Equivalence checking for Design-Level Refactoring Changes

ANONYMOUS AUTHOR(S)

Equivalence checking is the problem of deciding whether two program are semantically equivalent, i.e., for all inputs, they generate the same output. Equivalence checking has been extensively studied to compare two functions with the same signature. However, it is common to change a function's signature while evolving software.

In this work, we extend the notion of equivalence checking to compare functions with differing signatures. We introduce PolyCheck, a technique to find the equivalence of functions with differing signatures. PolyCheck builds upon Differential Symbolic Execution algorithm. The core idea to check equivalence: is it possible to transform all possible call sites such that the return values are the same? PolyCheck aims to build a transformation function that satisfies this condition. To evaluate PolyCheck, we create a benchmark of 8 pairs of functions with differing signatures. Our technique is find the equivalence functions in 4 cases.

ACM Reference Format:
Anonymous Author(s). 2025. Equivalence checking for Design-Level Refactoring Changes. 1, 1 (December 2025), 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Equivalence checking is the problem of deciding whether two programs are equivalent – i.e., two programs result in the same output, given the same input. Equivalence checking has many applications, such as checking the safetly of refactorings [7], evolving test suites, and tracking software evolution. Existing work in equivalence checking analyses two functions with matching signatures – i.e., the same input arguments and return types. The result of the analysis is a binary decision equivalent, or non-equivalent. However, software evolution is often accompanied by small logic or design changes which updates the signature of the method. To the best of our knowledge, equivalence of methods having different signatures has not been studied.

To bridge the gap, we consider the equivalence of functions with differing signatures (termed Poly-Methods). We extend the notion of equivalence to Poly-Methods, is there a way to transform all possible call sites to the original method in such a way that the return value of the method is the same?.

Further, we present a technique to check the equivalence of Poly-Methods, called PolyCheck. PolyCheck builds upon the Differential Symbolic Execution (DSE) algorithm to handle cases where the function signature is different. It do so by capturing the symbolic summaries, and introducing an adapter function. How does the technique work, at a high level? Transformation functions which are uninterpreted functions, symbolic summaries, and SMT solvers.

Eval. Be specific: add 1 parameter

We make the following contributions:

(i) Extending the notion of equivalence to Poly-Methods

(ii) PolyCheck– A technique to check the equivalence of Poly-Methods.

(iii) A benchmark of 8 Poly-Methods which are equivalent, and a baseline over those.

ACM XXXX-XXXX/2025/12-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

2 Overview

We motivate the need for equivalence checking of functions with differing signatures via the evolution of a `max` function. The `max` function takes two integers as inputs, and returns the larger one of them – see Figure 1 (a). As software evolves and grows, even a simple function like this is prone to change for multiple reasons: to make the function reusable, more idiomatic (following best-practices), or extensible.

Prior works in equivalence checking can reason about the equivalence of 1 (a) and 1 (b), which have matching signatures. That is, for all possible values of x and y, $\max_a$ and $\max_b$ return the same value.

As prior techniques only consider methods with matching signatures, the equivalence of 1 (a) and 1 (c) is not defined, as $\max_c$ has an extra boolean parameter. The boolean parameter `absolute` is a typical feature flag, which switches on alternate behaviour if set. If the value of `absolute` is `true`, the absolute value of the two integers is compared. Notably, we can adapt callers of $\max_a$ to use $\max_c$ in a manner such that the resulting values are the same. We can do so by passing `false` for the parameter `absolute`. The methods $\max_a$ and $\max_c$ are equivalent if their call sites are transformed in the right manner, leading to software whose behaviour is unchanged. We extend the notion of equivalence for Poly-Methods in this manner.

PolyCheck: Checking the Equivalence of Poly-Methods. To reason about the equivalence of Poly-Methods, a tool must figure out how to transform all possible call sites to the original method, such that the return value is the same. To transform $\max_a$ to $\max_c$, we must reason about the value for `absolute` which achieves this condition. PolyCheck can finds the default value to `absolute` by extending the Differential Symbolic Execution (DSE) algorithm. First, PolyCheck computes the symbolic summaries of $\max_a$ and $\max_c$ using symbolic execution. PolyCheck determines the symbolic summary of $\max_a$ to be the following formula: $(x > y \wedge \text{return} == x) \vee (x \leq y \wedge \text{return} == y)$. (some explanation of the formula?). Similarly, PolyCheck determines the symbolic summary of $\max_c$ to be: $(\text{absolute} == \text{true} \wedge \ldots) \vee (absolute == \text{false} \wedge \max_a(a, b))$. Next, PolyCheck models `absolute` as an uninterpreted function with inputs `x` and `y`. Finally, it passes the formula $\max_a \implies \max_b$ to the SMT solver for checking. The SMT solver confirms that the formula is satisfiable, and returns the definition of `absolute = false`.

In this case, the value of `absolute` is a single constant, which has no dependencies on the other input parameters.

Checking the equivalence of Figure 1a and 1d is more complicated. The function $\max_d$ takes an additional parameter `equalReturn`, which is returned when `x` and `y` are equal. It is possible to transform all possible call sites of $\max_a$(x, y) to $\max_d$(x, y, y) – thus maintaining equivalent behaviour. This is because, when `x` and `y` are equal, the original method returns the value `y`. Passing `y` as the `equalReturn` value achieves the same effect, when `x` and `y` are equal.

In this case, the value of the new parameter `equalReturn` is determined by one of the arguments in the original method, `y`. This is a more complex case, as PolyCheck must reason about finding a transformation of `x` or `y` that could be passed as `equalReturn`.

3 Checking the Equivalence of Poly-Methods

In this section, we first extend the notion of equivalence to methods with differing signatures (Poly-Methods). Then, we proceed to present our solution to check the equivalence of Poly-Methods.

3.1 Equivalence of Poly-Methods

Two methods m_1 and m_2 with the same signature are equivalent, if for all possible input values, they produce the same output.

```
int max_a(int x, int y){
 if (x>y) return x;
  else return y;
}
```

(a) A function computing the larger of two integers

```
int max_b(int x, int y){
  return x>y?x:y;
}
```

(b) Equivalent to original

```
int max_c(int x, int y, boolean absolute){
  if (absolute)
    return abs(x)>abs(y)? x: y;
  return x>y? x:y;
}
```

(c) Equivalent Under Transformation:
max(x, y) → max(x, y, false)

```
int max_d(int x, int y, int equalReturn){
  if (x>y) return x;
  else if (y>x) return y;
  else return equalReturn;
}
```

(d) Equivalent Under Transformation:
max(x, y) → max(x, y, y)

Fig. 1. An evolution of a max function over integers. (a) The original function, (b) An idiomatic rewrite, (c) and (d) introducing parameter for extensibility

Definition 1 (Direct Equivalence). Let a method be written as $m : (T_1, T_2, \ldots, T_k) \longrightarrow T_r$, where $T_1, \ldots, T_k$ are parameter types and T_r is the return type. Two methods m_1 and m_2 with the *same* signature are *equivalent* iff

$$\forall (v_1, \ldots, v_k) \in T_1 \times \cdots \times T_k : \quad m_1(v_1, \ldots, v_k) = m_2(v_1, \ldots, v_k).$$

Definition 2 (Equivalence of Poly-Methods). Let the signature of m_1 be

$$m_1 : (T_1, \ldots, T_k) \longrightarrow T_r$$

and the signature of m_2 be

$$m_2 : (S_1, \ldots, S_j) \longrightarrow T_r.$$

A call site of m_1 has the form $m_1(e_1, \ldots, e_k)$. We say that m_1 and m_2 are *transformationally equivalent* ($m_1 \rightsquigarrow m_2$) iff there exists a transformation function

$$\tau : T_1 \times \cdots \times T_k \longrightarrow S_1 \times \cdots \times S_j$$

such that

$$\forall (v_1, \ldots, v_k) \in T_1 \times \cdots \times T_k : \quad m_1(v_1, \ldots, v_k) = m_2\big(\tau(v_1, \ldots, v_k)\big).$$

That is, Poly-Methods (m_1 and m_2) are equivalent, if all possible call sites to m_1 can be transformed to call m_2 such that, the return values are the same.

Not Commutative. Transformational Equivalence is not commutative: $m_1 \rightsquigarrow m_2$ does not imply $m_2 \rightsquigarrow m_1$. Consider the examples (a) and (c) from Figure 1. As described previously, $\text{max}_a \rightsquigarrow \text{max}_c$, under the transformation: max_a(x, y) ⟶ max_c(x, y, false). However, the reverse transformation is not possible, in the general case. Only when `absolute` is false, max_c can be replaced by a call to max_a.

Transformational Equivalence can be achieved between functions with the same signature. expand.

The transformation function τ used to establish equivalence between Poly-Methods need not be unique, for a given pair of methods. Consider the problem of checking equivalence between max_a and max_d (see Figure 1 a and d). We can rewrite all calls to max_a in at least two ways: max_a(x, y) ⟶ max_d(x, y, x), and max_a(x, y) ⟶ max_d(x, y, y). This is because, if `x` and `y` are equal, `equalReturn` can be set to either one.

Transformational Equivalence doesn't consider changes to return types. expand.

3.2 PolyCheck: Checking Transformational Equivalence

The PolyCheck algorithm builds upon the Differential Symbolic Execution algorithm. Given two methods m_1 and m_2, with the following signatures:

$$m_1 : (T_1, \ldots, T_k) \rightarrow T_r \quad m_2 : (S_1, \ldots, S_j) \rightarrow T_r.$$

From a birds-eye view, PolyCheck follows this process:

(1) Compute the symbolic summaries of m_1 and m_2: s_1 and s_2 respectively.
(2) Model a transformation function, such that $\tau(t_1, ..., t_k) = (s_1, ..., s_j)$
(3) Ask an SMT solver to check the satisfiability of the formula:

$$\forall (t_1, ..., t_k) \in T_1 \times \cdots \times T_k. \quad s_1(t_1, ..., t_k) \implies s_2(\tau(t1, ..., t_k))$$

(4) If the solver say that F is satisfiable, claim that m_1 and m_2 are equivalent. Else, claim that m_1 and m_2 are not equivalent.

3.2.1 Symbolic Summary. Symbolic summaries are computed after performing symbolic execution. Explain how this is a formula. With example.

3.2.2 Progressive Modelling of τ. Talk about how passing tau as a complex function to z3 can increase time complexity, or make it run for too long. Talk about how τ is progressively increased in complexity. Starting with a constant, then uninterpreted function over inputs. This helps with feasibility.

3.3 Implementation

expand. SMT solver - Z3. Symbolic execution engine - JavaPathFinder. DSE implementation. Symbolic execution depth parameters??[1].

4 Evaluation

8 add 1 parameter examples.

Highlight the limitation of PolyCheck- to generate turing-machine like transition functions.

Baselines:

(1) Test generation tools like: evo-suite, test-spark, diffblue.
(2) Symbolic execution based equivalence checking tools like ArDiff, PASDA, etc.

5 Related Work

Equivalence checking is a well-studied problem which has received attention from many researchers. We divide the related work into two broad categories: (1) formal equivalence checking, (2) unit-test based equivalence checking.

5.1 Symbolic Execution Based Equivalence Checking

Differential symbolic execution [5] is a key technique which lays the foundation for equivalence checking. The core idea is to compare symbolic summaries of two functions, and use an SMT solver to prove equivalence. Many works [1, 4, 6] improve upon this idea.

Several techniques focus on checking the equivalence of two function with the same signature [1, 3], using a variety of techniques, such as symbolic execution, construction of a product program, etc. These techniques are fundamentally incapable of checking the equivalence of functions with differing suggestions (e.g. after performing extract parameter refactoring). Our work builds upon these techniques, overcoming their fundamental limitations by extending the notion of equivalence to non-matching signatures, and building a technique to check for equivalence.

5.2 Other Equivalence checking approaches

Other techniques such as computing a product program [3].

5.3 Unit-Test Based equivalence checking

Compiler level Testing LLM-based

6 Future Work

6.1 To get a full paper

expand.

(1) Expand the evaluation set. Create examples for delete parameter, type change.
(2) Extend the notion of equivalence to accomodate delete parameter. In this case, information can be lost, in many cases.
(3) Read and compare against literature for API-Migration/Fixing.

6.2 Limitations of existing tools

I found existing tooling on EqBench [2] to be lacking – they are not ready to be scaled for full-scale Java projects. Here, I list out several efforts required to bring existing tooling up to speed. Many of them are possibly engineering efforts, but are non-trivial.

(1) Tools only operate on functions with primitive types.
(2) Limited to Java 8 language features
(3) Analysis happens by compiling a single file. Projects are always written across many files and classes. Tools aren't yet engineered for that use case. Don't trigger existing build tools (gradle, mvn)to.
(4) Libraries. Only works with std libraries. When attempting to analyse functions which use third-party libraries, tools crash.

6.3 Possible extensions

(1) Equivalence checking for classes. Establish the notion of equivalence at a class level, and then an algorithm to check the equivalence. The notion of class-level equivalence must take into account: (1) object construction can be achieved in a limited number of ways. (2) methods can have side effects.
    A simple notion – for all ways to construct the object, for all functions, the output is the same.
(2) Building benchmarks over real projects. I found that EqBench [2] has made up toy math problems, which do not represent real-world business logic. Perhaps real-world code is easier?? Or harder to engineer tools for??

7 Conclusions

References

[1] Sahar Badihi, Faridah Akinotcho, Yi Li, and Julia Rubin. 2020. ARDiff: scaling program equivalence checking via iterative abstraction and refinement of common code. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, Virtual Event USA, 13–24. doi:10.1145/3368089.3409757

[2] Sahar Badihi, Yi Li, and Julia Rubin. 2021. EqBench: A Dataset of Equivalent and Non-equivalent Program Pairs. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. 610–614. doi:10.1109/MSR52588.2021.00084 ISSN: 2574-3864.

[3] Berkeley Churchill, Oded Padon, Rahul Sharma, and Alex Aiken. 2019. Semantic program alignment for equivalence checking. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, Phoenix AZ USA, 1027–1040. doi:10.1145/3314221.3314596

[4] Johann Glock, Josef Pichler, and Martin Pinzger. 2024. PASDA: A partition-based semantic differencing approach with best effort classification of undecided cases. *Journal of Systems and Software* 213 (July 2024), 112037. doi:10.1016/j.jss.2024.112037

[5] Suzette Person, Matthew B. Dwyer, Sebastian Elbaum, and Corina S. Păsăreanu. 2008. Differential symbolic execution. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*. ACM, Atlanta Georgia, 226–237. doi:10.1145/1453101.1453131

[6] Laboni Sarker and Tevfik Bultan. 2025. Hybrid Equivalence/Non-Equivalence Testing. In *2025 IEEE Conference on Software Testing, Verification and Validation (ICST)*. 36–46. doi:10.1109/ICST62969.2025.10988990 ISSN: 2159-4848.

[7] Gustavo Soares. [n. d.]. Making program refactoring safer. ([n. d.]).