Exploiting Synchrony and Symmetry in Relational Verification

Lauren Pick[(✉)], Grigory Fedyukovich[ID],
and Aarti Gupta

Princeton University, Princeton, USA
{lpick,grigoryf,aartig}@cs.princeton.edu

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 [4,5,8,27,31] or specific applications such as security verification [1,7,9], compiler validation [16,32], and differential program analysis [17,19,21–23]. 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. 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. Examples of techniques that exploit opportunities for such alignment include use of type-based analysis with self-composition [29], allowing flexibility in composition to be a mix of sequential and parallel [6], exploiting structurally equivalent programs for compiler validation [32], lockstep execution of loops in reasoning using Cartesian Hoare Logic [27], and merging Horn clause rules for relational verification [13,24].

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 [6,27], 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 novel 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 [27], and synchronizing at conditional blocks allows us to prune eagerly. More importantly, aligning different programs at conditional statements sets up our next strategy.

Our second strategy 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, expressed as formulas in first-order theories (e.g., linear integer arithmetic) with multi-index variables, to discover symmetries and exploit them to prune away redundant subtasks. Prior works on use of symmetry in model checking [11,14,15,20] are typically based on symmetric states satisfying the same set of indexed atomic propositions, and do not consider symmetries among different indices in specifications. To the best of our knowledge, ours is the first work to *extract* such symmetries in relational specifications, and to *use* them for pruning redundant subtasks during relational verification. For extracting these symmetries, we have lifted core ideas from symmetry-discovery and symmetry-breaking in SAT formulas [12] to richer formulas in first-order theories.

The strategies we propose for exploiting synchrony and symmetry via relational specifications are fairly general in that they can be employed in vari-

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

$y_1 > 20 \wedge y_2 > 20 \wedge y_3 > 20$

$y_1 > 20 \wedge y_2 > 20 \wedge y_3 \leq 20$

$y_1 > 20 \wedge y_2 \leq 20 \wedge y_3 > 20$

$y_1 > 20 \wedge y_2 \leq 20 \wedge y_3 \leq 20$

$y_1 \leq 20 \wedge y_2 > 20 \wedge y_3 > 20$

$y_1 \leq 20 \wedge y_2 > 20 \wedge y_3 \leq 20$

$y_1 \leq 20 \wedge y_2 \leq 20 \wedge y_3 > 20$

$y_1 \leq 20 \wedge y_2 \leq 20 \wedge y_3 \leq 20$

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

ous 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 [27]. 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(y₁ > 20); while (i₁ < 10) {x₁ *= i₁; i₁++;}
assume(y₂ > 20); while (i₂ < 10) {x₂ *= i₂; i₂++;}
assume(y₃ ≤ 20); while (i₃ < 10) {x₃++; i₃++;}
```

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

```
assume(y₁ ≤ 20); while (i₁ < 10) {x₁++; i₁++;}
assume(y₂ > 20); while (i₂ < 10) {x₂ *= i₂; i₂++;}
assume(y₃ > 20); while (i₃ < 10) {x₃ *= i₃; i₃++;}
```

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 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:

```
assume(y₁ > 20); while (i₁ < 10) {x₁ *= i₁; i₁++;}
```

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* $\boldsymbol{I_2}$:

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

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* $\boldsymbol{I}$:

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

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. We first describe our novel algorithm for inferring maximal sets of loops that can be executed in lockstep (Sect. 4.1). Next, we describe our technique for handling conditionals (Sect. 4.2). While this is similar to other prior work, the main purpose here is to set the stage for our novel methods for exploiting symmetry (Sect. 4.3).

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 loopy programs $\{P_1, \ldots, P_k\}$ and outputs a set of *maximal* 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 [27]. In this case, the entire set of input programs is one maximal 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 ith 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:

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

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 Q`$_i$` then R`$_i$` else S`$_i$, where $i \in \{1, 2\}$ and `R`$_i$ and `S`$_i$ are arbitrary blocks of code and could possibly have loops. Let them be a part of some RVP, which reduces to applying Propositions 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 Q`$_i$` then R`$_i$` else S`$_i$ for $i \in \{1, \dots, k\}$ and precondition $pre(\vec{x}_1, \dots, \vec{x}_k)$, we can simultaneously generate all possible combinations of decisions by querying a solver for all truth assignments to the `Q`$_i$s:

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

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 (V`$_1$`); U`$_1$`, `$\dots$`, assume (V`$_k$`); U`$_k\}$ where for each $i \in \{1, \dots, k\}$, if $v_i = true$, then `V`$_i$` = Q`$_i$ and `U`$_i$` = R`$_i$, else `V`$_i = \neg$`Q`$_i$ and `U`$_i$` = S`$_i$. We need to apply our verification algorithm on only the resulting sets of programs. For example, in our above RVP, if `Q`$_1$ is equivalent to `Q`$_2$ in all solutions, then the RVP reduces to verification of just two pairs of programs:

$$\text{assume (Q}_1\text{); R}_1 \quad \text{and} \quad \text{assume (Q}_2\text{); R}_2$$
$$\text{assume (}\neg\text{Q}_1\text{); S}_1 \quad \text{and} \quad \text{assume (}\neg\text{Q}_2\text{); S}_2$$

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** $(V, E) \leftarrow$ MAKECOLOREDAST(d) $\cup$ (V, E)

4:    **for** $v \in V$ with $x_i \in vars(color(v))$ **do**
5:        $V \leftarrow (V \setminus \{v\}) \cup \{$RECOLOR($v, v[x_i \mapsto x])\}$
6:        $E \leftarrow E \cup \{(v, v_i^{Id})\}$
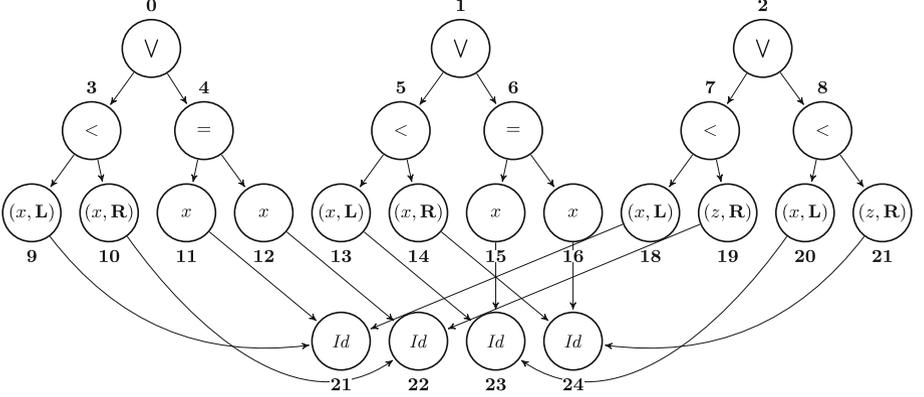

Fig. 2. Graph with vertex names (outside the vertices) and colors (inside the vertices).

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 due to symmetry. First, we discuss how to identify symmetries in formulas syntactically, and then we show how to use such symmetries.

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 Algorithm 1. The algorithm initializes a graph by 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 3) adds to the graph the union of the abstract syntax trees (AST) for the formula's conjuncts, where each vertex has a color corresponding to the type of its AST node. If a parent vertex has a color of an ordering-sensitive operation 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 4), 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 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 application of symmetry-breaking to generate 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) [12] 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 [12], 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 $l_i = \mathtt{Q}_i \implies \mathtt{Q}_{\pi(i)}$.

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 [24] are possible, but outside the scope of this work.

1: **procedure** VERIFY(pre, $Current$, Ifs, $Loops$, $post$)
2:     **while** $Current \neq \varnothing$ **do**
3:         **if** PROCESSSTATEMENT(pre, P_i, Ifs, $Loops$, $post$) = safe **then return** safe
4:     **if** $Loops \neq \varnothing$ **then** HANDLELOOPS(pre, $Loops$, $post$)
5:     **else if** $Ifs \neq \varnothing$ **then** HANDLEIFS(pre, Ifs, $Loops$, $post$)
6:     **else return** unsafe

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* = ∅), 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*, ∅, *post*).

Handling Conditionals. When *Current* = *Loops* = ∅, 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 implemented our algorithm from Sect. 5 on top of the DESCARTES tool for verifying k-safety properties, i.e., RVPs over k identical Java programs. We implemented two variants: SYN uses only synchrony (i.e., no symmetry is used), while SYNONYM uses both. All implementations (including DESCARTES) use the same guess-and-check invariant generator (the same originally used 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 equivalent formulas[1]. We compare the performance of our prototype implementations to DESCARTES[2]. We use two metrics for comparison: the time taken and the number of Hoare triples processed by the verification procedure. All experiments were conducted on a MacBook Pro, with a 2.7 GHz Intel Core i5 processor and 8 GB RAM.

6.1   Stackoverflow Benchmarks

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

---

[1] 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.

[2] While there are several tools for relational verification (e.g. ROSETTE/UNBOUND [25], VERIMAPREL [13], REVE [17], MOCHI [17], SYMDIFF [22]), 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.

P1: $\forall x, y. sgn(compare(x, y)) = -sgn(compare(y, x))$
P2: $\forall x, y, z.(compare(x, y) > 0 \land compare(y, z) > 0) \implies compare(x, z) > 0$
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 in Table 1. (Expanded versions and plots of these results are available in an extended version of the paper [26].)

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 to prove (all implementations can verify each example in under 0.3 s), so the overheads contribute more significantly to the runtime. For 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.

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* reports 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	7.83	2.62 (20)	3285	3.081 (16)	7.31	2.80 (19)	3224	3.140 (16)
			4.98	0.823 (13)	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)	1537	4.74 (16)
			6.18	0.584 (13)	6600	0.623 (17)	6.16	0.594 (14)	6600	0.623 (17)

6.2   Modified Stackoverflow Benchmarks

The original Stackoverflow examples are fairly small, with all implementations taking under 6 s to verify any example. To assess how we perform on larger examples, we modified several of the larger Stackoverflow comparator examples to be longer, take more arguments, 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 that allow reordering input arguments:

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

Note that P13 allows swapping the first two input arguments, while P14 allows any permutation of inputs, a useful hyperproperty.

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. For this property, note thatDESCARTES is unable to verify any of the examples within 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, with consistently faster performance 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-false	TO	TO	TO	TO	TO	TO

Summary of Experimental Results. Our experiments indicate that our performance improvements are consistent: on all DESCARTES benchmarks (in Table 1, which are all small) our techniques either have low overhead or show some improvement despite the overhead; and on modified (bigger) programs they lead to significant improvements. In particular, we report (Table 2) speedups up to 21.4x (on an example where the property doesn't hold) and 4.2x (on an example where it does). More importantly, we report (Table 3) that DESCARTES times out on 14 examples, where of these SYNONYM times out for 2 and cannot infer an invariant for one example.

7   Related Work

The work most closely related to ours is by Sousa and Dillig [27], 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 for loops, referred to as Cartesian Loop Logic (CLL). A generalization of CHL, called Quantitative Cartesian Hoare Logic was subsequently used by Chen et al. [10] to detect side-channel vulnerabilities in cryptographic implementations.

In terms of comparison, neither CHL nor CLL force alignment at conditional statements or take advantage of symmetries. We believe our algorithm for identifying a maximal set of lockstep loops is also novel and can be used in other

methods that do not rely on CHL/CLL. On the other hand, CLL proof rules allow not only fully lockstep loops, but also *partially* lockstep loops. Although we did not consider it here, our maximal lockstep-loop detection algorithm can be combined with their partial lockstep execution to further improve the efficiency of verification. For example, applying the Fusion 2 rule from CLL to our example while loops generated from RVP_1 (Sect. 2) would result in *three* subproblems and require reasoning twice about the second copy's loop finishing later. When combined with maximal lockstep-loop detection, 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 [6,13,17,21,22,24,28], with a possible reduction to Horn solving [13,17,21,24]. 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 [4,30] 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 [11,15,20] and concurrent programs [14]. Our work differs from these efforts in two main respects. First, the parametric systems considered in these efforts have components that interact with each other or share variables. Second, the 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 independent and do not share any common variables. The only interaction between them is via relational specifications. 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 [18] would use *visible* operations (i.e., synchronization operations or communication on shared state) as synchronizing points in the composed program. In our work, this selection is made based on the structure of the component programs and the ease of utilizing or deriving relational assertions for the code fragments. 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 synchrony requirements pertaining to visibility. For example, one may want to check for information leaks from private inputs to public outputs not only at the end of a program

but 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   Conclusions and Future Work

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 can be synchronized and identifying symmetries in relational specifications to avoid redundant subtasks. Our prototype 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 these ideas on top of a Horn-based relational verifier (e.g., [25]) and extending it to work with recursive data structures. We are also interested in developing an algorithm for finding symmetries in formulas that does not rely on an external graph automorphism tool.

References

1. Almeida, J.B., Barbosa, M., Barthe, G., Dupressoir, F., Emmi, M.: Verifying constant-time implementations. In: USENIX, pp. 53–70. USENIX Association (2016)
2. Aloul, F.A., Sakallah, K.A., Markov, I.L.: Efficient symmetry breaking for Boolean satisfiability. IEEE Trans. Comput. **55**(5), 549–558 (2006)
3. Antonopoulos, T., Gazzillo, P., Hicks, M., Koskinen, E., Terauchi, T., Wei, S.: Decomposition instead of self-composition for proving the absence of timing channels. In: PLDI, pp. 362–375 (2017)
4. Asada, K., Sato, R., Kobayashi, N.: Verifying relational properties of functional programs by first-order refinement. Sci. Comput. Program. **137**, 2–62 (2017)
5. Banerjee, A., Naumann, D.A., Nikouei, M.: Relational logic with framing and hypotheses. In: IARCS. LIPIcs, vol. 65, pp. 11:1–11:16. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2016)
6. Barthe, G., Crespo, J.M., Kunz, C.: Relational verification using product programs. In: Butler, M., Schulte, W. (eds.) FM 2011. LNCS, vol. 6664, pp. 200–214. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21437-0_17
7. Barthe, G., D'Argenio, P.R., Rezk, T.: Secure information flow by self-composition. In: CSFW, pp. 100–114. IEEE (2004)
8. Benton, N.: Simple relational correctness proofs for static analyses and program transformations. In: POPL, pp. 14–25 (2004)

9. Beringer, L., Hofmann, M.: Secure information flow and program logics. In: CSF, pp. 233–248. IEEE Computer Society (2007)
10. Chen, J., Feng, Y., Dillig, I.: Precise detection of side-channel vulnerabilities using quantitative Cartesian hoare logic. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, pp. 875–890 (2017)
11. Clarke, E.M., Filkorn, T., Jha, S.: Exploiting symmetry in temporal logic model checking. In: Courcoubetis, C. (ed.) CAV 1993. LNCS, vol. 697, pp. 450–462. Springer, Heidelberg (1993). https://doi.org/10.1007/3-540-56922-7_37
12. Crawford, J.M., Ginsberg, M.L., Luks, E.M., Roy, A.: Symmetry-breaking predicates for search problems. In: KR, pp. 148–159 (1996)
13. De Angelis, E., Fioravanti, F., Pettorossi, A., Proietti, M.: Relational verification through horn clause transformation. In: Rival, X. (ed.) SAS 2016. LNCS, vol. 9837, pp. 147–169. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-53413-7_8
14. Donaldson, A., Kaiser, A., Kroening, D., Wahl, T.: Symmetry-aware predicate abstraction for shared-variable concurrent programs. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 356–371. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_28
15. Emerson, E.A., Sistla, A.P.: Symmetry and model checking. In: Courcoubetis, C. (ed.) CAV 1993. LNCS, vol. 697, pp. 463–478. Springer, Heidelberg (1993). https://doi.org/10.1007/3-540-56922-7_38
16. Fedyukovich, G., Gurfinkel, A., Sharygina, N.: Property directed equivalence via abstract simulation. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9780, pp. 433–453. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41540-6_24
17. Felsing, D., Grebing, S., Klebanov, V., Rümmer, P., Ulbrich, M.: Automating regression verification. In: ASE, pp. 349–360. ACM (2014)
18. Godefroid, P.: VeriSoft: a tool for the automatic analysis of concurrent reactive software. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 476–479. Springer, Heidelberg (1997). https://doi.org/10.1007/3-540-63166-6_52
19. Godlin, B., Strichman, O.: Regression verification. In: DAC, pp. 466–471. ACM (2009)
20. Ip, C.N., Dill, D.L.: Verifying systems with replicated components in murϕ. In: Alur, R., Henzinger, T.A. (eds.) CAV 1996. LNCS, vol. 1102, pp. 147–158. Springer, Heidelberg (1996). https://doi.org/10.1007/3-540-61474-5_65
21. Kiefer, M., Klebanov, V., Ulbrich, M.: Relational program reasoning using compiler IR. In: Blazy, S., Chechik, M. (eds.) VSTTE 2016. LNCS, vol. 9971, pp. 149–165. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-48869-1_12
22. Lahiri, S.K., McMillan, K.L., Sharma, R., Hawblitzel, C.: Differential assertion checking. In: FSE, pp. 345–355. ACM (2013)
23. Logozzo, F., Lahiri, S.K., Fähndrich, M., Blackshear, S.: Verification modulo versions: towards usable verification. In: PLDI, p. 32. ACM (2014)
24. Mordvinov, D., Fedyukovich, G.: Synchronizing constrained horn clauses. In: LPAR. EPiC Series in Computing, vol. 46, pp. 338–355. EasyChair (2017)
25. Mordvinov, D., Fedyukovich, G.: Verifying safety of functional programs with rosette/unbound. CoRR, abs/1704.04558 (2017). https://github.com/dvvrd/rosette
26. Pick, L., Fedyukovich, G., Gupta, A.: Exploiting synchrony and symmetry in relational verification (extended version of CAV 2018 paper). https://cs.princeton.edu/%7Eaartig/papers/synonym-cav18.pdf

27. Sousa, M., Dillig, I.: Cartesian hoare logic for verifying k-safety properties. In: PLDI, pp. 57–69. ACM (2016)

28. Strichman, O., Veitsman, M.: Regression verification for unbalanced recursive functions. In: Fitzgerald, J., Heitmeyer, C., Gnesi, S., Philippou, A. (eds.) FM 2016. LNCS, vol. 9995, pp. 645–658. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-48989-6_39

29. Terauchi, T., Aiken, A.: Secure information flow as a safety problem. In: Hankin, C., Siveroni, I. (eds.) SAS 2005. LNCS, vol. 3672, pp. 352–367. Springer, Heidelberg (2005). https://doi.org/10.1007/11547662_24

30. Unno, H., Torii, S., Sakamoto, H.: Automating induction for solving horn clauses. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10427, pp. 571–591. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63390-9_30

31. Yang, H.: Relational separation logic. Theoret. Comput. Sci. **375**(1–3), 308–334 (2007)

32. Zaks, A., Pnueli, A.: CoVaC: compiler validation by program analysis of the cross-product. In: Cuellar, J., Maibaum, T., Sere, K. (eds.) FM 2008. LNCS, vol. 5014, pp. 35–51. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-68237-0_5