

1 Equivalence checking for Design-Level Refactoring Changes

2 ANONYMOUS AUTHOR(S)

3
4 Equivalence checking is the problem of deciding whether two program are semantically equivalent, i.e., for all
5 inputs, they generate the same output. Equivalence checking has been extensively studied to compare two
6 functions with the same signature. However, it is common to change a function's signature while evolving
7 software.

8 In this work, we extend the notion of equivalence checking to compare functions with differing signatures.
9 We introduce POLYCHECK, a technique to find the equivalence of functions with differing signatures. POLYCHECK
10 builds upon Differential Symbolic Execution algorithm. The core idea to check equivalence: is it possible
11 to transform all possible call sites such that the return values are the same? POLYCHECK aims to build a
12 transformation function that satisfies this condition. To evaluate POLYCHECK, we create a benchmark of 6
13 pairs of functions with differing signatures. Our technique is find the equivalence functions in 5 cases.

14 **ACM Reference Format:**

15 Anonymous Author(s). 2025. Equivalence checking for Design-Level Refactoring Changes. 1, 1 (December 2025),
16 8 pages. <https://doi.org/10.1145/nnnnnnn.nnnnnnn>

18 1 Introduction

19 Equivalence checking is the problem of deciding whether two programs are equivalent – i.e.,
20 two programs result in the same output, given the same input. Equivalence checking has many
21 applications, such as checking the safety of refactorings [10], evolving test suites, and tracking
22 software evolution. Existing work in equivalence checking analyses two functions with matching
23 signatures – i.e., the same input arguments and return types. The result of the analysis is a binary
24 decision: equivalent, or non-equivalent. However, software evolution is often accompanied by small
25 logic or design changes which updates the signature of the method. To the best of our knowledge
26 equivalence of methods having different signatures has not been studied.

27 To bridge the gap, we consider the equivalence of functions with differing signatures (which we
28 term POLY-METHODS). POLY-METHODS may introduce additional parameters, remove existing ones,
29 or change parameter types. We extend the notion of equivalence to POLY-METHODS by allowing a
30 transformational mapping between the arguments of the two methods. Intuitively, two methods
31 are considered equivalent if, for every call to the first method, there exists a systematic way to
32 construct a call to the second method such that the return values are identical. This perspective
33 captures common software evolution patterns, such as adding configuration parameters or default
34 values, while still preserving the observable behaviour of the original function.

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

41
42 Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee
43 provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the
44 full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored.
45 Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires
46 prior specific permission and/or a fee. Request permissions from permissions@acm.org.

47 © 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.

48 ACM XXXX-XXXX/2025/12-ART

49 <https://doi.org/10.1145/nnnnnnn.nnnnnnn>

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

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

58 This paper makes the following contributions:

- 59 (1) Formally extending the notion of equivalence to POLY-METHODS (methods with differing
 60 signatures)
- 61 (2) POLYCHECK – A technique to check the equivalence of POLY-METHODS.
- 62 (3) A benchmark of 6 POLY-METHODS which are equivalent, and a baseline over those.

64 2 Overview

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

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

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

81 **POLYCHECK: Checking the Equivalence of POLY-METHODS.** To reason about the equivalence
 82 of POLY-METHODS, a tool must figure out how to transform all possible call sites to the original
 83 method, such that the return value is the same. To transform max_a to max_c , we must reason about the
 84 value for `absolute` which achieves this condition. POLYCHECK can finds the default value to `absolute`
 85 by extending the Differential Symbolic Execution (DSE) algorithm. First, POLYCHECK computes
 86 the symbolic summaries of max_a and max_c using symbolic execution. POLYCHECK determines the
 87 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,
 88 POLYCHECK determines the symbolic summary of max_c to be: $(\text{absolute} == \text{true} \wedge \dots) \vee (\text{absolute} == \text{false} \wedge \text{max}_a(a, b))$. Next, POLYCHECK models `absolute` as an uninterpreted
 89 function with inputs x and y . Finally, it passes the formula $\text{max}_a \implies \text{max}_b$ to the SMT solver for
 90 checking. The SMT solver confirms that the formula is satisfiable, and returns the definition of
 91 `absolute = false`.

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

94 Checking the equivalence of Figure 1a and 1d is more complicated. The function max_d takes
 95 an additional parameter `equalReturn`, which is returned when x and y are equal. It is possible

99 to transform all possible call sites of $\max_a(x, y)$ to $\max_d(x, y, y)$ – thus maintaining equivalent
 100 behaviour. This is because, when x and y are equal, the original method returns the value y . Passing
 101 y as the `equalReturn` value achieves the same effect, when x and y are equal.

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

```
106 int max_a(int x, int y){  
107     if (x>y) return x;  
108     else return y;  
109 }
```

110 (a) A function computing the larger of two
111 integers

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

(b) Equivalent to original

```
112 int max_c(int x, int y, boolean absolute){  
113     if (absolute)  
114         return abs(x)>abs(y)? x: y;  
115     return x>y? x:y;  
116 }
```

117 (c) Equivalent Under Transformation:
118 $\max(x, y) \rightarrow \max(x, y, \text{false})$

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

119 (d) Equivalent Under Transformation:
120 $\max(x, y) \rightarrow \max(x, y, y)$

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 $m : (T_1, T_2, \dots, T_k) \rightarrow T_r$, where T_1, \dots, 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

$$135 \quad \forall(v_1, \dots, v_k) \in T_1 \times \dots \times T_k : \quad m_1(v_1, \dots, v_k) = m_2(v_1, \dots, v_k).$$

Definition 2 (Equivalence of POLY-METHODS). Let the signature of m_1 be

$$138 \quad m_1 : (T_1, \dots, T_k) \rightarrow T_r$$

and the signature of m_2 be

$$140 \quad m_2 : (S_1, \dots, S_j) \rightarrow T_r.$$

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

$$144 \quad \tau : T_1 \times \dots \times T_k \rightarrow S_1 \times \dots \times S_j$$

such that

$$146 \quad \forall(v_1, \dots, v_k) \in T_1 \times \dots \times T_k : \quad m_1(v_1, \dots, v_k) = m_2(\tau(v_1, \dots, v_k)).$$

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: $\max_a(x, y) \rightarrow \max_c(x, y, \text{false})$. 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:

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

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

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

such that

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

where (u_1, \dots, u_j) are the transformed arguments passed to m_2 .

- (3) Ask an SMT solver to check the satisfiability of the formula:

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

- (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 \max_a can then be written as the formula:

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

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

200
 201 3.2.2 *Progressive Modelling of τ* . The transformation function τ maps the inputs of m_1 to the inputs
 202 of m_2 in order to establish transformational equivalence. Encoding a fully general τ directly in
 203 an SMT solver is often infeasible: allowing an arbitrary complex function can cause the solver to
 204 explore an enormous search space, significantly increasing solving time or causing it to time out.

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

213 3.3 Implementation

214
 215 POLYCHECK builds on a combination of automated reasoning and program analysis techniques.
 216 Under the hood, it relies on the Z3 SMT solver to reason about symbolic constraints and establish
 217 equivalence conditions, and on JavaPathFinder ([cite](#)) as a symbolic execution engine to systematically
 218 explore program paths. Many components implemented by Badihi et al.,^[1] are reused in this
 219 work: the DSE implementation, integration with Z3, and the integration with JavaPathFinder.

221 4 Evaluation

222
 223 To evaluate our ideas, we first create a benchmark of POLY-METHODS, inspired by instances from
 224 EqBench^[2]. POLY-METHODS are created by adding a parameter to two math functions: sum (computing
 225 the sum over two input integers) max (computing the max over two input integers). Thus, we
 226 created 6 total variants of two math functions. All variants are designed to be transformationally
 227 equivalent to the original – i.e., there is a way to transform call sites to the original method to
 228 ensure equivalence.

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

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

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

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

```

246 int max_a(int x, int y){
247     if (x>y) return x;
248     else return y;
249 }

```

(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:
 $\max(a, b) \rightarrow \max(a, b, a>b?a:b)$

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

5 Related Work

We divide the related work into a few broad categories: (1) equivalence checking for functions, (2) Software API-migration.

5.1 Equivalence Checking for Functions

Equivalence checking had been extensively studied in the context of comparing functions with matching signatures. Differential symbolic execution [8] is a key technique which lays the foundation for this. The core idea is to compare symbolic summaries of two functions, and use an SMT solver to prove equivalence. Many works [1, 6, 9] improve upon this idea by proposing with techniques like abstraction refinement, fuzzing, etc. Other works from Churchill et. al., [3] perform equivalence checking on product programs.

We extend DSE-based techniques to check the equivalence of functions with differing suggestions (POLY-METHODS), an area which is previously not studied.

5.2 Software API Migration

Software library APIs are prone to breaking changes as developers maintain and update their code. A body of work focuses on automatically updating call sites to work with deprecated or updated APIs [4, 5, 7], effectively replacing old API usages with their newer counterparts.

Transformational equivalence checking (described in this paper) is closely related to this line of research and can be viewed as a first step toward API migration. By first determining whether two APIs are equivalent under a transformation, one can then safely update call sites. Existing techniques in the API-migration domain often rely on heuristic-based methods to establish equivalence between APIs. The more rigorous approach presented in this work could complement these methods by providing stronger guarantees and potentially uncovering subtle bugs during API migration.

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 [5]. Much of this work can be framed as an API-migration problem: given a breaking change to an API, how should

295 calls to the original API be transformed to maintain correctness? Investigating similarities
296 with prior approaches would answer questions about novelty.
297

298 6.2 Limitations of Existing Tools

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

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

313 6.3 Possible Extensions

314 We outline several potential directions for extending this work:

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

326 References

- 327 [1] Sahar Badihi, Faridah Akinotcho, Yi Li, and Julia Rubin. 2020. ARDiff: scaling program equivalence checking via
328 iterative abstraction and refinement of common code. In *Proceedings of the 28th ACM Joint Meeting on European
329 Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, Virtual Event USA,
330 13–24. doi:[10.1145/3368089.3409757](https://doi.org/10.1145/3368089.3409757)
- 331 [2] Sahar Badihi, Yi Li, and Julia Rubin. 2021. EqBench: A Dataset of Equivalent and Non-equivalent Program Pairs. In
332 *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. 610–614. doi:[10.1109/MSR52588.2021.00084](https://doi.org/10.1109/MSR52588.2021.00084) ISSN: 2574-3864.
- 333 [3] Berkeley Churchill, Oded Padon, Rahul Sharma, and Alex Aiken. 2019. Semantic program alignment for equivalence
334 checking. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*.
335 ACM, Phoenix AZ USA, 1027–1040. doi:[10.1145/3314221.3314596](https://doi.org/10.1145/3314221.3314596)
- 336 [4] Lukas Frunke and Jens Krinke. 2025. Automatically Fixing Dependency Breaking Changes. *Proc. ACM Softw. Eng.* 2,
337 FSE (June 2025), 2146–2168. doi:[10.1145/3729366](https://doi.org/10.1145/3729366)
- 338 [5] Xiang Gao, Arjun Radhakrishna, Gustavo Soares, Ridwan Shariffdeen, Sumit Gulwani, and Abhik Roychoudhury. 2021.
339 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](https://doi.org/10.1145/3485538)
- 340 [6] Johann Glock, Josef Pichler, and Martin Pinzger. 2024. PASDA: A partition-based semantic differencing approach with
341 best effort classification of undecided cases. *Journal of Systems and Software* 213 (July 2024), 112037. doi:[10.1016/j.jss.2024.112037](https://doi.org/10.1016/j.jss.2024.112037)

- 344 [7] Shangyu Li, Zhaoyang Zhang, Sizhe Zhong, Diyu Zhou, and Jiasi Shen. 2025. A Sound Static Analysis Approach to
345 I/O API Migration. *Proc. ACM Program. Lang.* 9, OOPSLA2 (Oct. 2025), 584–614. [doi:10.1145/3763071](https://doi.org/10.1145/3763071)
- 346 [8] Suzette Person, Matthew B. Dwyer, Sebastian Elbaum, and Corina S. Păsăreanu. 2008. Differential symbolic execution.
347 In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*. ACM, Atlanta
Georgia, 226–237. [doi:10.1145/1453101.1453131](https://doi.org/10.1145/1453101.1453131)
- 348 [9] Laboni Sarker and Tevfik Bultan. 2025. Hybrid Equivalence/Non-Equivalence Testing. In *2025 IEEE Conference on*
349 *Software Testing, Verification and Validation (ICST)*. 36–46. [doi:10.1109/ICST62969.2025.10988990](https://doi.org/10.1109/ICST62969.2025.10988990) ISSN: 2159-4848.
- 350 [10] Gustavo Soares. [n. d.]. Making program refactoring safer. ([n. d.]).

351

352

353

354

355

356

357

358

359

360

361

362

363

364

365

366

367

368

369

370

371

372

373

374

375

376

377

378

379

380

381

382

383

384

385

386

387

388

389

390

391

392