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 6 pairs of functions with differing signatures. Our technique is find the equivalence functions in 5 cases.

ACM Reference Format:
Anonymous Author(s). 2025. Equivalence checking for Design-Level Refactoring Changes. 1, 1 (December 2025), 8 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 safety of refactorings [8], 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 (which we term Poly-Methods). Poly-Methods may introduce additional parameters, remove existing ones, or change parameter types. We extend the notion of equivalence to Poly-Methods by allowing a transformational mapping between the arguments of the two methods. Intuitively, two methods are considered equivalent if, for every call to the first method, there exists a systematic way to construct a call to the second method such that the return values are identical. This perspective captures common software evolution patterns, such as adding configuration parameters or default values, while still preserving the observable behaviour of the original function.

Further, we present a technique to check the equivalence of Poly-Methods, called PolyCheck. PolyCheck builds upon the Differential Symbolic Execution (DSE) algorithm, extending it to handle cases where the function signatures differ. At a high level, PolyCheck first performs symbolic execution on both methods to generate symbolic summaries, which are logical formulas representing the relationship between the inputs and outputs of each method. It then introduces a transformation function τ that maps the inputs of the first method to the inputs of the second,

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

bridging the signature differences. Equivalence checking is performed by encoding the symbolic summaries and the transformation function into an SMT formula, which the solver then uses to determine whether the return values of the two methods are equal for all possible inputs under the transformation.

To evaluate our approach, we created a benchmark of Poly-Methods inspired by EqBench [2], by adding parameters to math functions. This results in 6 variants that are all transformationally equivalent to the originals. Running PolyCheck on this benchmark, we successfully confirmed equivalence in 5 cases.

This paper makes the following contributions:

(1) Formally extending the notion of equivalence to Poly-Methods (methods with differing signatures)
(2) PolyCheck– A technique to check the equivalence of Poly-Methods.
(3) A benchmark of 6 Poly-Methods which are equivalent, and a baseline over those.

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: $(x > y \wedge \text{return} == x) \vee (x \leq y \wedge \text{return} == y)$. Similarly, PolyCheck determines the symbolic summary of $\max_c$ to be: (absolute == true $\wedge$ ...) $\vee$ ($absolute$ == 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`.

```
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

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.

Definition 1 (Direct Equivalence). Let a method be written as $m : (T_1, T_2, \ldots, T_k) \to T_r$, 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

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

and the signature of m_2 be

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

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

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

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, $\max_a \rightsquigarrow \max_c$, under the transformation: $\max_a$(x, y) $\rightarrow$ $\max_c$(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) $\rightarrow$ $\max_d$(x, y, x), and $\max_a$(x, y) $\rightarrow$ $\max_d$(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:

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

PolyCheck aims to establish transformational equivalence. 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

$$\tau : T_1 \times \cdots \times T_k \rightarrow S_1 \times \cdots \times S_j,$$

such that

$$\tau(t_1, \ldots, t_k) = (u_1, \ldots, u_j),$$

where $(u_1, \ldots, u_j)$ are the transformed arguments passed to m_2.
(3) Ask an SMT solver to check the satisfiability of the formula:

$$\forall (t_1, ..., t_k) \in T_1 \times \cdots \times T_k. \quad s_1(t_1, \ldots, t_k) \implies s_2(u_1, \ldots, u_j)$$

(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. A *symbolic summary* represents the behavior of a method as a logical formula over its input parameters and return value. It is computed after performing symbolic execution, which explores the method's possible execution paths symbolically rather than concretely. Each path generates a path condition capturing the constraints on inputs that lead to that path, together with an expression for the return value along that path. The symbolic summary is then the disjunction of all path conditions paired with their corresponding return expressions.

Consider the $\max_a$ function in Figure 1(a). Symbolic execution generates two paths:

(1) Path 1: condition $x > y$, return value x
(2) Path 2: condition $x \leq y$, return value y

The symbolic summary for $\texttt{max}_a$ can then be written as the formula:

$$(x > y \wedge \text{return} = x) \ \vee \ (x \leq y \wedge \text{return} = y)$$

This formula compactly captures all possible executions of the method. Given concrete inputs for `x`, `y`, and `return`, evaluating the formula yields true/false, depending on whether the values fit the execution of the function.

3.2.2 Progressive Modelling of τ. The transformation function τ maps the inputs of m_1 to the inputs of m_2 in order to establish transformational equivalence. Encoding a fully general τ directly in an SMT solver is often infeasible: allowing an arbitrary complex function can cause the solver to explore an enormous search space, significantly increasing solving time or causing it to time out.

To address this, PolyCheck adopts a *progressive modelling* strategy, gradually increasing the expressiveness of τ in a controlled manner. We start with the simplest form, treating τ as a constant mapping, which corresponds to using fixed values for the additional arguments of m_2. If the solver fails to establish equivalence at this level, we increase the complexity of τ, for instance by modelling it as an uninterpreted function over the inputs of m_1. By progressively increasing the modelling power of τ, PolyCheck can efficiently find valid transformations when they exist, while keeping solver runtime manageable.

3.3 Implementation

PolyCheck builds on a combination of automated reasoning and program analysis techniques. Under the hood, it relies on the Z3 SMT solver to reason about symbolic constraints and establish equivalence conditions, and on JavaPathFinder (cite.) as a symbolic execution engine to systematically explore program paths. Many components implemented by Badihi et al.,[1] are reused in this work: the DSE implementation, integration with Z3, and the integration with JavaPathFinder.

4 Evaluation

To evaluate our ideas, we first create a benchmark of Poly-Methods, inspired by instances from EqBench[2]. Poly-Methods are created by adding a parameter to two math functions: `sum` (computing the sum over two input integers) `max` (computing the max over two input integers). Thus, we created 6 total variants of two math functions. All variants are designed to be transformationally equivalent to the original – i.e., there is a way to transform call sites to the original method to ensure equivalence.

Then, we run PolyCheck on this benchmark. PolyCheck is able to confirm the equivalence of 5 out of 6 cases. We describe the pattern where PolyCheck succeeds, and where it fails below:

PolyCheck is able to establish equivalence in cases where the transformation to the new function is simple: adding a constant parameter, or a linear transformation of inputs does the job.

However, PolyCheck fails to establish equivalence for the example-pair in Figure 2. In this example, $\max_e$ contains an additional parameter called `threshold`. When computing the `max`, the parameter acts as a cap on the return value. If the max exceeds the threshold value, the threshold value is returned.

Establishing equivalence in this case is more complex. This is because the transformation must reason over the input values `x` and `y` to fill out the threshold value accordingly. In this case, we must pick a threshold value which is at least the greater of `x` and `y`. The SMT solver alone is not able to reason about this. The solver must effectively "guess" how to choose the threshold value so that a particular branch of the code is never taken. This kind of reasoning, figuring out how to adapt inputs to control program behavior, is beyond what SMT solvers are designed to do. As a result, unless the transformation is explicitly provided, the solver cannot recognize the two methods as equivalent.

```
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_e(int x, int y, int threshold){
   int m = a>b?a:b;
   return m > threshold ? threshold : m;
}
```

(e) Equivalent under transformation: max(a, b) → max(a, b, a>b?a:b)

Fig. 2. A case where PolyCheck is not able to establish equivalence.

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 [6] 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, 5, 7] 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 Next Steps Toward a Complete Paper

This subsection outlines additional work needed to develop the current ideas into a complete paper suitable for submission to a programming languages (PL) conference.

(1) Expand the evaluation set to include additional transformation scenarios, such as deleting a parameter or changing the type of an existing parameter (e.g., int → double or String → Int).
(2) Extend the notion of equivalence to handle cases where a parameter is deleted. In these cases, some input information is lost, and complete equivalence with the original method may not be achievable. We may instead define a weaker, partial notion of equivalence over a subset of the input.
(3) Review and compare against existing literature on API migration [4]. Much of this work can be framed as an API-migration problem: given a breaking change to an API, how should calls to the original API be transformed to maintain correctness? Investigating similarities with prior approaches would answer questions about novelty.

6.2 Limitations of Existing Tools

Existing tools evaluated on EqBench [2] are not yet ready to scale to full-fledged Java projects. Below, we outline several key limitations that would need to be addressed to make these tools practical for real-world use. While some of these are primarily engineering challenges, they are non-trivial and require significant effort.

(1) **Primitive type support only:** Current tools operate exclusively on functions using primitive types (e.g., `int`, `double`, `boolean`, `String`), limiting their applicability.
(2) **Java version limitations:** Most tools are restricted to Java 8 language features and cannot handle newer constructs introduced in later versions.
(3) **Single-file analysis:** Equivalence checking is performed on individual files, whereas real-world projects span multiple files and classes. The tools are not yet integrated with standard build systems (e.g., Gradle, Maven) to handle multi-file dependencies.
(4) **Limited library support:** Analysis works only with standard libraries; functions that depend on third-party libraries often cause the tools to fail or crash.

6.3 Possible Extensions

We outline several potential directions for extending this work:

(1) **Equivalence Checking for Classes:** Extend the notion of equivalence from individual methods to entire classes, and develop algorithms to verify class-level equivalence. Such a notion must account for (1) the limited ways in which objects can be constructed, and (2) potential side effects of methods. A simple formulation could require that, for all valid object constructions and for all methods, the observable outputs are equivalent.
(2) **Building Benchmarks over Real Projects:** Current benchmarks, such as EqBench [2], consist primarily of toy math problems that do not capture the complexity of real-world business logic. Creating benchmarks from real projects could provide more realistic evaluation scenarios and help determine whether equivalence checking is easier or harder in practice.

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] Xiang Gao, Arjun Radhakrishna, Gustavo Soares, Ridwan Shariffdeen, Sumit Gulwani, and Abhik Roychoudhury. 2021. APIfix: output-oriented program synthesis for combating breaking changes in libraries. *Proc. ACM Program. Lang.* 5, OOPSLA (Oct. 2021), 1–27. doi:10.1145/3485538

[5] 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

[6] 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

[7] 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.

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