PEQtest: Testing Functional Equivalence[⋆]

Marie-Christine Jakobs(✉) and Maik Wiesner

Technical University of Darmstadt, Department of Computer Science, Darmstadt, Germany
jakobs@cs.tu-darmstadt.de

Abstract. Refactoring a program without changing the program's functional behavior is challenging. To prevent that behavioral changes remain undetected, one may apply approaches that compare the functional behavior of original and refactored programs. Difference detection approaches often use dedicated test generators and may be inefficient (i.e., execute (some of) the non-modified code twice). In contrast, proving functional equivalence often requires expensive verification. Therefore, we propose PEQtest, which aims at localized functional equivalence testing thereby relying on existing tests or test generators. To this end, PEQtest derives a test program from the original program by replacing each code segment being refactored with program code that encodes the equivalence of the original and its refactored code segment. The encoding is similar to program encodings used by some verification-based equivalence checkers. Furthermore, we prove that the test program derived by PEQtest indeed checks functional equivalence. Moreover, we implemented PEQtest in a prototype and evaluate it on several examples. Our evaluation shows that PEQtest successfully detects refactored programs that change the program behavior and that it often performs better than the state-of-the-art equivalence checker PEQcheck.

1 Introduction

Developers refactor programs [16] to improve quality attributes like e.g. performance. For instance, a developer may parallelize a program with OpenMP [30] to improve performance. While a refactoring changes the program code, e.g., adds OpenMP pragmas, to improve the program's quality, the changes must not alter the program's functional behavior. To ensure that a refactored program is reliable, we must check that the refactoring preservers the functional behavior.

Various approaches exist that aim to safeguard refactored programs from altered behavior. In practice, developers often perform regression testing [54], but the success of detecting altered behavior depends on the test suite and its test oracle(s). If refactoring rules are applied, one can prove the correctness of the applied refactoring rules [45,22,44]. In contrast, incremental verification techniques, e.g., [53,39,8,35], propose solutions for efficient re-verification of changed programs,

⋆ This work was funded by the Hessian LOEWE initiative within the Software-Factory 4.0 project.

Listing 1.1: Original program

```
void sum_seq(unsigned char N)
{
 int a[N+1];
 a[0] = 0;
 for(int i=1; i<=N; i++)
  a[i] = a[i-1] + i;
}
```

Listing 1.2: Refactored program

```
int sum_par(unsigned char N)
{
 int a[N+1];
 a[0] = 0;

#pragma omp parallel for
 for(int i=1; i <= N; i++)
  a[i] = (i*(i+1))/2;
}
```

Listing 1.3: Generated test program

```
int sum_test(unsigned char N)
{
 int a[N+1];
 a[0] = 0;

 store(a, 0);

 for(int i=1; i <= N; i++)
  a[i] = (i*(i+1))/2;

 store(a, 1);
 restore(a,0);

#pragma omp parallel for
 for(int i=1; i <= N; i++)
  a[i] = (i*(i+1))/2;

 store(a, 2);
 eq_store(a, 1, 2);
}
```

Fig. 1: Original, sequential program (top left), which initializes each array entry i with $\sum_{j=0}^{i} j$, the refactored program (bottom left), which parallelizes the array initialization using OpenMP and utilizing that $\sum_{j=0}^{i} j = \frac{i\cdot(i+1)}{2}$, as well as the generated program for testing functional equivalence (right)

but they typically need a specification of the functional behavior, which rarely exists. Another solution, which does not require a specification, is to inspect whether or when the original and the refactored program behave functionally equivalent. Approaches aiming to detect differences in the behavior [26,52,46,20,31,29,36,47] are inefficient, i.e., execute each test case on the original and the refactored program or function, and often use dedicated test generators. Approaches aiming to prove functional equivalence [5,56,40,14,13,43,49,41,34,4,15,17,38,23,42,19] use heavyweight verification techniques, rarely support parallel programs, and often consider all possible variables values.

Our goal is to develop a lightweight, test-based approach for functional equivalence checking, for which we can use existing tests or test generators. Inspired by equivalence checkers [17,38,23,51,42,2,19] that transform the equivalence of two programs into a set of verification tasks (i.e., programs with assertions), our PEQTEST approach transforms the equivalence of two programs into a test program. To restrict equivalence testing to relevant program values and to reduce the duplicate execution of non-modified code, PEQTEST generates a single test program (verification task) that executes the unchanged code only once and individually checks equivalence of each refactored code segment in the context of the original program. The individual checks use a similar idea as UC-KLEE [38], which verifies equivalence of functions. More concretely, PEQTEST derives the test program from the original program by extending each original code segment with (a) the refactored code segment and (b) code to store, restore, and compare variable values of modified variables. To store, restore, and compare the values of modified variables, PEQtest relies on checkpoints, which save the values of a given set of modified variables in a given program state.

In our example (Fig. 1), PEQtest first detects that the original (sequential) code segment (framed, dark blue) and the refactored (parallelized) code segment (frameless, light blue) modify variable `a`[1]. Thereafter, PEQtest derives the test program (right) from the original program (top left). It adds the parallelized code segment. To provide the same input to the original and refactored code segment, PEQtest uses checkpoint 0 to store modified variables. The test program calls `store(a, 0);` to save in checkpoint 0 the values of modified variable `a`before the original code segment and calls `restore(a,0);` to restore the values of modified variables before the refactored code segment. To make the result of both code segments available for equivalence checking, the test program stores the values of modified variable `a` after each code segment in checkpoint 1 and 2, respectively. Finally, the equivalence test `eq_store` checks whether the checkpoints 1 and 2 contain equivalent values for the modified variable `a`.

We proved that PEQtest generates test programs that can indeed detect inequivalence and that if no execution of the test program reveals an inequivalence, original and refactored program are equivalent. As a proof-of-concept, we implemented PEQtest and used it to check several program parallelizations and a few sequential refactorings. Our evaluation shows that PEQtest reliably detects inequivalences and typically outperforms the state-of-the-art equivalence checker PEQcheck [19].

2 Background

Program Syntax. To present our approach, we rely on a simple imperative language on integer variables.[2] Since synchronization issues, e.g., deadlocks, do not affect how our approach works and we want to keep the programming language simple, our language supports parallel execution, but no synchronization operations. Below, we show the grammar of the programming language that we use to present our approach.

$$S := E \mid v :=_\ell aexpr; \mid \mathbf{if}_\ell\ bexpr\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2 \mid \mathbf{while}_\ell\ bexpr\ \mathbf{do}\ S \mid S_1 S_2 \mid [S_1 \| \ldots \| S_n]$$

We use E to denote the empty program and assume that arithmetic expressions *aexpr* in assignments and Boolean expressions *bexpr* in if and while statements are built with standard operators on integers. To build more complex programs S, several subprograms S_i may be assembled into a sequence or into a parallel statement. To unambiguously identify the original and refactored code segments during test program generation[3] and any subprogram in our proofs,

[1] Both segments also modify variable `i`, but it is a local variable, which can be ignored.
[2] Our implementation supports a subset of C programs, which may use OpenMP pragmas for parallelization.
[3] For our implementation, one only needs to specify the start and end of code segments i.

$$\frac{\sigma(bexpr)=\text{true}}{(\textbf{if}_\ell\ bexpr\ \textbf{then}\ S_1\ \textbf{else}\ S_2,\sigma,\xi)\xrightarrow{bexpr}(S_1,\sigma,\xi)} \qquad \frac{\sigma(bexpr)=\text{false}}{(\textbf{if}_\ell\ bexpr\ \textbf{then}\ S_1\ \textbf{else}\ S_2,\sigma,\xi)\xrightarrow{\neg bexpr}(S_2,\sigma,\xi)}$$

$$\frac{\sigma(bexpr)=\text{true}}{(\textbf{while}_\ell\ bexpr\ \textbf{do}\ S,\sigma,\xi)\xrightarrow{bexpr}(S\ \textbf{while}_\ell\ bexpr\ \textbf{do}\ S,\sigma,\xi)} \qquad \frac{\sigma(bexpr)=\text{false}}{(\textbf{while}_\ell\ bexpr\ \textbf{do}\ S,\sigma,\xi)\xrightarrow{\neg bexpr}(E,\sigma,\xi)}$$

$$\frac{}{(v{:=}_\ell aexpr;,\sigma,\xi)\xrightarrow{v:=aexpr;}(E,\sigma[v:=\sigma(aexpr)],\xi)} \qquad \frac{(S_i,\sigma,\xi)\xrightarrow{op}(S_i',\sigma',\xi')}{([S_1\|\ldots\|S_i\|\ldots\|S_n],\sigma,\xi)\xrightarrow{op}([S_1\|\ldots\|S_i'\|\ldots\|S_n],\sigma',\xi')}$$

$$\frac{(S_1,\sigma,\xi)\xrightarrow{op}(S_1',\sigma',\xi')}{(S_1S_2,\sigma,\xi)\xrightarrow{op}(S_1'S_2,\sigma',\xi')} \qquad \frac{\forall v\in V{:}\xi(\sigma(aexpr1))(v)=\xi(\sigma(aexpr2))(v)}{(eq\_store(V,aexpr1,aexpr2);,\sigma,\xi)\xrightarrow{eq\_store}(E,\sigma,\xi)}$$

$$\frac{}{(restore(V,aexpr);,\sigma,\xi)\xrightarrow{restore}(E,\sigma[V\leftarrow\xi(\sigma(aexpr))],\xi)} \qquad \frac{}{(E\ \ S,\sigma,\xi)\xrightarrow{\textbf{nop}}(S,\sigma,\xi)}$$

$$\frac{}{(store(V,aexpr);,\sigma,\xi)\xrightarrow{store}(E,\sigma,\xi[\sigma(aexpr):=\xi(\sigma(aexpr))[V\leftarrow\sigma]])} \qquad \frac{}{([E\|\ldots\|E],\sigma,\xi)\xrightarrow{\textbf{nop}}(E,\sigma,\xi)}$$

Fig. 2: Rules for operational semantics

we assume that each basic statement is annotated with a label ℓ, which must be unique in the complete program. Moreover, we use the set $\mathcal{V}$ to refer to all program variables and subset $\mathcal{V}(S)\subseteq\mathcal{V}$ to refer to the variables occurring in (sub)program S. Similarly, subset $\mathcal{V}(expr)\subseteq\mathcal{V}$ represents all variables that occur in an arithmetic or Boolean expression *expr*.

While the programming language above is sufficient to represent original and refactored programs, the test programs derived by our approach also use checkpointing to store, restore, and compare relevant parts (e.g., modified variables) of program states. To support checkpointing and checkpoint comparison, we extend the programming language for test programs with the three checkpoint functions `eq_store`, `restore`, and `store`. All three functions get as input a subset $V\subseteq\mathcal{V}$ of relevant variables and one or two arithmetic expressions (typically an integer constant) to refer to the relevant checkpoints.

$$S := eq\_store(V, aexpr1, aexpr2);\ \mid restore(V, aexpr);\ \mid store(V, aexpr);$$

Program Semantics We formalize the program semantics using a fairly standard operational semantics that defines how a program executes. A program execution is a sequence of transitions between execution states. An execution state is a triple of a program, a data state, and an additional checkpoint state. A data state is a function $\sigma:\mathcal{V}\to\mathbb{Z}$ that provides an integer value for each program variable. We denote the set of all data states by Σ. A checkpoint state is a function $\xi:\mathbb{N}\to\Sigma$ that maps checkpoints i to data states σ. The set Ξ denotes all checkpoint states.

The 12 rules shown in Fig. 2, which consists of 7 standard rules plus 5 newly introduced rules highlighted in light gray, define the possible transitions. As usual, we write $\sigma(expr)$ for the evaluation of `expr` in data state $\sigma \in \Sigma$.[4] The state update $\sigma[v := \sigma(aexpr)]$, which is used in the rule for the assignment, returns a new data state σ_n with $\sigma_\mathrm{n}(w) = \sigma(w)$ for all $w \in \mathcal{V} \setminus \{v\}$ and $\sigma_\mathrm{n}(v) = \sigma(aexpr)$. Similarly, the multi state update $\sigma[V \leftarrow \sigma']$, which is used by the new store and restore rules, returns a new data state σ_n with $\sigma_\mathrm{n}(w) = \sigma(w)$ for all $w \in \mathcal{V} \setminus V$ and $\sigma_\mathrm{n}(v) = \sigma'(v)$ for all $v \in V$. In addition, the checkpoint update $\xi[c := \sigma_\mathrm{u}]$, which is used in the store rule, returns a new checkpoint state ξ_n with $\xi_\mathrm{n}(i) = \xi(i)$ for all $i \in \mathbb{N} \setminus \{c\}$ and $\xi_\mathrm{n}(c) = \sigma_\mathrm{u}$.[5] Also, note that instead of assuming that $E\ S$ and $[E\|\ldots\|E]S$ are equivalent to S, we introduce two nop rules, which make our proofs simpler. After we formalized the transitions, we now inductively define the executions $ex(S)$ of a program S with two inference rules:

1. $\frac{\sigma\in\Sigma,\xi\in\Xi}{(S,\sigma,\xi)\in ex(S)}$ and
2. $\frac{(S_0,\sigma_0,\xi_0)\xrightarrow{op_1}\ldots\xrightarrow{op_n}(S_n,\sigma_n,\xi_n)\in ex(S),\quad (S_n,\sigma_n,\xi_n)\xrightarrow{op_{n+1}}(S_{n+1},\sigma_{n+1},\xi_{n+1})}{(S_0,\sigma_0)\xrightarrow{op_1}\ldots\xrightarrow{op_n}(S_n,\sigma_n,\xi_n)\xrightarrow{op_{n+1}}(S_{n+1},\sigma_{n+1},\xi_{n+1})\in ex(S)}$.

We write $(S, \sigma, \xi) \rightarrow^* (S', \sigma', \xi')$ if the intermediate steps of the execution are unimportant. Furthermore, we say that execution $(S, \sigma, \xi) \rightarrow^* (S', \sigma', \xi')$ (i) terminates normally if $S' = E$ and (ii) violates a checkpoint equivalence if S' violates a checkpoint equivalence in (σ', ξ'). In general, a program S' violates a checkpoint equivalence in (σ', ξ') if either (a) there exists a statement $S_{eq} = eq\_store(V, aexpr1, aexpr2)$; such that $\exists v \in V : \xi'(\sigma'(aexpr1))(v) \neq \xi'(\sigma'(aexpr2))(v)$ and $S = S_{eq}$ or $S = S_{eq}S'$ or (b) $S = [S_1\|\ldots\|S_i\|\ldots\|S_n]$ or $S = [S_1\|\ldots\|S_i\|\ldots\|S_n]S'$ and there exists at least one subprogram S_i that violates a checkpoint equivalence in (σ', ξ'). In general, a program S violates a checkpoint equivalence if there exists an execution $(S, \sigma, \xi) \rightarrow^* (S', \sigma', \xi') \in ex(S)$ such that S' violates a checkpoint in (σ', ξ').

Partial Equivalence. We are interested whether two (sub)programs behave functionally equivalent, i.e., compute the same output when given the same input. Like many other approaches on equivalence checking, we focus on *partial equivalence*, i.e., we limit equivalence to executions that terminate normally.[6] In addition, we utilize that checkpoint functions are not used in programs, but are only introduced to test functional equivalence. Therefore, our definition of partial equivalence focuses on data states and ignores checkpoint states.

Definition 1. *(Sub)programs $S1$ and $S2$ are* partially equivalent *($S1 \equiv S2$) if*

$$\forall \sigma, \sigma', \sigma'' \in \Sigma, \xi, \xi', \xi'', \xi''' \in \Xi : ((S1, \sigma, \xi) \rightarrow^* (E, \sigma', \xi') \in ex(S1) \\ \wedge (S2, \sigma, \xi'') \rightarrow^* (E, \sigma'', \xi''') \in ex(S2)) \implies \sigma' = \sigma'' \ .$$

[4] Note that we do not specify the expression evaluation in detail because we have not fixed the expression syntax. However, we assume that the result of evaluating integer constant c in data state σ is the constant c (i.e., $\sigma(c) = c$) and that expression evaluation is deterministic (i.e., $\sigma(expr) = x \wedge \sigma(expr) = y \implies x = y$).

[5] The store rule determines the state σ_u using a multi state update and the index c evaluating an arithmetic expression (often a constant) in the current data state.

[6] Note that we still may detect that a refactoring introduces non-termination because if a refactoring introduces non-termination, our test program either detects inequivalence or does not terminate for some inputs.

Variable Modification. To make equivalence testing more efficient, we only want to checkpoint modified variables, i.e., the checkpoint should only store the value of those variables whose value may change. The following definition formalizes the set of variables modified by a (sub)program.

Definition 2. *Let S be a (sub)program. The variables* modified *by S are:*

$$\mathcal{M}(S) := \{v \in \mathcal{V} \mid \exists \sigma, \sigma' \in \Sigma, \xi, \xi' \in \Xi : (S, \sigma, \xi) \rightarrow^* (\cdot, \sigma', \xi') \wedge \sigma(v) \neq \sigma'(v)\}.$$

For instance, in programs written in our programming language that do not use restore statements variables can only be modified by assignments. For those programs, the set $\mathcal{M}(S)$ of modified variables can be overapproximated by the set of variables that occur in S on the left-hand side of an assignment. In the following, we describe any overapproximation of the modified variables, e.g. the one sketched above, by $M_{\approx} : S \rightarrow 2^{\mathcal{V}}$ and assume that $\mathcal{M}(S) \subseteq M_{\approx}(S)$.

3 Generating Test Programs with PEQtest

Our goal is to test equivalence between an original and refactored program, which both do not use checkpoint functions. As explained earlier, checkpoint functions are supposed to be used by test programs only. In this section, we describe how PEQtest generates the test program for equivalence testing, prove soundness of the generated test program, i.e., show that the generated test program checks functional equivalence, and discuss limitations of PEQtest's program generation as well as our implementation.

Sound Test Program Generation. To test functional equivalence of two subprograms, the idea of our PEQtest approach is to execute both subprograms with the same input and compare their outputs. The test program generated by PEQtest will execute the two subprograms sequentially to avoid that their executions can interfere with each other. Furthermore, it will ensure that both subprograms get equal inputs, which may be produced by the (original) program, and that their outputs can be compared. Many verification approaches for functional equivalence [17,38,23,51,42,2,19] use a similar setup, but do not restrict the inputs. To ensure equal inputs and make outputs available, these approaches either (1) duplicate (shared, modified) variables, replace the variables in one of the subprograms by the duplicated ones, and assign equal values to the original and duplicated inputs [17,42,2,19], (2) add additional variables to store the input and output values and restore the input after the execution of the first subprogram [51,23], or (3) use dedicated functions, e.g., checkpoint functions, to store and restore inputs and outputs [38]. For our test program, we choose option (3) because it does not change the subprograms and, thus, it simplifies test program generation as well as it eases the comprehensibility of the test program.

Next, we discuss how we implement option (3). To lower the test effort, we decide to only store and compare values of variables that may be modified by one of the two subprograms. Since this set cannot always be determined precisely and different overapproximations are imaginable, we use parameter V to provide this set to the test program generator. Moreover, we aim at localized equivalence testing. Thus, our test program likely includes more than one functional equivalence test, namely one for each pair of original and refactored subprogram. While the output must be stored directly after the execution of each subprogram, the output comparison can be done at the end of the test program or after the execution of original and refactored subprogram. We choose the second option because it allows us to reuse checkpoints and lets the test program stop at the first difference of outputs, which makes it easier to detect which pair of subprograms is responsible for the failure of the test program, i.e., which pair of subprograms is inequivalent. We stop at the first difference instead of e.g. logging the difference because test execution becomes faster, but we address the logging alternative when discussing the limitations. The following definition shows how we encode the functional equivalence test for an original subprogram $S1$ and the refactored subprogram $S2$ for a given overapproximation V of the set of modified variables.

$$\begin{aligned}&\mathit{test\_eq}(V, S1, S2) :=\\&\mathit{store}(V,0);\ S1\ \mathit{store}(V,1); \mathit{restore}(V,0);\ S2\ \mathit{store}(V,2); \mathit{eq\_store}(V,1,2);\end{aligned}$$

Next, we show that our test encoding is sound, i.e., it may detect inequivalences if the two subprograms $S1$ and $S2$ are inequivalent. Our encoding uses checkpoint equivalence to detect whether two subprograms $S1$ and $S2$ are inequivalent, i.e., differ in their outputs. Hence, it must violate a checkpoint equivalence if $S1$ and $S2$ are inequivalent. We can ensure even more and show that the test encoding is also complete. As shown by the following theorem our test encoding violates a checkpoint equivalence if and only if $S1$ and $S2$ are inequivalent.

Theorem 1. *Let $S1$ and $S2$ be (sub)programs without calls to checkpoint functions and $M_\approx$ be an overapproximation of the modified variables. Then, $S1 \equiv S2$ iff $\mathit{test\_eq}(M_\approx(S1) \cup M_\approx(S2), S1, S2)$ does not violate a checkpoint equivalence.*

Proof (Sketch). Let $M := M_\approx(S1) \cup M_\approx(S2)$.

$\Rightarrow$ Let $(\mathit{test\_eq}(M, S1, S2), \sigma, \xi) \rightarrow^* (\mathit{eq\_store}(M,1,2);, \sigma_5, \xi_5)$ be arbitrary. Show with semantics that there exists an execution
$(\mathit{test\_eq}(M, S1, S2), \sigma, \xi)$
$\rightarrow (S1\ \mathit{store}(M,1); \mathit{restore}(M,0);\ S2\ \mathit{store}(M,2); \mathit{eq\_store}(M,1,2);, \sigma_1, \xi_1)$
$\rightarrow^* (\mathit{store}(M,1); \mathit{restore}(M,0);\ S2\ \mathit{store}(M,2); \mathit{eq\_store}(M,1,2);, \sigma_2, \xi_2)$
$\rightarrow^* (S2\ \mathit{store}(M,2); \mathit{eq\_store}(M,1,2);, \sigma_3, \xi_3)$
$\rightarrow^* (\mathit{store}(M,2); \mathit{eq\_store}(M,1,2);, \sigma_4, \xi_4)$
$\rightarrow (\mathit{eq\_store}(M,1,2);, \sigma_5, \xi_5)$
with $\sigma = \sigma_1 = \sigma_3$, for all $v \in \mathcal{V} \setminus M$ also $\sigma(v) = \sigma_5(v)$, and for all $v \in M$ we have $\xi_5(1)(v) = \sigma_2(v)$ and $\xi_5(2)(v) = \sigma_4(v)$.
Conclude that exists $(S1, \sigma_1, \xi_1) \rightarrow^* (E, \sigma_2, \xi_2)$ and $(S2, \sigma_3, \xi_3) \rightarrow^* (E, \sigma_4, \xi_4)$ with $\sigma = \sigma_1 = \sigma_3$ and for all $v \in M$ we have $\xi_5(1)(v) = \sigma_2$ and $\xi_5(2)(v) = \sigma_4$. By assumption ($S1 \equiv S2$), $\sigma_2 = \sigma_4$ and, thus, $\xi_5(1)(v) = \sigma_2(v) = \sigma_4(v) = \xi_5(2)(v)$. By semantics, $(\mathit{test\_eq}(M, S1, S2), \sigma, \xi) \rightarrow^* (\mathit{eq\_store}(M,1,2);, \sigma_5, \xi_5)$ does not violate a checkpoint equivalence.

$\Leftarrow$ Let $(S1, \sigma_1, \xi_1) \rightarrow^* (E, \sigma_2, \xi_2)$ and $(S2, \sigma_3, \xi_3) \rightarrow^* (E, \sigma_4, \xi_4)$ be arbitrary with $\sigma_1 = \sigma_3$. Show with semantics that there exists an execution

$(\mathit{test\_eq}(M, S1, S2), \sigma, \xi)$
$\rightarrow (S1\ \mathit{store}(M, 1); \mathit{restore}(M, 0);\ S2\ \mathit{store}(M, 2); \mathit{eq\_store}(M, 1, 2); , \sigma_1, \xi_1)$
$\rightarrow^* (\mathit{store}(M, 1); \mathit{restore}(M, 0);\ S2\ \mathit{store}(M, 2); \mathit{eq\_store}(M, 1, 2); , \sigma_2, \xi_2)$
$\rightarrow^* (S2\ \mathit{store}(M, 2); \mathit{eq\_store}(M, 1, 2); , \sigma_3, \xi_3)$
$\rightarrow^* (\mathit{store}(M, 2); \mathit{eq\_store}(M, 1, 2); , \sigma_4, \xi_4)$
$\rightarrow (\mathit{eq\_store}(M, 1, 2); , \sigma_5, \xi_5)$
with $\sigma = \sigma_1 = \sigma_3$, for all $v \in \mathcal{V} \setminus M$ also $\sigma(v) = \sigma_5(v)$, and for all $v \in M$ we have $\xi_5(1)(v) = \sigma_2(v)$ and $\xi_5(2)(v) = \sigma_4(v)$.
Since the test program does not violate a checkpoint equivalence, for all $v \in M$ we know $\sigma_2(v) = \xi_5(1)(v) = \xi_5(2)(v) = \sigma_4(v)$. We conclude that $\sigma_2 = \sigma_4$. □

So far, we can use the test encoding to test or even verify functional equivalence of complete programs. Following the idea of PEQCHECK [19], which checks equivalence on the level of subprograms rather than on the level of functions or programs, our goal is to split testing of equivalence into multiple subtests, namely one subtest per pair of original and refactored subprogram. While PEQCHECK builds one equivalence task per pair and verifies all tasks on every input, our PEQTEST approach generates one single test program that only provides inputs produced by the original program[7]. More concretely, PEQTEST derives the test program from the original program by replacing the subprograms being refactored with the test encoding *test_eq* of the original and refactored subprogram.

Currently, we assume that PEQTEST is informed about the refactored subprograms. More concretely, given original program S and refactored program S', we assume that there exists a partial, injective *replacement function* $\gamma : 2^S \rightharpoonup 2^S$[8] such that S' can be derived from S by replacing all subprograms S_1 of S with $S_1 \in \mathit{preImg}(\gamma)$ by $\gamma(S_1)$. Generally, we write $S2 = \Gamma(S1, \gamma)$ to denote that $S2$ is derivable from $S1$ by replacing all subprograms S_s of $S1$ by $\gamma(S_s)$. For the PEQTEST approach, we assume that the replacement function γ only describes the refactoring of the original program S, i.e., $\mathit{preImg}(\gamma)$ only contains subprograms of S. In addition, the replacement must be unambiguous. Hence, we do not allow $S_1, S_2 \in \mathit{preImg}(\gamma)$ such that S_2 is a subprogram of S_1 nor $S_1, S_1S_2 \in \mathit{img}(\gamma)$ such that S_1 is a subprogram of S and $S_1 \notin \mathit{preImg}(\gamma)$.[9] We also require that $E, [E\|\ldots\|E] \notin (\mathit{preImg}(\gamma) \cup \mathit{img}(\gamma))$ and $\neg\exists S : E\ S, [E\|\ldots\|E]S \in (\mathit{preImg}(\gamma) \cup \mathit{img}(\gamma))$ because they are no proper programs. To avoid that interference of parallel statements can invalidate the result of a test, all subprograms in $\mathit{preImg}(\gamma)$ ($\mathit{img}(\gamma)$) must not occur in a parallel statement of the original (refactored) program. Thus, a refactoring in a parallel statement must be described by a refactoring of the parallel statement. Note that for proper programs one can always use $\gamma = \{(S, S')\}$.[10]

To generate our test program, PEQTEST requires a replacement function γ_{test} that maps the subprograms being refactored to their test encodings. PEQTEST

[7] If all original and refactored subprograms are equivalent (which we aim to inspect), the original and refactored program will provide the same inputs.
[8] If γ is not injective, one can make it injective by properly changing statement labels.
[9] One can achieve this by proper choices of code segments and statement labels.
[10] However, one may need to adapt some of the labels in S'.

derives γ_{test} from the replacement function γ, which describes the refactoring. For each subprogram in the domain, PEQtest replaces its image (the refactored subprogram) by the test encoding of that subprogram and its refactored subprogram thereby using an $M_{\approx}$ to determine the set of modified variables.

$$\gamma_{\text{test}}(\gamma, M_{\approx}) := \{(S1, \mathit{test\_eq}(M_{\approx}(S1) \cup M_{\approx}(\gamma(S1)), S1, \gamma(S1))) \mid S1 \in \mathit{preImg}(\gamma)\}$$

Let us briefly discuss why γ_{test} fulfills the requirements on a replacement function. Since the test encoding contains $\gamma(S1)$, function γ_{test} inherits injectivity from γ. By construction, test encodings are unequal to E, $E\ S$, $[E\|\ldots\|E]$, and $[E\|\ldots\|E]S$ and start with checkpoint functions, which we assume that the original program does not contain. The remaining requirements are fulfilled because we only replace refactored subprograms by the corresponding test encoding.

Now, we have everything at hand to generate the test program, which can then be used to detect inequivalences with an existing test approach, e.g., [12,1]. As explained, we derive the test program from the original program by replacing the subprograms being refactored with the test encoding *test_eq* of the original and refactored subprogram. To achieve this, we use the replacement function γ_{test}.

$$\mathit{test\_prog}(S, \gamma, M_{\approx}) := \Gamma(S, \gamma_{\text{test}}(\gamma, M_{\approx}))$$

Again, let us consider soundness, but now for the test program. Our goal is to detect inequivalences caused by a refactoring. Thus, we do not give any guarantees if the original program is non-deterministic, i.e., not equivalent to itself, which can only occur if it contains non-deterministic parallel statements or checkpoint functions. We already assumed that checkpoint functions are only used by the test program, but not by the original or refactored program. For our soundness discussion, we also exclude programs that contain non-replaced, non-deterministic parallel statements. More concretely, we assume that all parallel statements S_p that are not replaced, i.e., for whom there does not exist a subprogram $S_s \in \mathit{preImg}(\gamma)$ such that $S_p = S_s$ or S_p is a subprogram of S_s, are deterministic ($S_p \equiv S_p$). In this case, the following theorem ensures that our PEQtest approach can soundly detect inequivalences, i.e., the test program generated by PEQtest is able to detect a violation of a checkpoint equivalence if original and refactored program are inequivalent.

Theorem 2. *Let S and S' be programs without calls to checkpoint functions, $M_{\approx}$ an overapproximation of the modified variables, γ be a replacement function such that $S' = \Gamma(S, \gamma)$, and all non-replaced parallel statements S_p of S are deterministic ($S_p \equiv S_p$). If $S \not\equiv S'$, then there exists $(S_0, \sigma_0, \xi_0) \to^* (S_n, \sigma_n, \xi_n) \in ex(\mathit{test\_prog}(S, \gamma, M_{\approx}))$ that violates a checkpoint equivalence.*

Finally, let us look at the contraposition of the above theorem. While our intention for PEQtest is testing and detection of equivalence violations, the corollary below states that we can alternatively verify the test program generated by PEQtest to show functional equivalence.

Listing 1.4: Original program

```
void swapi_orig(int x, int y)
{
  tmp=y+1;
  y=x;
  x=tmp;
}
```

Listing 1.5: Refactored program

```
void swapi_mod(int x, int y)
{
  tmp=y;
  y=x;
  x=tmp+1;
}
```

Fig. 3: Behaviorally equivalent original and refactored program whose code segments are not equivalent

Corollary 1. *Let S and S' be programs without calls to checkpoint functions, $M_{\approx}$ an overapproximation of the modified variables, γ be a replacement function such that $S' = \Gamma(S, \gamma)$, and all non-replaced parallel statements S_p of S are deterministic ($S_p \equiv S_p$). If no execution $(S_0, \sigma_0, \xi_0) \rightarrow^* (S_n, \sigma_n, \xi_n) \in ex(test\_prog(S, \gamma, M_{\approx}))$ violates a checkpoint equivalence, then $S \equiv S'$.*

Discussion of Limitations. Functional equivalence of two programs is undecidable [17]. While our PEQTEST approach is sound under certain assumptions. PEQTEST may report violations of checkpoint equivalences, although original and refactored program are equivalent. Hence, it may be incomplete. One reason is the wrong choice of code segments. For example, consider Fig. 3. Although the two code segments of original and refactored program (highlighted in blue and green, respectively) are inequivalent, the programs are equivalent. For our experiments, we ensured that we do not make the wrong choice for the code segments. In practice, one may check whether a reported violation is a false alarm caused by a wrong choice of code segments by reusing the test input causing the violation to execute one or more test programs generated by PEQTEST that use the same original and refactored program but larger segments, e.g., using segments on function or program level, or iteratively merging segments until the violation is disproved or the segments become the programs.

Next, let us discuss the assumption used in Theorem 2. One can easily get rid of the assumption that non-replaced parallel statements must be deterministic. Basically, PEQTEST needs to extend γ with pairs (S_p, S_p) for all non-replaced parallel statements S_p. Supporting checkpoint functions is more challenging because PEQTEST must be able to store and restore checkpoints and it must ensure that its checkpoints and the program's checkpoints do not interfere. While one may find such an encoding, our definition of partial equivalence does not cover checkpoint states. Also, it does not support non-deterministic programs since our main motivation for PEQTEST is refactoring or parallelization of sequential programs not the refactoring of non-deterministic, parallel programs. To properly support checkpointing and all kinds of parallel programs, our definition of equivalence and PEQTEST need to be adapted significantly.

Also, the requirements on the replacement function restrict our PEQTEST approach. While many assumptions can be met by adapting labels of statements, the requirement that code segments must be subprograms and they must not occur in a parallel statement are major restrictions. However, note that this only limits the granularity of code segments, but not the applicability of the approach.

Finally, we want to mention that in our above formalization we chose to stop as soon as PEQTEST finds a violation because it simplified our proofs. To always inspect all refactored code segments, one can either move PEQTEST's checks at the end of the test program and use different checkpoints per test encoding, or only write a log but do not stop when detecting a difference. To ensure that one still tests on values of the original program, one must restore the output of the original program at the end of each test encoding or swap $S1$ and $S2$ in the test encoding $test\_eq$, i.e., execute the refactored subprogram $S2$ before the original subprogram $S1$. Our current implementation postpones PEQTEST's checks to the end of the test program and restores the output of the original program at the end of each test encoding.

Implementation. We support test program generation for a subset of C programs with or without OpenMP directives. So far, we do not support programs with pointer aliasing (except for parameter passing). While we allow pointers and dynamic memory allocation, we do not support the modification of dynamic data structures in original or refactored code segments. The reason is that we checkpoint arrays and structs by recursively checkpointing their elements and checkpoint pointers by dereferencing them and then checkpoint the dereferenced non-pointer element. Thus, our current implementation only works correctly in case that pointers that need to be checkpointed are non-null and do not change in original or refactored code segments.[11]

Our test program generation relies on the ROSE compiler framework [37]. To store and restore checkpoints, we use a minicpr library, but we built our own library to compare checkpoints. Our implementation assumes that the start and end of a code segment i is specified by pragma statements `#pragma scope_i` and `#pragma epocs_i`. Currently, we insert them manually. For OpenMP parallelization (our main field of application), insertion is mostly straightforward. Often, choosing the code blocks associated with the outermost OpenMP directives is a good choice. This can easily be automated, but has not been implemented yet.

For each code segment, our implementation runs ROSE's definition-use analysis to detect the modified variables $M_{\approx}$ that are visible after the code segment. If a code segment contains procedure calls, we also add all global variables and all variables occurring in the parameter expression of a pointer or array argument to the modified variables $M_{\approx}$. Based on the computed set $M_{\approx}$ of modified variables, we then extend the sequential code segment with the refactored code segment and the calls to the checkpoint library necessary to store and restore checkpoints. In contrast to our formalization, the store and restore operation only get the checkpoint name, while additional calls are used to inform the checkpoint library which variables V to consider. Also note that the test program generated by our implementation stores the output of the original and refactored code segments

[11] Due to internals of the used checkpoint library, pointers must not change after they are first checkpointed.

in checkpoints that differ for each execution of a test encoding and performs output comparison at the end of the test program, which allows us to inspect all checkpoints at once and to possibly find multiple violations.

Next, we describe the checkpoint comparison. For each variable in the two checkpoints[12], we check whether their content is equivalent. Except for floating point values, we rely on C's byte level comparison function `memcmp`. Often, implementations of floating point operations like + are not associative, but small differences of floating point values are tolerable. Thus, our comparison of floating point values succeeds when the difference of the values is within a tolerance ε[13].

4 Evaluation

The goals of our experiments are to (a) study how effective and efficient is PEQtest's detection of inequivalences and to (b) compare PEQtest to an existing equivalence checker. For our comparison, we choose PEQcheck because it also supports localized checking for OpenMP programs.

4.1 Experimental Setup

Benchmark. To check equivalence of sequential and parallelized programs, we use the tasks from the DataRaceBench (DRB) benchmark suite [24,50] (version 1.3.2), which addresses common mistakes in OpenMP parallelization and contains OpenMP programs with and without data races. From the DataRaceBench, we exclude all tasks with thread private directives, which we cannot cover with our segments and all tasks that require at least an OpenMP 4.5 compiler or that offload computation to a different device (i.e., use the `target` construct) because they are neither supported by PEQtest nor PEQcheck. In total, we get 132 tasks (26 equivalent and 106 inequivalent tasks). We manually selected the code segments following the idea discussed in the implementation paragraph and use the DataRaceBench programs without OpenMP constructs for the sequential (original) programs. To execute the generated test programs, we use the inputs provided by DataRaceBench.

To check equivalence of two sequential program versions, we consider all non-recursive programs from Rêve [15]. However, we exclude loop4 and loop5, which were not available, as well as digits10, digits!10, and barthe2, which declare different sets of output variables in original and refactored program and, thus, are detected inequivalent during test program generation. To make the programs executable, we remove the mark annotations, which have no implementation, and extend each of the programs with a test driver that generates random inputs. The code segments are the same as in the evaluation of PEQcheck [19]. In total, we get 15 sequential tasks (5 equivalent and 10 inequivalent tasks).

Tool Configurations. To study the trade-off between effectiveness and efficiency, we examine three PEQtest configurations, which differ in the resources

[12] By construction, checkpoints that are compared store the same variables.

[13] In our evaluation, we use $\varepsilon = 10^{-8}$.

used during test program execution. The low effort configuration uses one thread and runs the test program once. The other two configurations use two threads for the DRB tasks and one thread for the sequential tasks while running the test program 10- and 100-times. For the competitor PEQcheck [19], we use a setup similar to [19]. For the DRB tasks, PEQcheck combines the PEQcheck encoding[14] (revision 9dc36b) and verifier CIVL [42] (version 1.20_5259) using the theorem prover Z3 [27] (version 4.8.7). We restrict CIVL to two threads, set its timeout to 5 min, and disable the division by zero and memory leakage checks. For the sequential tasks, PEQcheck combines the PEQcheck encoding with verifier CPAchecker [7] (version 2.0). For verification, we use CPAchecker's default analysis, which is also limited to 5 min.

Environment. We use a time limit of 5 min per task and run our experiments on an Ubuntu 20.04 machine with an Intel Core i7 (1.8 GHz) and 32 GB of RAM.

4.2 Experiments

RQ 1: How effective is PEQtest with minimal resources? To answer this research question, we look at PEQtest's results for the low effort configuration (1 thread, 1 run). For the DataRaceBench (DRB) tasks (left) and the sequential (SEQ) tasks (right), Tab. 1 shows for all three PEQtest configurations the absolute and relative number[15] of correctly detected inequivalences, the number of missed inequivalences, i.e., inequivalences that are not detected, the number of equivalent tasks for which an inequivalence is incorrectly detected (i.e., the false alarms), and the number of equivalent tasks for which no inequivalence is detected. For the two classes in which no inequivalence is detected (missed inequivalence or correctly detected no inequivalence), we also distinguish between the two reasons for not detecting inequivalences: (1) no inequivalences are reported during test program execution and (2) task not completed, e.g., test program generation failed or a timeout occurred during test program generation or execution.

Looking at the first two columns of the DRB tasks and the two columns of the SEQ tasks in Tab. 1, which show the results of the low effort configuration, we observe that for our examples PEQtest does not report any false alarms, i.e., the number of incorrectly detected inequivalences is zero. Thus, we have 100 % precision for inequivalence detection. More surprisingly, PEQtest detects more than half of the inequivalences (i.e., recall $> 50\,\%$) with its low effort configuration and, thus, without parallel execution in case of the parallelized DRB tasks. Studying the detected inequivalences, we observe that almost all the detected inequivalent DRB tasks use a variable to which data-sharing attribute (first)private is assigned and that is visible, but typically not live after the parallelized code segment. The data-sharing attribute makes the variable thread-local during execution of the parallelized code segment and prevents that the thread-local variable values become available after the parallelized code segment.

[14] https://git.rwth-aachen.de/svpsys-sw/FECheck

[15] The relative numbers are the absolute numbers divided by the total number of equivalent and inequivalent tasks, respectively.

Table 1: For each of the three PEQTEST configurations, shows for the DRB and sequential (SEQ) tasks the absolute and relative number of tasks for which inequivalence is detected correctly, is missed, is detected incorrectly, and is correctly not detected. If no inequivalence is detected, the table also distinguishes between no inequivalence reported (i.e., no inequivalence observed in runs) and task is not completed due to a timeout or failure.

	DRB tasks						SEQ tasks	
	1 thread		2 threads				1 thread	
	1 run		10 runs		100 runs		1/10/100 runs	
correctly detected inequivalence	58	(55 %)	72	(68 %)	**74**	**(70 %)**	6	(60 %)
missed inequivalence	48	(45 %)	34	(32 %)	**32**	**(20 %)**	4	(40 %)
no inequivalence reported	38	(35 %)	24	(22 %)	**22**	**(17 %)**	3	(30 %)
task not completed	10	(10 %)	10	(10 %)	10	(10 %)	1	(10 %)
incorrectly reported inequivalence	0	(0 %)	0	(0 %)	0	(0 %)	0	(0 %)
correctly detected no inequivalence	26	(100 %)	26	(100 %)	26	(100 %)	5	(100 %)
no inequivalence reported	22	(85 %)	22	(85 %)	22	(85 %)	4	(80 %)
task not completed	4	(15 %)	4	(15 %)	4	(15 %)	1	(20 %)

Furthermore, many of the detected inequivalent sequential tasks are inequivalent for many different input values. We conclude that inequivalences caused by the discussed data-sharing attributes or input-insensitive inequivalences can easily be detected with a single run and thread.

RQ 2: Does PEQtest's effectiveness increase when given more resources and what are the costs? First, we examine whether PEQTEST performs better if we increase the resources for testing, i.e., the number of runs and for parallelized programs also the number of threads used during test program execution. Comparing the results of our three PEQTEST configurations (Tab. 1), we observe that there is no difference for the sequential tasks. The reason is that one can only detect the missed inequivalences with particular inputs whose random generation is unlikely. For the DRB tasks, however, the number of correctly detected inequivalences increases and the number of missed inequivalences decreases when providing more resources. All other entries stay the same. Hence, PEQTEST's effectiveness may increase (i.e., its recall increases) when we allow it to use more resources. Especially, using more than one thread for parallelized programs increases the effectiveness significantly, as one could have expected. For our examples, using 100 instead of 10 runs hardly improves PEQTEST's effectiveness. In general, PEQTEST misses inequivalences in the DRB tasks if the generation of the test program fails (10 tasks). In addition, it misses inequivalences for SIMD constructs (2 tasks), inequivalences depending on thread scheduling (13 tasks), and inequivalences in I/O behavior (7 tasks), e.g., values written via `printf`, which our implementation does not support yet[16].

[16] Support for I/O can be added by writing all outputs to the checkpoint.

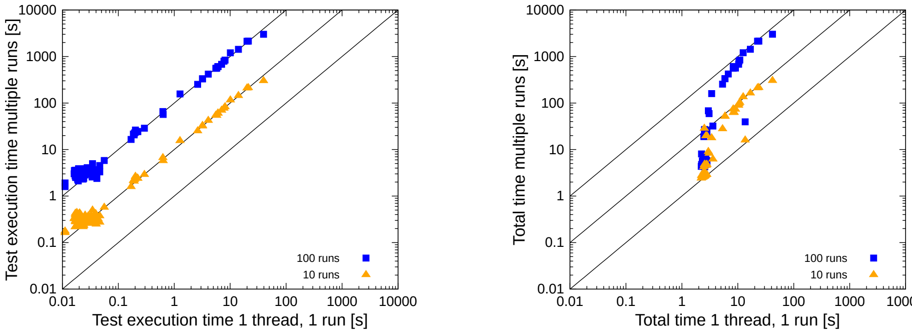

Fig. 4: Per task compare execution time of all test program runs (left) and total runtime of PEQTEST (right) in low effort configuration (1 thread, 1 run) against the other two configurations (2 threads for DRB tasks and 1 thread for sequential tasks, and 10 (▲) or 100 (■) runs)

Second, we examine the costs for increasing PEQTEST's resources for test program execution. To this end, we look at the execution times PEQTEST consumes for all test program runs and the total execution time (test program generation and execution). Figure 4 compares for each task that does not belong to one of the task not completed categories the times for the low effort configuration (1 thread, 1 run, x-axis) with the other two configurations of PEQTEST. As one could have expected, the scatter plot on the left-hand side of Fig. 4 shows that the execution times for the test programs scale linearly with the number of runs. A similar behavior can often be observed when the total time of the low effort configuration is not dominated by the test program generation ($> 3\,s$).

In summary, providing more resources often increases PEQTEST's effectiveness while causing at most a linear increase of runtime costs. In particular for parallelized tasks, using more than one thread is beneficial. However, we require many runs of the generated test program to find schedule-dependent or input-sensitive inequivalences.

RQ 3: How does PEQtest compare against state-of-the-art? We compare PEQTEST's configuration using 100 runs with equivalence checker PEQCHECK [19], which also performs localized checks, but relies on verification. Since PEQTEST's and PEQCHECK's definition of functional equivalence differ (PEQTEST considers all variables, while PEQCHECK only considers live variables), we restrict the comparison of PEQTEST and PEQCHECK to those 72 DRB tasks and 8 sequential tasks that (1) are either equivalent or inequivalent for both notions of equivalence and (2) in which the code segments affect at least one variable that is live afterwards.

Table 2 shows the results of PEQTEST and PEQCHECK on the restricted benchmark. The structure of Tab. 2 is the same as Tab. 1. Looking at Tab. 2, we first observe that both approaches do not incorrectly detect an inequivalence, i.e., they do not report false alarms. Hence, the precision for inequivalence detection is

Table 2: For PEQTEST and PEQCHECK, shows for the DRB and sequential (SEQ) tasks the absolute and relative number of tasks for which inequivalence is detected correctly, is missed, is detected incorrectly, and is correctly not detected. If no inequivalence is detected, also distinguishes between no inequivalence reported and task is not completed due to a timeout or failure.

	PEQTEST 100 runs				PEQCHECK			
	DRB		SEQ		DRB		SEQ	
correctly detected inequivalence	**39**	**(74 %)**	2	(67 %)	3	(6 %)	**3**	**(100 %)**
missed inequivalence	**14**	**(26 %)**	1	(%)	50	(94 %)	**0**	**(0 %)**
no inequivalence reported	11	(21 %)	0	(33 %)	**1**	**(2 %)**	0	(0 %)
task not completed	**3**	**(5 %)**	1	(33 %)	49	(92 %)	**0**	**(0 %)**
incorrectly detected inequivalence	0	(0 %)	0	(0 %)	0	(0 %)	0	(0 %)
correctly detected no inequivalence	19	(100 %)	5	(100 %)	19	(100 %)	5	(100 %)
no inequivalence reported	15	(79 %)	4	(80 %)	**5**	(26 %)	4	(80 %)
task not completed	**4**	(21 %)	1	(20 %)	14	(74 %)	1	(20 %)

100 %. For the sequential tasks, PEQCHECK detects one additional inequivalent task, for which PEQTEST times out. In contrast, PEQTEST detects significantly more inequivalent DRB tasks (i.e., has a higher recall) and, thus, misses less inequivalent DRB tasks. An important reason for the lower recall of PEQCHECK is that PEQCHECK's inspection fails in 87.5 % of the DRB tasks. The major failure causes are timeouts (30 %), missing support for OpenMP constructs in the verifier CIVL (31 %), and the detection of violations that are unrelated to functional equivalence, e.g., array out of bounds accesses in a verification task, which is generated by PEQCHECK to check functional equivalence. Despite PEQCHECK's worse performance, it can verify the task `DRB076-flush-orig-no.c`, for which PEQTEST failed. Finally, we remark that although PEQTEST has a higher time limit than PEQTEST (namely, 5 min per run instead of 5 min per verification task), there exist only two tasks in which PEQTEST requires more than 5 min in total and PEQCHECK could have profited from a higher time limit.

Summing up, PEQTEST is typically a better choice than PEQCHECK when aiming to find inequivalences. In particular, PEQTEST profits from relying on compiler support of OpenMP constructs and from checking equivalence only for the test inputs. Thus, PEQTEST is well-suited for inequivalence detection, but in contrast to PEQCHECK, which considers all inputs, it rarely proves equivalence.

5 Related Work

Approaches inspecting functional equivalence aim at proving equivalence or detecting behavioral differences. Alternatively, they characterize for which inputs equivalence is ensured.

Proving Functional Equivalence. Approaches proving functional equivalence may use relational verification [6,5],(bi)simulation relations [56,40,14,13], or domain-specific checks [55,9,10,18,25]. Other approaches transform the programs into models and check model equivalence [43,49,41]. ARDiff [4] compares symbolic summaries and Rêve [15] translates the equivalence into Horn constraints. Several approaches [17,38,23,51,42,2,19] encode equivalence checking into programs. Their encoding idea is similar to PEQtest's encoding of the functional equivalence tests. The closest encoding is the encoding of UCKLEE [38], which also use checkpointing, while the other approaches duplicate variables. Despite similar encodings, these approaches do not test, but verify the generated programs. A further difference is that they typically generate more than one program, namely one per changed unit (program [42], function [17,38,23,51], or refactored code segment [2,19]). Each generated program only consists of the functional equivalence check of the respective unit and typically considers all possible inputs. In contrast, PEQtest embeds the equivalence tests into the original program and only considers inputs produced by the original program.

Difference Detection. Relative debugging [3] executes the original and refactored program in parallel and compares the values of user-defined variables or data structure at user-defined program locations, which is more fine-grained than functional equivalence. Nevertheless, several techniques focus on detecting differences of the functional behavior. Differential monitoring [28] applies runtime monitoring that runs two programs, e.g., original and refactored program, in parallel, distributes any input to both programs, compares their outputs, and forwards equivalent outputs to the environment, while aborting in case of inequivalence. Following the idea of differential testing [26], BERT [20], shadow symbolic execution [31], and HyDiff [29] generate tests and execute the generated tests on original and refactored program to detect differences in the behavior. BERT [20] generates inputs to cover the changed code parts. Shadow symbolic execution [31] uses a more advanced test generation that is steered towards internal behavioral differences. HyDiff [29] combines shadow symbolic execution with fuzzing, using the tests from the shadow symbolic execution to steer the fuzzer AFL. In contrast, Qi et al. [36] and eXpress [47] directly aim at generating difference revealing tests. To this end, they steer the test generation to find test inputs that reach a change that affects the output. While the previous techniques use special test generators, Diffut [52] and DiffGen [46] rely on standard test generators. Diffut keeps shadow variables for the original program in the refactored program, wraps the method of the original program to extend it with equivalence checks, and uses JML annotations to force the execution of the wrapped method of the original program while testing the refactored program. DiffGen [46] generates one test driver per changed method that copies the input, executes original and refactored method with original and copied input, respectively, and contains one check per output. DiffGen's encoding idea is similar to PEQtest's encoding of functional equivalence tests, but PEQtest focuses on refactored code segments.

Semantic Characterization of Differences. To provide more information in case of non-equivalence, a few approaches compute or (under)approximate the condition when original and refactored program are equivalent. To this end, they use symbolic execution [34,48], abstract interpretation [21,32,33], or testing [11].

6 Conclusion

While refactorings are necessary to improve software quality, correct refactoring, i.e., a refactoring that does not change the functional behavior of the software, is challenging. Several solutions have been proposed to detect that refactored programs alter the behavior, some of them compare the functional behavior of original and refactored programs.

Approaches checking functional equivalence often use heavyweight (formal) verification. Furthermore, difference detection approaches frequently use dedicated test generators and execute (some of) the non-modified code twice, once for the original and once for the refactored program (function). To overcome these restrictions, we propose PEQtest, which can be used to test (the intended application) or to formally verify functional equivalence. The test program generated by PEQtest—for which we proved that it checks functional equivalence—allows us to rely on compiler support, e.g., for OpenMP, to reuse existing tests or test generators, and at the same time to utilize that refactorings are often local, thus, avoiding to execute non-modified code more than once in each test program execution. To this end, PEQtest replaces each refactored code segment in the program, e.g., a parallelized code segment, by a local check that inspects the equivalence of the corresponding original and refactored code segment.

We implemented PEQtest and evaluated it with the DataRaceBench benchmark suite and sequential refactorings already used to evaluate other functional equivalence checkers. Our experiments show that PEQtest detects many of the inequivalent tasks, e.g., incorrectly parallelized tasks, using a limited amount of resources, while reporting no false alarm. A comparison with the state-of-the-art equivalence checker PEQcheck reveals that PEQtest often performs better.

References

1. Technical "whitepaper" for afl-fuzz, http://lcamtuf.coredump.cx/afl/technical_details.txt (last accessed 2022-01-19)
2. Abadi, M., Keidar-Barner, S., Pidan, D., Veksler, T.: Verifying parallel code after refactoring using equivalence checking. Int. J. Parallel Program. **47**(1), 59–73 (2019)
3. Abramson, D., Foster, I.T., Michalakes, J., Sosič, R.: Relative debugging and its application to the development of large numerical models. In: Proc. SC. p. 51. ACM (1995)
4. Badihi, S., Akinotcho, F., Li, Y., Rubin, J.: ARDiff: scaling program equivalence checking via iterative abstraction and refinement of common code. In: Proc. FSE. pp. 13–24. ACM (2020)
5. Barthe, G., Crespo, J.M., Kunz, C.: Relational verification using product programs. In: Proc. FM. pp. 200–214. LNCS 6664, Springer (2011)
6. Benton, N.: Simple relational correctness proofs for static analyses and program transformations. In: Proc. POPL. pp. 14–25. ACM (2004)

7. Beyer, D., Keremoglu, M.E.: CPAchecker: A tool for configurable software verification. In: Proc. CAV. pp. 184–190. LNCS 6806, Springer (2011)
8. Beyer, D., Löwe, S., Novikov, E., Stahlbauer, A., Wendler, P.: Precision reuse for efficient regression verification. In: Proc. FSE. pp. 389–399. ACM (2013)
9. Blom, S., Darabi, S., Huisman, M.: Verification of loop parallelisations. In: Proc. FASE. pp. 202–217. LNCS 9033, Springer, Berlin (2015)
10. Blom, S., Darabi, S., Huisman, M., Safari, M.: Correct program parallelisations. STTT (2021)
11. Böhme, M., d. S. Oliveira, B.C., Roychoudhury, A.: Partition-based regression verification. In: Proc. ICSE. pp. 302–311. IEEE (2013)
12. Cadar, C., Dunbar, D., Engler, D.R.: KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proc. OSDI. pp. 209–224. USENIX Association (2008)
13. Churchill, B.R., Padon, O., Sharma, R., Aiken, A.: Semantic program alignment for equivalence checking. In: Proc. PLDI. pp. 1027–1040. ACM (2019)
14. Dahiya, M., Bansal, S.: Black-box equivalence checking across compiler optimizations. In: Proc. APLAS. pp. 127–147. LNCS 10695, Springer (2017)
15. Felsing, D., Grebing, S., Klebanov, V., Rümmer, P., Ulbrich, M.: Automating regression verification. In: Proc. ASE. pp. 349–360. ACM (2014)
16. Fowler, M.: Refactoring - Improving the Design of Existing Code. Addison-Wesley (1999)
17. Godlin, B., Strichman, O.: Regression verification. In: Proc. DAC. pp. 466–471. ACM (2009)
18. Jakobs, M.C.: PatEC: Pattern-based equivalence checking. In: Proc. SPIN. LNCS 12864, Springer, Cham (2021)
19. Jakobs, M.C.: PEQcheck: Localized and context-aware checking of functional equivalence. In: Proc. FormaliSE. pp. 130–140. IEEE (2021)
20. Jin, W., Orso, A., Xie, T.: Automated behavioral regression testing. In: Proc. ICST. pp. 137–146. IEEE (2010)
21. Kawaguchi, M., Lahiri, S., Rebelo, H.: Conditional equivalence. Tech. rep., Microsoft Research (2010)
22. Kundu, S., Tatlock, Z., Lerner, S.: Proving optimizations correct using parameterized program equivalence. In: Proc. PLDI. pp. 327–337. ACM (2009)
23. Lahiri, S.K., Hawblitzel, C., Kawaguchi, M., Rebêlo, H.: SYMDIFF: A language-agnostic semantic diff tool for imperative programs. In: Proc. CAV. pp. 712–717. LNCS 7358, Springer (2012)
24. Liao, C., Lin, P.H., Asplund, J., Schordan, M., Karlin, I.: DataRaceBench: A benchmark suite for systematic evaluation of data race detection tools. In: Proc. SC. pp. 11:1–11:14. ACM (2017)
25. Malík, V., Vojnar, T.: Automatically checking semantic equivalence between versions of large-scale C projects. In: Proc. ICST. pp. 329–339. IEEE (2021)
26. McKeeman, W.M.: Differential testing for software. Digital Technical Journal **10**(1), 100–107 (1998)
27. de Moura, L.M., Bjørner, N.: Z3: an efficient SMT solver. In: Proc. TACAS. pp. 337–340. LNCS 4963, Springer (2008)
28. Muehlboeck, F., Henzinger, T.A.: Differential monitoring. In: Proc. RV. pp. 231–243. LNCS 12974, Springer (2021)
29. Noller, Y., Pasareanu, C.S., Böhme, M., Sun, Y., Nguyen, H.L., Grunske, L.: HyDiff: Hybrid differential software analysis. In: Proc. ICSE. pp. 1273–1285. ACM (2020)
30. OpenMP: OpenMP application programming interface (version 5.1). Tech. rep., OpenMP Architecture Review Board (2020)

31. Palikareva, H., Kuchta, T., Cadar, C.: Shadow of a doubt: Testing for divergences between software versions. In: Proc. ICSE. pp. 1181–1192. ACM (2016)
32. Partush, N., Yahav, E.: Abstract semantic differencing for numerical programs. In: Proc. SAS. pp. 238–258. LNCS 7935, Springer (2013)
33. Partush, N., Yahav, E.: Abstract semantic differencing via speculative correlation. In: Proc. OOPSLA. pp. 811–828. ACM (2014)
34. Person, S., Dwyer, M.B., Elbaum, S.G., Pasareanu, C.S.: Differential symbolic execution. In: Proc. FSE. pp. 226–237. ACM (2008)
35. Person, S., Yang, G., Rungta, N., Khurshid, S.: Directed incremental symbolic execution. In: Proc. PLDI. pp. 504–515. ACM (2011)
36. Qi, D., Roychoudhury, A., Liang, Z.: Test generation to expose changes in evolving programs. In: Proc. ASE. pp. 397–406. ACM (2010)
37. Quinlan, D., Liao, C.: The ROSE source-to-source compiler infrastructure. In: Cetus users and compiler infrastructure workshop, in conjunction with PACT. vol. 2011, pp. 1–3. Citeseer (2011)
38. Ramos, D.A., Engler, D.R.: Practical, low-effort equivalence verification of real code. In: Proc. CAV. pp. 669–685. LNCS 6806, Springer (2011)
39. Sery, O., Fedyukovich, G., Sharygina, N.: Incremental upgrade checking by means of interpolation-based function summaries. In: Proc. FMCAD. pp. 114–121. IEEE (2012)
40. Sharma, R., Schkufza, E., Churchill, B.R., Aiken, A.: Data-driven equivalence checking. In: Proc. OOPSLA. pp. 391–406. ACM (2013)
41. Shashidhar, K.C., Bruynooghe, M., Catthoor, F., Janssens, G.: Functional equivalence checking for verification of algebraic transformations on array-intensive source code. In: Proc. DATE. pp. 1310–1315. IEEE (2005)
42. Siegel, S.F., Zheng, M., Luo, Z., Zirkel, T.K., Marianiello, A.V., Edenhofner, J.G., Dwyer, M.B., Rogers, M.S.: CIVL: the concurrency intermediate verification language. In: Proc. SC. pp. 61:1–61:12. ACM (2015)
43. Siegel, S.F., Zirkel, T.K.: TASS: the toolkit for accurate scientific software. Mathematics in Computer Science **5**(4), 395–426 (2011)
44. Steinhöfel, D.: REFINITY to model and prove program transformation rules. In: Proc. APLAS. pp. 311–319. LNCS 12470, Springer (2020)
45. Sultana, N., Thompson, S.J.: Mechanical verification of refactorings. In: Proc. PEPM. pp. 51–60. ACM (2008)
46. Taneja, K., Xie, T.: DiffGen: Automated regression unit-test generation. In: Proc. ASE. pp. 407–410. IEEE (2008)
47. Taneja, K., Xie, T., Tillmann, N., de Halleux, J.: eXpress: Guided path exploration for efficient regression test generation. In: Proc. ISSTA. pp. 1–11. ACM (2011)
48. Trostanetski, A., Grumberg, O., Kroening, D.: Modular demand-driven analysis of semantic difference for program versions. In: Proc. SAS. pp. 405–427. LNCS 10422, Springer (2017)
49. Verdoolaege, S., Janssens, G., Bruynooghe, M.: Equivalence checking of static affine programs using widening to handle recurrences. In: Proc. CAV. pp. 599–613. LNCS 5643, Springer (2009)
50. Verma, G., Shi, Y., Liao, C., Chapman, B.M., Yan, Y.: Enhancing DataRaceBench for evaluating data race detection tools. In: Proc. Correctness@SC. pp. 20–30. IEEE (2020)
51. Wood, T., Drossopoulou, S., Lahiri, S.K., Eisenbach, S.: Modular verification of procedure equivalence in the presence of memory allocation. In: Proc. ESOP. pp. 937–963. LNCS 10201, Springer (2017)

52. Xie, T., Taneja, K., Kale, S., Marinov, D.: Towards a framework for differential unit testing of object-oriented programs. In: Proc. AST. pp. 17–23. IEEE (2007)
53. Yang, G., Dwyer, M.B., Rothermel, G.: Regression model checking. In: Proc. ICSM. pp. 115–124. IEEE (2009)
54. Yoo, S., Harman, M.: Regression testing minimization, selection and prioritization: a survey. Softw. Test. Verification Reliab. **22**(2), 67–120 (2012)
55. Yu, F., Yang, S., Wang, F., Chen, G., Chan, C.: Symbolic consistency checking of OpenMP parallel programs. In: Proc. LCTES. pp. 139–148. ACM (2012)
56. Zaks, A., Pnueli, A.: CoVaC: Compiler validation by program analysis of the cross-product. In: Proc. FM. pp. 35–51. LNCS 5014, Springer (2008)