Exploiting Synchrony and Symmetry in Relational Verification

Lauren Pick, Grigory Fedyukovich, Aarti Gupta

Princeton University

Abstract. Relational safety specifications describe multiple runs of the same program or relate the behaviors of multiple programs. Approaches to automatic relational verification often compose the programs and analyze the result for safety, but a naively composed program can lead to difficult verification problems. We propose to exploit relational specifications for simplifying the generated verification subtasks. First, we maximize opportunities for synchronizing code fragments. Second, we compute symmetries in the specifications to reveal and avoid redundant subtasks. We have implemented these enhancements in a prototype for verifying k-safety properties on Java programs. Our evaluation confirms that our approach leads to a consistent performance speedup on a range of benchmarks.

1 Introduction

The verification of relational program specifications is of wide interest, having many applications. Relational specifications can describe multiple runs of the same program or relate the behaviors of multiple programs. An example of the former is the verification of security properties such as non-interference, where different executions of the same program are compared to check whether there is a leak of sensitive information. The latter is useful for checking equivalence or refinement relationships between programs after applying some transformations or during iterative development of different software versions.

There is a rich history of work on the relational verification of programs. Representative efforts include those that target general analysis using relational program logics and frameworks [8,29,5,25,4] or specific applications such as security verification [7,9,1], compiler validation [30,15], and differential program analysis [18,21,22,16,20]. These efforts are supported by tools that range from automatic verifiers to interactive theorem-provers. In particular, many automatic verifiers are based on constructing a *composition* over the programs under consideration, where the relational property over multiple runs (of the same or different programs) is translated into a functional property over a single run of a composed program. This has the benefit that standard techniques and tools for program verification can then be applied.

However, it is also well known that a naively composed program can lead to difficult verification problems for automatic verifiers. For example, a *sequential*

composition of two loops would require effective techniques for generating loop invariants; otherwise, the verifier could get stuck in exploring the first loop before even considering the second one. In contrast, a *parallel* composition would provide potential for aligning the loop bodies, where relational invariants may be easier to establish than a functional loop invariant across a single but complex loop. Examples of techniques that exploit opportunities for such alignment include use of type-based analysis with self-composition in security verification [27], allowing flexibility in composition to be a mix of sequential and parallel [6], exploiting structurally equivalent programs for compiler validation [30], lockstep execution of loops in reasoning using Cartesian Hoare Logic [25], and merging Horn clause rules for relational verification [12,23].

In this paper, we present a compositional framework that leverages relational specifications to further simplify the generated verification tasks on the composed program. Our framework is motivated by two main strategies. The first strategy, similar to the efforts mentioned above, is to exploit opportunities for *synchrony*, i.e., aligning code fragments across which relational invariants are easy to derive, perhaps due to functional similarity or due to similar code structure, etc. Specifically, we choose to *synchronize* the programs at conditional blocks as well as at loops. Similar to closely related efforts [6,25], we would like to execute loops in lockstep so that relational invariants can be derived over corresponding iterations over the loop bodies. Specifically, we propose a technique that analyzes the relational specifications to infer, under reasonable assumptions, *maximal sets of loops* that can be executed in lockstep. Synchronizing at conditional blocks in addition to loops enables simplification due to relational specifications and conditional guards that might result in infeasible or redundant subtasks. Pruning of such infeasible subtasks has been performed and noted as important in existing work [25], and synchronizing at conditional blocks allows us to prune eagerly. More importantly, aligning different programs at conditional statements gives us the opportunity to exploit symmetry to avoid redundant subtasks.

Our second strategy, which is set up by the first, is the exploitation of symmetry in relational specifications. Due to control flow divergences or non-lockstep executions of loops, even different copies of the same program may proceed along different code fragments. However, some of the resulting verification subtasks may be indistinguishable from each other due to underlying symmetries among related fragments. We analyze the relational specifications to discover symmetries and exploit them to prune away redundant subtasks. For discovering symmetries, we extend a known technique for symmetry-breaking on Boolean formulas [11] to handling relational specifications on equalities and linear integer arithmetic. Note that our two strategies work in synergy, where the first strategy aligns fragments for deriving relational assertions, and the second strategy analyzes these assertions to discover symmetries and perform pruning. To the best of our knowledge, symmetry has not been used in earlier efforts for relational verification, although it has been used very successfully in model checking parametric finite state systems [14,10,19] and concurrent programs [13].

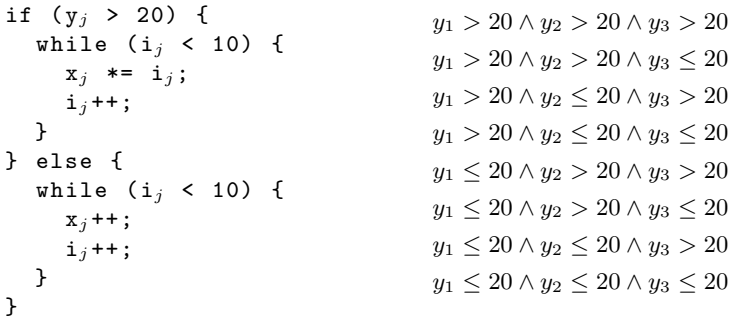

Fig. 1: Example program (**left**), and eight possible control-flow decisions (**right**).

The strategies we propose for exploiting synchrony and symmetry via relational specifications are fairly general in that they can be employed in various verification methods. We provide a generic logic-based description of these strategies at a high level (Sect. 4), and also describe a specific instantiation in a verification algorithm based on forward analysis that computes strongest-postconditions (Sect. 5). We have implemented our approach in a prototype tool called Synonym built on top of the Descartes tool [25]. Our experimental evaluation (Sect. 6) shows the effectiveness of our approach in improving the performance of verification in many examples (and a marginal overhead in smaller examples). In particular, exploiting symmetry is crucial in enabling verification to complete for some properties, without which Descartes exceeds a timeout on all benchmark examples.

2 Motivating Example

Consider three C-like integer programs $\{P_j\}$ of the form shown in Fig. 1 (**left**). They are identical modulo renaming, and we use indices $j \in \{1, 2, 3\}$ as subscripts to denote variables in the different copies. We assume that each variable initially takes a nondeterministic value in each program.

A relational verification problem (RVP) is a tuple consisting of programs $\{P_j\}$, a relational precondition *pre*, and a relational postcondition *post*. In the example RVPs below, we consider the three conditionals, which in turn lead to eight possible control-flow decisions (Fig. 1, **right**) in a composed program. Each RVP reduces to subproblems for proving that *post* can be derived from *pre* for each of these control-flow decisions. In the rest of the section, we demonstrate the underlying ideas behind our approach to solve these subproblems efficiently.

Maximizing Lockstep Execution. Given an RVP (referred to as RVP_1) with precondition $x_1 < x_3 \wedge x_1 > 0 \wedge i_1 > 0 \wedge i_2 \geq i_1 \wedge i_1 = i_3$ (*pre*) and postcondition $(x_1 < x_3 \vee y_1 \neq y_3) \wedge i_1 > 0 \wedge i_2 \geq i_1 \wedge i_1 = i_3$ (*post*), consider a control-flow decision $y_1 > 20 \wedge y_2 > 20 \wedge y_3 > 20$. This leads to another RVP, consisting of

three programs of the following form:

```
assume(y_j > 20); while (i_j < 10) { x_j *= i_j; i_j++; }
```

where $j \in \{1, 2, 3\}$, and the aforementioned *pre* and *post*. From *pre*, it follows that $i_1 = i_3$ and $i_2 \geq i_1$. We can thus infer that the first and third loops are always executed the same number of times, while the second loop may be executed for fewer iterations. This knowledge lets us infer a single relational invariant for the first and third loops and handle the second loop separately. Clearly, the relational invariant $x_1 < x_3 \wedge i_1 = i_3 \wedge i_1 \leq 10$ and the non-relational invariant $i_2 \leq 10$ are enough to derive *post*. If we were to handle the first and third loop separately, we would need complex nonlinear invariants such as $x_1 = \frac{x_{1,init} \times i_1!}{i_{1,init}!}$ and $x_3 = \frac{x_{3,init} \times i_3!}{i_{3,init}!}$, which involve auxiliary variables $x_{j,init}$ and $i_{j,init}$ denoting the initial values of x_j and i_j respectively.

Symmetry-Breaking. For the same program, and an RVP (referred to as RVP_2) with precondition $i_1 > 0 \wedge i_2 \geq i_1 \wedge i_1 = i_3$ and postcondition $i_1 > 0 \wedge i_2 \geq i_1 \wedge i_1 = i_3$. consider a control-flow decision $y_1 > 20 \wedge y_2 > 20 \wedge y_3 \leq 20$. We generate another RVP involving the following set of programs:

```
assume(y_1 > 20); while (i_1 < 10) { x_1 *= i_1; i_1++; }
assume(y_2 > 20); while (i_2 < 10) { x_2 *= i_2; i_2++; }
assume(y_3 ≤ 20); while (i_3 < 10) { x_3++; i_3++; }
```

Similarly, decision $y_1 \leq 20 \wedge y_2 > 20 \wedge y_3 > 20$ generates yet another RVP over the following:

```
assume(y_1 ≤ 20); while (i_1 < 10) { x_1++; i_1++; }
assume(y_2 > 20); while (i_2 < 10) { x_2 *= i_2; i_2++; }
assume(y_3 > 20); while (i_3 < 10) { x_3 *= i_3; i_3++; }
```

Both RVPs have the same precondition and postcondition as RVP_2. We can see that both RVPs differ only in their subscripts; by taking one of them and swapping the subscripts 1 and 3 due to symmetry, we arrive at the other. Thus, knowing the verification result for either RVP allows us to skip verifying the other one, by discovering and exploiting such symmetries.

3 Background and Notation

Given a loop-free program over input variables $\vec{x}$ and output variables $\vec{y}$ (such that $\vec{x}$ and $\vec{y}$ are disjoint), let $Tr(\vec{x}, \vec{y})$ denote its symbolic encoding.

Proposition 1. *Given two loop-free programs, $Tr_1(\vec{x}_1, \vec{y}_1)$ and $Tr_2(\vec{x}_2, \vec{y}_2)$, a precondition $pre(\vec{x}_1, \vec{x}_2)$, and a postcondition $post(\vec{y}_1, \vec{y}_2)$, the task of relational verification is reduced to checking validity of the following formula.*

$$pre(\vec{x}_1, \vec{x}_2) \wedge Tr_1(\vec{x}_1, \vec{y}_1) \wedge Tr_2(\vec{x}_2, \vec{y}_2) \Longrightarrow post(\vec{y}_1, \vec{y}_2)$$

Given a program with one loop (i.e., a transition system) over input variables $\vec{x}$ and output variables $\vec{y}$, let $Init(\vec{x}, \vec{u})$ denote a symbolic encoding of the block

of code before the loop, $Guard(\vec{u})$ denote the loop guard, and $Tr(\vec{u}, \vec{y})$ encode the loop body. Here, $\vec{u}$ is the vector of local variables that are live at the loop guard. For example, consider the program from our motivating example:

$$\texttt{assume(y}_1\texttt{ > 20); while (i}_1\texttt{ < 10) \{ x}_1\texttt{ *= i}_1\texttt{; i}_1\texttt{++; \}}$$

In its encoding, $\vec{x} = \vec{u} = (i_1, x_1, y_1)$, $\vec{y} = (i_1', x_1')$, $Init(\vec{x}, \vec{u}) = y_1 > 20$, $Guard(\vec{u}) = i_1' < 10$, and $Tr(\vec{u}, \vec{y}) = x_1' = x_1 \times i_1 \wedge i_1' = i_1 + 1$.

Proposition 2 (Naive parallel composition). *Given two loopy programs, $\langle Init(\vec{x}_1, \vec{u}_1), Guard(\vec{u}_1), Tr(\vec{u}_1, \vec{y}_1)\rangle$ and $\langle Init(\vec{x}_2, \vec{u}_2), Guard(\vec{u}_2), Tr(\vec{u}_2, \vec{y}_2)\rangle$, a precondition $pre(\vec{x}_1, \vec{x}_2)$, and a postcondition $post(\vec{y}_1, \vec{y}_2)$, the task of relational verification is reduced to the task of finding (individual) inductive invariants $\boldsymbol{I_1}$ and $\boldsymbol{I_2}$:*

$$\begin{aligned}
pre(\vec{x}_1, \vec{x}_2) \wedge Init(\vec{x}_1, \vec{u}_1) &\Longrightarrow \boldsymbol{I_1}(\vec{u}_1) \\
pre(\vec{x}_1, \vec{x}_2) \wedge Init(\vec{x}_2, \vec{u}_2) &\Longrightarrow \boldsymbol{I_2}(\vec{u}_2) \\
\boldsymbol{I_1}(\vec{u}_1) \wedge Guard_1(\vec{u}_1) \wedge Tr_1(\vec{u}_1, \vec{y}_1) &\Longrightarrow \boldsymbol{I_1}(\vec{y}_1) \\
\boldsymbol{I_2}(\vec{u}_1) \wedge Guard_2(\vec{u}_2) \wedge Tr_2(\vec{u}_2, \vec{y}_2) &\Longrightarrow \boldsymbol{I_2}(\vec{y}_2) \\
\boldsymbol{I_1}(\vec{y}_1) \wedge \boldsymbol{I_2}(\vec{y}_2) \wedge \neg Guard_1(\vec{y}_1) \wedge \neg Guard_2(\vec{y}_2) &\Longrightarrow post(\vec{y}_1, \vec{y}_2)
\end{aligned}$$

Note that the method of naive composition requires handling of multiple invariants, which is known to be difficult. Furthermore, it might lose some important relational information specified in $pre(\vec{x}_1, \vec{x}_2)$. One way to avoid this is to exploit the fact that loops could be executed in lockstep.

Proposition 3 (Lockstep composition). *Given two loopy programs, $\langle Init(\vec{x}_1, \vec{u}_1), Guard(\vec{u}_1), Tr(\vec{u}_1, \vec{y}_1)\rangle$ and $\langle Init(\vec{x}_2, \vec{u}_2), Guard(\vec{u}_2), Tr(\vec{u}_2, \vec{y}_2)\rangle$, a precondition $pre(\vec{x}_1, \vec{x}_2)$, and a postcondition $post(\vec{y}_1, \vec{y}_2)$. Let **both loops iterate exactly the same number of times**, then the task of relational verification is reduced to the task of finding one (relational) inductive invariant $\boldsymbol{I}$:*

$$\begin{aligned}
pre(\vec{x}_1, \vec{x}_2) \wedge Init(\vec{x}_1, \vec{u}_1) \wedge Init(\vec{x}_2, \vec{u}_2) &\Longrightarrow \boldsymbol{I}(\vec{u}_1, \vec{u}_2) \\
\boldsymbol{I}(\vec{u}_1, \vec{u}_2) \wedge Guard_1(\vec{u}_1) \wedge Tr_1(\vec{u}_1, \vec{y}_1) \wedge Guard_2(\vec{u}_2) \wedge Tr_2(\vec{u}_2, \vec{y}_2) &\Longrightarrow \boldsymbol{I}(\vec{y}_1, \vec{y}_2) \\
\boldsymbol{I}(\vec{y}_1, \vec{y}_2) \wedge \neg Guard_1(\vec{y}_1) \wedge \neg Guard_2(\vec{y}_2) &\Longrightarrow post(\vec{y}_1, \vec{y}_2)
\end{aligned}$$

In this paper, we do not focus on a specific method for deriving these invariants – a plethora of suitable methods have been proposed in the literature, and any of these could be used.

4 Leveraging Relational Specifications

In this section, we describe the main components of our compositional framework where we leverage relational specifications to simplify the verification subtasks. One major motivation is to exploit synchrony, i.e., align code fragments where relational invariants are relatively easy to derive. In particular, we synchronize at loop boundaries and conditional blocks. We first describe our algorithm for inferring maximal sets of loops that can be executed in lockstep (Sect. 4.1).

We also describe how we reduce the number of RVPs generated at points of divergent control-flow (Sect. 4.2) and how we can further reduce this number by exploiting symmetry (Sect. 4.3.2). For each of these components, we rely on the relational precondition (and sometimes the postcondition) to help us simplify the verification task.

4.1 Synchronizing loops

Given a set of loopy programs, we would like to determine which ones can be executed in lockstep. As mentioned earlier, relational invariants over lockstep loops are often easier to derive than loop invariants over a single copy.

Our algorithm CheckLockstep takes as input a set of programs $\{P_1, \ldots, P_k\}$ and outputs a set of equivalence classes of programs that can be executed in lockstep. The algorithm partitions its input set of programs and recursively calls CheckLockstep on the partitions.

First, CheckLockstep infers a relational inductive invariant over the loop bodies, synthesizing $\boldsymbol{I}(\vec{u}_1, \ldots, \vec{u}_k)$ in the following:

$$pre(\vec{x}_1, \ldots, \vec{x}_k) \wedge \bigwedge_{i=1}^{k} Init(\vec{x}_i, \vec{u}_i) \implies \boldsymbol{I}(\vec{u}_1, \ldots, \vec{u}_k)$$

$$\boldsymbol{I}(\vec{u}_1, \ldots, \vec{u}_k) \wedge \bigwedge_{i=1}^{k} Guard_i(\vec{u}_i) \wedge Tr_i(\vec{u}_i, \vec{y}_i) \implies \boldsymbol{I}(\vec{y}_1, \ldots, \vec{y}_k)$$

CheckLockstep then poses the following query:

$$\neg\left(\left(\boldsymbol{I}(\vec{u}_1, \ldots, \vec{u}_k) \wedge \bigvee_{i=1}^{k} \neg Guard(\vec{u}_i)\right) \implies \bigwedge_{i=1}^{k} \neg Guard(\vec{u}_i)\right) \tag{1}$$

The left-hand side of the implication holds whenever one of the loops has terminated (the relational invariant holds, and at least one of the loop conditions must be false), and the right-hand side holds only if all of the loops have terminated. If the formula is unsatisfiable, then the termination of one loop implies the termination of all loops, and all loops can be executed simultaneously [25]. In this case, the entire set of input programs is one equivalence class, and the set containing the set of all input programs is returned.

Otherwise, CheckLockstep gets a satisfying assignment and partitions the input programs into a set *Terminated* and a set *Unfinished*. The *Terminated* set contains all programs P_i whose guards $Guard(\vec{u}_i)$ are false in the model for the formula, and the *Unfinished* set contains the remaining programs. The CheckLockstep algorithm is then called recursively on both *Terminated* and *Unfinished*, with its final result being the union of the two sets returned by these recursive calls.

The following theorem assumes that any relational invariant $\boldsymbol{I}(\vec{u}_1, \ldots, \vec{u}_k)$, generated externally and used by the algorithm, is stronger than any relational invariant $\boldsymbol{I}(\vec{u}_1, \ldots, \vec{u}_{i-1}, \vec{u}_{i+1}, \ldots, \vec{u}_k)$ that could be synthesized over the same set of k loops with the i^{th} loop removed.

Theorem 1. *For any call to* CheckLockstep*, it always partitions its set of input programs such that for all $P_i \in$ Terminated and $P_j \in$ Unfinished, P_i and P_j cannot be executed in lockstep.*

Proof. Assume that CheckLockstep has partitioned its set of programs into the *Terminated* and *Unfinished* sets. Let $P_i \in$ *Terminated*, $P_j \in$ *Unfinished* be arbitrary programs. Based on how the partitioning is performed, we know that there is a model for Eq. 1 such that $Guard(\vec{u}_i)$ does not hold and $Guard(\vec{u}_j)$ does. We can thus conclude that the following formula is satisfiable:

$$\neg\Big(\boldsymbol{I}(\vec{u}_1, \ldots, \vec{u}_k) \wedge \neg Guard(\vec{u}_i) \implies \neg Guard(\vec{u}_j)\Big)$$

From the assumption on our invariant synthesizer, we conclude that the following is also satisfiable, indicating that P_i and P_j cannot be executed in lockstep:

$$\neg\Big(\boldsymbol{I}(\vec{u}_i, \vec{u}_j) \wedge \neg Guard(\vec{u}_i) \implies \neg Guard(\vec{u}_j)\Big)$$

where $\boldsymbol{I}(\vec{u}_i, \vec{u}_j)$ is the relational invariant for P_i and P_j that our invariant synthesizer infers. □

4.2 Synchronizing conditionals

Let two programs have forms `if` $\mathtt{Q}_i$ `then` $\mathtt{R}_i$ `else` $\mathtt{S}_i$, where $i \in \{1, 2\}$ and $\mathtt{R}_i$ and $\mathtt{S}_i$ are arbitrary blocks of code and could possibly have loops. Let them be a part of some RVP, which reduces to applying Prop. 1, 2, or 3, depending on the content of each block of code, to four pairs of programs. As we have seen in previous sections, each of the four verification tasks could be expensive. In order to reduce the number of verification tasks where possible, we use the relational preconditions to filter out pairs of programs for which verification conclusions can be derived trivially.

For k programs of the form `if` $\mathtt{Q}_i$ `then` $\mathtt{R}_i$ `else` $\mathtt{S}_i$ for $i \in \{1, \ldots, k\}$ and precondition $pre(\vec{x}_1, \ldots, \vec{x}_k)$, we can simultaneously generate all possible combinations of decisions by querying a solver for all truth assignments to the $\mathtt{Q}_i$s:

$$pre(\vec{x}_1, \ldots, \vec{x}_k) \wedge \bigwedge_{i=1}^{k} Q_i \tag{2}$$

We can then use the result of this All-SAT query to generate sets of programs in subtasks. For each assignment j, where each Q_i is assigned a Boolean value v_i, the following set is generated: {`assume(`$\mathtt{V}_1$`);` $\mathtt{U}_1, \ldots,$ `assume (`$\mathtt{V}_k$`);` $\mathtt{U}_k$} where for each $i \in \{1, \ldots, k\}$, if $v_i = true$, then $\mathtt{V}_i = \mathtt{Q}_i$ and $\mathtt{U}_i = \mathtt{R}_i$, else $\mathtt{V}_i = \neg\mathtt{Q}_i$ and $\mathtt{U}_i = \mathtt{S}_i$. We need to apply our verification algorithm on only the resulting sets of programs. For example, in our above RVP, if $\mathtt{Q}_1$ is equivalent to $\mathtt{Q}_2$ in all solutions, then the RVP reduces to verification of just two pairs of programs:

$$\begin{array}{rcl} \mathtt{assume(Q_1);\ R_1} & \text{and} & \mathtt{assume(Q_2);\ R_2} \\ \mathtt{assume(\neg Q_1);\ S_1} & \text{and} & \mathtt{assume(\neg Q_2);\ S_2} \end{array}$$

Algorithm 1 Algorithm for constructing a graph to find symmetries.

1: **procedure** MakeGraph(F)
2: $(V, E) \leftarrow (\{v_1^{Id}, \ldots, v_k^{Id}\}, \varnothing)$ where each v_i^{Id} has $color(v_i^{Id}) = Id$
3: **for** $d \in$ Clauses(F) **do**
4: $(V, E) \leftarrow$ MakeAST(d) $\cup (V, E)$
5: **for** $v \in V$ with $x_i \in vars(color(v))$ **do**
6: $V \leftarrow (V \setminus \{v\}) \cup \{$Recolor$(v, v[x_i \mapsto x])\}$
7: $E \leftarrow E \cup \{(v, v_i^{Id})\}$

4.3 Discovering and exploiting symmetries

Using the All-SAT query from Eq. 2 allows us to prune trivial RVPs. However, as we have seen in Sect. 2, some of the remaining RVPs could be regarded as equivalent. As before, we use our relational specifications to ease the verification task: symmetries in our relational precondition and postcondition can help us eliminate redundant RVPs. Here, we discuss how to identify symmetries in formulas syntactically, and then we show how to use such symmetries in our relational precondition and postcondition to define equivalent RVPs and ensure that we solve only one of them.

4.3.1 Identifying symmetries in formulas

Formally, symmetries in formulas are defined as permutations. Note that any permutation π of set $\{1, \ldots, k\}$ can be lifted to be a permutation of set $\{\vec{x}_1, \ldots, \vec{x}_k\}$.

Definition 1 (Symmetry). *Let $\vec{x}_1, \ldots, \vec{x}_k$ be vectors of the same size over disjoint sets of variables. A* symmetry *π of a formula $F(\vec{x}_1, \ldots, \vec{x}_k)$ is a permutation of set $\{\vec{x}_i \mid 1 \leq i \leq k\}$ such that $F(\vec{x}_1, \ldots, \vec{x}_k) \iff F(\pi(\vec{x}_1), \ldots, \pi(\vec{x}_k))$.*

The task of finding symmetries within a set of formulas can be performed syntactically by first canonicalizing the formulas, converting the formulas into a graph representation of their syntax, and then using a graph automorphism algorithm to find the symmetries of the graph. We demonstrate how this can be done for a formula φ over Linear Integer Arithmetic with the following example.

Let $\varphi = (x_1 \leq x_2 \wedge x_3 \leq x_4) \wedge (x_1 < z_2 \vee x_3 < z_4)$. Note that this formula is symmetric under a permutation of the subscripts that simultaneously swaps 1 with 3 and 2 with 4. Let $\{(x_1, z_1), (x_2, z_2), (x_3, z_3), (x_4, z_4)\}$ be the vectors of variables. We identify a vector by its subscript (e.g., we identify (x_1, z_1) by 1).

Our algorithm starts with canonicalizing the formula: $\varphi = (x_1 < x_2 \vee x_1 = x_2) \wedge (x_3 < x_4 \vee x_3 = x_4) \wedge (x_1 < z_2 \vee x_3 < z_4)$. It then constructs a colored graph for the canonicalized formula with the procedure in Alg. 1. The algorithm begins with graph containing only the set of k vertices $v_1^{Id}, \ldots, v_k^{Id}$ with color Id (vertices 21-24 in Fig. 2), where k is the number of identifiers. It then (Line 4) adds to the graph the union of the abstract syntax trees (AST) for the formula's conjuncts, where each vertex has a color indicating the type of AST node it corresponds to. If a parent vertex has a color of an ordering-sensitive operation

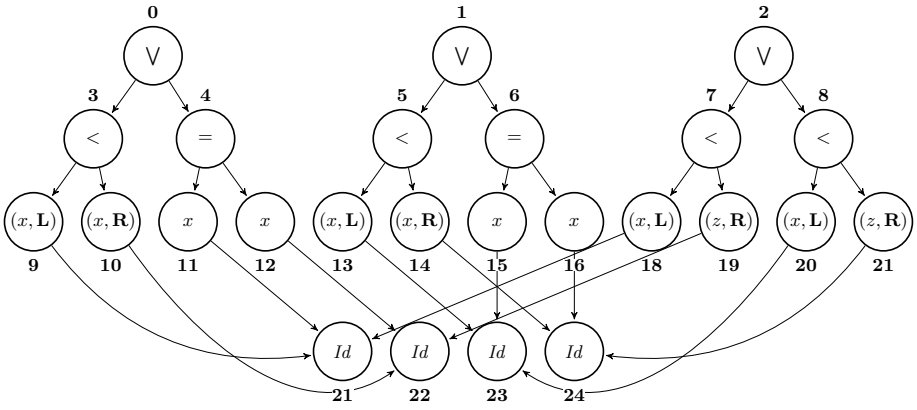

Fig. 2: Graph with vertex names (outside the vertices) and colors (inside the vertices).

or predicate, then the children should have colors that include a tag to indicate their ordering (e.g. vertices 9 and 10 in Fig. 2 have colors with tags because their parent has color $<$, but vertices 11 and 12 do not have tags because their parent has color $=$). Next (Line 5), the algorithm performs an appropriate renaming of vertex colors so that each indexed variable name x_i is replaced with a non-indexed version x while simultaneously adding edges from each vertex with a renamed color to v_i^{Id}. The resulting graph for φ is the one shown in Fig. 2. Finally, the algorithm applies a graph automorphism finder to get the following automorphism (in addition to the identity automorphism), which is shown here in a cyclic notation where $(x\ y)$ means that $x \mapsto y$ and $y \mapsto x$ (vertices that map to themselves are omitted):

$$(0\ 1)(3\ 5)(4\ 6)(7\ 8)(9\ 13)(10\ 14)(11\ 15)(12\ 16)(17\ 19)(18\ 20)(21\ 23)(22\ 24)$$

We are only interested in permutations of the vectors, so we project out the relevant parts of the permutation $(21\ 23)(22\ 24)$ and map them back to our vector identifiers to get the following permutation on the identifiers:

$$\pi = \{1 \mapsto 3, 2 \mapsto 4, 3 \mapsto 1, 4 \mapsto 2\}$$

4.3.2 Exploiting symmetries

We now define the notion of symmetric RVPs and present a novel application of symmetry breaking to generating a single representative per equivalence class of RVPs.

Definition 2 (Symmetric RVPs). *Two RVPs:* $\langle Ps, pre(\vec{x}_1, \ldots, \vec{x}_k), post(\vec{y}_1, \ldots, \vec{y}_k)\rangle$ *and* $\langle Ps', pre(\vec{x}_1, \ldots, \vec{x}_k), post(\vec{y}_1, \ldots, \vec{y}_k)\rangle$, *where* $Ps = \{P_1, \ldots, P_k\}$, *and* $Ps' = \{P'_1, \ldots, P'_k\}$, *are called* symmetric *under a permutation* π *iff*

1. π *is a symmetry of formula* $pre(\vec{x}_1, \ldots, \vec{x}_k) \wedge post(\vec{y}_1, \ldots, \vec{y}_k)$

2. *for every* $P_i \in Ps$, *and* $P_j \in Ps'$, *if* $\pi(i) = j$, *then* P_i *and* P_j *have the same number of inputs and outputs and have logically equivalent encodings for the same set of input variables* $\vec{x}_i$ *and output variables* $\vec{y}_i$

As we have seen in Sect. 4.3.1, identification of symmetries could be made purely on the syntactic level of the relational preconditions and postconditions. For each detected symmetry, it remains to check equivalence between the corresponding programs' encodings, which can be formulated as an SMT problem.

To exploit symmetries, we propose a simple but intuitive approach. First, we identify the set of symmetries using $pre \wedge post$. Then, we solve the All-SAT query from Eq. 2 and get a *reduced* set R of RVPs (i.e., one without all trivial problems). For each $RVP_i \in R$, we perform the relational verification only if no symmetric $RVP_j \in R$ has already been verified. Thus, the most expensive part of the routine, checking equivalence of RVPs, is performed on demand and only on a subset of all possible pairs $\langle RVP_i, RVP_j \rangle$.

Alternatively, in some cases (e.g., for parallelizing the algorithm) it might help to identify all symmetric RVPs prior to solving the All-SAT query from Eq. 2. From this set, we can generate symmetry-breaking predicates (SBPs) [11] and conjoin them to Eq. 2. Constrained with SBPs, this query will have fewer models, and will contain a single representative per equivalence class of RVPs. We describe how to construct SBPs in more detail in the next section.

4.3.3 Generating Symmetry-Breaking Predicates (SBPs)

SBPs have previously been applied in pruning the search space explored by SAT solvers. Traditionally, techniques construct SBPs based on symmetries in truth assignments to the literals in the formula, but SBP-construction can be adapted to be based on symmetries in truth assignments to conditionals, allowing us to break symmetries in our setting.

We can construct an SBP by treating each condition the way a literal is treated in existing SBP constructions. In particular, we can construct the common Lex-Leader SBP used for predicate logic [11], which in our case will force a solver to choose the lexicographically least representative per equivalence class for a particular ordering of the conditions. For the ordering of conditions where $\mathtt{Q}_i \leq \mathtt{Q}_j$ iff $i \leq j$ and a set of symmetries S over $\{1, \ldots, k\}$, we can construct a Lex-Leader SBP $SBP(S) = \bigwedge_{\pi \in S} PP(\pi)$ with the more efficient predicate chaining construction [2], where we have that

$$PP(\pi) = p_{\min(I)} \wedge \bigwedge_{i \in I} p_i \implies g_{prev(i,I)} \implies l_i \wedge p_{next(i,I)}$$

and that I is the support of π with the last condition for each cycle removed, $\min(I)$ is the minimal element of I, $prev(i, I)$ is the maximal element of I still less than i or 0 if there is none, $next(i, I)$ is the minimal element of I still greater than i or 0 if there is none, $p_0 = g_0 = true$, p_i is a fresh predicate for $i \neq 0$, $g_i = \mathtt{Q}_{\pi(i)} \implies \mathtt{Q}_i$ for $i \neq 0$, and $l_i = \mathtt{Q}_i \implies \mathtt{Q}_{\pi(i)}$.

```
1: procedure VERIFY(pre, Current, Ifs, Loops, post)
2:     while Current ≠ ∅ do
3:         if PROCESSSTATEMENT(pre, P_i, Ifs, Loops, post) = safe then return safe
4:     if Loops ≠ ∅ then HANDLELOOPS(pre, Loops, post)
5:     else if Ifs ≠ ∅ then HANDLEIFS(pre, Ifs, Loops, post)
6:     else return unsafe
```

After constructing the SBP, we conjoin it to the All-SAT query in Eq. 2. Our solver now generates sets of programs that, when combined with the relational precondition and postcondition, form a set of irredundant RVPs.

Example. Let us consider how SBPs can be applied to RVP_2 from Sect. 2 to avoid generating two of the eight RVPs we would otherwise generate.

First, we see that our three programs are all copies the same program and are at the same program point, so they will have the same encoding. Next, we find the set of permutations S over $\{1, 2, 3\}$ such that for each $\pi \in S$, we have that $i_1 > 0 \wedge i_2 \geq i_1 \wedge i_1 = i_3$ iff $i_{\pi(1)} > 0 \wedge i_{\pi(2)} \geq i_{\pi(1)} \wedge i_{\pi(1)} = i_{\pi(3)}$. In this case, we have that S is the set of permutations $\{\{1 \mapsto 1, 2 \mapsto 2, 3 \mapsto 3\}, \{1 \mapsto 3, 2 \mapsto 2, 3 \mapsto 3\}\}$. Now, we construct a Lex-Leader SBP (using the predicate chaining construction described above):

$$p_1 \wedge (p_1 \implies ((y_1 > 20) \implies (y_2 > 20)))$$

where p_1 is a fresh predicate. Conjoining this SBP to Eq. 2, leads to the RVPs arising from the control-flow decisions $y_1 > 20 \wedge y_2 > 20 \wedge y_3 \leq 20$ and $y_1 > 20 \wedge y_2 \leq 20 \wedge y_3 \leq 20$ no longer being generated.

5 Instantiation of Strategies in Forward Analysis

We now describe an instantiation of our proposed strategies in a verification algorithm based on forward analysis using a strongest-postcondition computation. Other instantiations, e.g., on top of a Horn solver based on Property-Directed Reachability [23] are possible but left outside the scope of the work.

Given an RVP in the form of a Hoare triple $\{Pre\}\ P_1 || \cdots || P_k\ \{Post\}$, where $||$ denotes parallel composition, the top-level VERIFY procedure takes as input the relational specification $pre = Pre$ and $post = Post$, the set of input programs $Current = \{P_1, \ldots, P_k\}$, and empty sets *Loops* and *Ifs*. It uses a strongest-postcondition computation to compute the next Hoare triple at each step until it can conclude the validity of the original Hoare triple.

Synchronization. Throughout verification, the algorithm maintains three disjoint sets of programs: one for programs that are currently being processed (*Current*), one for programs that have been processed up until a loop (*Loops*), and one for programs that have been processed up until a conditional statement (*Ifs*). The algorithm processes statements in each program independently, with PROCESSSTATEMENT choosing an arbitrary interleaving of statements from the

programs in *Current*. When the algorithm encounters the end of a program in its call to ProcessStatement, it removes this program from the *Current* set. At this point, the algorithm returns safe if the current Hoare triple is proven valid. When a program has reached a point of control-flow divergence and is processed by ProcessStatement, it is removed from *Current* and added to the appropriate set (*Loops* or *Ifs*).

Handling Loops. Once all programs are in the *Loops* or *Ifs* sets (i.e. *Current* = $\varnothing$), the algorithm handles the programs in the *Loops* set if it is nonempty. HandleLoops behaves like CheckLockstep but computes postconditions where possible; when a set of loops are able to be executed in lockstep, HandleLoops computes their postconditions before placing the programs into the *Terminated* set. After all loops have been placed in the *Terminated* set and a new precondition pre' has been computed, rather than returning *Terminated*, HandleLoops invokes Verify(pre', *Terminated*, *Ifs*, $\varnothing$, *post*).

Handling Conditionals. When *Current* = *Loops* = $\varnothing$, Verify handles conditional statements. HandleIfs exploits symmetries by using the All-SAT query with Lex-Leader SBPs as described in Sect. 4 and calls Verify on each generated verification problem.

6 Implementation and Evaluation

To evaluate the effectiveness of increased lockstep execution of loops and symmetry-breaking, we have implemented our forward analysis algorithm from Sect. 5 on top of the Descartes verification tool for verifying k-safety properties (i.e., RVPs over k identical programs) of Java programs. We compare the performance of our prototype implementation to Descartes[1]. We use two metrics for comparison: the time taken and the number of Hoare triples processed by the verification procedure.

We have implemented two variants of our algorithm: Syn uses only synchrony (i.e., no symmetry is used), while Synonym uses synchrony as well as SBPs to exploit symmetry. All implementations (including Descartes) use the same guess-and-check invariant generator. This invariant generator is the same as the one originally employed by Descartes, but modified to generate more candidate invariants.

In Synonym, we compute symmetries in preconditions and postconditions only when all program copies are the same. For our examples, it sufficed to compute symmetries simply by checking if each possible permutation leads to

[1] While there are several tools for relational verification (e.g. Rosette/Unbound [24], VeriMapRel [12], Reve [16], MoCHi [16], SymDiff [21]), most of these do not handle Java programs, and to the best of our knowledge, none of these tools has support for k-safety verification for k greater than 2.

equivalent formulas[2]. Even though our implementation does not apply symmetry at all points where possible, and uses a simple symmetry-finding method, we still see solid improvements in performance during verification as described later.

6.1 Stackoverflow Benchmarks

Table 1: Stackoverflow Benchmarks.
Total times (in seconds) and Hoare triple counts (HTC) for Stackoverflow benchmarks, where for each property, the results for Syn and Synonym are divided into those for examples where they exhibit a factor of improvement over Descartes that is greater or equal to 1 (top) and those for which they do not (bottom). Improv gives the factor of improvement over Descartes, where the number of examples is given in parentheses.

Prop	Descartes		Syn				Synonym			
	Time	HTC	Time	Improv	HTC	Improv	Time	Improv	HTC	Improv
P1	3.11	4422	1.91	1.39 (27)	2255	1.69 (27)	1.82	1.32 (25)	2401	1.82 (32)
			0.57	0.789 (6)	752	0.809 (6)	0.87	0.816 (8)	48	0.979 (1)
P2	24.6	13434	7.77	2.63 (19)	3285	3.081 (16)	7.31	2.80 (19)	3224	3.140 (16)
			5.04	0.825 (14)	4638	0.714 (17)	5.1	0.816 (14)	4638	0.714 (17)
P3	18.85	10938	5.22	2.92 (20)	1565	4.36 (16)	5.22	2.91 (19)	1412	4.74 (15)
			6.18	0.584 (13)	6600	0.623 (17)	6.16	0.594 (14)	6742	0.629 (18)

The first set of benchmarks we consider are the Stackoverflow benchmarks originally used to evaluate Descartes. These each (correctly or incorrectly) implement the Java `Comparator` or `Comparable` interface, and check whether or not their *compare* functions satisfy the following properties:

- P1: $\forall x, y. sgn(compare(x, y)) = -sgn(compare(y, x))$
- P2: $\forall x, y, z. (compare(x, y) > 0 \wedge compare(y, z) > 0) \implies compare(x, z) > 0$
- P3: $\forall x, y, z. (compare(x, y) = 0) \implies (sgn(compare(x, z)) = sgn(compare(y, z)))$

(One of the original 34 Stackoverflow examples is excluded from our evaluation here because of the inability of the invariant generator to produce a suitable invariant.) We compare the results of running Syn and Synonym vs. Descartes for each property (see Table 1).

Because property P1 contains a symmetry, we notice an improvement in terms of number of Hoare triples with the use of symmetry for this property; however, the overhead of computing symmetries leads to Synonym performing more slowly than Syn even for some examples that exhibit reduced Hoare triple counts. Property P1 is also the easiest example to prove (all implementations can verify each example in under 0.3 seconds), so the effect of these overheads contribute more significantly to the runtime.

[2] Our implementation includes the syntactic symmetry-finding algorithm from Sect. 4.3.1, though we do not use it for evaluation here due to its high overhead in using an external tool for finding graph automorphisms.

For particular examples on which our implementations do not perform as well as DESCARTES, we perform reasonably closely to DESCARTES. These examples are typically smaller, and again overheads play a larger role in our poorer performance. For property P3, we find that when our implementations perform worse than DESCARTES, they do not perform as closely to DESCARTES as in the other two properties. This result arises because DESCARTES chooses an interleaving of statements from different program copies that allows DESCARTES to detect when $compare(x, y) \neq 0$ more quickly. In particular, DESCARTES synchronizes on loops but not on conditionals and chooses statements from a particular program copy whenever it is not synchronizing. The formulation of P3 is such that DESCARTES will always initially choose statements from the $compare(x, y)$ program copy and, in the absence of loops, will completely avoid needing to handle any other copies. Our implementation synchronizes the different program copies at conditionals, so in these we end up processing more Hoare triples before we reach the end of the copy and conclude that $compare(x, y) \neq 0$. For the versions of the examples where the property does not hold, though, we tend to find counterexamples more quickly and with fewer Hoare triples than DESCARTES.

6.2 Modified Stackoverflow Benchmarks

The original Stackoverflow examples are fairly small, with all implementations taking under 6 seconds to verify any particular example. To assess how we perform on larger examples, we modified several of the larger Stackoverflow comparator examples to be longer and contain more control-flow decisions. The resulting functions take three arguments and pick the "largest" object's id, where comparison among objects is performed based on the original Stackoverflow example code. Ties are broken by choosing the least id. We check whether these *pick* functions satisfy the following properties about rearranging arguments:

- P13: $\forall x, y, z.pick(x, y, z) = pick(y, x, z)$
- P14: $\forall x, y, z.pick(x, y, z) = pick(y, x, z) \wedge pick(x, y, z) = pick(z, y, x)$

The results from running property P13 are shown in Table 2. We see here that for these larger examples, Hoare triple counts are more reliably correlated with the time taken to perform verification. SYN outperforms DESCARTES on 14 of the 16 examples, and SYNONYM outperforms both DESCARTES and SYN on all 16 examples.

The results from running property P14 are shown in Table 3. P14 is the property with the largest k ($k = 4$) and has more than one symmetry. For this property, DESCARTES is unable to verify any of the examples with a one-hour timeout. Meanwhile, SYN is able to verify 10 of the 16 examples without exceeding the timeout. Exploiting symmetries here exhibits an obvious improvement, with SYNONYM not only being able to verify the same examples as SYN, consistently performing more quickly on the larger examples, but also being able to verify an additional example within an hour.

Table 2: Verifying P13 for Modified Stackoverflow examples. Times (in seconds) and Hoare triple counts (HTC).

Example	Descartes		Syn		Synonym	
	Time	HTC	Time	HTC	Time	HTC
ArrayInt-pick3-false-simple	1.71	2573	1	1355	0.64	682
ArrayInt-pick3-false	1.55	2591	1.06	1439	0.8	724
ArrayInt-pick3-true-simple	1.71	2573	1.03	1355	0.65	682
ArrayInt-pick3-true	1.55	2591	1.08	1439	0.81	724
Chromosome-pick3-false-simple	0.9	1115	0.9	883	0.53	446
Chromosome-pick3-false	2.51	2891	2.94	3019	1.59	1514
Chromosome-pick3-true-simple	0.9	1115	0.9	883	0.53	446
Chromosome-pick3-true	2.51	2891	2.96	3019	1.59	1514
PokerHand-pick3-false-part1	5.87	5825	0.42	359	0.46	359
PokerHand-pick3-false-part2	9.74	10589	0.85	323	0.86	323
PokerHand-pick3-false	16.91	16475	0.73	159	0.79	159
PokerHand-pick3-true-part1	5.83	5825	3.98	3503	2.4	1756
PokerHand-pick3-true-part2	9.8	10565	7.36	5933	4.53	2971
PokerHand-pick3-true	17.25	16475	12.1	9293	7.34	4651
Solution-pick3-false	76.4	99910	25.05	20645	20.42	10327
Solution-pick3-true	64.5	99910	19.66	20645	15.21	10327
Total	219.64	283914	82.02	74252	59.15	37605
Improvement	1	1	2.68	3.8237	3.713	7.5499

Table 3: Verifying P14 for Modified Stackoverflow examples. Times (in seconds) and Hoare triple counts (HTC). - indicates that no sufficient invariant could be inferred.

Example	Descartes		Syn		Synonym	
	Time	HTC	Time	HTC	Time	HTC
ArrayInt-pick3-false-simple	TO	TO	4.12	1938	4.66	1734
ArrayInt-pick3-false	TO	TO	4.92	2017	6.03	1500
ArrayInt-pick3-true-simple	TO	TO	321.15	140593	170.43	58586
ArrayInt-pick3-true	TO	TO	366.98	149125	240.25	62141
Chromosome-pick3-false-simple	TO	TO	47.8	14097	1.67	834
Chromosome-pick3-false	TO	TO	264.21	93052	4.91	3043
Chromosome-pick3-true-simple	TO	TO	299.51	79613	135.56	33179
Chromosome-pick3-true	TO	TO	TO	TO	848.22	225044
PokerHand-pick3-false-part1	TO	TO	0.57	391	0.73	391
PokerHand-pick3-false-part2	TO	TO	0.81	228	0.81	228
PokerHand-pick3-false	-	-	-	-	-	-
PokerHand-pick3-true-part1	TO	TO	2277.03	819553	1272.58	341486
PokerHand-pick3-true-part2	TO	TO	-	-	-	-
PokerHand-pick3-true	-	-	-	-	-	-
Solution-pick3-false	TO	TO	TO	TO	TO	TO
Solution-pick3-true	TO	TO	TO	TO	TO	TO

7 Related Work

The work most closely related to ours is by Sousa and Dillig [25], which proposed Cartesian Hoare Logic (CHL) for proving k-safety properties and the tool DESCARTES for automated reasoning in CHL. In addition to the core program logic, CHL includes additional proof rules, referred to as Cartesian Loop Logic (CLL). Neither CHL nor CLL force alignment at conditional statements or take advantage of symmetries; however, CLL proof rules allow for the exploitation of synchrony not only for fully lockstep loops, but also for *partially* lockstep loops. Our maximal lockstep-loop detection algorithm can be combined with partial lockstep execution to improve the efficiency of verification further by avoiding redundant work. For example, applying the Fusion 2 rule from CLL to our example while loops generated from RVP_1 in Sect. 2 would result in three subproblems and require reasoning twice about the second copy's loop finishing later. In combination with our maximal-loop-identification, we could generate just two subproblems: one where the first and third loops terminate first, and another where the second loop terminates first.

Other automatic efforts for relational verification typically use some kind of product programs [6,21,26,16,20,12,23], with a possible reduction to Horn solving [16,20,12,23]. Similarly to our strategy for synchrony, most of them attempt to leverage similarity (structural or functional) in programs to ease verification. However, we have seen less focus on leveraging relational specifications themselves to simplify verification tasks, although this varies according to the verification method used. Some efforts do not reason over product programs at all, relying on techniques based on decomposition [3] or customized theories with theorem proving [4,28] instead. To the best of our knowledge, none of these efforts exploit symmetry in programs or in relational specifications.

On the other hand, symmetry has been used very successfully in model checking parametric finite state systems [14,10,19] and concurrent programs [13]. Our work differs from these efforts in two main respects. First, the parametric systems targeted in these efforts have components that interact with each other or share variables. Second, their correctness specifications are also parametric, usually single-index or double-index properties in a propositional (temporal) logic. In contrast, in our RVPs, the individual programs are completely independent and do not share any common variables. The only interaction between them is via relational specifications for the purpose of verification. Furthermore, we discover symmetries in these relational specifications over multi-index variables, expressed as formulas in first-order theories (e.g., Linear Integer Arithmetic). We then exploit these symmetries to prune redundant RVPs during verification.

There are also some similarities between relational verification and verification of concurrent/parallel programs. In the latter, a typical verifier [17] would use *visible* operations (i.e., synchronization operations or communication on shared state) as synchronizing points in the composed program, such that executions between synchronizing points can be regarded as transactions, without requiring any interleavings of operations within transactions. Further, partial order reduction is used to prune away redundant interleavings [17]. We too se-

lect synchronizing points in the composed program, but this selection does not depend on any shared state (there is none!). Furthermore, one does not need to consider different orderings in interleavings of programs in the RVPs. Since these fragments are independent, it suffices to explore any one ordering. Instead, we exploit symmetries in the relational assertions to prune away redundant RVPs.

Finally, specific applications may impose additional requirements pertaining to visibility. For example, one may want to check for information leaks in low-type outputs not only at the end of the program but also at other specified intermediate points, or information leakage models for side-channel attacks may check for leaks based on given observer models [1]. Such requirements can be viewed as relational specifications at selected synchronizing points in the composed program. Again, we can leverage these relational specifications to simplify the resulting verification subproblems.

8 Conclusion

We have proposed novel techniques for improving relational verification, which has several applications including security verification, program equivalence checking, and regression verification. Our two key ideas are maximizing the amount of code that could be synchronized to require drastically less expensive verification methods and identifying symmetries in relational specifications to avoid redundant subtasks during analysis. An implementation on top of the Descartes verification tool leads to consistent improvements on a range of benchmarks. In the future, we would be interested in implementing the approach on top of a Horn-based relational verifier (e.g., [24]) and extending it to work with recursive data structures. We are also interested in developing an algorithm for syntactically finding symmetries in formulas that does not rely on an external graph automorphism tool.

References

1. J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi. Verifying constant-time implementations. In *USENIX*, pages 53–70. USENIX Association, 2016.
2. F. A. Aloul, K. A. Sakallah, and I. L. Markov. Efficient symmetry breaking for boolean satisfiability. *IEEE Trans. Computers*, 55(5):549–558, 2006.
3. T. Antonopoulos, P. Gazzillo, M. Hicks, E. Koskinen, T. Terauchi, and S. Wei. Decomposition instead of self-composition for proving the absence of timing channels. In *PLDI*, pages 362–375, 2017.
4. K. Asada, R. Sato, and N. Kobayashi. Verifying relational properties of functional programs by first-order refinement. *Sci. Comput. Program.*, 137:2–62, 2017.
5. A. Banerjee, D. A. Naumann, and M. Nikouei. Relational logic with framing and hypotheses. In *IARCS*, volume 65 of *LIPIcs*, pages 11:1–11:16. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016.
6. G. Barthe, J. M. Crespo, and C. Kunz. Relational verification using product programs. In *FM*, volume 6664 of *LNCS*, pages 200–214. Springer, 2011.

7. G. Barthe, P. R. D'Argenio, and T. Rezk. Secure information flow by self-composition. In *CSFW*, pages 100–114. IEEE, 2004.
8. N. Benton. Simple relational correctness proofs for static analyses and program transformations. In *POPL*, pages 14–25, 2004.
9. L. Beringer and M. Hofmann. Secure information flow and program logics. In *CSF*, pages 233–248. IEEE Computer Society, 2007.
10. E. M. Clarke, T. Filkorn, and S. Jha. Exploiting symmetry in temporal logic model checking. In *CAV*, pages 450–462, 1993.
11. J. M. Crawford, M. L. Ginsberg, E. M. Luks, and A. Roy. Symmetry-breaking predicates for search problems. In *KR*, pages 148–159, 1996.
12. 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, 2016.
13. A. F. Donaldson, A. Kaiser, D. Kroening, and T. Wahl. Symmetry-aware predicate abstraction for shared-variable concurrent programs. In *CAV*, pages 356–371, 2011.
14. E. A. Emerson and A. P. Sistla. Symmetry and model checking. In *CAV*, pages 463–478, 1993.
15. G. Fedyukovich, A. Gurfinkel, and N. Sharygina. Property directed equivalence via abstract simulation. In *CAV*, volume 9780, Part II, pages 433–453. Springer, 2016.
16. D. Felsing, S. Grebing, V. Klebanov, P. Rümmer, and M. Ulbrich. Automating regression verification. In *ASE*, pages 349–360. ACM, 2014.
17. P. Godefroid. Verisoft: A tool for the automatic analysis of concurrent reactive software. In *CAV*, pages 476–479, 1997.
18. B. Godlin and O. Strichman. Regression verification. In *DAC*, pages 466–471. ACM, 2009.
19. C. N. Ip and D. L. Dill. Verifying systems with replicated components in mur*phi*. In *CAV*, pages 147–158, 1996.
20. M. Kiefer, V. Klebanov, and M. Ulbrich. Relational program reasoning using compiler IR. In *VSTTE*, volume 9971 of *LNCS*, pages 149–165. Springer, 2016.
21. S. K. Lahiri, K. L. McMillan, R. Sharma, and C. Hawblitzel. Differential assertion checking. In *FSE*, pages 345–355. ACM, 2013.
22. F. Logozzo, S. K. Lahiri, M. Fähndrich, and S. Blackshear. Verification modulo versions: towards usable verification. In *PLDI*, page 32. ACM, 2014.
23. D. Mordvinov and G. Fedyukovich. Synchronizing Constrained Horn Clauses. In *LPAR*, volume 46 of *EPiC Series in Computing*, pages 338–355. EasyChair, 2017.
24. D. Mordvinov and G. Fedyukovich. Verifying Safety of Functional Programs with Rosette/Unbound. *CoRR*, abs/1704.04558, 2017. https://github.com/dvvrd/rosette.
25. M. Sousa and I. Dillig. Cartesian hoare logic for verifying k-safety properties. In *PLDI*, pages 57–69. ACM, 2016.
26. O. Strichman and M. Veitsman. Regression verification for unbalanced recursive functions. In *FM*, volume 9995 of *LNCS*, pages 645–658, 2016.
27. T. Terauchi and A. Aiken. Secure information flow as a safety problem. In *SAS*, volume 3672 of *LNCS*, pages 352–367. Springer, 2005.
28. H. Unno, S. Torii, and H. Sakamoto. Automating induction for solving horn clauses. In *CAV, Part II*, volume 10427 of *LNCS*, pages 571–591. Springer, 2017.
29. H. Yang. Relational separation logic. *Theoretical Computer Science*, 375(1-3):308–334, 2007.
30. 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.