Lockstep Composition for Unbalanced Loops

Ameer Hamza and Grigory Fedyukovich

Florida State University, Tallahassee, FL, USA,
ahamza@fsu.edu, grigory@cs.fsu.edu

Abstract. Equivalence checking of two programs is often reduced to the safety verification of a so-called product program that aligns the programs in lockstep. However, this strategy is not applicable when programs have arbitrary loop structures, e.g., the numbers of loops vary. We introduce an automatic iterative abstraction-refinement-based technique for checking equivalence of a single-loop program and a program which has a series of consecutive loops. Our approach decomposes the single loop into a sequence of separate loops thus reducing the main problem to a series of equivalence-checking problems for pairs of loops. Since due to the decomposition, these problems become abstract, our approach iteratively refines the decomposed loops and lifts useful information across them. Our second contribution is a procedure for the alignment of loops with counters and explicit bounds that cannot be composed in lockstep. We have implemented the approach and successfully evaluated it on two suites, one with benchmarks containing different numbers of loops and the other containing benchmarks that need alignment.

1 Introduction

To gain performance benefits, optimizing compilers perform program transformations such as loop peeling, loop unrolling, loop unswitching. The reliance on many transformations lowers the trust in the computation and motivates us to use automated SMT-based verification to *verify equivalence* of the program before and after the transformation. Specifically, one should prove that for any equal inputs to both programs, their outputs are equal too. The problem is often reduced to construction of a *product program* by *aligning* (or merging) the instructions in lockstep and then determining if the product program meets a safety specification represented by the original relational specification. While effective for many pairs of programs that are relatively close to each other, this strategy may be insufficient for pairs of loopy programs with arbitrary control flow. We target the verification of pairs of programs in which the source program has a single loop, and the target program has a sequence of non-nested loops. Such programs have been extensively studied in the literature [4,23,31] but still are challenging for automated reasoning.

Before proving equivalence, our approach decomposes the loop in the source program into multiple loops such that the structure of this new loop exactly matches the one in the target program. With two structurally similar programs at hand, our approach targets pairs of loops and creates a lockstep composition for each pair. This lets us break our equivalence checking problem into smaller isolated problems, and if each such problem is successfully solved, then the given programs are indeed equivalent. An obvious downside of decomposition is the loss of context: if a program property is defined before the first loop, it may not be available for the second and later loops. For that reason, we have to *refine* the decomposition by extracting the requested properties in the previously considered pairs of loops and pulling them to the currently-considered loops. Technically, this process is driven by counterexamples.

Moreover, when attempting to create a lockstep composition for loops that have different numbers of iterations, we might need to *align* them. When our method can compute an exact number of iterations of both the source and the target, it rearranges the control flow in the source by grouping the iterations in the loop, and extracting selected iterations to either before the loop or after. Such rearranging helps with programs where the number of iterations of one loop is a multiple of other, or is off by few iterations, which is common for optimizations including loop vectorization and loop peeling.

We implemented our equivalence checking algorithm, along with the algorithms to refine and align the loops, in a tool called Alien. On many commonly used public benchmarks [23], Alien is an order of magnitude faster than the most recent (to our knowledge) state-of-the-art tool Counter [14]. Alien can prove equivalence of pairs of user-written programs and it is not bound to any particular compiler unlike many related tools based on translation validation.

We proceed with an overview of the related work in Sect. 2 and a motivating example in Sect. 3. Then, we formally introduce our problem in Sect. 4. The main ingredients of our algorithm are then discussed in Sect. 5, and in Sect. 6. The evaluation is reported in Sect. 7, and conclusion in Sect. 8.

2 Related Work

Relational verification aims at analyzing two different programs or two executions of the same program. This research field has been extensively studied, but since it reduces to safety verification, it is known to be undecidable in general. Relational verification has applications in checking program equivalence, information-flow leakage, incremental verification, etc. To reduce to safety, it is a common practice to convert the programs into a product. The product can be used for relational verification tasks by providing appropriate relational precondition and postcondition. This research trend is pioneered by Barthe et al. [3] who used product programs in Hoare-style proving. More recently, there has been a rise of automated product construnction techniques. e.g., [7, 16, 25, 26].

Creating product program requires that the two programs can be composed in some way, which is usually assumed to be trivial (e.g., lockstep), or provided to the verifier in some form. However, it is not always possible to get the trivial composition. The technique presented by Strichman et al. [36] extends the work of Godlin et al. [12] and it attempts to prove equivalence of two recursive functions having different base-cases and no lockstep composition, by creating an alignment between them. However, the alignment is done using unrolling factors, which are manually provided by the user, for both programs. The technique presented in [34] targets self-composition. It computes a scheduler for an asynchronous execution of both programs using counterexamples and a selection of predicates (e.g., from the user). A more recent work [38] is also a scheduler-driven but mainly targets mutual termination rather than full functional equivalence.

Translation validation techniques, [9, 17, 20, 22, 27, 28, 32, 35, 39], relate the source programs with their compiler outputs to check equivalence. However, it is usually the case that the compiler provides the manner of composition. Many data-driven techniques for proving equivalence, like [5,33], rely on finding a trace alignment between concrete executions of the programs. Such techniques might perform inefficiently when sufficient number of execution traces are not available. They might also require a lot of time for the data runs. The work in [22] performs bounded translation validation at the level of LLVM intermediate representation. The technique looks for a subset of behaviors of the source program in the target to infer equivalence. As the technique is bounded, it may not be sound.

The work by Gupta et al. [14] presents a counterexample-guided algorithm for translation validation of given programs. It explores the space of potential products to find a bisimulation relation between intermediate program locations of the two programs. and prove it via the generation of strong enough inductive invariants. Again, while making the approach flexible, reliance on counterexamples makes it slower, and as we will see from our evaluation (Sect. 7), this approach does not scale well in the cases an alignment needs larger unrollings.

Many techniques use relational verification for regression verification, where two versions of a program are compared for equivalence checking [1, 2, 11, 13, 15, 19, 24, 30, 36, 37]. Such techniques usually assume that two programs are closely related, hence the analysis is usually reduced by either pruning out or abstracting common parts of the programs. Many techniques simplify the process of equivalence checking. Some assume a static relationship between the number of iterations of two loops, in order to prove equivalence [6, 11, 21, 29, 33]. Other techniques create finite unrollings of loops and prove equivalence until a certain bound, e.g., [1,18,22,30]. Our work makes an attempt to relax such assumptions.

3 Illustration on Example

Fig. 1 gives two C programs, the source program contains a single loop and the optimized target programs contains two sequential loops. Our approach aims at proving the equivalence of the source and the target, that is, if variables are initially given equal values (`a = c, b = d, M = X, K = Y`), then their values at the end are equal too. A *lockstep composition* on the programs in Fig. 1 is challenging to construct: 1) it is difficult to compare one loop with two sequential loops, and 2) there are different numbers of iterations taken by programs.

Our method decomposes the source loop into two loops to make it easier to create a product program. It creates two copies of the loop in the source with the same loop body but different loop guards, shown in Fig 2 (left). Specifically,

```
1  int M = nondet(), K = nondet(),
2  a = 0, N = 2*M+1+K, b = 2*M+1;
3  assume(M >= 0 && K >= 0);
4  while(a != N) {
5    b = (a >= b) ? b + 1 : b;
6    a++;
7  }
```

```
1  int X = nondet(), Y = nondet(),
2  c = 1, d = 2*X+1;
3  assume(X >= 0 && Y >= 0);
4  while(c < 2*X+1) c+=2;
5  while(c != 2*X+1+Y) {
6    d++;
7    c++;
8  }
```

Fig. 1: Source (left) and target (right) programs.

```
1   int M = nondet(), K = nondet(),
2   a = 0, N = 2*M+1+K, b = 2*M+1;
3   assume(M >= 0 && K >= 0);
4   while(a != N && a < 2*M+1) {
5     b = (a >= b) ? b + 1 : b;
6     a++;
7   }
8   while(a != N) {
9     b = (a >= b) ? b + 1 : b;
10    a++;
11  }
```

```
1   int M = nondet(), K = nondet(),
2   a = 0, N = 2*M+1+K, b = 2*M+1;
3   assume(M >= 0 && K >= 0);
4   b = (a >= b) ? b + 1 : b; a++;
5   while(a != N && a < 2*M+1) {
6     b = (a >= b) ? b + 1 : b; a++;
7     b = (a >= b) ? b + 1 : b; a++;
8   }
9   assume(N == 2*M+1+K && b == 2*M+1);
10  while(a != N) {
11    b = (a >= b) ? b + 1 : b;  a++; }
```

Fig. 2: Decomposed (left) and refined (right) source programs.

it uses the loop guard for the first loop in the target program, i.e. `c < 2*X+1`, to create `a < 2*M+1` and add it to the guard of the first source loop. It then checks the equivalence of pairs of loops from the decomposed source and the target. However, the first pair of loops (lines 4-7 in the decomposed source, line 4 in the target) is not in lockstep, as for each iteration of the target, the source is expected to iterate twice. Thus, we attempt to construct a lockstep composition by grouping two iterations of the first loop in the decomposed source. However, this results in some residual iterations to be processed before the loop in the decomposed source. After conducting an analysis on the initial states of both loops and the body of the source loop, our approach moves one iteration to be before the loop in the source. This is sufficient to complete the lockstep composition and prove that the first pair of loops are equivalent.

Similarly, the approach considers the second pair of loops (lines 8-11 in the decomposed source, lines 5-8 in the target). To prove that the loops are in lockstep we are missing the information that `N = 2*M+1+K` and `b = 2*M+1`, which is available at the beginning of the program, but not in the middle of it. We say that these equalities *refine* the composition of the second loops, and they are added as an assumption before the start of the second loop (the refined source program is given in Fig. 2 (right)). The refinement makes it possible to both create the lockstep composition and prove the equivalence of both pairs of loops. The analysis terminates with the verdict that both programs are equivalent.

4 Preliminaries

We follow the *Satisfiability Modulo Theories* (SMT) background and notation to present the contributions. The goal of SMT is either to find an assignment to variables of a first-order logic formula that makes it true (written $m \models \varphi$, where m is a model, and φ is a formula), or prove its non-existence (also called unsatisfiability, denoted $\varphi \implies \bot$). For formulas φ, ψ, if every model of φ satisfies ψ, we say that φ is logically stronger than ψ (written $\varphi \implies \psi$). We write *ite* for an if-then-else.

4.1 Constrained Horn Clauses

Throughout the paper, we use the notion of *Constrained Horn Clauses (CHCs)* as a mean to represent the programs containing arbitrary number of loops.

Definition 1. A *Constrained Horn Clause C* over a set of uninterpreted relation symbols R is a (universally quantified, implicitly) formula in first-order logic that has the form of one of the three implications (namely a fact, an inductive clause and a query, respectively):

$$\phi(V_1) \implies L_1(V_1) \qquad L_1(V_1) \land \ldots \land L_n(V_n) \land \psi(V_1, \ldots, V_{n+1}) \implies L_{n+1}(V_{n+1})$$
$$L_1(V_1) \land \ldots \land L_k(V_k) \land \pi(V_1, \ldots, V_k) \implies \bot$$

where for all i, $L_i \in R$ are uninterpreted predicate symbols, V_i are implicitly quantified vectors of variables, and some L_i and L_j might be the same. All formulas ϕ, ψ, π are fully interpreted.

Throughout, we assume that each single loop is represented by two CHCs, e.g.:

$$\mathit{Init}(V) \implies L(V) \qquad L(V) \land \mathit{GTr}(V, V') \implies L(V')$$

where, *Init* represents the initial state of the loop, $\mathit{GTr}(V, V')$ represents one iteration of the loop, which we call a *guarded transition.* For convenience, we split $\mathit{GTr}(V, V')$ to $\mathit{Tr}(V, V') \land G(V)$, where G encodes a guard over the variables at the beginning of transition, and *Tr* has no additional guard.

Definition 2. Given a set R of uninterpreted predicates and a set H of CHCs over R, we say that H is *satisfiable* if there exists an interpretation for every $L \in R$ that makes all implications in H valid.

Solutions for CHC systems are called *inductive invariants.* If a CHC system is unsatisfiable, there exists a counterexample showing a bad state is reachable.

4.2 Relational Verification

The problems of equivalence checking and lockstep composability are the instances of a more general problem of *relational verification.* In this section, we introduce it in a simple case for two systems containing a single loop each.

Definition 3. Given two single-loop CHC systems over $L_{\{1,2\}} \in R$ with initial states $\mathit{Init}_{\{1,2\}}$ and guarded transition bodies $\mathit{GTr}_{\{1,2\}}$, resp., a relational precondition *pre* and a relational postcondition *post*, the problem of *relational verification* can be formulated as the satisfiability of the following CHC system:

$$\mathit{Init}_1(V) \implies L_1(V, V) \qquad \mathit{Init}_2(V) \implies L_2(V, V)$$
$$L_1(V_0, V) \land \mathit{GTr}_1(V, V') \implies L_1(V_0, V') \qquad L_2(V_0, V) \land \mathit{GTr}_2(V, V') \implies L_2(V_0, V')$$
$$\mathit{pre}(V_0, W_0) \land L_1(V_0, V) \land L_2(W_0, W) \land \neg \mathit{post}(V, W) \implies \bot$$

Here, both loop systems are augmented with an additional variable (at the first argument of $L_{\{1,2\}}$) to keep track of the initial values of variables.

To solve the problem, formulated as a complex nonlinear CHC, we need to find *individual* invariants for both loops, which is difficult [7,25]. Instead, we aim at simplifying the problem for certain classes of programs. Specifically, it often can be reduced to safety verification via so called *lockstep composition.*

Definition 4 (Lockstep-composability). Given two single-loop CHC systems and a relational precondition *pre*, a *lockstep composition* exists if 1) the following CHC system is satisfiable:

$$\begin{aligned} pre(V_1,V_2) \land Init_1(V_1) \land Init_2(V_2) &\implies L_{1,2}(V_1,V_2) \\ L_{1,2}(V_1,V_2) \land GTr_1(V_1,V'_1) \land GTr_2(V_2,V'_2) &\implies L_{1,2}(V'_1,V'_2) \\ L_{1,2}(V_1,V_2) \land G_1(V_1) \neq G_2(V_2) &\implies \bot \end{aligned}$$

where $L_{1,2} \in R$ is an uninterpreted predicate symbol, an interpretation of which corresponds to a *relational invariant*, and G_1 and G_2 represent the loop guards and 2) the body of the first CHC is satisfiable.

Intuitively, the first CHC constraints the values of input variables to be related through *pre* (and also, *pre* should be consistent with both *Init*-s.). The second CHC encodes a synchronous computation of both loops. The third CHC ensures that inside the product loop both G_1 and G_2 should be true, and outside the loop both G_1 and G_2 should be false. This implies that the numbers of steps in two lockstep-composable programs under some *pre* are the same.

The following lemma lets us reduce a relational verification problem to a safety verification problem computed after *merging* the loops and then use existing invariant generation techniques for solving relational verification problems. Note that due to the lockstep, both loop guards are always equal, so it is enough to conjoin the negation of only one of the loop guards to the query.

Lemma 1. *Given a relational verification problem over two systems over $L_{\{1,2\}} \in R$ representing single loops, pre, and post, if the systems are lockstep-composable under pre, and the following CHC problem is satisfiable, then post holds at the end of these loops.*

$$\begin{aligned} pre(V_1,V_2) \land Init_1(V_1) \land Init_2(V_2) &\implies L_{1,2}(V_1,V_2) \\ L_{1,2}(V_1,V_2) \land GTr_1(V_1,V'_1) \land GTr_2(V_2,V'_2) &\implies L_{1,2}(V'_1,V'_2) \\ L_{1,2}(V_1,V_2) \land \neg G_1(V_1) \land \neg post(V_1,V_2) &\implies \bot \end{aligned}$$

The problem of proving program equivalence is a special case of the relational verification problem where $pre = post$ is a pairwise equality over V_1 and V_2.

5 Equivalence Checking for Unbalanced Loops

In this section, we present our novel equivalence checking algorithm designed for the cases when the source and the target programs have different structures. We first describe a class of the input CHC systems that we target in Sect. 5.1. We then provide a procedure to decompose the source such that we can break the problem of equivalence checking under our limitations into a sequence of smaller problems in Sect. 5.2. We then finalize our core abstraction-refinement schema for equivalence checker in Sect. 5.3.

5.1 Input Limitations and Auxiliary Definitions

We support pairs of programs where the source contains a single loop, and the target possibly contains an arbitrary number of sequential loops. A CHC system of the latter sort that has n loops is called a *flat n-sequence* of loops further in the paper. Here and throughout, we assume that G_S and G_i encode the loop guard for the source loop and the i^{th} loop in the target, and that Tr_S and Tr_i encode respective loop bodies without the corresponding guard. Specifically, the shape of a source program that we consider is defined over a single predicate symbol S, and we thus refer to this system as S-system later in the text:

$$\mathit{Init}_S(V_S) \implies S(V_S) \qquad S(V_S) \wedge G_S(V_S) \wedge \mathit{Tr}_S(V_S, V'_S) \implies S(V'_S)$$

The flat n-sequence is defined over n predicate symbols $T_1,\ldots,T_n$, and is referred to as T-system in the paper:

$$\begin{aligned} \mathit{Init}_T(V_T) &\implies T_1(V_T) & T_1(V_T) \wedge G_1(V_T) \wedge \mathit{Tr}_1(V_T, V'_T) &\implies T_1(V'_T) \\ T_1(V_T) \wedge \neg G_1(V_T) &\implies T_2(V_T) & T_2(V_T) \wedge G_2(V_T) \wedge \mathit{Tr}_2(V_T, V'_T) &\implies T_2(V'_T) \\ & & &\ldots \\ T_{n-1}(V_T) \wedge \neg G_{n-1}(V_T) &\implies T_n(V_T) & T_n(V_T) \wedge G_n(V_T) \wedge \mathit{Tr}_n(V_T, V'_T) &\implies T_n(V'_T) \end{aligned}$$

There is one *fact* CHC, in which Init_T represents the initial state of the program. There are n *inductive* clauses, i.e., for each $i \in [1, n]$, the i^{th} inductive clause has occurrence of symbol T_i on both sides of the implication. There are also $n - 1$ non-inductive clauses that encode transitions between adjacent loops, so $\neg G_i$ represents the condition when loop i exits.

Example 1. The source in Fig. 1 is encoded to CHCs as follows:

$$\begin{aligned} a = 0 \wedge N = 2{*}M{+}1{+}K \wedge b = 2{*}M{+}1 \wedge M \geq 0 \wedge K \geq 0 &\implies S(a, b, M, K, N) \\ S(a, b, M, K, N) \wedge a \neq N \wedge a' = a{+}1 \wedge b' = \mathit{ite}(a \geq b, b{+}1, b) &\implies S(a', b', M, K, N) \end{aligned}$$

Example 2. The CHC encoding of the target program in Fig 1 is given as:

$$\begin{aligned} c = 1 \ \wedge d = 2{*}X + 1 \wedge X \geq 0 \wedge Y \geq 0 &\implies T_1(c, d, X, Y) \\ T_1(c, d, X, Y) \wedge c < 2{*}X + 1 \ \wedge c' = c + 2 &\implies T_1(c', d, X, Y) \\ T_1(c, d, X, Y) \wedge c \geq 2{*}X + 1 &\implies T_2(c, d, X, Y) \\ T_2(c, d, X, Y) \wedge c \neq 2{*}X + 1 + Y \ \wedge c' = c{+}1 \ \wedge d' = d{+}1 &\implies T_2(c', d', X, Y) \end{aligned}$$

We introduce a concept needed for the presentation in the next section, where by $A[B/C]$, we denote an expression with all instances of C replaced by B:

Definition 5. Given a CHC system H over predicate symbols $L_1, \ldots, L_n$, an L_i-projection of H (denoted $H \mid_i$) is defined as $\{C[\top/L_j(\cdot)] \mid C \in H, j \neq i\}$.

That is, our projection replaces all applications of all predicate symbols except of L_i by true. Clearly, some CHCs then can be simplified to true, and we assume that they are removed from the projection.

Example 3. Let H be a T-system from Example 2, then $H\mid_2$ has two CHCs:

$$\begin{aligned} c \geq 2{*}X + 1 &\implies T_2(c,d,X,Y)\\ T_2(c,d,X,Y) \land c \neq 2{*}X + 1 + Y \ \land c' = c{+}1 \ \land d' = d{+}1 &\implies T_2(c',d',X,Y) \end{aligned}$$

5.2 Equivalence Checking by Decomposition

Our main insight on checking equivalence of a source loop and a flat n-sequence is that if the source breaks into n distinct loop-chunks, and if each of these chunks is equivalent to the corresponding loop from the n-sequence, then the actual programs are equivalent too. We thus present a decomposition of the source into a sequence of n new loops that gives us the basis for comparing the two CHC systems. A decomposition of S-system into an n-flat sequence is done by:

1. introducing n fresh predicate symbols $S_1, \ldots, S_n$,
2. cloning the inductive CHC n times and replacing S with S_i in each clone,
3. creating $n-1$ non-inductive CHCs between S_i and S_{i+1}, and
4. introducing additional guard predicates $P_1, \ldots, P_{n-1}$ to schedule chunks of iterations of the S-loop to either of the new n loops. To sum up:

$$\begin{aligned} \mathit{Init}_S(V_S) &\implies S_1(V_S)\\ S_1(V_S) \land G_S(V_S) \land P_1(V_S) \land \mathit{Tr}_S(V_S, {V'}_S) &\implies S_1({V'}_S)\\ S_1(V_S) \land \neg(G_S(V_S) \land P_1(V_S)) &\implies S_2(V_S)\\ &\ldots\\ S_n(V_S) \land G_S(V_S) \land \mathit{Tr}_S(V_S, {V'}_S) &\implies S_n({V'}_S) \end{aligned}$$

For *any interpretation* of $P_1, \ldots, P_{n-1}$, the CHC system constructed above is semantically equivalent to the S-system, for the following three reasons. First, no matter how many iterations the first $n-1$ loops conduct, all the remaining ones will be conducted in the last loop. Second, all n loops still use the original guard G, and if it is exceeded in some i^{th} loop, then all the remaining $i+1^{\text{th}}, \ldots, n^{\text{th}}$ loops will be just skipped. Lastly, all these loops perform exactly the same operations as the original loop since Tr_S is copied to all of them. We will instantiate all the P-predicates on demand in our CounterExample Guided Abstraction Refinement (CEGAR) loop.

The CEGAR loop for our equivalence checking problem is outlined in Alg. 1. It begins with decomposing the S-system into a flat n-sequence, as defined above. The P-predicates are created from G_i guards in T-system by rewriting T-variables to S-variables, $i \in [1, n-1]$:

$$P_i(V) \overset{\text{def}}{=} \exists V'. G_i(V') \land \mathit{pre}(V, V')$$

Algorithm 1: DecomposeAndCheck(S, T, Pre, $Post$)

```
Input: $S$-system, $T$-system, relational pre and post-conditions
       $Pre = \langle pre_1, pre_2, \ldots, pre_n \rangle$ and $Post = \langle post_1, post_2, \ldots, post_n \rangle$
Output: $res \in \langle$EQUIV, UNKNOWN$\rangle$
1  $S' \leftarrow$ DECOMPOSE$(S, n)$;
2  for $i \leftarrow 1; i \leq n; i \leftarrow i{+}1$ do
3      $S_i \leftarrow S' |_i$; $T_i \leftarrow T |_i$;
4      while $true$ do
5          $aligned \leftarrow \bot$; $refined_{1,2} \leftarrow \bot$;
6          $ST_i \leftarrow$ GETPRODUCT$(S_i, T_i, pre_i)$;
7          Let $Init$ be the body of the fact CHC in $ST_i$;
8          $res \leftarrow$ CHECKSAT$(Init)$;
9          if $res$ then
10             $\langle inv, cex \rangle \leftarrow$ CHECKSAT$(ST_i \cup \{L \wedge (G_s \wedge P_i) \neq G_i \implies \bot\})$;
11         if $\neg res \vee cex \notin \varnothing$ then
12             $\langle aligned, S_i \rangle \leftarrow$ ALIGNCHCS$(S_i, T_i, pre_i)$;
13             if $aligned$ then continue;
14         else
15             $\langle inv, cex \rangle \leftarrow$ CHECKSAT$(ST_i \cup \{L \wedge \neg G_i \wedge \neg post_i \implies \bot\})$;
16             if $cex \in \varnothing$ then break;
17         $\langle refined_1, S_1, \ldots, S_i \rangle \leftarrow$ REFINE$(S_1, \ldots, S_i, cex)$;
18         $\langle refined_2, T_1, \ldots, T_i \rangle \leftarrow$ REFINE$(T_1, \ldots, T_i, cex)$;
19         if $\neg(refined_1 \vee refined_2 \vee aligned)$ then return UNKNOWN;
20 return EQUIV;
```

Note that the relational precondition *pre* is assumed to be a conjunction of equalities. This gives us two flat n-sequences, which lets us consider pairs of loops (line 2) from both systems separately. Each such CHC system is created by applying the projection from Def. 5. In a sense, this is an *abstraction* of the original system since by isolating one loop (say, i^{th}), we lose the state computed all the way from the entry to the program by iterating $i-1$ loops. Aiming to check equivalence for each pair of projections, the algorithm first figures out how/if a lockstep-composition is applicable. We write: $res \leftarrow$ CHECKSAT(fla) to denote a satisfiability check for a (first order) formula fla, and we write:

$$\langle inv, cex \rangle \leftarrow \textsc{checkSAT}(ST_i \cup \{L \wedge \ldots \implies \bot\})$$

to denote this check for the CHC-product ST_i over predicate symbol L with respect to the query written in $\{\ldots\}$. The check returns either an inductive invariant (i.e., an interpretation of L) or a counterexample. Before checking for lockstep, the compatibility of the initial states needs to be checked, i.e., if the body of the fact is satisfiable (line 8). If it succeeds, each check of the lockstep-composability is reduced by Def. 4 to a CHC satisfiability check, and it uses both guards in the CHC query (line 9). If either the initial-states check or the lockstep check fails, the algorithm uses a method for alignment of projections discussed in detail in Sect. 6. If aligned, we continue with the next iteration of the loop, attempting to prove lockstep composition and equivalence of the projections.

Algorithm 2: $\textsc{Refine}(Q_1, \ldots, Q_i, \mathit{cex})$

Input: Set of i CHC systems $Q_1, \ldots, Q_i$ over L; and counterexample cex
Output: $\mathit{res} \in \langle\langle \bot, \cdot \rangle, \langle \top,$ refined systems $Q_1, \ldots, Q_i \rangle\rangle$

```
1  if $i = 1$ then return$\langle \bot, \cdot \rangle$;
2  while $\mathit{cex} \notin \varnothing$ do
3      $\langle \mathit{inv}, \mathit{cex}' \rangle \leftarrow \textsc{checkSAT}(Q_{i-1} \cup \{L(V) \wedge \neg G_{i-1}(V) \wedge \bigwedge_{v \in V} v = \mathit{cex}(v) \implies \bot\})$;
4      if $\mathit{cex}' \in \varnothing$ then
5          assert$(\mathit{inv} \notin \varnothing)$;
6          $\mathit{Fact} \leftarrow \{C \in Q_i \mid C$ has form $\mathit{Init}(V) \implies L(V)\}$;
7          $Q_i \leftarrow Q_i \setminus \{\mathit{Fact}\} \cup \{\mathit{Init}(V) \wedge \mathit{inv}(V) \implies L(V)\}$;
8          return$\langle \top, Q_1, \ldots, Q_i \rangle$;
9      else
10         $\langle \mathit{res}, Q_1, \ldots, Q_{i-1} \rangle \leftarrow \textsc{refine}(Q_1, \ldots, Q_{i-1}, \mathit{cex}')$;
11         if $\neg \mathit{res}$ then return$\langle \bot, \cdot \rangle$;
```

Example 4. Recall CHC systems defined in Examples 1 and 2. In the first iteration, Alg. 1 considers the first pair of loops. The initial-states check at line 8 fails, and thus the loops are aligned at line 12 (to be explained in Example 8).

Whenever two CHC systems are in lockstep, the algorithm utilizes Lemma 1 and checks the product system computed for two isolated loops (line 15) for safety. The success of the check lets the algorithm to continue with the next pair of loops. Otherwise, we receive a counterexample, which might be spurious because of the abstraction. Our refinement procedure then searches for a strengthening of either of the CHC systems (lines 17-18), which is described in more details in the next subsection. If it cannot refine further using the given technique, it returns UNKNOWN (line 19).

5.3 Refinement

Due to the decomposition presented in the previous section, there could be sensitive information that is available in the earlier parts of the programs, but not in the later parts. Alg. 2 gives a refinement procedure needed to propagate useful properties about the programs towards queries. Intuitively, we have to strengthen our relational preconditions, thus improving the chances to prove the safety of the i^{th} CHC product. Recall that in Alg. 1, refinement is invoked for each counterexample which is technically, an assignment to the variables at the initial state of either of the programs being composed into the product CHC.

The key idea is to check if the counterexample is spurious by constructing a scenario in which the $i-1^{\text{th}}$ system can eventually reproduce the values from the counterexample at the end of its execution (line 3). This is reduced technically to a satisfiability check of the corresponding CHC system w.r.t. the "negation" of the counterexample. If it succeeds, then an inductive invariant can be used to strengthen (line 7) the i^{th} system. Otherwise, the algorithm might recursively descend to refining the $i-1^{\text{th}}$ system via finding an invariant for the $i-2^{\text{nd}}$

product, and so on (line 10). For this reason, the algorithm has the while-loop (line 2) that lets to repeat the satisfiability check for some (already strengthened) systems, and it continues till the current system has been refined.

Example 5. Continuing with Example 4, in the second iteration of Alg. 1, the lockstep check[1] does not succeed:

$$a = c \land b = d \land M = X \land Y = K \land (a = N \lor a \geq 2{*}M + 1) \land c \geq 2{*}X + 1 \implies L_2(V)$$
$$L_2(V) \land a \neq N \land a' = a + 1 \land b' = ite(a \geq b, b + 1, b) \land$$
$$c \neq 2{*}X + 1 + Y \land c' = c + 1 \land d' = d + 1 \implies L_2(V')$$
$$L_2(V) \land (a \neq N) \neq (c \neq 2{*}X + 1 + Y) \implies \bot$$

For the CHC system above, a counterexample could be $cex = \{a, c, b, d \mapsto 110, M, K \mapsto 50, N \mapsto 0, X, Y \mapsto 50\}$ because we miss that $N = 2{*}M + 1 + K$, hence lockstep is not possible. Alg. 2 then confirms that this counterexample is spurious by learning this inductive invariant. After adding it to the fact CHC of S_2 and recomputing the product system ST_2, it becomes satisfiable. We then add the following query for equivalence check:

$$L_2(V) \land c = 2{*}X + 1 + Y \land (a \neq c \lor b \neq d \lor M \neq X \lor K \neq Y) \implies \bot$$

which fails because of missing invariant $b = 2{*}M + 1$. After adding it to the fact CHC of S_2 and recomputing the product CHC system, it becomes satisfiable.

As can be seen from this example, the refinement procedure is beneficial for both the lockstep-composability and the equivalence checks in Alg. 1, thus the inner loop in the algorithm can iterate multiple times before terminating with a positive verdict. We note that inductive invariants are in general tricky for finding. Thus, our approach has essential limitations and cannot prove equivalence of programs that require complicated (e.g., quantified) inductive invariants.

6 Aligning Unbalanced Loops

In this section, we present an algorithm for creating alignment between two single-loop CHC systems that have different number of loop iterations. Our new method of *alignment* of an S-projection and a T-projection is based on restructuring the former to become lockstep-composable with the latter. The algorithm identifies if any iterations of the former have to be extracted and placed before the loop and if any iterations have to be grouped and performed at once. These numbers (called *alignment bounds* in the rest of the section) are identified if exact loop bounds of both projections are computable.

6.1 Finding the Number of Iterations

We aim first at computing a function that returns the exact number of iterations of a single loop in terms of input variables, based on the CHC representation.

[1] We abbreviate $\langle a,b,M,K,N,c,d,X,Y \rangle$ with V, and $\langle a',b',M,K,N,c',d',X,Y \rangle$ with V'.

In the technique presented below, the input systems need to have a counter variable that monotonically increments between two extremes that do not change in the loop.[2] Focusing on a single-loop CHC system with initial states *Init* and guarded transition body $G \land \mathit{Tr}$ where G encodes a guard over the variables at the beginning of the transition, and Tr has no additional guard, we wish to find the exact number of the iterations of the corresponding loop. In general, for that, we could consider an augmented CHC system with a fresh decrementing counter.

Definition 6. The exact number of iterations is an interpretation of the function symbol $\mathcal{N}$ that makes the augmented CHC system satisfiable:

$$\begin{aligned} \mathit{Init}(V) \land j = \mathcal{N}(V) &\implies L(V,j) \\ L(V,j) \land G(V) \land \mathit{Tr}(V,V') \land j' = j - 1 &\implies L(V',j') \\ L(V,j) \land \neg G(V) \land j \neq 0 &\implies \bot \end{aligned}$$

For an arbitrary loop, finding $\mathcal{N}$ is difficult and often not possible (e.g., for problems with nondeterminism in the loop). However, for some CHC systems encoding *range-based* loops, i.e., that already have counters, we can attempt to synthesize $\mathcal{N}$ from the information obtained from syntax of CHCs. Specifically, we assume that formula *Init* has the form $i = \mathcal{S}(V) \land \mathit{Init}'(V,i)$ for some variable i and some function $\mathcal{S}$, We also assume that the guard of the transition has the form $i < \mathcal{F}(V) \land G'(V,i)$ for some function $\mathcal{F}$, and Tr has the form $i' = i + \mathcal{D} \land \mathit{Tr}'(V,i,V',i')$ for some positive constant $\mathcal{D} > 0$.

Definition 7. A *range-based* CHC system is the one that has the following form

$$\begin{aligned} \mathit{Init}'(V,i) \land i = \mathcal{S}(V) &\implies T(V,i) \\ T(V,i) \land i < \mathcal{F}(V) \land i' = i + \mathcal{D} \land G'(V,i) \land \mathit{Tr}'(V,i,V',i') &\implies T(V',i') \end{aligned}$$

such that for some inductive invariant *inv* the following hold:

$$\mathit{Tr}'(V,i,V',i') \land \mathit{inv}(V,i) \implies \mathcal{S}(V) = \mathcal{S}(V') \tag{1}$$
$$\mathit{Tr}'(V,i,V',i') \land \mathit{inv}(V,i) \implies \mathcal{F}(V) = \mathcal{F}(V') \tag{2}$$
$$i < \mathcal{F}(V) \land \mathit{inv}(V,i) \implies G'(V,i) \tag{3}$$

To guarantee soundness of our construction, the constraints in the definition above ensure that $\mathcal{S}$ and $\mathcal{F}$ are the tightest bounds for the counter variable i. Specifically, (1) and (2) ensure that i has the lower and the upper bound that do not change throughout the execution, and (3) ensures that the loop does not break before i exceeds $\mathcal{F}(V)$. An invariant *inv* could in simple cases be just $\top$ but often it needs to bring important information from an initial state to an arbitrary iteration. For instance, if a loop has two counters with their own upper and lower bounds, then our analysis can proceed only when we can prove that

[2] A similar technique for a *decrementing counter* is straightforward but omitted for brevity of presentation.

either of the counters exceeds its upper bound *always faster* than another does so. Our running example makes another use of (3), to ensure that the residual guard $G'(V, i)$ is weaker than $i < \mathcal{F}(V)$ strengthened by the invariant.

Example 6. Recall the first loop of the decomposed source of Example 1. It has the guard $a \neq N \wedge a < 2{*}M+1$. We can find invariant $N = 2{*}M+1+K \wedge K \geq 0$. Clearly, since $N = 2{*}M + 1 + K \wedge K \geq 0 \wedge a < 2{*}M + 1 \implies a \neq N$, then $\mathcal{F}(M) \stackrel{\text{def}}{=} 2{*}M + 1$ satisfies (3). With no invariant, $a < 2{*}M + 1 \;\not\Longrightarrow\; a \neq N$.

Lemma 2. *An integer function $\mathcal{N}$ computes the exact number of iterations for a range-based CHC system:*

$$\mathcal{N} \stackrel{\text{def}}{=} (\mathcal{F} - \mathcal{S}) \;\; \textit{div} \;\; \mathcal{D} + (\textit{if} \;\; ((\mathcal{F} - \mathcal{S}) \;\; \textit{mod} \;\; \mathcal{D} = 0) \;\; \textit{then} \;\; 0 \;\; \textit{else} \;\; 1)$$

In practice, the approach is limited to the invariant generation capabilities. If a sufficient invariant for Def. 7 (and thus, Lemma 2) is found, the approach proceeds to align loops. Otherwise, it returns Unknown.

6.2 Identifying Unrolling Depths

If the numbers of iterations can be computed, the approach proceeds to finding alignment bounds ℓ and m that define respectively the number of iterations to be extracted and placed before the loop and the number of iterations to be grouped and performed at once in the loop. These bounds are obtained from the following ingredients:

1. functions $\mathcal{N}_S$ and $\mathcal{N}_T$ to compute the numbers of iterations of the S-projection and the T-projection, respectively;
2. fresh integer variable v_ℓ to represent (a yet unknown) number of iterations to be moved out of the loop in the S-projection,
3. fresh integer variable v_m to represent (a yet unknown) number of iterations to be grouped inside the loop for the S-projection.

Values ℓ and m can be directly taken from a satisfying assignment to variables v_ℓ and v_m for the following SMT query. Intuitively, it equates the total numbers of iterations in the S-projection and the T-projection:

$$Q_{ST} \stackrel{\text{def}}{=} \exists v_\ell, v_m \,.\, \forall V_S, V_T \,.\, (v_\ell \geq 0 \wedge v_m > 0) \wedge \textit{pre}(V_S, V_T) \implies \\ \mathcal{N}_S(V_S) - v_\ell = v_m * \mathcal{N}_T(V_T)$$

Thus, the SMT formula has the form of implication: if *pre* holds, then the number of iterations of one program can be expressed over the number of iterations of another program (and vice versa). If $\mathcal{M} \models Q_{ST}$, then $\ell \stackrel{\text{def}}{=} \mathcal{M}(v_\ell)$, and $m \stackrel{\text{def}}{=} \mathcal{M}(v_m)$.

Example 7. For the first projections in the decomposed source and the target, we generate the following (simplified) SMT query:

$$Q_{ST} = \exists v_\ell, v_m \,.\, (v_\ell \geq 0 \wedge v_m > 0) \wedge M = X \implies 2{*}M + 1 - v_\ell = v_m * X$$

and the solver generates model $\mathcal{M} = \{v_\ell \mapsto 1, v_m \mapsto 2\}$, and $\ell = 1$, and $m = 2$.

6.3 Rearrangement of the Source Projection

Finally, we present the restructuring of the S-projection based on two alignment bounds, ℓ and m, computed in the previous section. The former represents the number of iterations to be moved before the loop, and the latter represents the number of iterations to make a batch inside the loop.[3] We assume that an S-projection is defined using the following two CHCs over a single predicate symbol L: $\mathit{Init}_S(V) \implies L(V)$ and $L(V) \land \mathit{GTr}(V, V') \implies L(V')$.

We define an auxiliary predicate $U(u, V, V')$ that allows us to create an unrolling of arbitrary length: if $u = 0$, the result is the identity formula, otherwise we create u unrollings of the system (GTr_S conjoined u times), then define $\mathit{Init}_S^{(\ell)}$ and $\mathit{GTr}_S^{(m)}$, as follows:

$$\begin{aligned}
U(u, V, V') &\overset{\text{def}}{=} \mathit{ite}(u = 0,\ V' = V,\\
&\qquad \exists V'', \ldots, V^{(u)} \,.\, \mathit{GTr}_S(V, V'') \land \ldots \land \mathit{GTr}_S(V^{(u)}, V'))\\
\mathit{Init}_S^{(\ell)}(V') &\overset{\text{def}}{=} \exists V \,.\, \mathit{Init}_S(V) \land U(\ell, V, V')\\
\mathit{GTr}_S^{(m)}(V, V') &\overset{\text{def}}{=} U(m, V, V')
\end{aligned}$$

Finally, we are ready to define the aligned CHC product used in Alg. 1 (ALIGN-CHCs(S, T, pre)).

Definition 8. Let S and T be two range-based CHC systems, as defined in Def. 7. Let $\mathcal{M} \models Q_{ST}(\mathcal{N}_S, \mathcal{N}_T, v_\ell, v_m, \mathit{pre})$, as defined in Sect. 6.2. Then, the rearranged system S_R is defined as follows:

$$\mathit{Init}_S^{(\mathcal{M}(v_\ell))}(V) \implies L(V) \qquad L(V) \land \mathit{GTr}_S^{(\mathcal{M}(v_m))}(V, V') \implies L(V')$$

Note that S^R and T are in lockstep, and S^R is equivalent to S, both by construction. Thus, after such alignment, our Alg. 1 will proceed to checking the equivalence of S and T by means of checking equivalence of S^R and T.

Example 8. For the first projections in the decomposed source and the target, the lockstep check does not succeed because the body of the fact is unsatisfiable:

$$\begin{aligned}
&a = c \land b = d \land M = X \land Y = K \land a = 0 \land N = 2{*}M{+}1{+}K \land b = 2{*}M{+}1 \land M \geq 0 \land\\
&\qquad K \geq 0 \land c = 1 \land d = 2{*}X{+}1 \land X \geq 0 \land Y \geq 0 \implies L_1(a, b, M, K, N, c, d, X, Y)
\end{aligned}$$

With the bounds computed in Example 7, we compute the following product:

$$\begin{aligned}
&a = 0 \land N = 2{*}M + 1 + K \land b = 2{*}M + 1 \land M \geq 0 \land K \geq 0 \land\\
&a \neq N \land a < 2{*}M + 1 \land a' = a{+}1 \land b' = \mathit{ite}(a \geq b, b{+}1, b) \land\\
&c = 1\ \land d = 2{*}X + 1 \land X \geq 0 \land Y \geq 0 \land\, a' = c \land b' = d \land M = X \land Y = K\\
&\qquad \implies L_1(a', b', M, K, N, c, d, X, Y)\\
&L_1(a, b, M, K, N, c, d, X, Y) \land a \neq N \land a < 2{*}M{+}1 \land a' = a{+}1\ \land\ b' = \mathit{ite}(a \geq b, b{+}1, b)\\
&a' \neq N \land a' < 2{*}M{+}1 \land a'' = a'{+}1 \land b'' = \mathit{ite}(a' \geq b', b'{+}1, b')\\
&c < 2{*}X + 1\ \land c' = c + 2 \implies L_1(a'', b'', M, K, N, c', d, X, Y)
\end{aligned}$$

[3] In practice, it could also be required to move some iterations to after the loop (and our implementation supports it). Then, we split m into $m_1 + m_2$ heuristically and move m_1 iterations to before the loop, and m_2 to after the loop.

7 Evaluation

We have implemented the algorithm for equivalence checking in a tool called Alien[4] on top of the invariant synthesizer FreqHorn that supports integers and arrays (over integers) [10]. Alien takes as input an S-system and a T-system, automatically decomposes the former, creates a sequence of product programs, and delegates the inductive invariant generation to FreqHorn. For solving SMT queries, it uses Z3 [8]. We considered two benchmark suites:

– Test Suite of Vectorization Compilers (TSVC) [23], preprocessed in the way suggested by [5]. TSVC has 152 benchmarks, and 48 of which are either not vectorizable, contain floating point operations, intrinsic functions, or need some extra processing like loop rerolling. We thus experimented on a set of remaining 104 remaining benchmarks. We check equivalence of these programs w.r.t. their optimized versions, both translated to CHCs.
– A subset of 24 multi-phase benchmarks taken from [4,31] in which the phases can be "extracted" from the loops. The optimized versions of these benchmarks have more than one loop, thus necessitating to use our decomposition.

We considered the state-of-the-art tools LLREVE [16], an equivalence checker by Churchill et al. [5], Counter [14], and CHC-Product [25]. However, only Counter was able to solve some of our benchmarks in reasanoble time: Churchill et al. report that the minimum time any benchmark takes to solve is around 2 hours, and it was largely outperformed by Counter in [14].

We thus evaluate our Alien against Counter for both benchmark suites. To run Counter on a pair of manually provided C programs[5], it was configured to apply no optimization to any of the programs. For TSVC benchmarks, we manually pass an unrolling factor 8 required by each benchmark (compare to our approach in which the tool automatically identifies this number). For Alien, we provide two CHC encodings of the program before and after the optimization. We specified a timeout of 15 minutes for both tools.

Alien solved 103 out of 104 TSVC benchmarks. Alien times out on the `s279` benchmark because its invariant synthesizer struggles with finding a helper invariant. Benchmark `s113` requires the approach to automatically synthesize an extra lemma (i.e., cnt>0), in addition to the variable equalities. Alien took 3.7 seconds to solve a benchmark on average: from 1.3 in the best case to 27.4 in the worst case. Among all, 26 (resp. 2) benchmarks require moving iterations before (resp. after) the loop. Counter proved equivalence for 15 benchmarks, it failed to prove equivalence for 9 benchmarks, while the rest (81 benchmarks) timed-out. Its minimum running time is 50.2 seconds, maximum 704 seconds and average 117.4 seconds.

[4] The tool and benchmarks are available at https://github.com/a-hamza-r/aeval/tree/equiv-check.

[5] We consulted https://github.com/compilerai/counter to run tool in our setting. Note that in their paper, the authors evaluated Counter only on compiler-optimized targets. Our case study is different, and it shows that checking equivalence between two arbitrary programs is a harder problem for Counter.

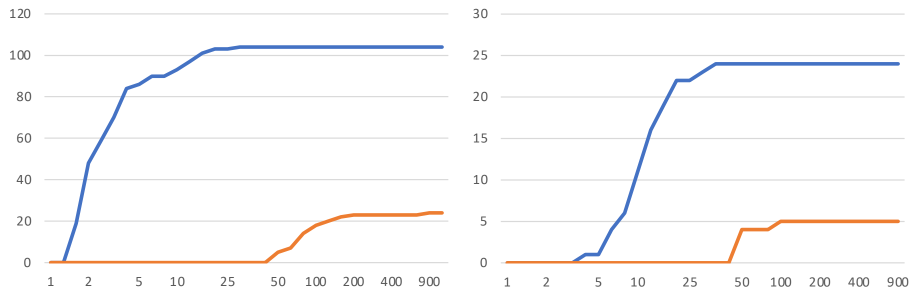

Fig. 3: Cactus plots (left: for TSVC benchmarks, right: for multi-phase benchmarks) comparing running times of ALIEN (blue line) and COUNTER (orange line).

For 24 multi-phase benchmarks, ALIEN proved all of them. COUNTER proved equivalence for 5 benchmarks, it failed to prove equivalence for 3 benchmarks, while the remaining benchmarks timed-out. The minimum, maximum and average times are 3.2, 32.6, and 11.5 seconds, respectively for ALIEN; and 43.8, 106.9, and 56.2 seconds respectively for COUNTER.

A larger picture on the experimental results is given in Fig. 3. The horizontal axes in the cactus plots represent time limit (logarithmic scale), and the vertical axes represent the numbers of benchmarks (linear scale) solved within the corresponding time limits. Intuitively, the plots demonstrate that COUNTER is an order of magnitude slower than our novel approach.

8 Conclusion

We have presented a novel CEGAR-based approach for checking equivalence of two programs containing possibly different number of loops. The technique involves automatic decomposition of one of the programs to match the loops structure of the other, so that the task of equivalence checking of two given programs can be split into a sequence of tasks of equivalence checking of single loops, each of which is solved easier. Since such decomposition comes at a cost of possible loss of information, we developed a refinement schema that is intuitively based on propagation of lemmas on demand. Moreover, in case we deal with loops with provably-different number of iterations, our technique automatically rearranges the iterations in the loops making them lockstep-composable for each subtask. We developed the ALIEN tool and empirically demonstrated that our approach to equivalence checking is more efficient than state-of-the-art on two classes of public benchmarks. In future, it would be interesting to extend these techniques to more general program structures, e.g., where both programs have multiple and possibly nested loops.

Acknowledgments The work is supported in parts by a gift from Amazon Web Services and by the National Science Foundation grant 2106949.

References

1. J. D. Backes, S. Person, N. Rungta, and O. Tkachuk. Regression verification using impact summaries. In *SPIN*, volume 7976 of *LNCS*, pages 99–116. Springer, 2013.
2. S. Badihi, F. Akinotcho, Y. Li, and J. Rubin. Ardiff: scaling program equivalence checking via iterative abstraction and refinement of common code. In *ESEC/FSE*, pages 13–24. ACM, 2020.
3. G. Barthe, J. M. Crespo, and C. Kunz. Relational verification using product programs. In *FM*, volume 6664 of *LNCS*, pages 200–214. Springer, 2011.
4. M. Blicha, G. Fedyukovich, A. E. J. Hyvärinen, and N. Sharygina. Transition Power Abstractions for Deep Counterexample Detection. In D. Fisman and G. Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems*. Springer Berlin Heidelberg, 2022.
5. B. R. Churchill, O. Padon, R. Sharma, and A. Aiken. Semantic program alignment for equivalence checking. In *PLDI*, pages 1027–1040. ACM, 2019.
6. B. R. Churchill, R. Sharma, J. F. Bastien, and A. Aiken. Sound loop superoptimization for google native client. In *ASPLOS*, pages 313–326. ACM, 2017.
7. E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. Relational Verification Through Horn Clause Transformation. In *SAS*, volume 9837 of *LNCS*, pages 147–169. Springer, 2016.
8. L. M. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *TACAS*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.
9. S. Dutta, D. Sarkar, A. Rawat, and K. Singh. Validation of loop parallelization and loop vectorization transformations. In *ENASE*, pages 195–202. SciTePress, 2016.
10. G. Fedyukovich, S. Prabhu, K. Madhukar, and A. Gupta. Quantified Invariants via Syntax-Guided Synthesis. In *CAV, Part I*, volume 11561 of *LNCS*, pages 259–277. Springer, 2019.
11. D. Felsing, S. Grebing, V. Klebanov, P. Rümmer, and M. Ulbrich. Automating regression verification. In *ASE*, pages 349–360. ACM, 2014.
12. B. Godlin and O. Strichman. Inference rules for proving the equivalence of recursive procedures. *Acta Informatica*, 45(6):403–439, 2008.
13. B. Godlin and O. Strichman. Regression verification: proving the equivalence of similar programs. *Softw. Test. Verification Reliab.*, 23(3):241–258, 2013.
14. S. Gupta, A. Rose, and S. Bansal. Counterexample-guided correlation algorithm for translation validation. *Proc. ACM Program. Lang.*, 4(OOPSLA):221:1–221:29, 2020.
15. M. Jakobs. PEQCHECK: localized and context-aware checking of functional equivalence. In S. Bliudze, S. Gnesi, N. Plat, and L. Semini, editors, *9th IEEE/ACM International Conference on Formal Methods in Software Engineering, FormaliSE@ICSE 2021, Madrid, Spain, May 17-21, 2021*, pages 130–140. IEEE, 2021.
16. V. Klebanov, P. Rümmer, and M. Ulbrich. Automating regression verification of pointer programs by predicate abstraction. *Formal Methods Syst. Des.*, 52(3):229–259, 2018.
17. S. Kundu, Z. Tatlock, and S. Lerner. Proving optimizations correct using parameterized program equivalence. In *PLDI*, pages 327–337. ACM, 2009.
18. S. K. Lahiri, C. Hawblitzel, M. Kawaguchi, and H. Rebêlo. SYMDIFF: A language-agnostic semantic diff tool for imperative programs. In *CAV*, volume 7358 of *LNCS*, pages 712–717. Springer, 2012.

19. S. K. Lahiri, K. L. McMillan, R. Sharma, and C. Hawblitzel. Differential assertion checking. In *FSE*, pages 345–355. ACM, 2013.
20. J. P. Lim, V. Ganapathy, and S. Nagarakatte. Compiler optimizations with retrofitting transformations: Is there a semantic mismatch? In *PLAS@CCS*, pages 37–42. ACM, 2017.
21. J. P. Lim and S. Nagarakatte. Automatic equivalence checking for assembly implementations of cryptography libraries. In *CGO*, pages 37–49. IEEE, 2019.
22. N. P. Lopes, J. Lee, C. Hur, Z. Liu, and J. Regehr. Alive2: bounded translation validation for LLVM. In S. N. Freund and E. Yahav, editors, *PLDI '21: 42nd ACM SIGPLAN PLDI, Virtual Event, Canada, June 20-25, 2021*, pages 65–79. ACM, 2021.
23. S. Maleki, Y. Gao, M. J. Garzar, T. Wong, D. A. Padua, et al. An Evaluation of Vectorizing Compilers. In *2011 PACT*, pages 372–382. IEEE, 2011.
24. V. Malík and T. Vojnar. Automatically checking semantic equivalence between versions of large-scale C projects. In *14th IEEE Conference on Software Testing, Verification and Validation, ICST 2021, Porto de Galinhas, Brazil, April 12-16, 2021*, pages 329–339. IEEE, 2021.
25. D. Mordvinov and G. Fedyukovich. Synchronizing Constrained Horn Clauses. In *LPAR*, volume 46 of *EPiC Series in Computing*, pages 338–355. EasyChair, 2017.
26. D. Mordvinov and G. Fedyukovich. Property Directed Inference of Relational Invariants. In *FMCAD*, pages 152–160. IEEE, 2019.
27. K. S. Namjoshi and A. Xue. A self-certifying compilation framework for webassembly. In F. Henglein, S. Shoham, and Y. Vizel, editors, *VMCAI - 22nd International Conference, VMCAI 2021, Copenhagen, Denmark, January 17-19, 2021, Proceedings*, volume 12597 of *LNCS*, pages 127–148. Springer, 2021.
28. G. C. Necula. Translation validation for an optimizing compiler. In *PLDI*, pages 83–94. ACM, 2000.
29. N. Partush and E. Yahav. Abstract semantic differencing for numerical programs. In *SAS*, volume 7935 of *LNCS*, pages 238–258. Springer, 2013.
30. S. Person, M. B. Dwyer, S. G. Elbaum, and C. S. Pasareanu. Differential symbolic execution. In *FSE*, pages 226–237. ACM, 2008.
31. D. Riley and G. Fedyukovich. Multi-phase invariant synthesis. In A. Roychoudhury, C. Cadar, and M. Kim, editors, *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*, pages 607–619. ACM, 2022.
32. T. A. L. Sewell, M. O. Myreen, and G. Klein. Translation validation for a verified OS kernel. In *PLDI*, pages 471–482. ACM, 2013.
33. R. Sharma, E. Schkufza, B. R. Churchill, and A. Aiken. Data-driven Equivalence Checking. In *OOPSLA*, pages 391–406. ACM, 2013.
34. R. Shemer, A. Gurfinkel, S. Shoham, and Y. Vizel. Property directed self composition. In *CAV, Part I*, volume 11561, pages 161–179. Springer, 2019.
35. M. Stepp, R. Tate, and S. Lerner. Equality-based translation validator for LLVM. In *CAV*, volume 6806 of *LNCS*, pages 737–742. Springer, 2011.
36. O. Strichman and M. Veitsman. Regression verification for unbalanced recursive functions. In *FM*, volume 9995 of *LNCS*, pages 645–658, 2016.
37. A. Trostanetski, O. Grumberg, and D. Kroening. Modular demand-driven analysis of semantic difference for program versions. In *SAS*, volume 10422 of *LNCS*, pages 405–427. Springer, 2017.

38. H. Unno, T. Terauchi, and E. Koskinen. Constraint-based relational verification. In A. Silva and K. R. M. Leino, editors, *CAV - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part I*, volume 12759 of *LNCS*, pages 742–766. Springer, 2021.
39. A. Zaks and A. Pnueli. Covac: Compiler validation by program analysis of the cross-product. In *FM*, volume 5014 of *LNCS*, pages 35–51. Springer, 2008.