, we get this property by line 1 in Equation (4). This concludes Case (i).

Case (ii): $r'.\mathit{heap}(x)(\mathit{target}) \neq \texttt{null}$. Analogously to Case (i), we derive $[\![e]\!]s \neq \texttt{null}$. Therefore, line 1 in Equation (4) applies and we have to prove:

$$r.\mathit{heap}(x) = r'.\mathit{heap}(x)[f \leftarrow r'.\mathit{heap}(r'.\mathit{heap}(x)(\mathit{target}))(f)]$$

Again, using the definitions of r and r' as well as $\beta(s, s')$ reveals that this is exactly the case if the following equation holds.

$$s.\mathit{heap}(x)(f) = s'.\mathit{heap}([\![e]\!]s)(f)$$

Due to the assumption of this case, Assertion (7) is known to guarantee:

$$\begin{array}{l}(s'.\mathit{heap}(x)(\mathit{target}) = \texttt{null} \wedge \mathit{zero}(\mathrm{ctt}(f)) = s'.\mathit{heap}([\![e]\!]s)(f)) \vee \\ (s'.\mathit{heap}(x)(\mathit{target}) \neq \texttt{null} \wedge s'.\mathit{heap}(s'.\mathit{heap}(x)(\mathit{target}))(f) = s'.\mathit{heap}([\![e]\!]s)(f))\end{array}$$

If the first disjunct of this condition holds then line 2 in Equation (4) implies $s.\mathit{heap}(x)(f) = \mathit{zero}(\mathrm{ctt}(f))$, rendering it equal to $s'.\mathit{heap}([\![e]\!]s)(f)$.

If the second disjunct holds, line 1 in Equation (4) yields $s.\mathit{heap}(x)(f) = s'.\mathit{heap}(s'.\mathit{heap}(x)(\mathit{target}))(f)$. This concludes Case (ii) and, thereby, the case for updates of *target*. □

5 Related Work

There is a vast literature on refactoring, but little on its formalization. Most of the related work discusses the design rationale [12] of individual refactorings or treats refactorings on a syntactic level [14]. In this section, we discuss work that is geared towards reasoning about refactorings.

Opdyke [20] mentions explicitly that refactorings have correctness conditions and argued informally for the correctness of refactorings. He defined the notion of equivalence for refactorings that is also used in this paper, namely identical sequences of I/O operations.

The Smalltalk Refactoring Browser [21] samples the program at runtime to estimate properties that are difficult or impossible to infer statically. Samples are taken before the refactoring because their results are sometimes needed for the transformation itself. For instance, because Smalltalk is untyped, the classes that are receivers of a certain method call have to be determined by sampling before "Rename Method" can be applied. Representative program executions must be available for this approach, which is a serious restriction as explained in Sect. 1. While typed languages remove the need to sample programs in order to carry out the refactoring, sampling the program before refactoring could still be used to check correctness conditions a-priori. However, our approach of adding a-posteriori conditions as assertions to the refactored program has several advantages. It reduces the dependence on a complete unit test suite, enables static verification, and improves program documentation.

Streckenbach and Snelting [22] use the results of static or dynamic analyses to determine possible refactorings of class hierarchies automatically. The analyses guarantee that the refactorings are correct. Two class hierarchies are considered equivalent if the behavior observable by clients is identical. This notion of equivalence is too restrictive for many non-local refactorings such as renaming a public field or method. In a similar approach, Logozzo and Cortesi [18] solve this problem by defining explicitly what aspects of the program behavior are observable. They use abstract interpretation to determine possible refactorings of class hierarchies. We do not aim at finding possible refactorings automatically, but require the user to apply the desired refactorings. Our approach supports complex refactorings whose correctness conditions cannot be checked efficiently by static analyses.

Cornélio [6] uses a refinement relation as the equivalence criterion for refactorings. He shows correctness by decomposing a refactoring into various refinement steps. The refinement relation of the calculus [19] per se does not guarantee external equivalence however. In particular, visible intermediary states may be different. We solve this problem by introducing an explicit I/O model. Cornélio's work does not support important language features such as references and therefore avoids some of the most challenging problems. Our formalism is based on an operational semantics, which allows us to handle realistic languages.

Refactorings have also been applied to models of the program such as a UML diagrams rather than to the source code. There are various efforts to formalize such refactorings [4, 11, 17, 23]. Our work focuses on source code because the main application of refactoring is changing code quickly and correctly with all its intricacies For instance, method calls cannot be adjusted in class diagrams.

Like our work, investigations on representation independence [1] aim at proving that certain changes in the code preserve its external behavior. Representation independence relies on encapsulation to guarantee that modifications of the internal representation of a module cannot be observed by client code. In contrast, refactorings are typically not local to a module and, therefore, require very different correctness conditions.

6 Conclusion

We have presented a technique that guarantees that the application of a refactoring preserves the external behavior of the program if the transformed program satisfies the refactoring's correctness conditions. These conditions are added to the transformed program and can be used for runtime assertion checking, to generate unit tests, and for static verification. We applied our approach successfully to 15 representative refactorings [2].

An important virtue of our approach is that it automatically improves program documentation by adding assertions. Thereby, it prepares the program for the application of specification-based test tools and program verifiers.

We have implemented a prototype for "Move Field" for Spec# in Visual Studio and for JML in Eclipse. As future work, we plan to develop a more comprehensive refactoring tool based on the technique presented here. Moreover, we plan to investigate how the generated assertions can be refactored into invariants and method contracts.

References

1. A. Banerjee and D. A. Naumann. Ownership confinement ensures representation independence for object-oriented programs. *J. ACM*, 52(6):894–960, 2005.
2. F. Bannwart. Changing software correctly. Technical Report 509, Department of Computer Science, ETH Zürich, 2006.
3. M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# Programming System: An Overview. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors, *CASSIS*, volume 3362 of *LNCS*, pages 49–69. Springer-Verlag, 2005.
4. P. L. Bergstein. Object-preserving class transformations. In *OOPSLA*, pages 299–313. ACM Press, 1991.
5. D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *OOPSLA*, pages 48–64. ACM Press, 1998.
6. M. Cornélio. *Refactorings as Formal Refinements.* PhD thesis, Universidade de Pernambuco, 2004.
7. W. Dietl and P. Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology (JOT)*, 4(8):5–32, 2005.
8. M. Fähndrich and R. DeLine. Adoption and focus: practical linear types for imperative programming. In *PLDI*, pages 13–24. ACM Press, 2002.
9. M. Fähndrich and K. R. M. Leino. Declaring and checking non-null types in an object-oriented language. In *OOPSLA*, pages 302–312. ACM Press, 2003.
10. M. Fowler. *Refactoring: improving the design of existing code.* Addison-Wesley, 1999.
11. R. Gheyi, T. Massoni, and P. Borba. An abstract equivalence notion for object models. *Electr. Notes Theor. Comput. Sci.*, 130:3–21, 2005.
12. J. Kerievsky. *Refactoring to Patterns.* Addison-Wesley Professional, August 2004.
13. J. R. Kiniry and D. R. Cok. ESC/Java2: Uniting ESC/Java and JML. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors, *CASSIS*, volume 3362 of *LNCS*, pages 108–128. Springer-Verlag, 2005.
14. R. Lämmel. Towards Generic Refactoring. In *Workshop on Rule-Based Programming (RULE).* ACM Press, 2002.
15. G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06-rev28, Iowa State University, 2005.
16. K. R. M. Leino and P. Müller. Object invariants in dynamic contexts. In M. Odersky, editor, *ECOOP*, volume 3086 of *LNCS*, pages 491–516. Springer-Verlag, 2004.
17. K. J. Lieberherr, W. L. Hürsch, and C. Xiao. Object-extending class transformations. *Formal Aspects of Computing*, (6):391–416, 1994.
18. F. Logozzo and A. Cortesi. Semantic hierarchy refactoring by abstract interpretation. In E. A. Emerson and K. S. Namjoshi, editors, *VMCAI*, volume 3855 of *LNCS*, pages 313–331. Springer-Verlag, 2006.
19. C. Morgan. *Programming from specifications.* Prentice-Hall, 1990.
20. W. F. Opdyke. *Refactoring object-oriented frameworks.* PhD thesis, University of Illinois at Urbana-Champaign, 1992.
21. D. B. Roberts. *Practical analysis for refactoring.* PhD thesis, University of Illinois at Urbana-Champaign, 1999.
22. M. Streckenbach and G. Snelting. Refactoring class hierarchies with KABA. In *OOPSLA*, pages 315–330. ACM Press, 2004.
23. L. Tokuda and D. Batory. Evolving object-oriented designs with refactorings. In *Automated Software Engineering*, pages 174–182. IEEE Computer Society, 1999.