KestRel: Relational Verification Using E-Graphs for Program Alignment

ROBERT DICKERSON, Purdue University, USA
PRASITA MUKHERJEE, Purdue University, USA
BENJAMIN DELAWARE, Purdue University, USA

Many interesting program properties involve the execution of *multiple* programs, including observational equivalence, noninterference, co-termination, monotonicity, and idempotency. One strategy for verifying such *relational properties* is to construct and reason about an intermediate program whose correctness implies that the individual programs exhibit those properties. A key challenge in building an intermediate program is finding a good *alignment* of the original programs. An alignment puts subparts of the original programs into correspondence so that their similarities can be exploited in order to simplify verification. We propose an approach to intermediate program construction that uses e-graphs, equality saturation, and algebraic realignment rules to efficiently represent and build programs amenable to automated verification. A key ingredient of our solution is a novel data-driven extraction technique that uses execution traces of candidate intermediate programs to identify solutions that are semantically well-aligned. We have implemented a relational verification engine based on our proposed approach, called KestRel, and use it to evaluate our approach over a suite of benchmarks taken from the relational verification literature.

1 Introduction

Program verification tools have matured considerably in recent years, to the point that many mainstream programming languages are targeted by at least one verification tool [Baudin et al. 2021; Cao et al. 2018; Gurfinkel et al. 2015; Lattuada et al. 2023], and verification-aware languages now support a rich set of features [Leino 2010; Müller et al. 2016; Swamy et al. 2016]. These tools and languages are designed to automate verification of semantically rich program properties, focusing on *individual* program executions; for example, showing that every final state of a program satisfies a desired postcondition. Many interesting program behaviors, however, involve the executions of *multiple* programs. To verify correctness of a program optimization, we must show that an original program p_1 and its optimized version p_2 are *observationally equivalent*. That is, when p_1 and p_2 are executed in the same initial state, they arrive at the same final state. Proving this sort of *relational* behavior requires reasoning jointly about the executions of *both* p_1 and p_2. A variety of important program behaviors are inherently relational [Barthe et al. 2011a; Clarkson and Schneider 2010], including refinement, idempotence, non-interference, and co-termination.

Somewhat surprisingly, reasoning about relational properties does not require the development of specialized tooling, as most common relational verification problems can be immediately reduced to non-relational ones. This reduction is straightforward: given a pair of target programs, we can construct an *intermediate program* that encodes their joint execution by renaming any shared variables and concatenating the programs together [Barthe et al. 2011b; Francez 1983]. The desired relational property can then be established by applying single-program verification techniques to this intermediate program. While this approach is theoretically sound, the concatenated intermediate program is often prohibitively difficult to verify in practice.

To demonstrate the limitations of this strategy, consider the pair of programs, p_1 and p_2, shown in Fig. 1. Both programs iterate over a list of employees, scheduling bonus payments for the identified workers via some black-box financial services API. Program p_2 does so slightly more efficiently

```
int i1 = 0;
while (i1 < length(bonuses1)) {
 int id1 = bonuses1.get(i1);
 int sal1 = emp1.getSalary(id1);
 payments1.schedule(id1, sal1 * calc_bonus(rate1));
 i1 += 1; }
```

p_1

```
int i2 = 0;
int bonus2 = calc_bonus(rate2);
while (i2 < length(bonuses2)) {
 int id2 = bonuses2.get(i2);
 int sal2 = emp2.getSalary(id2);
 payments2.schedule(id2, sal2 * bonus2);
 i2 += 1; }
```

p_2

Fig. 1. Two programs for calculating employee bonuses.

than p_1, however, as it caches part of the bonus calculation prior to entering the loop. To establish that this optimization is safe, we need to prove that, starting from the same initial state, p_1 and p_2 schedule the same set of payments. Since the two programs already operate over disjoint variables, it suffices to show that $\texttt{payments}_1$ and $\texttt{payments}_2$ are the same after executing the program $p_1\,;\ p_2$.

In theory, we could do so using any verifier for the source language of p_1 and p_2. However, this task is beyond the capabilities of many automated program verifiers, as reasoning about this intermediate program requires a pair of loop invariants that *completely* characterize how each loop in $p_1\,;\ p_2$ mutates its copy of `payments`. Even given such loop invariants, establishing that they hold requires a complete specification of the `schedule` method. In essence, we have reduced the problem of showing this optimization is correct to proving full functional correctness of p_1 and p_2 [Francez 1983]. As this example illustrates, the simple strategy of constructing an intermediate program by concatenation can result in a disproportionately difficult verification task. In fact, depending on the input programs, the loop invariants and function specifications required to reason about the intermediate program may not be expressible in a decidable logic [Shemer et al. 2019].

One solution to this problem is to find an intermediate program that *aligns* the original programs in a way that that better captures the commonalities between their subparts [Barthe et al. 2011a]. Better alignments enable verifiers to exploit these similarities, simplifying the proof effort. To see how, consider the program $p_{1\times2}$ in Fig. 2 to the right, which also encodes the semantics of p_1 and p_2. In contrast to $p_1\,;\ p_2$, $p_{1\times2}$ features a single loop that encodes the simultaneous, or *lockstep*, execution of the loops in p_1 and p_2. This single loop captures the fact that each iteration of these loops schedules the same payment, as they both call $\texttt{payments.schedule}(\ldots)$ with identical arguments. This property is straightforwardly captured by a simple loop invariant expressible in the theory of equality with uninterpreted functions (EUF). In addition, rather than a full specification of `schedule`, we need only know that calling it with the same arguments yields the same result, a property that can also be captured in EUF. Importantly, EUF is supported by all modern SMT solvers, unlocking the possibility of tractable automated verification. As this example suggests, more sophisticated strategies for constructing aligned intermediate programs can greatly simplify relational reasoning.

```
int i1 = 0; int i2 = 0;
int bonus2 = calc_bonus(rate2);
while (i1 < length(bonuses1)) {
 int id1 = bonuses1.get(i1);
 int id2 = bonuses2.get(i2);
 int sal1 = emp1.getSalary(id1);
 int sal2 = emp2.getSalary(id2);
 payments1.schedule(id1, sal1 * calc_bonus(rate1));
 payments2.schedule(id2, sal2 * bonus2);
 i1 += 1;  i2 += 1; }
```

$p_{1\times2}$

Fig. 2. A program that combines p_1 and p_2 from Fig. 1

This paper presents an automated approach to intermediate program construction which aligns a pair of target programs in order to effectively reason over their joint execution. While strategies exist for constructing the kind of intermediate program given in $p_{1\times2}$, they tend to

operate *syntactically* [Zaks and Pnueli 2008] by identifying locations or *cut points* in the input programs to bring into alignment. This places strong constraints on the shape of the input programs, effectively requiring a one-to-one correspondence between program locations. Such approaches fail to take into account how the *semantics* of programs affect their alignment. Consider the pair of programs in Fig. 3 taken from Unno et al. [2021]. Each program sets its version of `y` to `2x`, but the loop in $\texttt{double}_1$ executes twice as many times as the loop in $\texttt{double}_2$. A single loop that encodes the behaviors of both loops must align each iteration of the loop in $\texttt{double}_2$ with two iterations of the one in $\texttt{double}_1$. One challenge when considering these kinds of semantic alignments is that they naturally lead to larger sets of candidate intermediate programs. While the number of ways to align the locations in a pair of programs is already large, $O(|p_1| \cdot |p_2|)$, the space of semantic alignments involving these kinds of loop schedulings and unrollings is potentially unbounded; in this example, an arbitrary number of iterations of the loop in $\texttt{double}_1$ can be aligned with an arbitrary number of iterations of the loop in $\texttt{double}_2$. Even if we limit ourselves to matching a bounded number n of iterations between loops, there are now $O(n \cdot n)$ ways of aligning *every* pair of loops in the target programs. A successful semantic alignment strategy must be able to efficiently represent and explore a large (possibly infinite) set of candidate intermediate programs.

```
int z1 = 0;
int y1 = 0;
z1 = 2*x1;
while (z1>0) {
  z1 = z1 − 1;
  y1 = y1 + x1;
}
```

$\texttt{double}_1$

```
int z2 = 0;
int y2 = 0;
z2 = x2;
while (z2>0) {
  z2 = z2 − 1;
  y2 = y2 + x2; }
y2 *= 2;
```

$\texttt{double}_2$

Fig. 3. A good alignment matches every iteration of the loop in $\texttt{double}_1$ with two iterations of the loop in $\texttt{double}_2$.

Our solution to this challenge is to use e-graphs [Nelson 1980; Nieuwenhuis and Oliveras 2005; Willsey et al. 2021] to compactly represent a space of semantically aligned intermediate programs. While e-graphs have previously been used to efficiently represent and explore sets of equivalent programs in order to find one with the best performance [Tate et al. 2009], we use them to identify intermediate programs that are amenable to verification. Instead of representing these programs directly, we instead embed them in a relational calculus equipped with algebraic *realignment rules* in the spirit of Antonopoulos et al. [2023]. This allows us to frame the search for a good alignment as a search for a set of realignment rule applications starting from a naïve concatenative embedding of the original program. To compactly represent this space of rule applications, we initialize an e-graph with the naïve alignment and saturate it with the set of realignment rules. A key invariant of the resulting e-graph is it only contains elements corresponding to intermediate programs that are semantically equivalent to the original pair of programs (Theorem 3.4). A pleasant consequence of our strategy is that it can easily incorporate existing semantics-preserving transformations on individual program, e.g., loop unrolling, that can unlock better alignments.

As the name suggests, finding a good semantic alignment depends on the *semantics* of the target programs. Thus, our approach examines traces from concrete program executions to look for indicators of promising alignments to guide our search. For example, loops in the intermediate program which combine loops from the input programs, e.g., $\texttt{p}_{1\times2}$, are likely not well-aligned unless the loop conditions from each input program become false at the same time. When exploring the space of alignments, we can favor candidate programs whose execution traces exhibit this property. In contrast to other data-driven approaches to semantic alignment [Churchill et al. 2019] which are only guaranteed to produce programs that cover a set of test cases, our approach always finds a semantically equivalent intermediate program. This allows us to find alignments even when no finite set of traces is capable of capturing the relationship between the input program.

Our approach is able to find an intermediate program with a desirable alignment for the programs in Fig. 4, which Churchill et al. [2019] identifies as particularly challenging. To the best of our knowledge, no other existing technique is capable of building an aligned intermediate program for these programs.

```
int f(int m, int n) {
  int k := 0;
  for i in 0..n {
    k += m;
  }
  return k;
}
```

```
int g(int m, int n) {
  int k := 0;
  for i in 0..n {
    for j in 0..m {
      k++;
  }}
  return k; }
```

Fig. 4. The function `f` optimizes `g` by collapsing its inner loop into a single addition operation.

To demonstrate the feasibility of our approach, we have built a prototype relational verification tool, KestRel. KestRel targets a subset of C and supports Dafny [Leino 2010] and SeaHorn [Gurfinkel et al. 2015] as backends, enabling it to verify relational properties of programs that use APIs to manage abstract data types with hidden internal state, as well as array-manipulating C programs. We have used KestRel to produce aligned intermediate programs for a diverse suite of benchmarks and relational properties taken from the literature, including examples that fall outside the reach outside of similar tools. Our experimental results show that KestRel discovers alignments that enable verification to succeed where simpler alignment strategies fail.

We pause here to note that an alternative strategy is to instead develop bespoke relational verification tools tailored to assertions over multiple program executions [Farzan and Vandikas 2019a,b; Francez 1983; Itzhaky et al. 2024; Shemer et al. 2019; Unno et al. 2021]. These tools also try to exploit syntactic and semantic commonalities between the target programs in order to simplify the verification task.[1] A popular example of this approach is *relational program logics*, which adapt the judgment of traditional program logics [Hoare 1969], to range over multiple programs [Aguirre et al. 2017; Banerjee et al. 2016; Barthe et al. 2013b; Benton 2004; Francez 1983; Maillard et al. 2019; Yang 2007]. These logics are equipped with specialized rules for reasoning about the joint behaviors of programs. Many logics include a specialized rule for reasoning about "lockstep" loops whose iterations are in a perfect 1-1 correspondence, for example, enabling much simpler loop invariants [Sousa and Dillig 2016]. While other bespoke relational verification tools differ in their alignment strategies, they all attempt to exploit commonalities between the target programs to simplify the verification task. Section 6 provides a more detailed discussion of these approaches.

In summary, this paper describes the following contributions:

- We present a novel application of e-graphs to build and compactly represent the space of aligned intermediate programs expressed in a domain of relational alignments equipped with algebraic realignment rules.
- We develop a hybrid extraction technique that combines a syntactic cost metric with a novel non-local extraction technique that uses dynamic execution traces to identify alignments amenable to automated verification.
- We present a relational verification framework, KestRel, that implements this approach, and demonstrate its utility by evaluating it on a diverse set of challenging relational verification benchmarks drawn from the literature.

The remainder of the paper is structured as follows. We begin with a brief primer on e-graphs, followed by an overview of our approach. Section 3 then formalizes our approach to relational verification using a core calculus for relational alignment. Next, Section 4 describes how we use e-graphs

[1] What we refer to as "intermediate programs" in this work are often called "product programs" in the literature. However, many bespoke relational verification approaches use the term "product program" to refer to their embeddings of multiple programs. This is fundamentally distinct from our notion of intermediate programs as standalone programs that are written in the same language as the input programs ans that are independent of a specific verification technique. To avoid confusion, we use "intermediate program" for the latter connotation of "product program" throughout this paper.

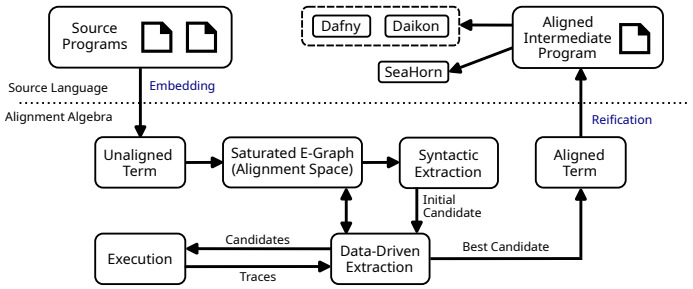


Fig. 5. High-level overview of KestRel.

to construct and represent a set of candidate intermediate programs, and then presents our data-driven technique for identifying and extracting an intermediate program amenable to automated verification from this space. Section 5 presents an empirical evaluation of KestRel, a relational verification tool based on our approach. The paper ends with related work and conclusions.

2 Overview

Fig. 5 presents a high-level overview of our approach to relational verification. Given a pair of programs as input, our goal is to output a single intermediate program which can be handed off to automated, non-relational verification tools. To do so, we first embed the input programs as a term in a relational algebra and insert this term into an e-graph. We construct a space of aligned programs by saturating this e-graph using a set of algebraic realignment rules. Using purely syntactic criteria (e.g., the number of fused loops in the alignment), we identify an initial candidate. We then give this initial alignment to a data-driven extraction method which uses the e-graph to look for better alignments, examining execution traces to measure the quality of a candidate alignment. The best alignment from this process is then reified into a single intermediate program. This program can then be handed to an off-the-shelf verifier like Dafny or SeaHorn.

2.1 Introduction to E-Graphs

We begin with a brief overview of e-graphs and equality saturation; readers familiar with these topics may safely skip this section.

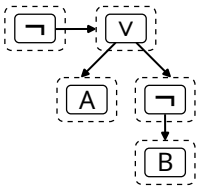
E-Graphs. An e-graph [Nelson 1980; Nieuwenhuis and Oliveras 2005] is a data structure which compactly represents equivalence classes on sets of terms. An e-graph contains *e-nodes* and *e-classes*, where each e-node associates a symbol with a (possibly empty) list of child e-classes and each e-class contains a collection of equivalent e-nodes. Each e-class has a unique identifier, and an equivalence relation between e-class identifiers is stored in an union-find structure [Tarjan 1975].

Definition 2.1 (E-Graph). An *e-graph* is a triple (U, M, H) where:

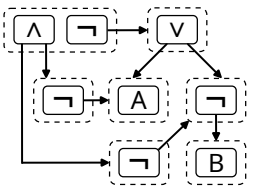
- U is a union-find data structure over e-class identifiers.
- M is a mapping from e-class identifiers to to e-classes such that all identifiers equivalent in U map to the same e-class: $\forall i, j.\ M[i] = M[j] \iff \mathsf{find}(U, i) = \mathsf{find}(U, j)$.
- H is a mapping from e-nodes to e-class identifiers.

A term t is *represented* in an e-graph or e-class if there exists an e-node d in the e-graph or e-class such that:

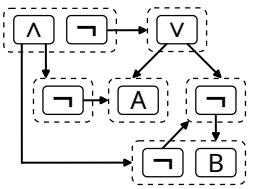
(1) d's symbol matches the symbol at t_r, the root of t's syntax tree,
(2) d has the same number of children as t_r, and
(3) each child $t_0, \ldots, t_n$ of t_r is represented by a corresponding child e-class $c_0, \ldots, c_n$ of d.



(a) An e-graph containing only $\neg(A \vee \neg B)$.



(b) E-graph (a) with $\neg(x \vee y) \rightarrow \neg x \wedge \neg y$ applied.



(c) E-graph (b) with $\neg\neg x \rightarrow x$ applied.

Fig. 6. Example e-graphs.

Example 2.2. Using classical Boolean logic as an example language, Fig. 6a shows the term $\neg(A \vee \neg B)$ represented as an e-graph. E-nodes and e-classes are displayed as solid and dashed boxes, respectively. Our initial e-graph has e-nodes for each node in the term's syntax tree, and each of these e-nodes is in a singleton e-class. While each e-class in this e-graph contains a single e-node, note that the children of e-nodes are *e-classes*, not other e-nodes. To extract a term represented by an e-graph, we start at its root e-class and recursively chose an e-node from its child e-classes. In this case, each e-class has exactly one choice, so this e-graph only represents the original term.

Rewrites Rules. E-graphs can compactly represent the term equivalences induced by a set of rewrite rules. Rewrite rules have the form $l \rightarrow r$, which intuitively says any subterm of an e-graph that matches l may be replaced by the term r. Both l and r can contain variables; these are instantiated when the rule is applied. An example of a sound rewrite rule in Boolean algebra is $x \vee \bot \rightarrow x$, which says a disjunction of false with any term x is equivalent to x.

To add representations of terms rewritten according to some rule $l \rightarrow r$ to an e-graph, we:

(1) locate all pairs (c_i, σ_i) where c_i is an e-class representing a term that matches l and σ_i is an appropriate instantiation of the variables in l,
(2) for each c_i, add a new e-class e_i to the e-graph such that e_i represents $r[\sigma_i]$, where $r[\sigma_i]$ is r with substitutions in σ_i applied, and finally
(3) merge each c_i with each e_i.

The first task is accomplished using *e-matching* [Moura and Bjørner 2007], while the second and third are handled by the `add` and `merge` operations of the e-graph's union-find structure U.

Example 2.3. One sound rewrite rule we can apply in the context of classical Boolean logic is given by De Morgan's laws: $\neg(x \vee y) \rightarrow \neg x \wedge \neg y$. The upper left e-class in Fig. 6a has a match with the left-hand side of this rewrite rule where $\sigma = [x \mapsto A, y \mapsto \neg B]$. Fig. 6b shows the e-graph in Fig. 6a with this rewrite applied; a new $\wedge$ e-node has been created and merged with the original matching e-class. We can extract the rewritten term $\neg A \wedge \neg\neg B$ from this e-graph by choosing the $\wedge$ e-node as the representative of the root e-class, and recursively selecting the singleton e-nodes as the representatives of each of its descendants.

Example 2.4. Another sound rewrite rule in classical Boolean logic is double negation elimination: $\neg\neg x \rightarrow x$. Fig. 6c depicts Fig. 6b with this rewrite applied. The e-nodes corresponding to the terms $\neg\neg B$ and B are now equivalent as the two bottommost e-classes have been merged. We can find the term $\neg A \wedge B$ by starting with $\wedge$ in the upper left e-class as before, choosing B for its first child in the lower right e-class, and choosing the only available e-node in all other e-classes. Note also that the e-graph in Fig. 6c represents infinite terms of the form $(\neg\neg)^* B$, where * is Kleene star, as we can traverse the cycle between the bottom right e-classes an arbitrary number of times.

Equality Saturation and Extraction. Each application of a rewrite rule to an e-graph is additive, potentially creating new e-nodes and merging e-classes, but continuing to represent all previously

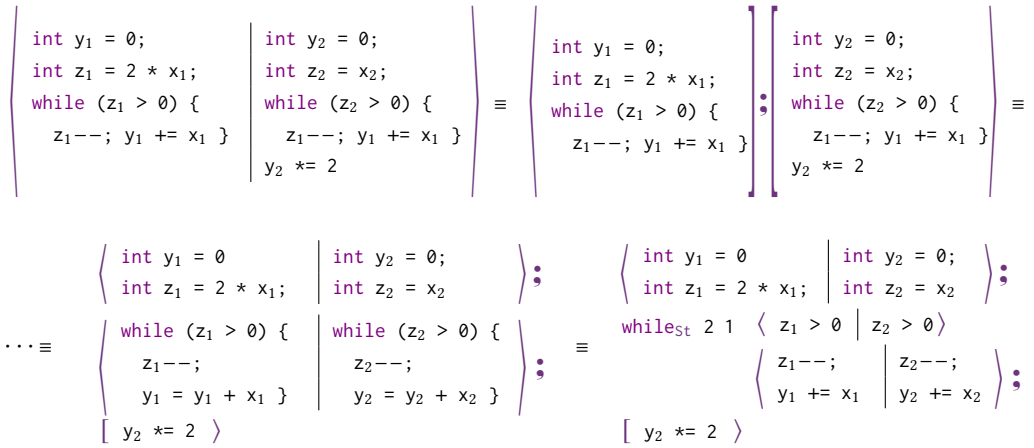


Fig. 8. Abbreviated derivation of an alignment using the rewrite rules presented in Section 4. The initial term relates two programs, both of which set y to $2x$. The program on the left does this by counting to $2x$, while the program on the right counts to x before multiplying by 2. The final term aligns the pre-loop initializations, the loop executions (with two iterations of the left program's loop for every one of the right's), and does not align the right-only `y *= 2` with anything.

represented terms. *Equality saturation* [Tate et al. 2009] is the process of initializing an e-graph with some term, and then continually matching and applying rewrite rules to that e-graph until either no new opportunities for rewrites are found or some bound or timeout is reached. The result is an e-graph which represents a (potentially infinite) space of terms which are equivalent to the original under some sequence of rewrite rule applications.

Once an e-graph has been saturated in this way, the challenge is how to select or *extract* the best term, according to some metric. Metrics that are defined in terms of subterms often admit a simple extraction technique. In order to extract the smallest representative term in an e-graph, for example, we first recursively assign each e-node a *cost* of one more than the sum of the costs of its child e-class, and each e-class the minimum cost of its constituent e-nodes. Extraction then proceeds in a top-down manner by selecting the lowest cost e-node in each e-class. More complicated metrics may require more sophisticated extraction techniques. In the case of KestRel, the quality of a program alignment depends on the semantics of the intermediate program it represents; this is the motivation for our data-driven approach to extraction.

2.2 Example KestRel Workflow

We illustrate the key pieces of our proposed approach to intermediate program construction by showing how we build $\texttt{double}_{1\times2}$ shown in Fig. 7, given $\texttt{double}_1$ and $\texttt{double}_2$ from Fig. 3. Importantly, verifying that $y_1 = y_2$ after executing $\texttt{double}_{1\times2}$ requires a loop invariant that is a simple equality between the values of $y_1 = 2 * y_2$, while verifying $\texttt{double}_1\texttt{;}\ \texttt{double}_2$ requires two loop invariants, each of which involve x, y, and z.

```
int y1 = 0; int y2 = 0;
int z1 = 2*x1; int z2 = x2;
while (z2 > 0)
  { z1−−; y1 += x1; z1−−; y1 += x1;
    z2−−; y2 += x2 }
y2 *= 2;
```

$\texttt{double}_{1\times2}$

Fig. 7. An intermediate program encoding the behaviors of $\texttt{double}_1$ and $\texttt{double}_2$.

An Algebra of Alignments. Our first step in constructing $\texttt{double}_{1\times2}$ is to embed $\texttt{double}_1$ and $\texttt{double}_2$ into a richer domain that provides a more structured representation of intermediate programs. We refer to elements of this domain as *alignments* of

(a pair of) programs. The simplest alignment has the form $\langle p_1 | p_2 \rangle$; this alignment represents an intermediate program which fully executes p_1 and then p_2, i.e. p_1; p_2. Our domain also includes finer-grained alignments that group together subterms of the intermediate program. The alignment $\langle s_1 \mid t_1; t_2 \rangle ; \langle s_2 \mid t_3 \rangle$, for example, groups together the first statement of s_1; s_2 with the first two statements of t_1; t_2; t_3 and aligns the last statements of both programs; these sub-alignments are composed together with the ; operator. This domain is equipped with other relational operators for aligning different control flow operators. The most important of these is the $\texttt{while}_R\ \langle b_1 | b_2 \rangle\ \langle c_1 | c_2 \rangle$ operator, which represents an intermediate program that executes the bodies of two loops in lockstep. The final alignment in Fig. 8 encodes $\texttt{double}_{1\times2}$ using a variant of this operator, $\texttt{while}_{St}\ m\ n\ \langle b_1 | b_2 \rangle\ \langle c_1 | c_2 \rangle$, which executes c_1 m times and c_2 n times on each iteration.

Alignments are equipped with an equivalence relation, ≡. Intuitively, equivalent alignments represent semantically equivalent intermediate programs. This equivalence admits several relational *realignment laws* which can be used to reason about the equivalence of different alignments. The equivalence of all of the alignments shown in Fig. 8 are justified by these laws, for example. Importantly, the alignment that encodes $\texttt{double}_{1\times2}$ can be automatically derived from $\langle \texttt{double}_1 \mid \texttt{double}_2 \rangle$ via a sequence of rewriting steps.

Representing Possible Alignments with E-Graphs. While the chain of rewrites shown in Fig. 8 yields a desirable alignment, many other equivalent alignments can be similarly derived via realignment laws. To explore the set of equivalent alignments, we use e-graphs as a compact representation of the space of alignments. Fig. 9(a) gives a simplified representation of the $\langle \texttt{double}_1 \mid \texttt{double}_2 \rangle$ as an e-graph, while Fig. 9(b) depicts an e-graph that simultaneously encodes both the first and second alignments in 8, as reflected by the inclusion of the $\langle | \rangle$ and ; e-nodes in the same e-class. Saturating the e-graph in Fig. 9(a) with a set of realignment rules results in an e-graph that includes the alignment corresponding to $\texttt{double}_{1\times2}$.

Searching for Desirable Alignments. Once a fully saturated e-graph that represents the space of possible alignments of $\langle \texttt{double}_1 \mid \texttt{double}_2 \rangle$ is in hand, our next step is to *extract* $\texttt{double}_{1\times2}$ from the set of intermediate programs embedded in the e-graph. Modern e-graph libraries [Willsey et al. 2021] are equipped with a mechanism that greedily extracts terms by recursively using a cost function to select the "best" representative of each equivalence class. This strategy is inherently *syntactic*, selecting nodes based on the terms they represent. However, identifying the best alignment often

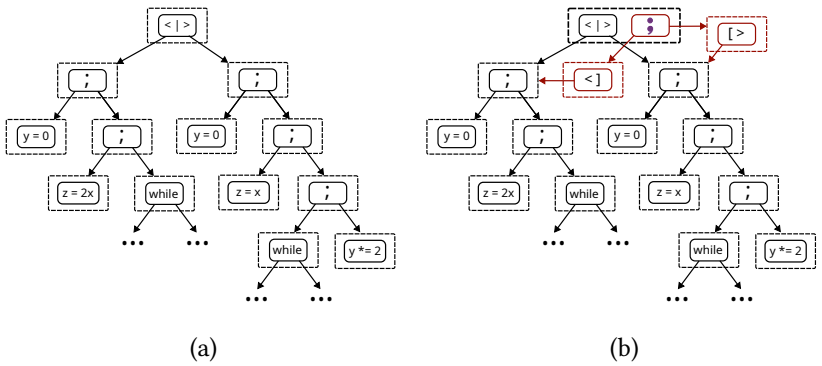


Fig. 9. E-graphs containing representations of possible alignments between $\texttt{double}_1$ and $\texttt{double}_2$. The e-graph on the left (a) contains only the initial embedding. The e-graph on the right (b) contains the initial embedding plus an application of the REL-DEF given in Fig. 13. (Added nodes are depicted in red.) It is possible to extract both the first and second terms in Fig. 8 from (b).

```
a ::=      Integer Expressions
     n | x | a + a | a − a | a * a
b ::=      Boolean Expressions
     true | false | a = a | a < a
     | not b | b && b
c ::=            Commands
     skip | c; c | x := a | while b c
     | if b then c else c

if b then c ≜ if b then c else skip
```

```
r ::=          Aligned Commands
     ⟨ c | c ⟩
     | r ; r
     | if_R ⟨ b | b ⟩ then r else r
     | while_R ⟨ b | b ⟩ r

⟨s] ≜ ⟨s | skip⟩
[s⟩ ≜ ⟨skip | s⟩
while_St n m ⟨b1 | b2⟩ ⟨c1 | c2⟩ ≜
  while_R ⟨b1 | b2⟩ ⟨if b1 then c1^n | if b2 then c2^m⟩
```

Fig. 10. Syntax and notations for Imp and CoreRel.

involves *semantic* properties of the intermediate program it represents. Finding the alignment that produces $\texttt{double}_{1\times2}$, for example, requires the observation that the body of the loop in $\texttt{double}_1$ must be executed twice for every execution of $\texttt{double}_2$. To find alignments with this kind of semantic property, we use a data-driven extraction technique that examines traces of states generated by from candidate intermediate program executions to determine alignment quality. Observing dynamic traces allows our extraction mechanism to observe this semantic relationship. We use a Markov-Chain Monte-Carlo (MCMC)-based [Hastings 1970] algorithm to sample programs from promising parts of the search space, using the e-graph to provide neighboring extraction candidates during the search. Once a promising candidate alignment has been found, our final step is to reify it into an intermediate program, e.g., $\texttt{double}_{1\times2}$, which can then be given to an off-the-shelf single program verifier like Dafny [Leino 2010] or SeaHorn [Gurfinkel et al. 2015].

3 The CoreRel Calculus

This section describes a core calculus for program alignment, called CoreRel, which we use to formalize our approach to intermediate program construction.[2] Our target programming language is the simple imperative programming language, Imp, whose syntax is shown on the lefthand side of Fig. 10. The calculus assumes an infinite set of identifiers for program variables; program states are partial functions from identifiers to integers. The semantics of an Imp program $\texttt{c}$ is given by a completely standard big-step reduction relation from input states σ to output states σ': $\sigma, \texttt{c} \Downarrow \sigma'$. Imp is also equipped with a straightforward program logic that acts as our "off-the-shelf" verifier for Imp programs. Formally, this logic proves partial Hoare triples of the form $\vdash \{\phi\}\ \texttt{c}\ \{\phi\}$, and is parameterized over the underlying assertion language. We write $\sigma \models \phi$ to denote that a state σ satisfies the assertion ϕ.

Equipped with these ingredients, it is straightforward to state our relational verification problem:

Definition 3.1 (Relational Safety). Given a pair of Imp programs, $\texttt{c}_1$ and $\texttt{c}_2$, we say that $\texttt{c}_1$ and $\texttt{c}_2$ are *safe* with respect to the relational pre- and postconditions ϕ and ψ if every pair of final states reachable from input states meeting ϕ is guaranteed to satisfy ψ. We denote relational safety as:

$$\models_R \{\Phi\}\ \texttt{c}_1 \otimes \texttt{c}_2\ \{\Psi\} \triangleq \forall \sigma_1, \sigma_2.\ \sigma_1 \uplus \sigma_2 \models \phi \implies \forall \sigma_1', \sigma_2'. \sigma_1, \texttt{c}_1 \Downarrow \sigma_1' \wedge \sigma_2, \texttt{c}_2 \Downarrow \sigma_2' \implies \sigma_1' \uplus \sigma_2' \models \psi$$

An essential property of this definition is that both $\texttt{c}_1$ and $\texttt{c}_2$ operate over *disjoint* state spaces (hence the use of $\uplus$ to merge states)– as we shall see, this property plays a key role in the equations used to align programs. Following the convention of Benton [2004], we use the subscripts 1 and 2 to disambiguate references to any identifiers shared between the left- and right-hand programs.

[2]The anonymized supplementary material includes a complete Coq formalization of CoreRel and its metatheory.

$$\frac{\sigma_1,\ \mathtt{b}_1 \Downarrow \mathtt{true} \quad \sigma_2,\ \mathtt{b}_2 \Downarrow \mathtt{true} \quad (\sigma_1,\ \sigma_2)\ \mathtt{r} \Downarrow (\sigma_1',\ \sigma_2') \quad (\sigma_1',\ \sigma_2'),\ \mathtt{while}_R\ \langle \mathtt{b}_1 \mid \mathtt{b}_2 \rangle\ \mathtt{r} \Downarrow (\sigma_1'',\ \sigma_2'')}{(\sigma_1,\ \sigma_2),\ \mathtt{while}_R\ \langle \mathtt{b}_1 \mid \mathtt{b}_2 \rangle\ \mathtt{r} \Downarrow (\sigma_1'',\ \sigma_2'')}\ \text{E-WhileT}$$

$$\frac{(\sigma_1,\ \sigma_2),\ \mathtt{r}_1 \Downarrow (\sigma_1',\ \sigma_2') \quad (\sigma_1',\ \sigma_2'),\ \mathtt{r}_2 \Downarrow (\sigma_1'',\ \sigma_2'')}{(\sigma_1,\ \sigma_2)\ \mathtt{r}_1 ; \mathtt{r}_2 \Downarrow (\sigma_1'',\ \sigma_2'')}\ \text{E-Seq}$$

$$\frac{\sigma_i,\ \mathtt{b}_i \Downarrow \mathtt{false}}{(\sigma_1,\ \sigma_2)\ \mathtt{while}_R\ \langle \mathtt{b}_1 \mid \mathtt{b}_2 \rangle\ \mathtt{r} \Downarrow (\sigma_1,\ \sigma_2)}\ \text{EWhileF}$$

$$\frac{\sigma_1,\ \mathtt{s}_1 \Downarrow \sigma_1' \quad \sigma_2,\ \mathtt{s}_2 \Downarrow \sigma_2'}{(\sigma_1,\ \sigma_2),\ \langle \mathtt{s}_1 \mid \mathtt{s}_2 \rangle \Downarrow (\sigma_1',\ \sigma_2')}\ \text{E-Align}$$

$$\frac{\sigma_i,\ \mathtt{b}_i \Downarrow \mathtt{false} \quad (\sigma_1,\sigma_2),\ \mathtt{r}_2 \Downarrow (\sigma_1',\sigma_2')}{(\sigma_1,\sigma_2),\ \mathtt{if}_R\ \langle \mathtt{b}_1 \mid \mathtt{b}_2 \rangle\ \mathtt{then}\ \mathtt{r}_1\ \mathtt{else}\ \mathtt{r}_2 \Downarrow (\sigma_1',\sigma_2')}\ \text{EIfF}$$

$$\frac{\sigma_1,\ \mathtt{b}_1 \Downarrow \mathtt{true} \quad \sigma_2,\ \mathtt{b}_2 \Downarrow \mathtt{true} \quad (\sigma_1,\sigma_2),\ \mathtt{r}_2 \Downarrow (\sigma_1',\sigma_2')}{(\sigma_1,\sigma_2),\ \mathtt{if}_R\ \langle \mathtt{b}_1 \mid \mathtt{b}_2 \rangle\ \mathtt{then}\ \mathtt{r}_1\ \mathtt{else}\ \mathtt{r}_2 \Downarrow (\sigma_1',\sigma_2')}\ \text{EIfT}$$

Fig. 11. Big-step semantics of CoreRel

Thus, the assertion $\mathtt{x}_1 > \mathtt{x}_2$ is satisfied by any pair of states σ and σ' in which σ maps x to a larger number than σ'.

While several specialized verification techniques for directly reasoning about relational safety have been proposed, including, e.g., *relational* program logics [Benton 2004; Maillard et al. 2019; Sousa and Dillig 2016], our goal here is to reduce a relational verification problem about a pair of Imp programs to logically equivalent claim involving a *single* intermediate Imp program. This claim can then be established using the program logic for Imp that we already have in hand. Of course, there are many such programs, some of which are more amenable to automated verification than others. As $\mathtt{double}_{1\times2}$ demonstrated, aligning similar control flow paths (e.g., loops) with each other helps a verifier to exploit similarities between the paths in order to simplify verification. Our strategy is to represent intermediate programs in a richer domain which explicitly *aligns* subcomponents of the original programs. This domain is equipped with relational variants of the control flow structures of the original program; intuitively, the relational variants group together control flow paths of the two programs.

Syntax. The syntax of aligned programs in CoreRel is given on the righthand side of Fig. 10. A basic alignment, $\langle\ \mathtt{c}_1 \mid \mathtt{c}_2\ \rangle$, consists of a pair of Imp programs $\mathtt{c}_1$ and $\mathtt{c}_2$ whose control flows are completely independent. In contrast, the relational control flow operators $\mathtt{while}_R$, $\mathtt{if}_R$, and $;$ align the control flows of their subexpressions. The branches of the relational conditional $\mathtt{if}_R$, for example, are themselves aligned programs. Fig. 10 also defines some additional notations which capture common alignment strategies. $\langle\mathtt{s}]$ and $[\mathtt{s}\rangle$ embed a single Imp program into the left and right sides of a relational representation, respectively. We also write $\mathtt{while}_{St}$ to denote 'stuttered' versions of aligned loops. Intuitively, the left and right hand loop bodies execute a different number of times at each loop iteration. The aligned expression $\mathtt{while}_{St}\ 2\ 1\ \langle \mathtt{b}_1 \mid \mathtt{b}_2 \rangle\ \langle \mathtt{c}_1 \mid \mathtt{c}_2 \rangle$, for example, represents a loop that executes $\mathtt{c}_1$ twice for every execution of $\mathtt{c}_2$.

Semantics. The big-step operational semantics of CoreRel are given by the rules shown in Fig. 11. As aligned programs are used to encode the behaviors of two programs, the reduction relation$(\sigma_1,\ \sigma_2)\ \mathtt{r} \Downarrow (\sigma_1',\ \sigma_2')$ states that the aligned command $\mathtt{r}$ takes a pair of disjoint initial states to a pair of disjoint output states. The basic alignment $\langle\ \mathtt{c}_1 \mid \mathtt{c}_2\ \rangle$ yields a pair of output states by combining the results of independently executing $\mathtt{c}_1$ and $\mathtt{c}_2$ (E-Align). Evaluating the aligned program $\langle \mathtt{x} := 2 \mid \mathtt{y} := 3 \rangle$, for example, results in a pair of final states where the value of $\mathtt{x}$ in the

first state is 2 and the value of y in the second state is 3; the value of y (x) is left unchanged in the first (second) state. Evaluating a relational loop $\texttt{while}_R\ \langle \texttt{b}_1 \mid \texttt{b}_2 \rangle\ \langle \texttt{c}_1 \mid \texttt{c}_2 \rangle$, in contrast, simulates a "lockstep" execution of a pair of loops by executing $\texttt{c}_1$ and $\texttt{c}_2$ in tandem (E-While$_R$T) until one of its conditions is falsified (E-While$_R$F).

Embedding and Reification. Any pair of Imp programs $\texttt{c}_1$ and $\texttt{c}_2$ can be embedded into CoreRel as the aligned program $\langle \texttt{c}_1 \mid \texttt{c}_2 \rangle$. Importantly, this embedding preserves the semantics of the original pair of programs:

Theorem 3.2 (Embedding is Sound). *The pair of Imp programs,* $\texttt{c}_1$ *and* $\texttt{c}_2$*, is semantically equivalent to their embedding in CoreRel,* $\langle \texttt{c}_1 \mid \texttt{c}_2 \rangle$*:*

$$\forall \sigma_1\ \sigma_2\ \sigma_1'\ \sigma_2'.\ \sigma_1,\ \texttt{c}_1 \Downarrow \sigma_1'\ \wedge\ \sigma_2,\ \texttt{c}_2 \Downarrow \sigma_2' \implies (\sigma_1,\ \sigma_2)\ \langle \texttt{c}_1 \mid \texttt{c}_2 \rangle \Downarrow (\sigma_1',\ \sigma_2')$$

More importantly, every CoreRel alignment can be *reified* back into a Imp program via the $[\![\cdot]\!]$ function shown in Fig. 12. This function uses a pair of renaming functions, $[\![\cdot]\!]_L$ and $[\![\cdot]\!]_R$, to ensure that variables that come from the lefthand side of the aligned program are distinct from those on the right. The control flow of a reified program mimics that of the aligned program that produced it. Each iteration of the reified version of an aligned loop $[\![\texttt{while}_R\ \langle \texttt{b}_1 \mid \texttt{b}_2 \rangle\ \langle \texttt{c}_1 \mid \texttt{c}_2 \rangle]\!]$ simulates lockstep evaluation of $[\![\texttt{c}_1]\!]$ and $[\![\texttt{c}_2]\!]$, for example, while reifying a pair of unaligned loops $[\![\langle \texttt{while b}_1\ \texttt{c}_1 \mid \texttt{while b}_2\ \texttt{c}_2 \rangle]\!]$ produces a program that fully evaluates $[\![\texttt{while b}_1\ \texttt{c}_1]\!]$ before $[\![\texttt{while b}_2\ \texttt{c}_2]\!]$, obscuring any intermediate relationships beween variables and $[\![\texttt{c}_1]\!]$ and $[\![\texttt{c}_2]\!]$.

$$\begin{aligned}
&[\![\langle \texttt{s}_1 \mid \texttt{s}_2 \rangle]\!] \triangleq [\![\texttt{s}_1]\!]_L;\ [\![\texttt{s}_2]\!]_R \\
&[\![\texttt{r}_1\ \texttt{;}\ \texttt{r}_2]\!] \triangleq [\![\texttt{r}_1]\!];\ [\![\texttt{r}_2]\!] \\
&[\![\texttt{while}_R\ \langle \texttt{b}_1 \mid \texttt{b}_2 \rangle\ \texttt{r}\ ]\!] \triangleq \\
&\quad \texttt{while}\ ([\![\texttt{b}_1]\!]_L\ \texttt{\&\&}\ [\![\texttt{b}_2]\!]_R)\ [\![\texttt{r}]\!] \\
&[\![\texttt{if}_R\ \langle \texttt{b}_1 \mid \texttt{b}_2 \rangle\ \texttt{then}\ \texttt{r}_1\ \texttt{else}\ \texttt{r}_2]\!] \triangleq \\
&\quad \texttt{if}\ ([\![\texttt{b}_1]\!]_L\ \texttt{\&\&}\ [\![\texttt{b}_2]\!]_R)\ \texttt{then}\ [\![\texttt{r}_1]\!]\ \texttt{else}\ [\texttt{r}_2]
\end{aligned}$$

Fig. 12. Reification of CoreRel into Imp

On their own, the reification and embedding functions enable us to reduce a relational safety problem about a pair of programs $\texttt{c}_1$ and $\texttt{c}_2$ to one involving a single program, i.e., $[\![\langle \texttt{c}_1 \mid \texttt{c}_2 \rangle]\!]$. When coupled with a notion of equivalence on aligned programs, however, they provide the formal foundation for defining a space of intermediate programs that are sufficient for relational reasoning.

Definition 3.3 (Alignment Equivalence). Two aligned programs are equivalent if they take the same pair of initial states to the same pair of final states:

$$\texttt{r}_1 \equiv \texttt{r}_2 \triangleq \forall \sigma_1\ \sigma_2\ \sigma_1'\ \sigma_2'.\ (\sigma_1,\ \sigma_2)\ \texttt{r}_1 \Downarrow (\sigma_1',\ \sigma_2') \Leftrightarrow (\sigma_1,\ \sigma_2)\ \texttt{r}_2 \Downarrow (\sigma_1',\ \sigma_2')$$

Reification is equivalence preserving, in that reifying equivalent aligned programs yields equivalent Imp programs:

Theorem 3.4 (Reification preserves Equivalence). *Any equivalent pair of aligned programs* $\texttt{r}_1$ *and* $\texttt{r}_2$ *represent equivalent intermediate programs,* $[\![\texttt{r}_1]\!]$ *and* $[\![\texttt{r}_2]\!]$*:*
$\texttt{r}_1 \equiv \texttt{r}_2 \implies \forall \sigma\ \sigma'.\ \sigma, [\![\texttt{r}_1]\!] \Downarrow \sigma' \Leftrightarrow \sigma, [\![\texttt{r}_2]\!] \Downarrow \sigma'$

A direct consequence of Theorems 3.2 and 3.4 is that we can reduce the relational verification problem to reasoning about an equivalent intermediate program:

Corollary 3.5. *Given a pair of Imp programs,* $\texttt{c}_1$ *and* $\texttt{c}_2$*, in order to prove that* $\texttt{c}_1$ *and* $\texttt{c}_2$ *are safe with respect to a pair of relational pre- and postconditions* Φ *and* Ψ*, it suffices to prove that an equivalent intermediate program* $\texttt{r}$ *is safe:* $\langle \texttt{c}_1 \mid \texttt{c}_2 \rangle \equiv \texttt{r} \wedge \vdash \{\Phi\}\ [\![\texttt{r}]\!]\ \{\Psi\} \implies \models_R \{\Phi\}\ \texttt{c}_1 \otimes \texttt{c}_2\ \{\Psi\}$

Unfortunately, this corollary does not provide any guidance on which $\texttt{r}$ to use. While equivalent aligned programs are *extensionally* equal, they may be *intensionally* different, in the sense that one may be more amenable to verification than the other. We now turn to the problem of how to automatically construct a good alignment.

4 The Alignment Algorithm

We first present our high-level algorithm before continuing to discussion of our realignment rules, data-driven extraction, and the details of our implementation. Our approach to constructing an aligned intermediate program is given in Algorithm 1. The algorithm takes as input two programs $\mathtt{p}_1$ and $\mathtt{p}_2$, a `Cost` function over candidate alignments, and a parameter μ bounding the number of candidates our data-driven extraction phase should consider. The programs $\mathtt{p}_1$ and $\mathtt{p}_2$ are assumed to have disjoint variable namespaces, which can be easily accomplished through an automated α-renaming pass. Any time the algorithm needs to compute the `Cost` of a candidate alignment, it first collects execution traces for that candidate over a set of randomly generated test inputs. (This set of test inputs does not change between successive invocations of `Cost`.) Traces are collected and scored as described in Section 4.3.

Algorithm 1: KestRel

Inputs : p_1 and p_2: programs,
`Cost`: cost metric for alignments,
μ: number of SA iterations
Output : intermediate program $p_1 \times p_2$

```
1  E ← CreateEGraph()
2  Insert(E, ⟨p1|p2⟩)
3  EQSat(E, CoreRel)
4  best ← ExtractLocal(E)
5  η̂ ← Cost(best)
6  Σ ← RandomStartStates(⟨p1|p2⟩)
7  for k ← 0 to μ do
8      τ ← Temperature(k, μ)
9      N ← RandomNeighbor(E, best)
10     T ← Evaluate(Reify(N), Σ)
11     η ← Cost(N, T)
12     if η < η̂ ∨ Jump(τ, best, η̂, N, η) then
13         (best, η̂) ← (N, η)
14 return Reify(best)
```

Algorithm 1 procedes as follows: lines 1 — 2 create a new e-graph from the initial embedding of the input programs, $\langle p_1 | p_2 \rangle$. Line 3 then applies equality saturation (Section 4.2) using a collection of CoreRel realignment rules (Section 4.1) to construct the set of candidate alignments. Next, an initial term is extracted using a cost function which minimizes the number of unfused loops in each e-class (Line 4) and that term's `Cost` is computed (Line 5). Line 6 generates a set of random relational start states for the term $\langle p_1 | p_2 \rangle$ which will be used to collect execution traces. The algorithm then proceeds to the data-driven extraction phase, which uses a simulated annealing loop (Lines 7–13) that queries the saturated e-graph to provide candidate alignments by perturbing the selection of representative nodes for the e-classes in the current alignment (Section 4.3).

4.1 CoreRel Rewrite Rules

Observing that program equivalence is a congruence relation, we frame the search for a good intermediate program as rewriting problem in which we attempt to *realign* the naïve embedding of a pair of programs into a form more amenable for automated verification. Fig. 13 provides several equivalences that we can use to explore the space of possible alignments. Our notion of equivalence naturally admits any equivalences on non-relational Imp programs; for example, it is sound to unroll one iteration of a loop on one side of an aligned term (unroll-l). More interestingly, the richer structure of CoreRel programs also includes a set of rules that allow us to realign terms. The first three rules (rel-def, hom-l, and hom-r) allow us to de- and re-compose subprograms into different alignments, while the rel-assoc rule reassociates relational sequences of statements, and the rel-comm rule leverages the fact that the left- and right-hand programs operate over distinct state spaces to rearrange two embedded programs. Observe that the alignments on the two sides of rel-comm reify into different intermediate programs: $[\![\langle \mathtt{c}_1]\!]; [\mathtt{c}_2 \rangle]\!] \coloneqq \mathtt{c}_1; \mathtt{c}_2$, while $[[\![\mathtt{c}_2 \rangle]\!]; \langle \mathtt{c}_1]\!]] \coloneqq \mathtt{c}_2; \mathtt{c}_1$. A similar rule over sequences of Imp commands $\mathtt{c}_1; \mathtt{c}_2 \equiv \mathtt{c}_2; \mathtt{c}_1$ is obviously incorrect in the general case, as $\mathtt{c}_1$ and $\mathtt{c}_2$ may modify the same variables.

$$\langle c_1 \mid c_2\rangle \equiv \langle c_1\rangle ; [c_2] \quad \text{REL-DEF} \qquad \langle \texttt{while b c} \rangle \equiv \langle \texttt{if b then c; while b c}\rangle \quad \text{UNROLL-L}$$

$$\langle c_1; c_2\rangle \equiv \langle c_1\rangle ; \langle c_2\rangle \quad \text{HOM-L} \qquad \langle c_1\rangle ; [c_2] \equiv [c_2] ; \langle c_1\rangle \quad \text{REL-COMM}$$

$$[c_1; c_2\rangle \equiv [c_1\rangle ; [c_2\rangle \quad \text{HOM-R} \qquad r_1 ; (r_2 ; r3) \equiv (r_1 ; r_2) ; r3 \quad \text{REL-ASSOC}$$

$$\langle \texttt{while } b_1\ c_1 \mid \texttt{while } b_2\ c_2\rangle \equiv \texttt{while}_{St}\ \texttt{n m}\ \langle b_1 \mid b_2\rangle\ \langle c_1 \mid c_2\rangle ; \langle\ \texttt{while } b_1\ c_1\ ]; [\ \texttt{while } b_2\ c_2\rangle \quad \text{WHILE-ALIGN}$$

$$\langle \texttt{if } b_1 \texttt{ then } c_1 \texttt{ else } c_2 \mid \texttt{if } b_2 \texttt{ then } c_3 \texttt{ else } c_4\rangle \equiv \texttt{if}_R\ \langle b_1 \mid b_2\rangle \texttt{ then } \langle c_1 \mid c_3\rangle \texttt{ else if}_R\ \langle b_1 \mid \texttt{not } b_2\rangle \texttt{ then } \langle c_1 \mid c_4\rangle \texttt{ else if}_R\ \langle \texttt{not } b_1 \mid b_2\rangle \texttt{ then } \langle c_2 \mid c_3\rangle \texttt{ else } \langle c_2 \mid c_4\rangle \quad \text{IF-ALIGN}$$

$$\texttt{while}_R\ \langle b_1 \mid b_2\rangle\ r \equiv \texttt{if}_R\ \langle b_1 \mid b_2\rangle \texttt{ then } r \texttt{ else } \langle \texttt{skip} \mid \texttt{skip}\rangle ; \texttt{while}_R\ \langle b_1 \mid b_2\rangle\ r \quad \text{UNROLL-BOTH}$$

$$\langle \texttt{if } b_1 \texttt{ then } c_1 \texttt{ else } c_2 \mid c_3\rangle \equiv \texttt{if}_R\ \langle b_1 \mid \texttt{true}\rangle \texttt{ then } \langle c_1 \mid c_3\rangle \texttt{ else } \langle c_2 \mid c_3\rangle \quad \text{COND-L}$$

$$\langle c_1 \mid \texttt{if } b_1 \texttt{ then } c_2 \texttt{ else } c_3\rangle \equiv \texttt{if}_R\ \langle \texttt{true} \mid b_1\ \rangle \texttt{ then } \langle c_1 \mid c_2\rangle \texttt{ else } \langle c_1 \mid c_3\rangle \quad \text{COND-R}$$

Fig. 13. Selected CoreRel realignment laws

The WHILE-ALIGN rule is particularly important, as it merge two loops together so that their bodies execute in lockstep. Note that since $\texttt{while}_{St}$ terminates as soon as either condition is false, WHILE-ALIGN must add trailing "runoff" `while` loops after the joint loop in order for this equivalence to hold. In the case that the original loops always have the same number of iterations, these loops will never execute. A similar argument explains the IF-ALIGN rule.

4.2 Realignment via Equality Saturation

Using CoreRel as the underlying language, e-graphs can be used to compactly represent a (potentially infinite) number of program realignments. By Theorem 3.4, extractions from an e-graph saturated with sound CoreRel rewriting rules like those given in Fig. 13 are semantically equivalent to the naïve alignment *by construction*.

To build an e-graph representing a space of potential alignments of programs p_1 and p_2, we start by constructing an e-graph that contains the naïve alignment term $\langle p_1 \mid p_2\rangle$. For example, given the CoreRel term $\langle$`i := 3; while (i > 0) { i--; } | j := 3; while (j > 0) { j --; }`$\rangle$, Fig. 14(a) depicts the initial e-graph. We then run equality saturation on the e-graph using CoreRel rewrite rules like those listed in Fig. 13.

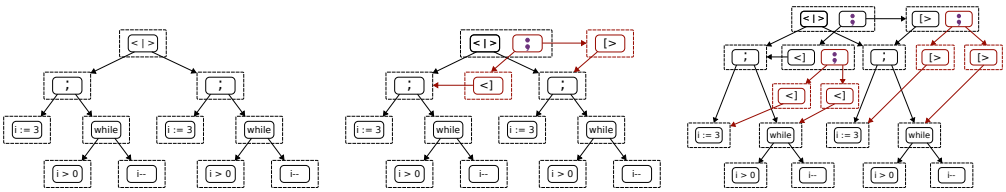


Fig. 14. E-graphs representing the space of alignments that result from applying the rewrite rules from Fig. 13 to the aligned term $\langle$`i := 3; while (i > 0) { i--; } | j := 3; while (j > 0) { j --; }`$\rangle$. The leftmost e-graph (a) represents this initial alignment. The middle e-graph (b) additionally includes the alignment that results from applying the REL-DEF rule. The last e-graph (c) includes additional alignments that result from applying both the HOM-L and HOM-R rules. The additional nodes that result from each rule are highlighted in red. Some e-nodes (<, :=, and --) have been combined into a single node for brevity.

Example 4.1. As an example, the root node of Fig. 14(a) matches the left-hand side of the REL-DEF rule from Fig. 13, with c_1 and c_2 corresponding to the root left and right e-classes, respectively. Fig. 14(b) depicts the e-graph that results from applying this rewrite. Observe that there is a new node in the root e-class corresponding to the right-hand side of REL-DEF. We now have a choice when extracting a term from this e-graph; if we choose the ⟨ | ⟩ node in the root e-class, we get the original term. If we instead choose the ⨟ node, we get ⟨`i := 3; while (i > 0) { i--; }`⟩ ⨟ ⟨`j := 3; while (j > 0) { j--; }`⟩, i.e. the original term rewritten according to the REL-DEF rule.

Performing additional rewrites to this e-graph will further grow the set of equivalent programs it represents. Fig. 14(c) depicts the e-graph that results from applying HOM-L and HOM-R. Included in the elements of this e-graph is a fully decomposed version of the original alignment: ⟨`i := 3`⟩ ⨟ ⟨`while (i > 0) { i--; }`⟩ ⨟ ⟨`j := 3`⟩ ⨟ ⟨`while (j > 0) { j--; }`⟩. Further applications of the REL-COMM, WHILE-ALIGN, and REL-DEF rules eventually yield an e-graph that includes the (likely) desired alignment:

⟨`i := 3; j := 3`⟩ ⨟
`while`$_R$ ⟨`i > 0 | j > 0`⟩ ⟨`i-- | j-- `⟩ ⨟ ⟨`while (i > 0) { i--; } | while (j > 0) { j--; }`⟩

4.3 Data-Driven Extraction

After we have built an e-graph representation of the space of possible alignments, we still need to *extract* a desirable relational program which can be reified and handed off to a program verifier. Before we present our approach to extraction, however, we first need to define what constitutes a "good" alignment. The ultimate answer is that any alignment that produces a intermediate program that can be automatically verified is good, and an alignment that does not is bad. Verification is too expensive to use as a measure of the quality of a candidate alignment, so we require an alternative metric. One immediate solution is to define a cost function that uses syntactic features to identify good alignments. While such a syntactic approach allows programs to be quickly extracted, a purely syntactic measure fails to capture important semantic properties of an alignment. For example, if the "runoff" loops generated by an application of WHILE-ALIGN never execute, it suggests a semantic similarity between the loops, as both loop conditions became false at the same time. However, this semantic property is not obvious from the syntax of an alignment alone. Our solution is to combine a syntactic extraction strategy with a *data-driven* approach [Ernst et al. 2007; Padhi et al. 2016; Sharma et al. 2013; Zhu et al. 2016] that examines concrete executions of a candidate alignment to approximate its *semantic* fitness.

4.3.1 Traces. The data-driven component of our extraction mechanism executes candidate alignments in order to gather a set of *traces*, sequences of intermediate states that summarizes the semantic behaviors of an alignment. The extraction then applies a *cost* function to each set of traces in order to compare the relative quality of each alignment. While there are many aspects of an execution trace which may have bearing on the quality of alignment, enabling straightforward loop invariants is usually the ultimate goal, and so we focus on loop executions as the most effective measure of trace quality. We therefore construct traces of loop head and end tags:

(1) wB_R, wH_R, and wE_R occur, respectively, at the entry, beginning of each iteration, and exit of each relational loop (`while`$_R$),
(2) wB_O, wH_O, and wE_O occur in similarly for runoff loops generated by WHILE-ALIGN, and
(3) wB, wH, and wE do the same for standard IMP loops (`while`).

Example 4.2. The program on the left emits the trace on the right when executed with a pair of empty initial states:

```
⟨y1 := 0; z1 := 2 * x1 | y2 := 0; z2 := x2⟩;
  while_R ⟨z1 > 0 | z2 > 0⟩ ⟨z1−−; y1 := y1 + x1 | z2−−; y1 = y1 + x1 ⟩;
  ⟨while (z1 > 0) {z1−−; y1 := y1 + x1} | while (z2 > 0) {z2−−; y1 = y1 + x1} ⟩;
  [y2 := 2 * y2⟩
```

Fig. 16. A suboptimal alignment of $\texttt{double}_1$ and $\texttt{double}_2$ from Section 2.

```
⟨i1 := 3 | i2 := 2⟩;
while_R ⟨i1 > 0 | i2 > 0⟩ ⟨i1−−;| i2−−;⟩;
⟨while (i1 > 0) i1++];
[while (i2 > 0) i2−− ⟩
```

(a) CoreRel program.

$\texttt{wB}_R$	– entering the $\texttt{while}_R$ loop
$\texttt{wH}_R, \texttt{wH}_R$	– two iterations of $\texttt{while}_R$
$\texttt{wE}_R,$	– exiting $\texttt{while}_R$
$\texttt{wB}_O, \texttt{wH}_O, \texttt{wE}_O$	– execution of left runoff `while` loop

(b) Corresponding trace.

Example 4.3. `⟨i := 0 | j := 2; while (j > 0) { j−−; }⟩` emits the following trace: wB, wH, wH, wE

4.3.2 Comparing Alignments. Before describing our particular cost function over traces, we first discuss what a desirable trace looks like, using the traces that are generated by the different alignments of $\texttt{double}_1$ and $\texttt{double}_2$ from Section 2. On one hand, we have the initial embedding of these programs, $\langle\texttt{double}_1|\texttt{double}_2\rangle$, and on the other hand we have the target alignment corresponding to $\texttt{double}_{1\times2}$. Consider what features of the traces generated by $\texttt{double}_{1\times2}$ indicate that it should be preferred over $\langle\texttt{double}_1|\texttt{double}_2\rangle$. An immediate difference is that $\texttt{double}_{1\times2}$ includes a relational loop, $\texttt{while}_R$, which manifests in its execution traces as a sequence of $\texttt{wH}_R$'s followed by a $\texttt{wE}_R$. In contrast, the trace of $\langle\texttt{double}_1|\texttt{double}_2\rangle$ contains *only* non-relational wH's. This suggests a straightforward heuristic of preferring traces with more relational loop tags. However, consider the suboptimal alignment shown in Fig. 16. While this is close to $\texttt{double}_{1\times2}$ in that it combines loop bodies in a single $\texttt{while}_R$, it does not properly stutter the body of the relational loop using $\texttt{while}_{St}$. This manifests in a trace that includes several $\texttt{wH}_O$'s that are generated by the lefthand runoff loop after the relational loop has ended ($\texttt{wE}_R$), suggesting another straightforward strategy of favoring traces with fewer runoff loop executions.

Based on these observations, we define a cost function for a trace T that has the following two components, where $\#(\texttt{tag}, T)$ is the number of occurrences of tag in T:

(1) The ratio of unmerged to merged loop executions:

$$r_{unmerged}(T) = \#(\texttt{wB}, T)/(\#(\texttt{wB}_R, T) + \#(\texttt{wB}, T))$$

(2) The ratio of runoff to non-runoff loop executions:

$$r_{runoff}(T) = \#(\texttt{wH}_O, T)/(\#(\texttt{wH}_R, T) + \#(\texttt{wH}_R, T) + \#(\texttt{wH}, T))$$

The overall cost of a trace is then calculated as $\texttt{cost}(T) = 0.5 * r_{unmerged}(T) + 0.5 * r_{runoff}(T)$.

4.3.3 Neighboring Alignments. Our data-driven extraction loop uses the `RandomNeighbor` function to find an alignment "near" the current one. Our implementation of `RandomNeighbor` uses an e-graph saturated with CoreRel rewrite rules to locate neighbors as follows:

(1) Given a CoreRel term, find a set of e-classes and choices of e-nodes within those classes that represents the term. If cycles exist in the e-graph, the same e-class may appear multiple times, potentially with a different chosen e-node each time.
(2) Pick one of the e-classes within this set which has more than one e-node.
(3) Select a different e-node from this e-class. Assign children to this e-node, re-using children of the previous choice when possible and randomly choosing children otherwise.
(4) Construct a new CoreRel term from this modified set of e-node selections.

This approach assumes that e-nodes in the same e-class represent alignments which are sufficiently similar to be considered "neighbors". This assumption deserves some caveats, however, as e-nodes in the same e-class may represent terms an arbitrary number of rewrites away from each other. Identifying simultaneous rewrites which produce new alignments is a powerful aspect of what an e-graph representation provides us, but we must take care to examine reasonable neighborhoods of transformations to keep the search tractable. In practice, this becomes an issue when loop unrolling and scheduling rules place e-nodes representing arbitrary numbers of unrollings in the same e-class. To combat this, our implementation of `RandomNeighbor` only considers a fixed number of unrollings one away from the input term. Additionally, randomly constructing children of new selections in cases where the modified choice does not share a child e-class with the original choice can lead to arbitrarily large subterms. Instead of building truly random terms in these cases, KestRel biases towards terms with small ASTs. In practice, we have found that these choices identify good neighbors for our suite of benchmarks.

4.4 Implementation

We have implemented a relational verification engine based on Algorithm 1, called KestRel. KestRel is written in Rust and uses the Egg library [Willsey et al. 2021] to represent spaces of candidate alignments as e-graphs. Internally, KestRel operates over a superset of CoreRel, but it is equipped with a frontend that accepts a subset of C (it does not support `for` loops or `structs`, for example) and backends for outputting intermediate programs in Dafny and C (the latter is used to target SeaHorn). Our implementation of KestRel hands off the initial alignment found by syntactic extraction (Line 5) to a verifier; if this program successfully verifies, KestRel halts and reports its success. Otherwise, it proceeds to its data-driven extraction phase, and the result of verifying the intermediate program produced by this phase is reported to the user.

Equality Saturation Optimizations. Basic blocks whose internal realignment cannot impact the verifiability of the intermediate program are encoded using a distinguished `basic-block` structure to which the HOM-L and HOM-R rules do not apply. This avoids unnecessarily polluting the search space with useless permutations of realigned straightline code.

Instrumentation. In order to generate traces during its data-driven extraction phase, KestRel produces instrumented programs that are augmented with `log` commands that produce traces. To generate traces from an instrumented program, KestRel randomly generates a set of starting states which meet the verification problem's relational precondition using test input generators in the style of property based testing frameworks [Claessen and Hughes 2000].

Reification. While the work of finding alignments is carried out in the language of CoreRel, KestRel translates CoreRel alignments into intermediate programs annotated with `assume` and `assert` statements. These intermediate programs can be given directly to off-the-shelf verifiers. Currently KestRel has backends for C, targeting SeaHorn, and Dafny.

Invariant Inference. While KestRel tries to find aligned intermediate programs that have simple loop invariants, not every verification tool, e.g., Dafny, implements automated invariant inference. Thus, we have implemented an

Algorithm 2: HouDafny

Inputs : p: aligned intermediate program
Output : loop invariant annotations

```
1 I ← Daikon(p) ∪ BaseInv(p)
2 p′ ← Annotate(p, I)
3 X ← Dafny(p′)
4 while X ≠ ∅ do
5     I ← I \ X
6     p′ ← Annotate(p, I)
7     X ← Dafny(p′)
8 return p′
```

invariant inference algorithm, shown in Algorithm 2, for KestRel's Dafny backend. This algorithm implements a Houdini-style [Flanagan and Leino 2001] iterative refinement invariant inference procedure. Starting from a pool of candidate invariants that combines invariants suggested by Daikon [Ernst et al. 2007] with a built-in set of candidate invariants (`BaseInv`), the algorithm iteratively tries to verify the program using Dafny, collecting and removing non-invariant clauses from the set of candidates each iteration until a fixpoint is reached. The algorithm showcases one of the strengths of using intermediate programs for relational verification, as it leverages an existing, non-relational verification tool, Daikon.

5 Evaluation

Our experimental evaluation investigates five key questions regarding our approach to building aligned intermediate programs:

RQ1 Is our approach *effective*, i.e., does KestRel construct intermediate programs that enable verification tools to be used for relational reasoning?
RQ2 Is our approach *expressive* enough to cover a diverse set of programs and relational properties?
RQ3 Is our approach *efficient*? Is KestRel able to find useful intermediate programs within a reasonable time frame?
RQ4 How much does each component of KestRel contribute to its effectiveness?
RQ5 Is our approach *general*? Can KestRel build intermediate programs suitable for multiple backend verifiers?

5.1 Benchmark Construction (RQ2)

To answer these questions, we evaluate KestRel on a diverse corpus of benchmarks[3] drawn from the relational verification literature, and that includes examples from both the intermediate program-based and bespoke approaches [Antonopoulos et al. 2023; Barthe et al. 2011a; Churchill et al. 2019; Sousa and Dillig 2016; Unno et al. 2021]. Our benchmark suite includes clients of a variety of abstract data types (ADTs), including key value stores, lists, binary search trees, and 2-3 trees (**RQ2**). Our evaluation considers several categories of relational properties (**RQ2**), including:

- **Equivalence:** Two programs exhibit equivalent behaviors, for example always returning the same value given the same inputs.
- **Anticommutativity:** Swapping the arguments of a function inverts its result: `compare(a, b) = !compare(b, a)`, for example.
- **Monotonicity:** Under certain conditions, one program always yield a result greater than (or less than) another.
- **Noninterference:** An information security property that requires observable ("low") outputs of multiple executions to independent of any secret ("high") inputs.

All benchmarks were run on ArchLinux with an 8 core Intel i7-6700K 4GHz CPU and 16 GB RAM.

5.2 Quality of Aligned Intermediate Programs (RQ1, RQ3)

Our first set of experiments addresses our alignment strategy's ability to produce aligned intermediate programs that can be effectively and efficiently verified by an existing, non-relational program verifier, i.e., KestRel's Dafny backend (**RQ1, RQ3**). We compare KestRel against two baselines: a **Naïve** strategy which α-renames variables and concatenates the input programs together (see Section 2), and a **Syntactic** strategy which greedily aligns the loops in a program.[4]

[3] All the benchmarks and results from our evaluation are provided in the anonymized supplementary material.
[4] In practice, the syntactic strategy is equivalent to only using the first component of KestRel's extraction mechanism.

Table 1. Verification results for different alignment strategies. Verification failures and successes are marked with ✖ and ✔, respectively. The total time to construct and verify intermediate programs are given in seconds; ✖ ∞indicates that a five minute timeout was exceeded. The upper and bottom tables present the results for programs with only basic types and ADTs, respectively. Benchmark names are annotated with their source: Antonopoulos et al. [2023] (†), Barthe et al. [2011a] (★), Sousa and Dillig [2016](◦), Shemer et al. [2019] (◇), Cormen et al. [2009] (□), and Churchill et al. [2019](‡). For benchmarks with simple types, the **Loops** column indicates the presence of a loop in the benchmark. For clients of ADTs, the **ADTs** column lists the ADTs used– all these benchmarks had loops. The **Property** column gives the relational property being verified. Results in the **Naïve** column are for a concatenative alignment strategy, the **KestRel (Syntactic)** column presents results when only the syntactic component of Kestrel's extraction mechanism, while the **KestRel (Full)** column uses both components.

Benchmark	Loops	Property	Naïve	KestRel (Syntactic)	KestRel (Full)
commute	✔	commutativity	✖ 6.90	✔ 4.92	✔ 4.88
data-alignment†	✔	monotonicity	✖ 8.09	✖ 8.46	✖ 25.50
double-square◇	✔	equivalence	✖ ∞	✖ 21.32	✔ 36.22
half-square◇	✔	noninterference	✖ 5.04	✔ 9.06	✔ 12.90
payments	✔	equivalence	✖ 9.12	✔ 7.22	✔ 7.21
shemer◇	✔	equivalence	✖ ∞	✖ 23.05	✔ 46.50
simple†	✔	equivalence	✖ 4.35	✔ 2.60	✔ 2.61
strength-reduction★	✔	equivalence	✖ 8.10	✔ 6.29	✔ 6.31
square-sum◇	✔	equivalence	✖ 11.84	✔ 4.60	✔ 4.60
unroll★	✔	equivalence	✖ 6.26	✖ 19.10	✔ 21.77
col-item◦	✖	anticommutativity	✔ 2.56	✔ 2.63	✔ 2.59
container◦	✖	anticommutativity	✔ 2.58	✔ 2.69	✔ 2.68
file-item◦	✖	anticommutativity	✔ 2.52	✔ 2.56	✔ 2.48
match◦	✖	anticommutativity	✔ 2.59	✔ 2.56	✔ 2.53
node◦	✖	anticommutativity	✔ 2.45	✔ 2.56	✔ 2.65

Benchmark	ADTs	Property	Naïve	KestRel (Syntactic)	KestRel (Full)
array-insert†	kvstore	equivalence	✖ 13.48	✔ 18.67	✔ 14.47
array-int◦	kvstore	anticommutativity	✖ 14.44	✔ 17.24	✔ 17.14
bst-min-search□	bst	monotonicity	✖ 6.61	✔ 4.54	✔ 4.57
bst-sum□	bst	monotonicity	✖ 7.12	✔ 6.99	✔ 7.09
bubble-sort★	kvstore	robustness	✖ 20.41	✔ 19.97	✔ 17.18
chromosome◦	kvstore	anticommutativity	✖ 14.02	✔ 20.14	✔ 19.90
code-sinking★	kvstore	equivalence	✖ 11.44	✔ 11.65	✔ 11.49
flatten‡	kvstore	equivalence	✖ 132.40	✔ 153.52	✔ 10.07
linked-list-ni	list	noninterference	✖ 10.71	✔ 28.48	✔ 28.17
list-array-sum□	kvstore, list	equivalence	✖ 7.04	✔ 5.10	✔ 4.96
list-length□	list	equivalence	✖ 4.34	✔ 3.35	✔ 3.23
loop-alignment★	kvstore	equivalence	✖ 11.50	✖ 17.57	✔ 36.68
loop-pipelining★	kvstore	equivalence	✖ 10.87	✖ 17.56	✔ 56.08
loop-tiling†	kvstore	equivalence	✖ ∞	✖ 15.32	✖ 83.76
loop-unswitching★	kvstore	equivalence	✖ 8.35	✔ 5.86	✔ 5.96
static-caching★	kvstore	equivalence	✖ 21.63	✖ 2.42	✖ 185.22

The results of these experiments are presented in Table 1. The experiments are grouped into two tables: the first is comprised of benchmarks that only require basic datatypes, while the second consists of benchmarks that use ADTs. The first set is further subdivided into benchmarks with and without loops (all of our ADT benchmarks contain loops). As the set of paths through loop-free programs is finite, we expect alignment to be unnecessary for verification in these cases. Indeed, Dafny is able to verify all alignments for all five loop-free benchmarks. Nevertheless these

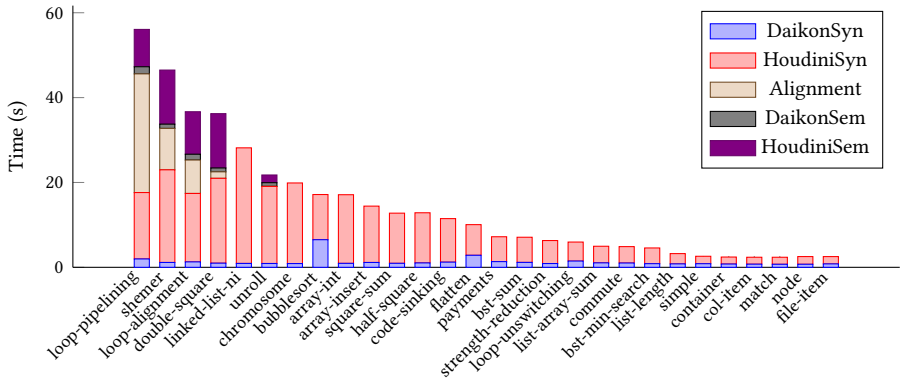


Fig. 17. Breakdown of KestRel runtimes by subtask. "DaikonSyn" refers to generating initial invariant candidates for syntactic extraction, "HoudiniSyn" refers to elimination of invalid invariant candidates for syntactic extractions, and "Alignment" refers to semantic extraction of an intermediate program from the e-graph. "DaikonSem" and "HoudiniSem" refer to the analogous invariant inference tasks over semantic extractions. Subtasks which take negligible amounts of time (for example, extracting a purely syntactic alignment from the e-graph) are not depicted.

benchmarks show that the alignments computed by KestRel do not adversely affect verifiability in cases where alignments are not strictly necessary.

Dafny failed to verify the naïve alignments of the remaining 26 benchmarks, suggesting that combining control flow is beneficial. For all but three of these benchmarks, Dafny was able to verify the aligned intermediate programs produced by KestRel (**RQ1**). Of these verified benchmarks, five did not verify with a syntactic extraction, suggesting a need for semantic methods. In addition, verification was reasonably efficient, finishing in under 30 seconds in most cases (**RQ3**). Fig. 17 presents per-subtask timings for the individual components of the KestRel pipeline. For most of these benchmarks, invariant inference dominates the total runtime; see discussion below.

Two of the three benchmarks that our pipeline fails to verify require the insertion of sophisticated guards inside the loops of the aligned program, transformations that are not currently supported by KestRel. The `data-alignment` benchmark must skip executing certain loop iterations (for example, when the loop index `mod` 3 is zero), and `loop-tiling` requires the creation of new inner loops which subdivide iterations at certain tile sizes. The remaining failure case, `static-caching`, requires complex loop invariants; a stronger invariant inference engine could enable the alignments produced by KestRel to be automatically verified.

5.2.1 Invariant Inference. As described in Section 4.4, the Dafny backend of KestRel performs a Houdini-style [Flanagan and Leino 2001] invariant inference, drawing from a set of predicates generated by Daikon [Ernst et al. 2007] and provided by users. This approach requires multiple verification attempts per benchmark to locate non-invariant clauses. Our pipeline was able to verify 16 of these benchmarks using only the predicates suggested by Daikon. Cases which required additional hints ranged from invariants with simple equalities between, e.g., loop indices which Daikon did not discover, invariants which simply carry forward axiomitizations of ADT methods, or invariants involving more complex relationships requiring insights into how these ADT axioms interact. Augmenting Daikon with a more robust hypothesis space or otherwise using invariant inference methods which specifically target theories of these ADTs has the potential to improve the success rate without user hints.

5.3 Contribution of Each Component (RQ4)

To demonstrate the utility of each of the components in KestRel's extraction procedure, we have conducted a pair of ablation studies to evaluate their individual impact on the quality of aligned programs. The first experiment considers the effectiveness of KestRel's combined approach to program extraction. This experiment uses the benchmarks in Table 1 that contain loops and which our Dafny backend can automatically verify. We gave each of these benchmarks to two modified versions of KestRel. The first performs only local extraction ("Syntactic"), while the second performs *only* the data driven phase, starting from a naïve concatenative alignment ("Semantic"). Fig. 18 presents the results of this experiment.

Benchmark	Syntactic	Semantic	Combined
array-insert	✔ 16.54	✔ 34.76	✔ 14.47
array-int	✔ 17,35	✔ 34.14	✔ 17.14
bst-min-search	✔ 4.78	✖ 16.52	✔ 4.57
bst-sum	✔ 6.88	✔ 18.53	✔ 7.09
bubble-sort	✔ 18.70	✖ 97.14	✔ 17.18
chromosome	✔ 19.93	✔ 31.45	✔ 19.90
code-sinking	✔ 9.30	✔ 12.97	✔ 11.49
col-item	✔ 2.63	✖ ∞	✔ 2.59
commute	✔ 4.91	✔ 15.19	✔ 4.88
container	✔ 2.73	✖ ∞	✔ 2.68
double-square	✖ 21.03	✔ 23.51	✔ 36.22
flatten	✔ 9.22	✔ 38.11	✔ 10.07
file-item	✔ 2.55	✔ 57.15	✔ 2.48
half-square	✔ 13.10	✔ 10.57	✔ 12.90
linked-list-ni	✔ 29.01	✖ 35.19	✔ 28.17
list-array-sum	✔ 5.12	✔ 16.93	✔ 4.96
list-length	✔ 3.37	✖ ∞	✔ 3.23
loop-alignment	✖ 19.41	✔ 31.15	✔ 36.68
loop-pipelining	✖ 15.65	✔ 37.89	✔ 56.08
loop-unswitching	✔ 6.03	✔ 69.28	✔ 5.97
match	✔ 2.71	✔ 57.90	✔ 2.53
node	✔ 2.61	✔ 10.49	✔ 2.65
payments	✔ 7.36	✔ 10.59	✔ 7.21
shemer	✖ 23.21	✖ 40.17	✔ 46.50
simple	✔ 2.56	✔ 2.63	✔ 2.61
strength-reduction	✔ 6.28	✔ 5.84	✔ 6.31
square-sum	✔ 4.50	✔ 12.74	✔ 4.60
unroll	✖ 19.15	✔ 2.75	✔ 21.77

Fig. 18. Results of an ablation study over benchmarks with successful KestRel verifications. The "Syntactic" column lists verification results using just local extraction. The "Semantic" column gives results for programs constructed via data-driven extractions starting from a naïve concatenation and using a maximum 12000 iterations. The "Combined" column contains verification results for the default KestRel workflow. All times are shown in seconds.

In most cases, our syntactic extraction technique, which minimizes the total number of loops (fewer loops likely means more fused loops) is sufficient for verification. In some cases (e.g., `bst-min-search`, `linked-list-ni`), this approach succeeded where data-driven simulated annealing failed; the likely cause is a large alignment space which causes the MCMC search to converge too slowly. In several cases (e.g. `file-item`, `match`), the programs produced by both approaches were able to verify, but simulated annealing took much longer than the syntactic approach. Taken together, these points indicate that using a purely syntactic extraction is an effective starting point for KestRel's simulated annealing approach. For one of these benchmarks (`shemer`), neither alignment strategy identified a verifiable alignment on its own, demonstrating the benefits of KestRel's combined approach.

We next evaluated the effectiveness of the cost function used by KestRel's simulated annealing loop by constructing two variants of KestRel: the first uses a cost function that only considers the number of fused loops as a proportion of all loops, and the second only considers the number of runoff loop iterations as a proportion of all loop iterations. We ran an ablation study which examined KestRel's performance over the five benchmarks from Table 1 that required semantic extraction.

Benchmark	Runoff	Fusion	Combined
double-square	✖ 31.84	✖ 52.38	✔ 44.52
loop-alignment	✖ 33.22	✖ 57.58	✔ 45.82
loop-pipelining	✖ 36.89	✖ 66.89	✔ 53.50
shemer	✖ 55.17	✖ 58.30	✔ 45.24
unroll	✖ 25.54	✖ 49.02	✔ 21.80

Fig. 19. Results of an ablation study of the KestRel cost function across the five benchmarks which required semantic extraction. The **Runoff** column considers only number of runoff loop iterations, the **Fusion** column considers only fused loops, and the **Combined** column considers both. All times reported in seconds.

Fig. 19 presents the results from this experiment: in all cases, the combined cost function generated verifiable alignments where the individual cost function components alone failed, suggesting that both components of the cost function are useful in discovering good alignments.

5.4 Relational Verification with SeaHorn (RQ5)

To demonstrate that KestRel's ability to find good intermediate programs is not tied to a particular verifier (**RQ5**), we translated a subset of the benchmarks from Table 1 into array-manipulating C programs and then verified them using KestRel's SeaHorn backend. Fig. 20 reports the verification times for naïve aligned programs, alignments produced by just the local cost function ("Syntactic"), and the alignments produced by KestRel's default workflow ("Combined"). Our results are grouped into three categories, shown at the the top, middle, and bottom of Fig. 20. The top and middle groups comprise benchmarks where SeaHorn was able to verify the intermediate programs produced by KestRel, and the top group includes the cases where SeaHorn was not able to verify the naïve alignment. For two of these benchmarks, `schemer` and `simple`, SeaHorn failed to verify the program found by syntactic methods, while our data-driven approach was able to find intermediate programs that successfully verified. Taken together, these results provide evidence that our approach can support multiple verification backends (**RQ4**).

Benchmark	Naïve	KestRel (Syntactic)	KestRel (Full)
double-square	✖ 0.16	✖ 0.19	✔ 1.81
shemer	✖ 0.19	✔ 0.18	✔ 2.05
simple	✖ ∞	✔ 0.28	✔ 1.68
unroll	✖ ∞	✖ ∞	✔ 1.58
array-insert	✔ 3.52	✔ 14.32	✔ 19.56
array-int	✔ 0.17	✔ 0.18	✔ 5.35
bubble-sort	✔ 0.14	✔ 0.22	✔ 16.26
chromosome	✔ 0.15	✔ 0.16	✔ 3.57
col-item	✔ 0.13	✔ 0.19	✔ 2.51
container	✔ 0.16	✔ 0.15	✔ 3.94
file-item	✔ 0.13	✔ 0.18	✔ 21.63
half-square	✔ 0.15	✔ 0.19	✔ 3.98
loop-alignment	✔ 0.13	✔ 0.15	✔ 6.12
loop-unswitching	✔ 1.19	✔ 1.23	✔ 6.25
match	✔ 0.14	✔ 0.16	✔ 40.01
node	✔ 0.12	✔ 0.14	✔ 2.09
code-sinking	✖ 0.67	✖ 0.29	✖ 19.21
data-alignment	✖ ∞	✖ ∞	✖ 33.84
loop-pipelining	✖ 5.32	✖ 15.82	✖ 15.83
loop-tiling	✖ ∞	✖ ∞	✖ ∞
strength-reduction	✖ ∞	✖ 0.20	✖ 2.74
square-sum	✖ 0.17	✖ 0.23	✖ 1.22

Fig. 20. Results from using KestRel's SeaHorn backend to verify a suite of array benchmarks to a suite of array benchmarks. All times reported in seconds.

Analogous to the previous experiment, the middle group of benchmarks SeaHorn was able to verify using only the naïve alignment includes the loop-free programs at the bottom of Table 1. However, it additionally contains several other benchmarks whose naïve alignment Dafny was not able to verify. We conjecture that this is due to SeaHorn's requirement that programs only use arrays with static sizes, allowing it use bounded verification techniques not possible in the presence of datatypes of arbitrary size. As before, while not unlocking new verifications, these benchmarks provide evidence that KestRel "does no harm": programs that verify before alignment continue to verify after, in a comparable amount of time (minus the overhead of finding an alignment).

The last group of benchmarks represents the six cases where SeaHorn was unable to verify KestRel alignments. As before, `data-alignment` and `loop-tiling` require synthesizing loop conditions currently beyond KestRel's scope. In the remaining cases, the complexity of the required loop invariants appears to put verification out of reach for SeaHorn. To verify these alignments are nevertheless valid, we manually verified each of these aligned programs using VST [Cao et al. 2018]. For the cases verified with VST, invariants did not require specification of full functional correctness.

5.5 Discussion and Limitations

Data-Dependent Alignments. The alignments KestRel considers are limited by the rewrite rules given to it. In some cases, these rewrites are insufficient to express conditions needed for a verifiable alignment. *Data-dependent alignments* examine program state to decide control flow, for example by performing an unrolled or duplicated loop iteration only when some condition over program variables is met. In general this may require synthesizing boolean expressions that fall outside the terms KestRel's rewrite rules can generate; for example, a good alignment of our `data-alignment` benchmark requires conditionals over a program variable modulus 2 and 3. Synthesizing these kinds of expressions is beyond the scope of KestRel's rewrite-based approach.

Neighbor and Cost Function Heuristics. Because KestRel operates in a verifier independent way, its quality of alignments depends on the quality of heuristics in its neighbor and cost functions. This is a purposeful design tradeoff to avoid tightly coupling with bespoke relational verifiers, as leveraging off-the-shelf single-execution verifiers is one of the goals of intermediate program approaches to relational verification. However, heuristic approaches are susceptible to missing uncommon or unintuitive results. We believe exploring the state space in a more principled way by using feedback from the backing single-execution verifiers to drive intermediate program construction for relational reasoning may be an interesting direction for future work.

Backend Verifiers. While KestRel inherits the strengths of any backend verifier it targets, this means it also inherits its limitations. (Lack of invariant inference in Dafny is why we needed to develop an invariant inference engine based using Daikon for our Dafny backend, for example.) Similarly, the predicates used to specify the behaviors of our ADT operations in EUF were not included in Daikon's hypothesis space; these predicates comprise several of the user-defined predicates included in our invariant inference engine. We also had to add several axioms about the semantics of the uninterpreted function symbols used. For example, the `loop-alignment` benchmark needed to be given the invariant left.b = store(right.b, right.j, read(right.a, right.j)) after the lists `left.b` and `right.b` had both stored values at equal indicies from equal locations in equal lists. While this is a straightforward consequence of the axioms modeling the lists, these kinds of predicates are outside of Daikon's predicate space.

6 Related Work

Francez [1983] identified two high-level strategies for reasoning about *product properties*, i.e. what we call relational properties: "indirect" methods, which explicitly construct an auxiliary *product program* that can be verified using standard techniques for single programs; and "direct" approaches, which adapt single-program techniques to the relational setting. The product program construction proposed by Francez was equivalent to our naive construction; he observed that reasoning over this simple construction often amounted to proving full functional correctness of the input programs. In response to this challenge, Francez advocated for techniques tailored to the relational setting, and proposed two direct approaches: a relational variant of Floyd's inductive assertions method [Floyd 1967], and a relational variant of Hoare logic [Hoare 1969]. Instead of discarding indirect approaches entirely, this paper presents a technique for building better intermediate programs, constructing an *aligned* program that exposes the similarities between the input programs in a way that single-program verifiers can take advantage of.

6.1 Building Aligned Intermediate Programs

Barthe et al. [2011a] also observed that more sophisticated alignment strategies can yield programs more amenable to verification using standard techniques. Building on prior work which employed

self composition [Barthe et al. 2011b]— effectively our naive construction applied to two copies of the same program— to formalize information flow properties, Barthe et al. [2011a] proposed a set of inference rules for constructing intermediate programs which *synchronize* the execution of certain instructions, and shows how they can be used to manually construct several intermediate programs that can be automatically verified by Why. Subsequent work extended construction to cover a broader class of *asymmetric properties*, which include refinement [Barthe et al. 2013a]. Neither of these works present an algorithm for automatically constructing an intermediate program.

A key challenge to this approach is that the set of intermediate programs is potentially infinite, so the set of candidate programs must thus be constrained in some way to find a good alignment. One strategy to tame this complexity is to restrict the shape of the input programs. *Hyperproperties* are a restricted form of relational properties that involve multiple executions of the same program [Clarkson and Schneider 2010]. In some sense the two copies of the same programs are already in perfect alignment; the construction of Barthe et al. [2014] exploits this fact to build a product program construction for reasoning about differential privacy, which is a hyperproperty. In order to validate compiler optimizations, Zaks and Pnueli [2008] provides an algorithm for constructing intermediate programs for *consonant*, i.e., structurally similar, programs by syntactically aligning a predetermined set of locations in the source and target programs. The consonance requirement rules out programs where locations cannot be perfectly synchronized, which includes programs pairs whose loops execute for different numbers of iterations, e.g., the programs in Fig. 3.

Richer alignment strategies broaden the set of candidate programs. Like KestRel, Churchill et al. [2019] also try to leverage concrete program executions to guide the search for intermediate programs. Instead of trying to align syntactic program locations, that work uses an *alignment predicate* to identify "semantic" points in program traces that should be aligned, and then constructs a candidate program based on that alignment. The resulting intermediate program is not guaranteed to be semantically equivalent to the input programs, so after construction, candidate are checked to ensure they capture all the behaviors of the input programs. When the set of traces fails to cover all the paths in the original program, or a poor alignment predicate was chosen, this check can fail. In contrast, KestRel produces intermediate programs that are guaranteed to be equivalent *by construction*: KestRel first builds a space of equivalent programs, and then uses data gathered from executions to identify the solution with the best alignment. In addition, there are some alignments which cannot be covered by a finite set of executions, e.g., when the alignment depends on an unbounded input. Because KestRel only relies on data to rank (and not construct) candidate solutions, it is able to find aligned intermediate programs when this occurs, allowing it to align the programs in Fig. 4, which the approach of Churchill et al. [2019] cannot directly handle.

6.2 Direct Approaches

Since Francez first advocated for direct relational verification techniques, a diverse range of approaches that adapt single-program verification methodologies to the relational setting have been proposed. Given the wide range of direct approaches to relational verification, we limit our discussion to how some of the most prominent examples exploit relationships between target programs to simplify relational reasoning.

Relational Program Logics. One particular popular category of direct techniques are *relational program logics*, which extend traditional program logics to the relational setting [Aguirre et al. 2017; Banerjee et al. 2016; Barthe et al. 2013b; Benton 2004; Dardinier et al. 2024; Dardinier and Müller 2024; Dickerson et al. 2022; Francez 1983; Gladshtein et al. 2024; Maillard et al. 2019; Sousa and Dillig 2016; Yang 2007]. First introduced by Francez, and more famously rediscovered by Benton [2004], the judgements of these logics use relational assertions to reason over multiple programs, and/or

multiple copies of the same program. One feature of many of these relational logics is that they are not syntax-directed: Cartesian Hoare Logic (CHL) [Sousa and Dillig 2016], for example, provides four rules for reasoning about loops, each corresponding to a different program alignment. CHL's rule for reasoning about loops that execute in lockstep is roughly analogous to our While-Align law, for example. Rules corresponding to closer alignments enable simpler loop invariants, so finding a good alignment in these logics is often analogous to identifying which of these rules to apply. One exception is the relational logic of Banerjee et al. [2016], which uses a"biprogram" syntax to allow users to explicitly express alignments between multiple programs, enabling users to account for different interleavings of biprogram control flow when constructing proofs.

Automated verifiers based on these logics typically rely on built-in heuristics to select which rule to apply [Dardinier et al. 2024; Sousa and Dillig 2016]. Chen et al. [2019] use reinforcement learning to find effective proof strategies for different categories of programs. Alignments are effectively discovered via learned proof strategies which dictate the order in which to apply rules of the relational logic. In contrast, KestRel uses concrete program traces to identify promising alignments *before* verification. An interesting direction for future work is to investigate if execution traces could be used to guide rule selection in verifiers based on relational logics.

Relational Verification via CHC Solving. Several works have considered how to adapt verification approaches based on constrained Horn clause (CHC) solving to the relational setting [De Angelis et al. 2016; Itzhaky et al. 2024; Shemer et al. 2019; Unno et al. 2021]. These approaches generally operate by encoding the semantics of the target programs as a set of CHCs, and then augmenting or transforming these in a clauses to enable verification using standard CHC solvers. This process often amounts to identifying how to align subparts of the programs' semantics (which these works sometimes call "scheduling"). While these works share our goal of finding and exploiting correspondences between program structures to enable tractable automated verification, they all operate directly over an embedding of the semantics of the target programs and furthermore do not explicitly construct an intermediate program that is independent from a particular verifier.

De Angelis et al. [2016] first encode the target programs as CHCs and then constructing an equisatisfiable set of clauses that combines pairs of predicates into single predicates. This linearizes the relationships between variables, making resulting in clauses that can be handled by standard CHC solvers. Inferring how to pair predicates is tantamount to discovering alignments between subparts of the represented programsShemer et al. [2019] verify k-safety properties by inferring a self-composition function that interleaves control flow from multiple executions along with an invariant sufficient to verify the composite program. This procedure is parameterized over a set of predicates that are used to construct the self-composition and invariant pair. The inferred self-composition function constitutes an alignment for the semantic embedding of the target programs.Unno et al. [2021] make alignment constraints manifest in templated verification conditions, which are expressed in an extension to constrained Horn clauses. Finding alignment then becomes a concern of a CEGIS-based verifier for this extended class of CHCs. Itzhaky et al. [2024] present a technique for reducing $\forall\exists$-hyperproperties over infinite-state transition systems to a set of CHCs. Program trace alignments are represented using uninterpreted predicates whose constraints can be expressed as a CHC problem.

Language-Theoretic Approaches to Relational Verification. Farzan and Vandikas [2019a,b] use infinite tree automata to represent sets of semantic reductions of multiple program runs, such that proving a relational property for one of these reductions is sufficient to establish that property over the original programs. Proofs are discovered via a counterexample-guided refinement loop, iteratively strengthening a candidate proof until a covered reduction is found. Valid reductions are defined via a dependence relation over how original program statements may be reordered. The

infinite tree automata in this approach are roughly analogous to our use of e-graphs (both data structures represent a space of program reductions or alignments, respectively), and the dependence relation here controls the space of reductions in a similar way to how our rewrite rules control the space of possible alignments. However, instead of reducing a semantic model of program traces to produce a proof, our approach uses syntactic realignment laws to construct an aligned intermediate program which can be handed to an off-the-shelf verifier for the original source language.

6.3 Extracting Terms from E-Graphs

Extracting a desirable term with a heuristic cost function is a core piece of synthesis and optimization techniques based on equality saturation [Tate et al. 2009]. Using local cost functions to greedily select subterms is a common strategy, and forms the default extraction mechanism of the popular Egg library [Willsey et al. 2021]. Wang et al. [2020] propose an alternative non-local approach based on mixed-integer linear programming (MILP). Although this approach requires assigning a single, static cost for each e-graph node. In contrast, alignment problems are most naturally expressed using variable node costs that depend on, e.g., sibling extractions. Although it is possible to set up MILP encodings for alignment problems, our initial experiments using this technique did not scale to the majority of the benchmarks in our evaluation.

7 Conclusion

One way to reason about relationships between multiple programs is to construct a single intermediate program that captures their behaviors and then apply standard program reasoning techniques to that intermediate program to establish the desired relational property. A key hurdle to employing this strategy is finding an intermediate program that exposes semantic similarities between the original programs in way that can be exploited by a verifier. In this paper, we have presented an approach to automatically building aligned intermediate program that are amenable to automated verification. We first embed the target programs in an algebra that captures how they should be aligned, insert that term into an e-graph, and use realignment rules to construct a space of aligned intermediate programs. We use a novel data-driven technique which examines execution traces to identify the most promising alignment; we extract an intermediate program from this alignment and hand it off to an off-the-shelf verifier. We have implemented this approach in a tool, called KestRel, which supports both Dafny and SeaHorn as backend verifiers. We have evaluated KestRel on a diverse suite of benchmarks taken from the relational verification literature. Our experiments show that KestRel is capable of discovering alignments that enable verification to succeed where a naïve alignment strategy would otherwise fail.

Data-Availablity Statement

Our supplementary material includes an anonymized artifact. This artifact contains the source code for KestRel, our suite of benchmark programs, scripts to reproduce the experimental results in Section 5, and a Coq formalization of CoreRel and its metatheory. We intend to submit this artifact for evaluation by the artifact evaluation committee should this paper be accepted.

References

Alejandro Aguirre, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Pierre-Yves Strub. 2017. A Relational Logic for Higher-order Programs. *Proc. ACM Program. Lang.* 1, ICFP, Article 21 (Aug. 2017), 29 pages.

Timos Antonopoulos, Eric Koskinen, Ton Chanh Le, Ramana Nagasamudram, David A Naumann, and Minh Ngo. 2023. An Algebra of Alignment for Relational Verification. *Proceedings of the ACM on Programming Languages* 7, POPL (2023), 573–603.

Anindya Banerjee, David A Naumann, and Mohammad Nikouei. 2016. Relational logic with framing and hypotheses. In *36th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2016)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik.

Gilles Barthe, Juan Manuel Crespo, and César Kunz. 2011a. Relational verification using product programs. In *FM 2011: Formal Methods: 17th International Symposium on Formal Methods, Limerick, Ireland, June 20-24, 2011. Proceedings 17*. Springer, 200–214.

Gilles Barthe, Juan Manuel Crespo, and César Kunz. 2013a. Beyond 2-safety: Asymmetric product programs for relational program verification. In *Logical Foundations of Computer Science: International Symposium, LFCS 2013, San Diego, CA, USA, January 6-8, 2013. Proceedings*. Springer, 29–43.

Gilles Barthe, Pedro R D'argenio, and Tamara Rezk. 2011b. Secure information flow by self-composition. *Mathematical Structures in Computer Science* 21, 6 (2011), 1207–1252.

Gilles Barthe, Marco Gaboardi, Emilio Jesús Gallego Arias, Justin Hsu, César Kunz, and Pierre-Yves Strub. 2014. Proving Differential Privacy in Hoare Logic. In *Proceedings of the 2014 IEEE 27th Computer Security Foundations Symposium (CSF '14)*. IEEE Computer Society, USA, 411–424. https://doi.org/10.1109/CSF.2014.36

Gilles Barthe, Boris Köpf, Federico Olmedo, and Santiago Zanella-Béguelin. 2013b. Probabilistic Relational Reasoning for Differential Privacy. *ACM Trans. Program. Lang. Syst.* 35, 3, Article 9 (nov 2013), 49 pages. https://doi.org/10.1145/2492061

Patrick Baudin, François Bobot, David Bühler, Loïc Correnson, Florent Kirchner, Nikolai Kosmatov, André Maroneze, Valentin Perrelle, Virgile Prevosto, Julien Signoles, and Nicky Williams. 2021. The dogged pursuit of bug-free C programs: the Frama-C software analysis platform. *Commun. ACM* 64, 8 (jul 2021), 56–68. https://doi.org/10.1145/3470569

Nick Benton. 2004. Simple Relational Correctness Proofs for Static Analyses and Program Transformations. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Venice, Italy) *(POPL '04)*. ACM, New York, NY, USA, 14–25.

Qinxiang Cao, Lennart Beringer, Samuel Gruetter, Josiah Dodds, and Andrew W. Appel. 2018. VST-Floyd: A Separation Logic Tool to Verify Correctness of C Programs. *J. Autom. Reason.* 61, 1-4 (2018), 367–422. https://doi.org/10.1007/S10817-018-9457-5

Jia Chen, Jiayi Wei, Yu Feng, Osbert Bastani, and Isil Dillig. 2019. Relational verification using reinforcement learning. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–30.

Berkeley Churchill, Oded Padon, Rahul Sharma, and Alex Aiken. 2019. Semantic program alignment for equivalence checking. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 1027–1040.

Koen Claessen and John Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00)*. Association for Computing Machinery, New York, NY, USA, 268–279. https://doi.org/10.1145/351240.351266

Michael R. Clarkson and Fred B. Schneider. 2010. Hyperproperties. *J. Comput. Secur.* 18, 6 (sep 2010), 1157–1210.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms, Third Edition* (3rd ed.). The MIT Press.

Thibault Dardinier, Anqi Li, and Peter Müller. 2024. Hypra: A Deductive Program Verifier for Hyper Hoare Logic. *Proceedings of the ACM on Programming Languages* 8, OOPSLA2 (2024), 1279–1308.

Thibault Dardinier and Peter Müller. 2024. Hyper Hoare Logic: (Dis-)Proving Program Hyperproperties. *Proc. ACM Program. Lang.* 8, PLDI, Article 207 (June 2024), 25 pages. https://doi.org/10.1145/3656437

Emanuele De Angelis, Fabio Fioravanti, Alberto Pettorossi, and Maurizio Proietti. 2016. Relational verification through horn clause transformation. In *Static Analysis: 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8-10, 2016, Proceedings 23*. Springer, 147–169.

Robert Dickerson, Qianchuan Ye, Michael K Zhang, and Benjamin Delaware. 2022. RHLE: Modular Deductive Verification of Relational ∀∃ Properties. In *Programming Languages and Systems: 20th Asian Symposium, APLAS 2022, Auckland, New Zealand, December 5, 2022, Proceedings*. Springer, 67–87.

Michael D Ernst, Jeff H Perkins, Philip J Guo, Stephen McCamant, Carlos Pacheco, Matthew S Tschantz, and Chen Xiao. 2007. The Daikon system for dynamic detection of likely invariants. *Science of computer programming* 69, 1-3 (2007), 35–45.

Azadeh Farzan and Anthony Vandikas. 2019a. Automated hypersafety verification. In *Computer Aided Verification: 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I 31*. Springer, 200–218.

Azadeh Farzan and Anthony Vandikas. 2019b. Reductions for safety proofs. *Proceedings of the ACM on Programming Languages* 4, POPL (2019), 1–28.

Cormac Flanagan and K Rustan M Leino. 2001. Houdini, an annotation assistant for ESC/Java. In *International Symposium of Formal Methods Europe*. Springer, 500–517.

Robert W. Floyd. 1967. Assigning Meanings to Programs. *Proceedings of Symposium on Applied Mathematics* 19 (1967), 19–32.

Nissim Francez. 1983. Product properties and their direct verification. *Acta informatica* 20 (1983), 329–344.

Vladimir Gladshtein, Qiyuan Zhao, Willow Ahrens, Saman Amarasinghe, and Ilya Sergey. 2024. Mechanised Hypersafety Proofs about Structured Data. *Proc. ACM Program. Lang.* 8, PLDI, Article 173 (June 2024), 24 pages. https://doi.org/10.1145/3656403

Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A Navas. 2015. The SeaHorn verification framework. In *Computer Aided Verification: 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*. Springer, 343–361.

W Keith Hastings. 1970. Monte Carlo Sampling Methods using Markov Chains and their Applications. (1970).

C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12, 10 (Oct. 1969), 576–580.

Shachar Itzhaky, Sharon Shoham, and Yakir Vizel. 2024. Hyperproperty verification as chc satisfiability. In *European Symposium on Programming*. Springer, 212–241.

Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. 2023. Verus: Verifying Rust Programs using Linear Ghost Types. *Proc. ACM Program. Lang.* 7, OOPSLA1, Article 85 (apr 2023), 30 pages. https://doi.org/10.1145/3586037

K. Rustan M. Leino. 2010. Dafny: an automatic program verifier for functional correctness. In *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning* (Dakar, Senegal) *(LPAR'10)*. Springer-Verlag, Berlin, Heidelberg, 348–370.

Kenji Maillard, Cătălin Hriţcu, Exequiel Rivas, and Antoine Van Muylder. 2019. The next 700 Relational Program Logics. *Proc. ACM Program. Lang.* 4, POPL, Article 4 (dec 2019), 33 pages. https://doi.org/10.1145/3371072

Leonardo Moura and Nikolaj Bjørner. 2007. Efficient E-Matching for SMT Solvers. In *Proceedings of the 21st International Conference on Automated Deduction: Automated Deduction* (Bremen, Germany) *(CADE-21)*. Springer-Verlag, Berlin, Heidelberg, 183–198. https://doi.org/10.1007/978-3-540-73595-3_13

Peter Müller, Malte Schwerhoff, and Alexander J. Summers. 2016. Viper: A Verification Infrastructure for Permission-Based Reasoning. In *Verification, Model Checking, and Abstract Interpretation*, Barbara Jobstmann and K. Rustan M. Leino (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 41–62.

CG Nelson. 1980. *Techniques for program verification [Ph. D. Thesis]*. Stanford University, CA, USA.

Robert Nieuwenhuis and Albert Oliveras. 2005. Proof-producing congruence closure. In *International Conference on Rewriting Techniques and Applications*. Springer, 453–468.

Saswat Padhi, Rahul Sharma, and Todd Millstein. 2016. Data-Driven Precondition Inference with Learned Features. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Santa Barbara, CA, USA) *(PLDI '16)*. Association for Computing Machinery, New York, NY, USA, 42–56. https://doi.org/10.1145/2908080.2908099

Rahul Sharma, Eric Schkufza, Berkeley Churchill, and Alex Aiken. 2013. Data-Driven Equivalence Checking. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications* (Indianapolis, Indiana, USA) *(OOPSLA '13)*. Association for Computing Machinery, New York, NY, USA, 391–406. https://doi.org/10.1145/2509136.2509509

Ron Shemer, Arie Gurfinkel, Sharon Shoham, and Yakir Vizel. 2019. Property directed self composition. In *Computer Aided Verification: 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I 31*. Springer, 161–179.

Marcelo Sousa and Isil Dillig. 2016. Cartesian hoare logic for verifying k-safety properties. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Santa Barbara, CA, USA) *(PLDI '16)*. Association for Computing Machinery, New York, NY, USA, 57–69. https://doi.org/10.1145/2908080.2908092

Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoue, and Santiago Zanella-Béguelin. 2016. Dependent types and multi-monadic effects in F*. *SIGPLAN Not.* 51, 1 (jan 2016), 256–270. https://doi.org/10.1145/2914770.2837655

Robert Endre Tarjan. 1975. Efficiency of a good but not linear set union algorithm. *Journal of the ACM (JACM)* 22, 2 (1975), 215–225.

Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. 2009. Equality saturation: a new approach to optimization. *SIGPLAN Not.* 44, 1 (Jan. 2009), 264–276. https://doi.org/10.1145/1594834.1480915

Hiroshi Unno, Tachio Terauchi, and Eric Koskinen. 2021. Constraint-based relational verification. In *Computer Aided Verification: 33rd International Conference, CAV 2021, Virtual Event, July 20–23, 2021, Proceedings, Part I*. Springer, 742–766.

Yisu Remy Wang, Shana Hutchison, Jonathan Leang, Bill Howe, and Dan Suciu. 2020. SPORES: sum-product optimization via relational equality saturation for large scale linear algebra. *arXiv preprint arXiv:2002.07951* (2020).

Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. 2021. egg: Fast and extensible equality saturation. *Proc. ACM Program. Lang.* 5, POPL, Article 23 (Jan. 2021), 29 pages. https://doi.org/10.1145/3434304

Hongseok Yang. 2007. Relational separation logic. *Theor. Comput. Sci.* 375, 1–3 (apr 2007), 308–334. https://doi.org/10.1016/j.tcs.2006.12.036

Anna Zaks and Amir Pnueli. 2008. CoVaC: Compiler Validation by Program Analysis of the Cross-Product. In *Proceedings of the 15th International Symposium on Formal Methods* (Turku, Finland) *(FM '08)*. Springer-Verlag, Berlin, Heidelberg, 35–51. https://doi.org/10.1007/978-3-540-68237-0_5

He Zhu, Gustavo Petri, and Suresh Jagannathan. 2016. Automatically Learning Shape Specifications. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Santa Barbara, CA, USA) *(PLDI '16)*. Association for Computing Machinery, New York, NY, USA, 491–507. https://doi.org/10.1145/2908080.2908125