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 popular approach to reasoning about these sorts of relational properties is to construct and verify a *product program*: a program whose correctness implies that the individual programs exhibit the desired relational property. A key challenge in product program construction 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 product program construction that uses e-graphs, equality saturation, and algebraic realignment rules to efficiently represent and build verifiable product programs. A key ingredient of our solution is a novel data-driven extraction technique that uses execution traces of product programs to identify candidate 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

First proposed by Hoare [19] and Floyd [16], deductive program logics are a popular foundation for many modern program verification tools [17, 20, 21, 27, 29, 30]. The majority of these tools consider *single* executions of a program, certifying that each run of the program results in a state meeting some postcondition. Many interesting behaviors involve the executions of *multiple* programs, however. For example, say we wish to show two programs p_1 and 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 jointly reasoning about the executions of both p_1 and p_2.

A variety of important program behaviors are relational properties, including observational equivalence, refinement, idempotence, non-interference, and co-termination. Several verification techniques for reasoning about relational properties have been proposed. These approaches can be roughly grouped into two camps: those relying on bespoke relational logics [1, 5, 6, 12, 13, 23, 33, 36], and those that reduce a relational problem to reasoning about a single execution of an equivalent *product program* [3, 4]. Relational program logics operate directly over multiple programs, and usually include rules for reasoning about parallel control flow structures from each program simultaneously. Techniques based on product programs, in contrast, attempt to build a single program that encodes the behaviors of multiple programs, and that is then verified using existing single-program verification tools. As a consequence, product program-based approaches inherit any advances in single program verification technologies, e.g. automatic invariant inference, essentially 'for free'.

The effectiveness of both approaches hinges on finding proper *alignments* of the underlying programs [24]. Alignments identify subparts of the original programs, e.g. control flow paths, whose similarities can be exploited by the underlying verifier. In the case of relational logics, alignments implicly drive the application of rules for simultaneous reasoning, while in the case of product programs, alignment dictates how statements from each program are grouped in their product.

To illustrate the role proper alignment can play in relational reasoning, consider the pair of programs shown in Fig. 1. Both programs iterate over a list of employees, scheduling bonus

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

p_1

```
int i₂ = 0;
int bonus₂ = calc_bonus(rate);
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.

payments for the identified workers via some black-box financial services API. The program on the right does so slightly more efficiently than the one on the left, however, as it caches part of the bonus calculation prior to entering the loop. To establish that this optimization is safe, we might wish to verify that, starting from the same initial state, each program schedules the same set of payments. To do so, it suffices to verify the *product program* on the right, a single program that encodes the semantics of both programs. We can see from the product program that `payments.schedule(...)` is indeed called with the same arguments in each iteration of the loop. As long as the API methods are deterministic, we can be confident that p_1 and p_2 have the same effect. This property is easily expressed in the theory of equality with uninterpreted functions (EUF), a logic supported by all modern SMT solvers. The combined loop maintains the straightforward invariant that $\texttt{payments}_1$ and $\texttt{payments}_2$ are equivalent after each loop iteration; this invariant is also expressible in EUF. Contrast this with what is needed to reason about a product program that naïvely concatenates the two program together, $p_1;\ p_2$. This requires invariants that completely characterize how the individual loops mutate their respective copies of `payments` in order to show their final states are equivalent; this invariant may not even be expressible in a decidable logic [32]. Even if the loop invariant is expressible, establishing that it holds requires specifications encoding the full functional correctness of the `schedule` method.

```
int i₁ = 0; int i₂ = 0;
int bonus₂ = calc_bonus(rate);
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(rate));
 payments₂.schedule(id₂, sal₂ * bonus₂);
 i₁ += 1;  i₂ += 1; }
```

$p_{1\times 2}$

Fig. 2. A product of the two programs in Fig. 1

Unfortunately, automatically finding a product program that facilitates verification presents several challenges. First, the number of product programs is exponential in the size of the programs involved, making it difficult to effectively search for good product programs. Next, finding a good alignment can demand more than a simple syntactic grouping of statements and control flow constructs; the optimal alignment may require transforming the original programs to exploit semantic similarities between them. As a simple example, it is only after the loop in Fig. 3a is unrolled by one iteration that it is fruitful to align the resulting loop with the one in Fig. 3b. Finally, after constructing a searchable space of alignments, it remains to identify a good alignment for use in (relational) verification. For these reasons, most relational verification techniques that rely on product programs either punt on the question of how to algorithmically construct a product

```
int i₁ := 0;
while (i₁ ≤ n) {
  x₁ += i₁;
  i₁++;
}
```

(a)

```
int i₂ := 1;
while (i₂ ≤ n) {
  x₂ += i₂;
  i₂++;
}
```

(b)

Fig. 3. The loop on the left should be unrolled once when constructing a product program (from Barthe et al. [3]).

program [2–4], or are tightly coupled to a particular language and verification task, e.g., proving observational equivalence between x86 programs [9, 31].

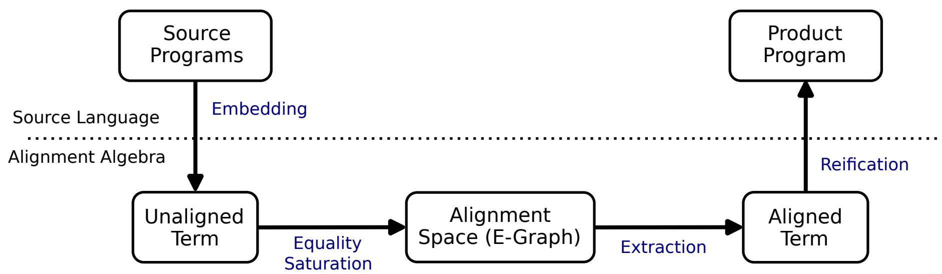


Fig. 4. High-level overview of KestRel.

The present work addresses these challenges by leveraging recent advances in equality saturation [38] and algebraic approaches to program alignment [2] to build product programs amenable to verification. Fig. 4 presents a high-level overview of our proposed approach. The process begins by naïvely embedding the input programs in CoreRel, a relational calculus equipped with algebraic *realignment rules* in the spirit of Antonopoulos et al. [2]. This approach allows us to easily exploit existing semantics-preserving transformations on individual program (e.g., loop unrolling) that unlock better alignments. We then use these rules to build an e-graph [25, 26, 38] that compactly represents the space of possible alignments. In order to effectively explore the space of candidate alignments, we use a novel, data-driven approach to program extraction that examines program traces to identify promising alignments. The most promising product program is then reified back into the original source language and handed off to an off-the-shelf solver for verification. In contrast to approaches that rely on specialized relational verifiers, our solution is capable of repurposing existing verifiers for single programs, allowing users to obtain a relational verifier at little cost. We have implemented our approach in a tool, KestRel, that features Dafny [21] and SeaHorn [17] backends, enabling it to reason about relational properties of programs that use API to manage abstract data types with hidden internal state, as well as array-manipulating C programs. We have evaluated KestRel on a diverse suite of benchmarks and relational properties taken from the literature. Our experimental results show that KestRel discovers alignments that enable verification to succeed where simpler alignment strategies would otherwise fail.

In summary, this paper describes the following contributions:

- We show how to use e-graphs to build and compactly represent the space of possible product 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 an overview of our approach. We then formally define our core calculus for relational alignment, and formalize our approach to relational verification using this calculus. Section 4 describes our equational approach to program realignment, and details how we represent spaces of possible alignments using e-graphs. Section 5 then explains our data-driven technique for extracting a product program from this space. Section 6

```
⟨ int y₁ = 0;              | int y₂ = 0;             ⟩       ⟦ int y₁ = 0;              ⟧ ⨟ ⟦ int y₂ = 0;             ⟧
⟨ int z₁ = 2 * x₁;         | int z₂ = x₂;            ⟩   ≡   ⟦ int z₁ = 2 * x₁;         ⟧   ⟦ int z₂ = x₂;            ⟧   ≡
⟨ while (z₁ > 0) {         | while (z₂ > 0) {        ⟩       ⟦ while (z₁ > 0) {         ⟧   ⟦ while (z₂ > 0) {        ⟧
⟨   z₁--; y₁ += x₁ }       |   z₁--; y₁ += x₁ }      ⟩       ⟦   z₁--; y₁ += x₁ }       ⟧   ⟦   z₁--; y₁ += x₁ }      ⟧
⟨                          | y₂ *= 2                 ⟩       ⟦                          ⟧   ⟦ y₂ *= 2                 ⟧

      ⟨ int y₁ = 0             | int y₂ = 0;     ⟩ ⨟          ⟨ int y₁ = 0          | int y₂ = 0; ⟩ ⨟
      ⟨ int z₁ = 2 * x₁;       | int z₂ = x₂     ⟩            ⟨ int z₁ = 2 * x₁;    | int z₂ = x₂ ⟩
· · · ≡ ⟨ while (z₁ > 0) {     | while (z₂ > 0) { ⟩ ⨟   ≡     while_St 2 1 ⟨ z₁ > 0 | z₂ > 0 ⟩
      ⟨   z₁--;                |   z₂--;         ⟩                         ⟨ z₁--;    | z₂--;    ⟩ ⨟
      ⟨   y₁ = y₁ + x₁ }       |   y₂ = y₂ + x₂ } ⟩                        ⟨ y₁ += x₁ | y₂ += x₂ ⟩
      [ y₂ *= 2 ⟩                                             [ y₂ *= 2 ⟩
```

Fig. 5. 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.

and Section 7 describe our implementation and evaluation of KestRel. Related work and conclusions are given in Sections 8 and 9.

2 OVERVIEW

We begin by illustrating the key pieces of our proposed approach to automatically constructing product programs, using the programs labelled $\texttt{double}_1$ and $\texttt{double}_2$ in the figure below as examples:

$\texttt{double}_1$	$\texttt{double}_2$	$\texttt{double}_{1\times 2}$

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

Each program sets its version of y to 2x: $\texttt{double}_1$ does so by incrementing y_1 $2x_1$ times, while $\texttt{double}_2$ increments y_2 x_2 times and then doubles the result. To verify that both programs update their ys to the same final value, it suffices to verify that the program labeled $\texttt{double}_{1\times 2}$ always ends in a state in which $y_1 = y_2$ when executed from a state in which $x_1 = x_2$. Here, $\texttt{double}_{1\times 2}$ is an example of a *product program*. Many such product programs exist: the simplest one simply sequences $\texttt{double}_1$ and $\texttt{double}_2$; another swaps the first and second lines of $\texttt{double}_{1\times 2}$. While all of these product programs are semantically equivalent, some of them are easier to verify than others. For example, $\texttt{double}_{1\times 2}$, requires a loop invariant that is a simple equality between the values of $y_1 = 2 * y_2$, while $\texttt{double}_1$; $\texttt{double}_2$ requires two loop invariants, each of which involve x, y, and z.

An Algebra of Alignments. Our first step in constructing $\texttt{double}_{1\times 2}$ is to embed $\texttt{double}_1$ and $\texttt{double}_2$ into a richer domain that provides a more structured representation of product 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 a product 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

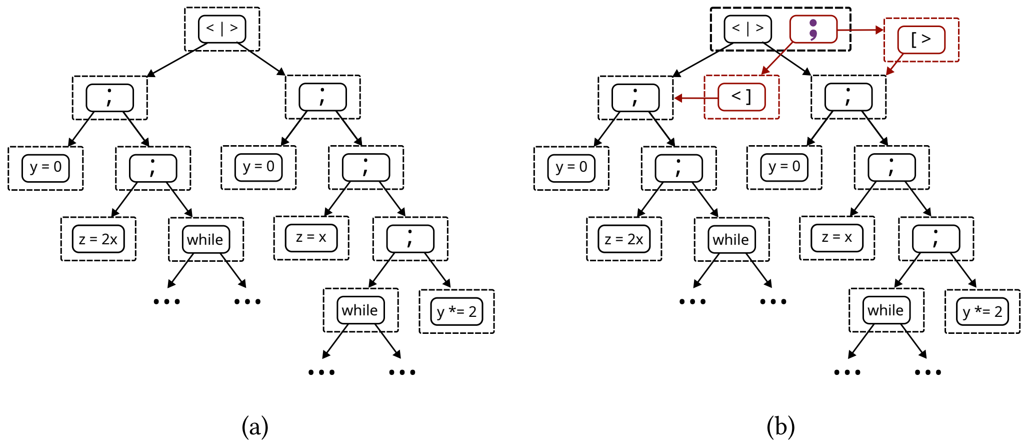


Fig. 6. E-graphs containing representations of possible alignments between $\texttt{double}_1$ and $\texttt{double}_2$. The e-graph on the left (a) contains only the initial embedding. The e-graph on the right (b) contains the initial embedding plus an application of the REL-DEF given in Fig. 10. (Added nodes are depicted in red.) It is possible to extract both the first and second terms in Fig. 5 from (b).

of the product program: the alignment $\langle \texttt{s}_1 \mid \texttt{t}_1;\ \texttt{t}_2 \rangle \fatsemi \langle \texttt{s}_2 \mid \texttt{t}_3 \rangle$, for example, groups together the first statement of $\texttt{s}_1;\ \texttt{s}_2$ with the first two statements of $\texttt{t}_1;\ \texttt{t}_2;\ \texttt{t}_3$ and aligns the last statements of both programs; these subalignments are composed together with the $\fatsemi$ operator. Our domain is equipped with other relational operators for aligning different control flow operators. The most important of these is the $\texttt{while}_R\ \langle \texttt{b}_1 | \texttt{b}_2 \rangle\ \langle \texttt{c}_1 | \texttt{c}_2 \rangle$ operator, which encodes a product program that executes the bodies of two loops in lockstep. The final alignment in Fig. 5 encodes $\texttt{double}_{1\times2}$ using a variant of this operator, $\texttt{while}_{St}\ \texttt{m}\ \texttt{n}\ \langle \texttt{b}_1 | \texttt{b}_2 \rangle\ \langle \texttt{c}_1 | \texttt{c}_2 \rangle$, which executes $\texttt{c}_1$ $\texttt{m}$ times and $\texttt{c}_2$ $\texttt{n}$ times on each iteration.

Alignments are equipped with an equivalence relation, $\equiv$. Intuitively, equivalent alignments represent semantically equivalent product 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. 5 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. 5 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 [25] as a compact representation of the space of alignments. Each node in an e-graph is a member of an equivalence class, and equivalence classes can be updated when the e-graph is extended with new information on equivalent subterms. Fig. 6(a) gives a simplified representation of the $\langle \texttt{double}_1 \mid \texttt{double}_2 \rangle$ as an e-graph, while Fig. 6(b) depicts an e-graph that simultaneously encodes both the first and second alignments in 5, for example. Importantly, the set of terms an e-graph encodes can be algorithmically grown through a process known as *equality saturation* [35], which repeatedly updates an e-graph with a set of equivalences until either a fixpoint (saturation) or a bound on the number of iterations is reached. Saturating the e-graph in Fig. 6(a) with a set of realignment rules results in an e-graph that includes the alignment corresponding to $\texttt{double}_{1\times2}$.

```
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. 7. Syntax and notations for Imp and CoreRel.

Searching for Desirable Alignments. Once a fully saturated e-graph that represents the space of possible alignments of ⟨$\texttt{double}_1$ | $\texttt{double}_2$⟩ is in hand, our next step is to *extract* $\texttt{double}_{1\times 2}$ from the set of product programs embedded in the e-graph. Modern e-graph libraries [38] 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 involves *semantic* properties of the product program it represents. Finding the alignment that produces $\texttt{double}_{1\times 2}$, 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 product 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 [18] 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 a product program, e.g., $\texttt{double}_{1\times 2}$, which can then be given to an off-the-shelf single program verifier like Dafny [21] or SeaHorn [17].

3 THE CORERel LANGUAGE

This section describes a core calculus for program alignment, called CoreRel, which we use to formalize our approach to product program construction.[1] The starting point for our formalization is a completely standard core imperative programming language, Imp, whose syntax is shown on the lefthand side of Fig. 7. The calculus is parameterized over an infinite set of identifiers for program variables $\mathcal{V}$. Program states are partial mappings from identifiers to integers, and the semantics of an Imp program c is given by the expected big-step reduction relation from input states σ to output states σ': $\sigma, c \Downarrow \sigma'$. This language is also equipped with a completely standard program logic that acts as our "off-the-shelf" verifier for Imp programs. Formally, this logic proves Hoare triples of the form $\vdash \{\phi\}\ c\ \{\phi\}$, 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. Given a pair of Imp programs, c_1 and c_2, we say that c_1 and 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 $\models_R \{\Phi\}\ c_1 \otimes c_2\ \{\Psi\}$:

$$\models_R \{\Phi\}\ c_1 \otimes c_2\ \{\Psi\} \triangleq \forall \sigma_1, \sigma_2.\ \sigma_1 \uplus \sigma_2 \models \phi \implies \forall \sigma_1', \sigma_2'. \sigma_1, c_1 \Downarrow \sigma_1' \land \sigma_2, c_2 \Downarrow \sigma_2' \implies \sigma_1' \uplus \sigma_2' \models \psi$$

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

$$\frac{(\sigma_1,\ \sigma_2),\ \mathtt{r}_1 \Downarrow (\sigma_1',\ \sigma_2') \dashv \alpha_1 \quad (\sigma_1',\ \sigma_2'),\ \mathtt{r}_2 \Downarrow (\sigma_1'',\ \sigma_2'') \dashv \alpha_2}{(\sigma_1,\ \sigma_2)\ \mathtt{r}_1 \,;\, \mathtt{r}_2 \Downarrow (\sigma_1'',\ \sigma_2'') \dashv \alpha_1 +\!\!+ \alpha_2}\ \text{E-Seq}$$

$$\frac{\sigma_1,\ \mathtt{s}_1 \Downarrow \sigma_1' \dashv \alpha_1 \quad \sigma_2,\ \mathtt{s}_2 \Downarrow \sigma_2' \dashv \alpha_2}{(\sigma_1,\ \sigma_2),\ \langle \mathtt{s}_1 | \mathtt{s}_2 \rangle \Downarrow (\sigma_1',\ \sigma_2') \dashv \mathrm{rB}\langle\sigma_1,\sigma_2\rangle +\!\!+ \alpha_1 +\!\!+ \mathrm{rM}\langle\sigma_1',\sigma_2\rangle +\!\!+ \alpha_2 +\!\!+ \mathrm{rE}\langle\sigma_1',\sigma_2'\rangle}\ \text{E-Align}$$

$$\frac{\sigma_1,\ \mathtt{b}_1 \Downarrow \mathtt{true} \quad \sigma_2,\ \mathtt{b}_2 \Downarrow \mathtt{true} \quad (\sigma_1,\ \sigma_2)\ \mathtt{r} \Downarrow (\sigma_1',\ \sigma_2') \dashv \alpha 1 \quad (\sigma_1',\ \sigma_2'),\ \mathtt{while}_\mathtt{R}\ \langle \mathtt{b}_1 \mid \mathtt{b}_2 \rangle\ \mathtt{r} \Downarrow (\sigma_1'',\ \sigma_2'') \dashv \alpha 2}{(\sigma_1,\ \sigma_2),\ \mathtt{while}_\mathtt{R}\ \langle \mathtt{b}_1 \mid \mathtt{b}_2 \rangle\ \mathtt{r} \Downarrow (\sigma_1'',\ \sigma_2'') \dashv \mathrm{wH}_R\langle\sigma_1,\sigma_2\rangle +\!\!+ \alpha_1 +\!\!+ \alpha_2}\ \text{E-WhileT}$$

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

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

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

Fig. 8. Big-step semantics of CoreRel

An essential property of this definition is that both c_1 and 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 we use to align programs. By convention, we use the subscripts 1 and 2 to disambiguate between occurences of the same variable in the left- and right-hand programs [5]. Thus, the assertion $x_1 = x_2$ is satisfied by any pair of states in which x maps to the same number.

While several specialized *relational* program logics have been developed for proving relational triples directly [5, 22, 33], our goal in this work is to instead reduce a relational verification problem to an equivalent claim that only involves a single *product program* and that can additionally be proven using the program logic for Imp that we already have in hand. As discussed in Section 2, there may be many such programs, some of which are more amenable to automated verification than others. Our approach is to represent product 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. 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.

Syntax. Concretely, the syntax of aligned programs in CoreRel is given on the righthand side of Fig. 7. A basic alignment, $\langle\ c_1\ \mid\ c_2\ \rangle$, consists of a pair of Imp programs c_1 and c_2 whose control flows are completely disjoint; this is effectively the naïve embedding of c_1 and c_2. In contrast, the relational control flow operators $\mathtt{while}_\mathtt{R}$, $\mathtt{if}_\mathtt{R}$, and ; group together the control flows of their subexpressions: the branches of the relational conditional command $\mathtt{if}_\mathtt{R}$, for example, are themselves aligned programs. Fig. 7 also defines some additional notations for convenience: $\langle s]$ and $[s\rangle$ embed a single Imp program into the left and right sides of a relational representation, respectively. We use $\mathtt{while}_\mathtt{St}$ to denote 'stuttered' versions of aligned loops, where the left and right hand loop bodies execute a different number of times at each loop iteration: the aligned expression $\mathtt{while}_\mathtt{St}\ 2\ 1\ \langle \mathtt{b}_1 \mid \mathtt{b}_2 \rangle\ \langle \mathtt{c}_1 \mid \mathtt{c}_2 \rangle$, for example, represents a loop that executes c_1 twice for every execution of c_2.

Semantics. CoreRel is equipped with its own big-step operational semantics. The evaluation rules of the language are shown in Fig. 8. As aligned commands are meant to encode the behaviors of two programs, the reduction relation $(\sigma_1,\ \sigma_2)\ \mathsf{r} \Downarrow (\sigma_1',\ \sigma_2') \dashv \alpha$ defines how an aligned command r takes a pair of disjoint initial states to a pair of disjoint output states, and also records intermediate states via a trace. We will use these traces to rank the quality of an alignment; they can safely be ignored until Section 5. A pair of aligned programs produces a pair of output states by combining the results of independently running its constituent programs (E-Align). Executing the aligned program $\langle \mathsf{x}_1 := 2 \mid \mathsf{y}_2 := 3 \rangle$, for example, results in a pair of final states where the value of x in the first state is 2 and the value of y in the second state is 3; the values stored by all other variables in both states remain unchanged. A relational loop, in constrast, simulates a "lockstep" execution of a pair of loops, executing both behaviors of its aligned body in tandem (E-While$_R$T) until either of its conditions fails (E-While$_R$F).

Any pair of Imp programs c_1 and c_2 can be represented as the aligned CoreRel program $\langle \mathsf{c}_1 \mid \mathsf{c}_2 \rangle$. Importantly, this embedding preserves the semantics of the original pair of programs:

Theorem 3.2 (Embedding is Sound). *The embedding of $\langle \mathsf{c}_1 \mid \mathsf{c}_2 \rangle$ of a pair of Imp programs, c_1 and c_2, has the same semantics as the original programs:*

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

Equivalence. These semantics admit a natural notion of equivalence on aligned programs: equivalent alignments take the same initial states to the same final states:

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

3.1 Reifying CoreRel into Imp

Every aligned CoreRel program can be interpreted as a semantically equivalent Imp program. This interpretation is given by the reification function, $[\![\cdot]\!]$ shown in Fig. 9. This function constructs a product program in Imp from an aligned program, using a pair of variable renaming functions $[\![\cdot]\!]_L$ and $[\![\cdot]\!]_R$ to ensure that the programs operate over disjoint states. Importantly, reification is equivalence preserving, in that reifying equivalent aligned programs yields equivalent Imp programs:

$[\![\langle \mathsf{s}_1 \mid \mathsf{s}_2 \rangle]\!] \triangleq [\![\mathsf{s}_1]\!]_L;\ [\![\mathsf{s}_2]\!]_R$
$[\![\mathsf{r}_1 \,\mathbf{;}\, \mathsf{r}_2]\!] \triangleq [\![\mathsf{r}_1]\!];\ [\![\mathsf{r}_2]\!]$
$[\![\mathsf{while}_R\ \langle \mathsf{b}_1 \mid \mathsf{b}_2 \rangle\ \mathsf{r}\ ]\!] \triangleq$
$\quad \mathsf{while}\ ([\![\mathsf{b}_1]\!]_L\ \&\&\ [\![\mathsf{b}_2]\!]_R)\ [\![\mathsf{r}]\!]$
$[\![\mathsf{if}_R\ \langle \mathsf{b}_1 \mid \mathsf{b}_2 \rangle\ \mathsf{then}\ \mathsf{r}_1\ \mathsf{else}\ \mathsf{r}_2]\!] \triangleq$
$\quad \mathsf{if}\ ([\![\mathsf{b}_1]\!]_L\ \&\&\ [\![\mathsf{b}_2]\!]_R)\ \mathsf{then}\ [\![\mathsf{r}_1]\!]\ \mathsf{else}\ [\![\mathsf{r}_2]\!]$

Fig. 9. Reification of CoreRel into Imp

Theorem 3.3 (Reification preserves Equivalence). *Any equivalent pair of aligned programs r_1 and r_2 have equivalent product programs, $[\![\mathsf{r}_1]\!]$ and $[\![\mathsf{r}_2]\!]$:*
$\mathsf{r}_1 \equiv \mathsf{r}_2 \implies \forall \sigma\, \sigma'.\ \sigma, [\![\mathsf{r}_1]\!] \Downarrow \sigma' \Leftrightarrow \sigma, [\![\mathsf{r}_2]\!] \Downarrow \sigma'$

A direct consequence of Theorems 3.2 and 3.3 is that we can reduce the relational verification problem to reasoning about an equivalent product program:

Corollary 3.4. *Given a pair of Imp programs, c_1 and c_2, in order to prove that c_1 and c_2 are safe with respect to a pair of relational pre- and postconditions Φ and Ψ, it suffices to prove that an equivalent product program r is safe:* $\langle \mathsf{c}_1 | \mathsf{c}_2 \rangle \equiv \mathsf{r} \wedge\ \vdash \{\Phi\}\ [\![\mathsf{r}]\!]\ \{\Psi\} \implies \models_R \{\Phi\}\ \mathsf{c}_1 \otimes \mathsf{c}_2\ \{\Psi\}$

Unfortunately, this corollary does not provide any guidance on which 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, as we saw in Section 2. We now turn to the problem of how to construct such an r automatically.

$\langle c_1 \mid c_2 \rangle$	$\equiv$	$\langle c_1] \fatsemi [c_2 \rangle$	REL-DEF
$\langle c_1;\ c_2]$	$\equiv$	$\langle c_1] \fatsemi \langle c_2]$	HOM-L
$[c_1;\ c_2 \rangle$	$\equiv$	$[c_1 \rangle \fatsemi [c_2 \rangle$	HOM-R
$\langle \texttt{while}\ b\ c\ ]$	$\equiv$	$\langle \texttt{if}\ b\ \texttt{then}\ c;\ \texttt{while}\ b\ c]$	UNROLL-L
$\langle c_1] \fatsemi [c_2 \rangle$	$\equiv$	$[c_2 \rangle \fatsemi \langle c_1]$	REL-COMM
$r_1 \fatsemi (r_2 \fatsemi r3)$	$\equiv$	$(r_1 \fatsemi r_2) \fatsemi r3$	REL-ASSOC
$\langle \texttt{while}\ b_1\ c_1 \mid \texttt{while}\ b_2\ c_2 \rangle$	$\equiv$	$\texttt{while}_{\texttt{St}}\ n\ m\ \langle b_1 \mid b_2 \rangle\ \langle c_1 \mid c_2 \rangle \fatsemi$ $\langle\ \texttt{while}\ b_1\ c_1\ ] \fatsemi [\ \texttt{while}\ b_2\ c_2 \rangle$	WHILE-ALIGN
$\langle \texttt{if}\ b_1\ \texttt{then}\ c_1\ \texttt{else}\ c_2$ $\mid \texttt{if}\ b_2\ \texttt{then}\ c_3\ \texttt{else}\ c_4 \rangle$	$\equiv$	$\texttt{if}_{\texttt{R}}\ \langle b_1 \mid b_2 \rangle\ \texttt{then}\ \langle c_1 \mid c_3 \rangle$ $\texttt{else}\ \texttt{if}_{\texttt{R}}\ \langle b_1 \mid \texttt{not}\ b_2 \rangle\ \texttt{then}\ \langle c_1 \mid c_4 \rangle$ $\texttt{else}\ \texttt{if}_{\texttt{R}}\ \langle \texttt{not}\ b_1 \mid b_2 \rangle\ \texttt{then}\ \langle c_2 \mid c_3 \rangle$ $\texttt{else}\ \langle c_2 \mid c_4 \rangle$	IF-ALIGN
$\texttt{while}_{\texttt{R}}\ \langle b_1 \mid b_2 \rangle\ r$	$\equiv$	$\texttt{if}_{\texttt{R}}\ \langle b_1 \mid b_2 \rangle\ \texttt{then}\ r\ \texttt{else}\ \langle \texttt{skip} \mid \texttt{skip} \rangle \fatsemi$ $\texttt{while}_{\texttt{R}}\ \langle b_1 \mid b_2 \rangle\ r$	UNROLL-BOTH
$\langle \texttt{if}\ b_1\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \mid c_3 \rangle$	$\equiv$	$\texttt{if}_{\texttt{R}}\ \langle b_1 \mid \texttt{true} \rangle\ \texttt{then}\ \langle c_1 \mid c_3 \rangle\ \texttt{else}\ \langle c_2 \mid c_3 \rangle$	COND-L
$\langle c_1 \mid \texttt{if}\ b_1\ \texttt{then}\ c_2\ \texttt{else}\ c_3 \rangle$	$\equiv$	$\texttt{if}_{\texttt{R}}\ \langle \texttt{true} \mid b_1 \rangle\ \texttt{then}\ \langle c_1 \mid c_2 \rangle\ \texttt{else}\ \langle c_1 \mid c_3 \rangle$	COND-R

Fig. 10. Selected CoreRel realignment laws

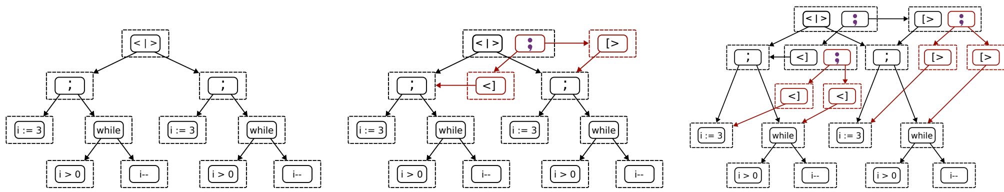
4 RELATIONAL REALIGNMENT VIA EQUALITY SATURATION

Observing that program equivalence is a congruence relation, we frame the search for a good product 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. 10 provides several equivalences that we can use to explore the space of possible alignments. Our notion of equivalence naturally admits any equivalences on Imp programs. It is sound to unroll one iteration of a loop on one side of an aligned term, for example (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 product programs: $[\![\langle c_1] \fatsemi [c_2\rangle]\!] \coloneqq c_1;\ c_2$, while $[[\![c_2\rangle \fatsemi \langle c_1]]\!] \coloneqq c_2;\ c_1$. A similar rule over sequences of Imp commands $c_1;\ c_2 \equiv c_2;\ c_1$ is obviously incorrect in the general case, as c_1 and c_2 may modify the same variables.

The while-align rule is particularly important, as it merge two loops together so that their bodies execute in lockstep. Note that since $\texttt{while}_{\texttt{R}}$ 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 justifies the shape of if-align.

4.1 E-Graphs

E-Graphs are a union find [34] data structure which efficiently represents a congruence relation between a set of terms in some language. When the underlying language is a programming language, e-graphs can be used to compactly represent a (potentially exponential) number of programs. An e-graph representation of a (set of) programs lifts AST nodes to *e-nodes* whose children are *e-classes*, i.e., a set of equivalent e-nodes. Since the members of an e-class are equivalent by construction, different selection of elements from the child e-classes of an e-node correspond to different (but equivalent) programs. Thus, *extracting* a particular program from a set of programs encoded as an



(a) (b) (c)

Fig. 11. E-graphs representing the space of alignments that result from applying the rewrite rules from Fig. 10 to the aligned term `⟨i := 3; while (i > 0) { i--; } | j := 3; while (j > 0) { j --; }⟩`. 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.

e-graph amounts to recursively selecting a representative member for each of the children of its e-nodes, starting from the root of the e-graph.

To build an e-graph which represents a space of potential rewritings of an aligned term, we start by constructing an e-graph that contains each AST node of the original term in a separate e-class. The structure of the e-graph mirrors the structure of the term's AST, but with each e-node pointing to the *e-class* of the node's original children. For example, given the CoreRel term `⟨i := 3; while (i > 0) { i--; } | j := 3; while (j > 0) { j --; }⟩`, Fig. 11(a) depicts the initial e-graph constructed over the term's AST. Note that there is only one term which can be constructed from this e-graph (the original term), as each equivalence class contains only a single node.

To apply a rewrite rule to an e-graph, we first pattern match the left-hand side of the rule against all e-nodes in the e-graph. We then add a new node to the e-class containing the matching node corresponding to the right-hand side of the rewrite rule. The process of repeatedly applying rewrite rules to an e-graph until a fixpoint or some finite bound is reached is called *equality saturation*. In order to compactly represent a space of possible realignments of a CoreRel term, we insert its naïve embedding into an e-graph and apply equality saturation using the CoreRel realignment rules.

Example 4.1. As an example, the root node of Fig. 11(a) matches the left-hand side of the REL-DEF rule from Fig. 10, with c_1 and c_2 corresponding to the root left and right e-classes, respectively. Fig. 11(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 grow the set of equivalent programs it represents further. Fig. 11(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 what is likely the desired alignment, `⟨i := 3; j := 3⟩⁏ while`$_R$ `⟨i > 0 | j > 0⟩ ⟨i-- | j-- ⟩`.

5 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.

$$\frac{}{\sigma,\ \texttt{skip} \Downarrow \sigma_1 \dashv \epsilon}\ \textsc{E-Skip} \qquad \frac{\sigma,\ \texttt{s}_1 \Downarrow \sigma' \dashv \alpha_1 \qquad \sigma',\ \texttt{s}_2 \Downarrow \sigma'' \dashv \alpha_2}{\sigma,\ \texttt{s}_1;\ \texttt{s}_2 \Downarrow \sigma'' \dashv \alpha_1 +\!\!+ \alpha_2}\ \textsc{E-Seq}$$

$$\frac{\sigma,\ \texttt{b} \Downarrow \texttt{true} \qquad \sigma,\ \texttt{s}_1 \Downarrow \sigma' \dashv \alpha}{\sigma,\ \texttt{if b then s}_1\ \texttt{else s}_2 \Downarrow \sigma' \dashv \alpha}\ \textsc{E-IfT} \qquad \frac{\sigma,\ \texttt{b} \Downarrow \texttt{true} \quad \sigma,\ \texttt{s} \Downarrow \sigma' \dashv \alpha_1 \qquad \sigma',\ \texttt{while b s} \Downarrow \sigma'' \dashv \alpha_2}{\sigma,\ \texttt{while}_\texttt{R}\ \texttt{b s} \Downarrow \sigma'' \dashv \texttt{wH}(\sigma) +\!\!+ \alpha_1 +\!\!+ \alpha_2}\ \textsc{E-WhileT}$$

$$\frac{\sigma,\ \texttt{b} \Downarrow \texttt{false} \qquad \sigma,\ \texttt{s}_2 \Downarrow \sigma' \dashv \alpha}{\sigma,\ \texttt{if b then s}_1\ \texttt{else s}_2 \Downarrow \sigma' \dashv \alpha}\ \textsc{E-IfT} \qquad \frac{\sigma,\ \texttt{b} \Downarrow \texttt{false}}{\sigma,\ \texttt{while b s} \Downarrow \sigma \dashv \texttt{wE}(\sigma)}\ \textsc{E-WhileF}$$

Fig. 12. Big-step semantics of Imp

Before we present our approach to extraction, however, we first need to define what constitutes a "good" alignment. The ultimate answer, of course, is that a good alignment is one that produces a product program that can be easily verified; a bad one does not. Verification is too expensive to use as an oracle for 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 simlarity between the loops, as both loop conditions became false at the same time. However, this semantic property is not obvious from the syntactic form the alignment alone. Our solution is to combine a syntactic extraction strategy with a *data-driven* approach [14, 28, 31, 39] that examines concrete executions of a candidate alignment to approximate its *semantic* fitness.

5.1 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. The big-step semantics of both CoreRel and Imp (given in Fig. 8 and Fig. 12, respectively) produce both a final state and a *trace* of intermediate states. Traces include relational and individual program states tagged with one of the following identifiers:

(1) rB, rM, and rE are used to mark the states at the entry, midpoint, and end of a pair of aligned Imp programs ($\langle c_1 \mid c_2 \rangle$),
(2) $\texttt{wH}_R$ and $\texttt{wE}_R$ tag states at, respectively, the start and exit of each relational loop iteration ($\texttt{while}_\texttt{R}$), and
(3) wH and wE do the same for standard Imp loops (while).

Example 5.1. $\langle \texttt{i}_1 := 3 \mid \texttt{i}_2 := 2 \rangle\,;\ \texttt{while}_\texttt{R}\ \langle \texttt{i}_1 > 0 \mid \texttt{i}_2 > 0 \rangle\ \langle \texttt{i}_1\texttt{--;} \mid \texttt{i}_2\texttt{--;} \rangle$ emits the following trace when executed with a pair of empty initial states:

$\texttt{rB}\langle\{\},\{\}\rangle,\ \texttt{rM}\langle\{\texttt{i}_1\mapsto 3\},\{\}\rangle,\ \texttt{rE}\langle\{\texttt{i}_1\mapsto 3\},\{\texttt{i}_2\mapsto 2\}\rangle,$ – initial $\langle\ \mid\ \rangle$
$\texttt{wH}_R\langle\{\texttt{i}_1\mapsto 3\},\{\texttt{i}_2\mapsto 2\}\rangle,\ \texttt{rB}\langle\{\texttt{i}_1\mapsto 3\},\{\texttt{i}_2\mapsto 2\}\rangle,\ \texttt{rM}\langle\{\texttt{i}_1\mapsto 2\},\{\texttt{i}_2\mapsto 2\}\rangle,\ \texttt{rE}\langle\{\texttt{i}_1\mapsto 2\},\{\texttt{i}_2\mapsto 1\}\rangle,$ – first iteration of $\texttt{while}_\texttt{R}$
$\texttt{wH}_R\langle\{\texttt{i}_1\mapsto 2\},\{\texttt{i}_2\mapsto 1\}\rangle,\ \texttt{rB}\langle\{\texttt{i}_1\mapsto 2\},\{\texttt{i}_2\mapsto 1\}\rangle,\ \texttt{rM}\langle\{\texttt{i}_1\mapsto 1\},\{\texttt{i}_2\mapsto 1\}\rangle,\ \texttt{rE}\langle\{\texttt{i}_1\mapsto 1\},\{\texttt{i}_2\mapsto 0\}\rangle,$ – second iteration of $\texttt{while}_\texttt{R}$
$\texttt{wE}_R\langle\{\texttt{i}_1\mapsto 1\},\{\texttt{i}_2\mapsto 0\}\rangle,$ – exiting $\texttt{while}_\texttt{R}$

Example 5.2. $\langle\ \texttt{i := 0} \mid \texttt{j := 2; while (j > 0) \{ j--; \}}\ \rangle$ emits the following execution trace:

$$\texttt{rB}\langle\{\},\{\}\rangle,\ \texttt{rM}\langle\{\texttt{i}\mapsto 0\},\{\}\rangle,\ \texttt{wH}(\{\texttt{j}\mapsto 2\}),\ \texttt{wH}(\{\texttt{j}\mapsto 1\}),\ \texttt{wE}(\{\texttt{j}\mapsto 0\}),\ \texttt{rE}\langle\{\texttt{i}\mapsto 0\},\{\texttt{j}\mapsto 0\}\rangle$$

```
⟨y₁ ≔ 0; z₁ ≔ 2 * x₁ | y₂ ≔ 0; z₂ ≔ x₂⟩⨾
whileR ⟨z₁ > 0 | ⊤⟩ ⟨z₁--; y₁ ≔ y₁ + x₁ | skip ⟩⨾
whileR ⟨⊤ | z₂ > 0⟩ ⟨skip | z₂--; y₁ ≔ y₁ + x₁ ⟩⨾
[y₂ ≔ 2 * y₂⟩
```
(a)

```
⟨y₁ ≔ 0; z₁ ≔ 2 * x₁ | y₂ ≔ 0; z₂ ≔ x₂⟩⨾
whileR ⟨z₁ > 0 | z₂ > 0⟩
  ⟨z₁--; y₁ ≔ y₁ + x₁ | z₂--; y₁ = y₁ + x₁ ⟩⨾
[y₂ ≔ 2 * y₂⟩
```
(b)

Fig. 13. Suboptimal alignments of $\texttt{double}_1$ and $\texttt{double}_2$ from Section 2.

Intuitively, semantically similar components should modify variables in similar ways. We thus introduce a notion of a trace summary which describes how program variables change between different program points.

Definition 5.3. Let τ be a trace containing non-relational events $a\langle\sigma\rangle$ and $b\langle\sigma'\rangle$ at indices m and n, respectively. Then a summary $\Delta^{m\to n}$ is a mapping on $dom(\sigma')$ such that

$$\Delta^{m\to n}(x) = \begin{cases} \sigma'(x) - \sigma(x) & \text{if } x \in \sigma, \\ \sigma'(x) & \text{otherwise} \end{cases}$$

Similarly, if τ contains relational events $a\langle\sigma_1,\sigma_2\rangle$ and $b\langle\sigma_1',\sigma_2'\rangle$ at indices m and n, respectively, then $\Delta_1^{m\to n}$ and $\Delta_2^{m\to n}$ are mappings over $dom(\sigma_1')$ and $dom(\sigma_2')$, respectively, where

$$\Delta_1(x) = \begin{cases} \sigma_1'(x) - \sigma_1(x) & \text{if } x \in \sigma_1, \\ \sigma_1'(x) & \text{otherwise} \end{cases} \qquad \Delta_2(x) = \begin{cases} \sigma_2'(x) - \sigma_2(x) & \text{if } x \in \sigma_2, \\ \sigma_2'(x) & \text{otherwise} \end{cases}$$

For example, the initial relational block `⟨i ≔ 3 | j ≔ 2⟩` in Example 5.1 can be summarized as $\Delta_1^{0\to 2} = \{i \mapsto 3\}$ and $\Delta_2^{0\to 2} = \{j \mapsto 2\}$, while the effect of the `while` loop in Example 5.2 can be summarized as $\Delta^{2\to 4} = \{j \mapsto -2\}$.

5.2 Comparing Alignments

Before describing a 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 $\texttt{double}_{1\times 2}$. Consider what features of the traces generated by $\texttt{double}_{1\times 2}$ indicate that it should be preferred over $\langle\texttt{double}_1|\texttt{double}_2\rangle$.

One immediate difference is that $\langle\texttt{double}_1|\texttt{double}_2\rangle$ contains the entirety of both program executions in a single aligned term. This will manifest in traces that contain a single `rB` at index 0 and a single `rE` at the last index n. Contrast this with $\texttt{double}_{1\times 2}$'s traces, whose `rB` and `rE` events appear much more frequently and closer together. Alignments like $\langle\texttt{double}_1|\texttt{double}_2\rangle$ which group many instructions together do not give us many opportunities to realign smaller subprograms. As Example 4.1 demonstrated, it is often useful to apply the REL-DEF, HOM-L, and HOM-R rules to decompose an alignment into smaller pieces that can be rearranged. In order to guide our extraction mechanism towards these sorts of alignments, a cost function should favor alignments whose traces feature smaller gaps between `rB` and `rE` tags.

Another difference is that $\texttt{double}_{1\times 2}$ includes a relational loop, $\texttt{while}_\texttt{R}$, which manifests in its execution traces as a sequence of $\texttt{wH}_R$'s followed by a $\texttt{wE}_R$: $\texttt{wH}_R\langle\sigma_1,\sigma_2\rangle$, $\texttt{wH}_R\langle\sigma_1',\sigma_2'\rangle, \ldots,$ $\texttt{wE}_R\langle\sigma_1'',\sigma_2''\rangle$. Note that the pair of runoff `while` loops in the reified program never execute, and thus never affect the *semantic* trace despite appearance in the *syntax* of $\texttt{double}_{1\times 2}$. In contrast, the trace of $\langle\texttt{double}_1|\texttt{double}_2\rangle$ contains *only* non-relational `wH`'s. While this suggests a straightforward heuristic of preferring traces with more relational loop tags, consider the (somewhat contrived) alignment given in Fig. 13(a). While this alignment features more loops, it does not allow a verifier to leverage the similarities between the loops of $\texttt{double}_1$ and $\texttt{double}_2$. This manifests in the traces of the correct alignments, where each pair of successive $\texttt{wH}_R$'s at indices m and n will have the property $dom(\Delta_1^{m\to n}) = dom(\Delta_2^{m\to n})$. In other words, the relational loop updates the same values on the left and right side in a similar way, a property which the trace of the program in Fig. 13(a) will not have. This suggests a improved heuristic of preferring relational loops that modify variables from $\texttt{double}_1$ and $\texttt{double}_2$ in similar ways.

Finally, consider the suboptimal alignment shown in Fig. 13(b). While this is close to $\texttt{double}_{1\times 2}$, it does not properly stutter the body of the relational loop using $\texttt{while}_{\texttt{St}}$. This manifests in a trace that includes several of wH's that are generated by the lefthand "runoff" loop after the relational loop has ended ($\texttt{wE}_R$). This suggests another straightforward strategy of favoring traces with fewer runoff loop executions.

While not exhaustive, the above discussion illustrates how execution traces may be used to quantify desirable alignments. While our extraction algorithm is parameterized over the particular cost function, so that it is possible to use functions which prefer different alignment features, the implementation presented in Section 6 scores traces across several dimensions:

- **Relational block size**, preferring relational blocks which modify fewer variables.
- **Relational block symmetry**, preferring relational blocks which update the same variables on both sides.
- **Loop head matching**, preferring loops which update the same variables on both sides.
- **Loop update linearity**, preferring loops which change variables by the same amount on each iteration.
- **Loop executions**, preferring loops which execute fewer times. This especially favors alignments whose "runoff" loops never execute.

5.3 Data-Driven Extraction of Product Programs

Our complete algorithm for constructing aligned product programs is given in Algorithm 1. The algorithm takes as input two programs p_1 and p_2, a `Cost` function over candidate alignments, and a parameter μ bounding the number of candidates our data-driven extraction phase should consider. 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 5.

Algorithm 1: KestRel

Inputs : p_1 and p_2: programs,
`Cost`: cost function over alignments,
μ: number of SA iterations

Output : product program $p_1 \times p_2$

1 **begin**
2 $\quad E \leftarrow$ `CreateEGraph()`
3 $\quad$ `Insert`$(E, \langle p_1 | p_2 \rangle)$
4 $\quad$ `EQSat`$(E,$ CoreRel$)$
5 $\quad best \leftarrow$ `ExtractLocal`(E)
6 $\quad \hat{\eta} \leftarrow$ `Cost`$(best)$
7 $\quad$ **for** $k \leftarrow 0$ *to* μ **do**
8 $\quad\quad \tau \leftarrow$ `Temperature`(k, μ)
9 $\quad\quad N \leftarrow$ `RandomNeighbor`$(E, best)$
10 $\quad\quad \eta \leftarrow$ `Cost`(N)
11 $\quad\quad$ **if** $\eta < \hat{\eta} \vee$ `Jump`$(\tau, best, \hat{\eta}, N, \eta)$ **then**
12 $\quad\quad\quad (best, \hat{\eta}) \leftarrow (N, \eta)$
13 $\quad$ **return** `Reify`$(best)$

Lines 1 — 3 create a new e-graph from the initial embedding of the input programs, $\langle p_1 | p_2 \rangle$, and then uses equality saturation (Line 4) with a collection of realignment rules to build the set of candidate alignment. The algorithm then uses a standard local

$$\llbracket \langle s_1 \mid s_2 \rangle \rrbracket^{I} \triangleq \mathtt{log}(\mathtt{relBegin});\ \llbracket s_1 \rrbracket^{I}_{L};\ \mathtt{log}(\mathtt{relMid});\ \llbracket s_2 \rrbracket^{I}_{R};\ \mathtt{log}(\mathtt{relEnd})$$

$$\llbracket r_1 \,;\, r_2 \rrbracket^{I} \triangleq \llbracket r_1 \rrbracket^{I};\ \llbracket r_2 \rrbracket^{I}$$

$$\llbracket \mathtt{while}_R\ \langle b_1 \mid b_2 \rangle\ r\ \rrbracket^{I} \triangleq \mathtt{log}(\mathtt{whileBegin});\ \mathtt{while}\ (\llbracket b_1 \rrbracket^{I}_{L}\ \&\&\ \llbracket b_2 \rrbracket^{I}_{R})\ \{\ \mathtt{log}(\mathtt{loopHead});\ \llbracket r \rrbracket^{I} \};\ \mathtt{log}(\mathtt{whileEnd})$$

$$\llbracket \mathtt{if}_R\ \langle b_1 \mid b_2 \rangle\ \mathtt{then}\ r_1\ \mathtt{else}\ r_2 \rrbracket^{I} \triangleq \mathtt{if}\ (\llbracket b_1 \rrbracket^{I}_{L}\ \&\&\ \llbracket b_2 \rrbracket^{I}_{R})\ \mathtt{then}\ \llbracket r_1 \rrbracket^{I}\ \mathtt{else}\ \llbracket r_2 \rrbracket^{I}$$

Fig. 14. Instrumented reification of CoreRel into Imp

cost function, `ExtractLocal`, that minimizes the number of loops in each e-class to extract an initial candidate program (Line 5), and then computes its `Cost` (Line 6). The algorithm then proceeds to the data-driven extraction phase, which is implemented as a simulated annealing loop (Lines 7–12) that leverages the e-graph to provide candidate alignments by perturbing an alignment's selection of e-class representative nodes. On each iteration, we calculate the temperature, use the e-graph to select a neighbor of the current candidate, *best*, and calculate the cost of that neighbor. The `Temperature` function may implement any cooling schedule; our implementation uses a linear schedule. If the neighbor's `Cost` is lower than the current candidate alignment, we adopt it as our new candidate alignment. The algorithm will also accept a neighbor with a higher `Cost` with some probability that decreases as the temperature lowers. After this loop finishes, the algorithm returns the current candidate alignment.

6 IMPLEMENTATION

We have implemented a relational verification engine based on Algorithm 1, called KestRel. KestRel is written in Rust and uses the Egg library [38] 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 `struct`s, for example) and backends for outputting product programs in Dafny and C (the latter is used to target SeaHorn). To improve performance, 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 product program produced by this phase is reported to the user.

Equality Saturation Optimizations. To ensure KestRel can efficiently saturate its e-graph, it optimizes its representation of the space of alignments in several ways. Firstly, basic blocks whose internal realignment cannot impact the verifiability of the product 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. Secondly, we limit the number of duplicate loop bodies in product programs by capping the arguments to $\mathtt{while}_{St}$ to 2. This limit can be relaxed or eliminated at the cost of increasing the number of candidate alignments.

Instrumentation. In order to generate traces during its data-driven extraction phase, KestRel produces instrumented product programs that are augmented with `log` commands which record the intermediate states used in traces. The instrumented variant of $\llbracket\cdot\rrbracket$, shown in Fig. 14, adds appropriate logging commands to the beginnings, middles, and ends of relations; the beginning and ends of loops; and entry points of loop heads. Fig. 15 shows the instrumented variant of $\langle \mathtt{double}_1 | \mathtt{double}_2 \rangle$ produced

```
log(relBegin);
y1 := 0; z1 := 2 * x1;
log(whileBegin); while (z1 > 0) {
  log(loopHead); z1--; y1 += x1;
} log(whileEnd);
log(relMid);
y2 := 0; z2 := x2;
log(whileBegin); while (z2 > 0) {
  log(loopHead); z2--; y2 += x2;
} log(whileEnd);
y2 := 2 * y2;
log(relEnd);
```

Fig. 15. Output of $\llbracket\langle \mathtt{double}_1 | \mathtt{double}_2 \rangle\rrbracket^{I}$.

by this function. 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 [10].

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

Invariant Inference. After constructing a candidate product program, the Dafny backend of KestRel uses Daikon [14] along with Houdini-style[15] iterative refinement to automatically infer basic loop invariants. For cases where this invariant inference is insufficient, KestRel allows users to provide additional hints about possible relational loop invariants.

7 EVALUATION

Our expermintal evaluation investigates four key questions regarding our approach to relational verification:

RQ1 Is our approach *effective*, i.e., does KestRel enable existing verification tools to be used for relational reasoning?
RQ2 Is our approach *expressive* enough to verify a diverse set of programs and relational properties?
RQ3 Is our approach *efficient*? Is KestRel able to find useful product programs within a reasonable time frame?
RQ4 Is our approach *general*? Can KestRel build product programs suitable for multiple backend verifiers?

To answer these questions, we evaluate KestRel on a diverse corpus of benchmarks[2] that includes programs drawn from the relational verification literature, and incorporates examples from both product program- and relational logic-based approaches [2, 3, 33, 36]. 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.

[2]All the benchmarks and results from our evaluation are provided in the supplementary material.

Table 1. Comparing naïve alignments with the alignments produced by KestRel. The top 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. [2] (†), Barthe et al. [3] (★), Sousa and Dillig [33](°), Shemer et al. [32] (◇), and Cormen et al. [11] (□). For benchmarks with basic types, the **Loops** column indicates the presence of a loop in the benchmark. For clients of ADTs, the **ADTs** column lists the ADTs used. The **Property** column gives the relational property of interest. The **Unaligned** and **Aligned** columns indicate verification times in seconds for, respectively, the naïve product program and the program produced by KestRel. A red ✖ indicates verification failure and a green ✔ indicates success. The simulated annealing phase of program extraction is capped at 12000 iterations. The **Inv** column contains the number of loop invariant hints given to KestRel.

Benchmark	Loops	Property	Unaligned	Aligned	Inv
commute	✔	commutativity	✖ 6.47	✔ 4.68	
data-alignment†	✔	monotonicity	✖ 5.98	✖ 18.36	
half-square◇	✔	noninterference	✖ 12.01	✔ 12.38	2
payments	✔	equivalence	✖ 8.59	✔ 6.71	
simple†	✔	equivalence	✖ 4.16	✔ 2.48	1
strength-reduction★	✔	equivalence	✖ 7.76	✔ 5.88	1
square-sum◇	✔	equivalence	✖ 10.98	✔ 4.27	
unroll★	✔	equivalence	✖ 5.95	✔ 23.57	
col-item°	✖	anticommutativity	✔ 2.40	✔ 2.38	
container°	✖	anticommutativity	✔ 2.49	✔ 2.44	
file-item°	✖	anticommutativity	✔ 2.34	✔ 2.35	
match°	✖	anticommutativity	✔ 2.38	✔ 2.38	
node°	✖	anticommutativity	✔ 2.35	✔ 2.37	

Benchmark	ADTs	Property	Unaligned	Aligned	Inv
array-insert†	kvstore	equivalence	✖ 12.64	✔ 15.00	
array-int°	kvstore	anticommutativity	✖ 13.52	✔ 15.92	
bst-min-search□	bst	monotonicity	✖ 6.17	✔ 4.36	
bst-sum□	bst	monotonicity	✖ 6.49	✔ 6.51	
bubble-sort★	kvstore	robustness	✖ 21.67	✔ 17.48	4
chromosome°	kvstore	anticommutativity	✖ 13.29	✔ 13.46	
code-sinking★	kvstore	equivalence	✖ 10.80	✔ 9.02	17
linked-list-ni	list	noninterference	✖ 10.10	✔ 26.90	4
list-array-sum□	kvstore, list	equivalence	✖ 6.70	✔ 4.83	
list-length□	list	equivalence	✖ 3.74	✔ 3.14	
loop-alignment★	kvstore	equivalence	✖ 10.78	✔ 34.66	3
loop-pipelining★	kvstore	equivalence	✖ 9.94	✔ 35.12	7
loop-tiling†	kvstore	equivalence	✖ 12.42	✖ 40.20	
loop-unswitching★	kvstore	equivalence	✖ 7.51	✔ 5.42	3
static-caching★	kvstore	equivalence	✖ 19.51	✖ 77.42	

7.1 Enabling Relational Verification

Our first set of experiments explores whether KestRel is able to identify alignments that enable verifiers to exploit semantic similarities between programs **RQ1**. Recall from Section 2 that a straightforward way to construct a product program is to simply concatenate multiple programs together, after α-renaming variables to ensure distinct namespaces. Our experiments use these *unaligned* programs as the baseline, as this naïve alignment strategy does not perform any loop fusion, unrolling, or other transformations that expose similarities between programs to the verifier. We verified both naïve and aligned product programs using our Dafny backend, as its module system natively supports reasoning about clients of ADTs with algebraic specifications.

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

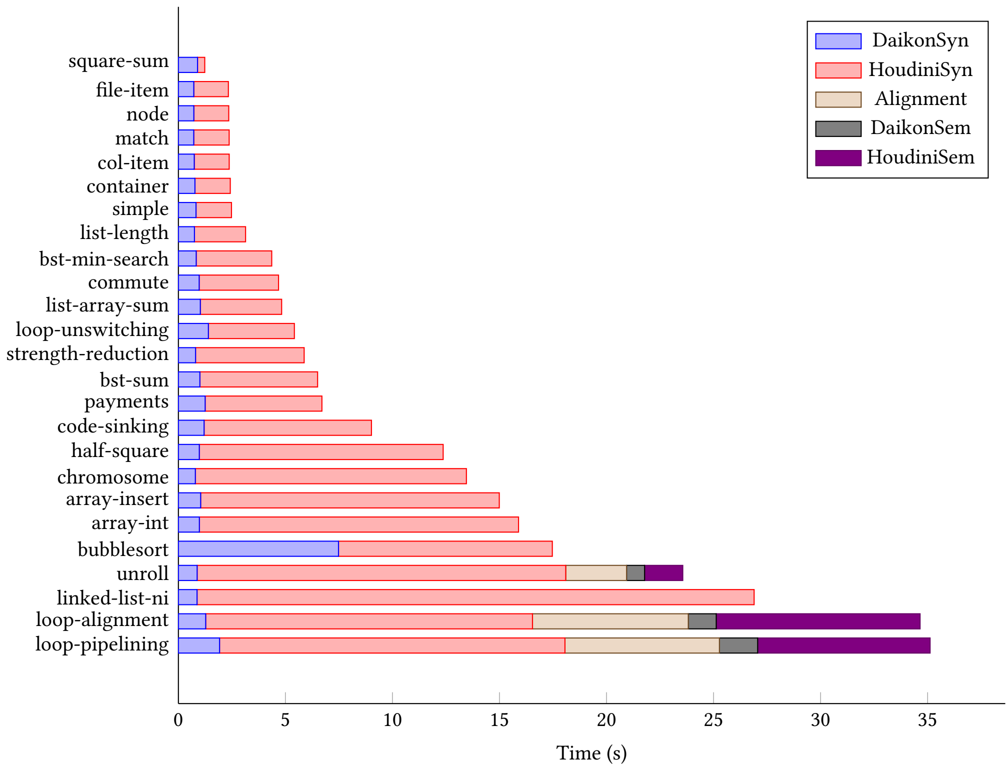


Fig. 16. Breakdown of KESTREL runtimes by subtask. "DaikonSyn" refers to generating initial invariant candidates for syntactic extraction, "HoudiniSyn" refers to elimination of invalid invariant candidates for syntactic extractions, and "Alignment" refers to semantic extraction of a product program from the e-graph. "DaikonSem" and "HoudiniSem" refer to the analagous invariant inference tasks over semantic extractions. Subtasks which take negligable amounts of time (for example, extracting an initial syntactic product from the e-graph) are not depicted.

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 the unaligned versions of all five loop-free benchmarks. Nevertheless these 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 unaligned versions of the remaining 23 benchmarks, suggesting that a better alignment could be beneficial. Indeed, for all but three of these benchmarks, Dafny was able to verify the aligned product programs produced by KESTREL (**RQ1**). Our pipeline was able to verify over half of these benchmarks (16) fully automatically; the remaining 9 required us to provide hints about additional predicates that needed inclusion in the set of candidate loop invariants. In addition, verification was relatively efficient, finishing in under 30 seconds in most cases (**RQ3**). For most of these benchmarks, invariant inference dominates the total runtime; Fig. 16 presents per-subtask timings for the individual componets of the KESTREL pipeline.

Two of the five 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 interations (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.

7.2 Impact of the Extraction Method

As discussed in Sections 5 and 6, KestRel uses a two-phase approach to extract a product program from the space of alignments. The first phase uses a local, syntactic cost function to quickly identify promising alignments, while the second uses a slower non-local cost function that uses data-driven simulated annealing to examine semantic properties of potential product alignments. KestRel uses the former technique to quickly arrive at a reasonable starting alignment, and the latter to refine that starting point into an alignment with desirable semantic behavior.

To demonstrate the utility of our combined approach to extraction, we perform an ablation study which uses each extraction method individually to construct an aligned product program. This experiment uses the benchmarks in Table 1 that contain loops and which KestRel 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 simulated annealing, starting from a random point in the alignment space ("Semantic"). Fig. 17 presents the results of this experiment.

Benchmark	Syntactic	Semantic	Combined
array-insert	✔ 13.64	✔ 25.67	✔ 15.00
array-int	✔ 20.03	✖ 25.04	✔ 15.92
bst-min-search	✔ 4.33	✖ 7.98	✔ 4.36
bst-sum	✔ 6.55	✔ 9.79	✔ 6.51
bubble-sort	✔ 17.86	✖ 15.40	✔ 17.48
chromosome	✔ 15.04	✖ 4.54	✔ 13.46
code-sinking	✔ 9.06	✖ 32.31	✔ 14.02
col-item	✔ 2.64	✖ ∞	✔ 2.38
commute	✔ 4.78	✔ 7.61	✔ 4.68
container	✔ 2.81	✖ ∞	✔ 2.44
file-item	✔ 2.45	✔ 29.51	✔ 2.35
half-square	✔ 12.38	✔ 10.57	✔ 12.38
linked-list-ni	✔ 27.52	✖ 17.65	✔ 26.90
list-array-sum	✔ 4.70	✔ 6.70	✔ 4.83
list-length	✔ 3.17	✖ ∞	✔ 3.14
loop-alignment	✖ 16.61	✖ 26.25	✔ 34.66
loop-pipelining	✖ 19.88	✔ 18.68	✔ 35.12
loop-unswitching	✔ 7.31	✖ 36.63	✔ 5.42
match	✔ 2.43	✔ 49.39	✔ 2.38
node	✔ 2.43	✔ 5.93	✔ 2.37
payments	✔ 6.71	✖ 18.51	✔ 6.71
simple	✔ 2.49	✔ 3.84	✔ 2.48
strength-reduction	✔ 6.77	✖ 41.42	✔ 5.88
square-sum	✔ 4.33	✔ 5.26	✔ 4.27
unroll	✖ 21.28	✔ 4.80	✔ 23.57

Fig. 17. Results of an ablation study over benchmarks with successful KestRel verifications. The "Syntactic" column lists verification results for programs obtained using just local extraction. The "Semantic" gives results for programs constructed via data-driven simulated annealing extractions starting from a random (instead of minloops-extracted) 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) was sufficient for verification. In some cases (e.g., `array-int`, `loop-unswitching`), this approach succeeded where data-driven simulated annealing failed; the likely cause is a bad random start in a large alignment space. 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 local minloops extraction is an effective starting point for KestRel's simulated annealing approach.

In three benchmarks, minloops alone was insufficient to produce a verifiable alignment. These represent cases where data-driven alignment is necessary to discover the semantic transformations that enable automatic verification. One of these benchmarks, `loop-alignment`, did not arrive at

Benchmark	Verification Time (s)		
	Unaligned	Syntactic	Combined
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. 18. Results from using KestRel's SeaHorn backend to verify a suite of array benchmarks to a suite of array benchmarks and. All times reported in seconds. A ∞ indicates the process did not finish within 10 minutes. A ✖ indicates SeaHorn was unable to verify, either due to timeout, inability to find a loop invariant, or inability to decide verification conditions. A ✔ indicates successful verification.

a working alignment when performing either local minloops extraction or simulating annealing from a random starting point, showcasing the benefits of KestRel's combined approach.

7.3 Relational Verification with SeaHorn

To demonstrate that KestRel's utiltiy is not restricted to a particular verifier (**RQ4**), we used KestRel to produce product programs for verification using SeaHorn [17], a popular tool for automatically verifying safety properties of C programs. We translated a subset of the benchmarks from Table 1 to C programs which use arrays. Currently, SeaHorn only supports reasoning about arrays up to a statically-known size, as opposed to, e.g., arbitrary key-value stores. The results of this evaluation are presented in Fig. 18. Combined with the results from our Dafny backend, these verification times suggest that the KestRel produces programs that can be efficiently verified (**RQ2**).

Fig. 18 reports verification times for unaligned 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. 18. The top and middle groups comprise benchmarks where SeaHorn was able to verify the product programs produced by KestRel, and the top group includes the cases where SeaHorn was not able to verify the naïve product program. For two of these benchmarks, `schemer` and `simple`, SeaHorn failed to verify the product program found by syntactic methods, while our data-driven approach was able to find product programs that successfully verified. Taken together, these results provide evidence that our approach can support multiple verification backends (**RQ4**).

Analagous to the previous experiment, the middle group of benchmarks SeaHorn was able to verify using only the naïve product program 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 use arrays with static sizes, affording it the opportunity to 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.[3] To verify these alignments are nevertheless valid, we manually verified each of these aligned product programs using VST [7]. For the cases verified with VST, invariants did not require specification of full functional correctness.

8 RELATED WORK

Relational Program Logics. Much recent work on relational verification has focused on custom relational program logics [5, 8, 13, 33, 36]. Relational alignment in these logics arise (often implicitly) from how the deductive rules are applied during verification. Cartesian Hoare Logic (CHL) [33], for example, provides a special set of rules for reasoning over loop executions in lock step, but in order to take advantage of these rules the verifier must step over program statements from different executions in a way that aligns related loops. The CHL verification algorithm uses lightweight syntactic heuristics to attempt to maximize opportunities for loop alignment, while our approach which uses a data-driven technique based on execution traces to find alignments. Chen et al. [8] use reinforcement learning to identify proof strategies that are likely to lead to a successful verification using their relational logic. While their approach uses machine learning to identify promising applications of relational proof rules for different classes of programs, e.g., programming assignments, our approach instead examines execution traces of individual programs on to identify alignments amenable to verification with existing tools. Unno et al. [36] make alignment constraints, which they call a "schedule", explicit in templated verification conditions, which are expressed in an extension to constrained Horn clauses (CHCs). Finding alignment then becomes a concern of a CEGIS-based verifier for this extended class of CHCs.

Aligned Product Programs. Barthe et al. [3, 4] present systems for soundly combining multiple programs into a single product program, although does not propose generic algorithms for constructing products with good alignments. Indeed, efficiently searching the large space of possible product programs for desirable alignments is one of the primary hurdles of the product program approach. Sharma et al. [31] perform data-driven equivalence checking of x86 loops by examining instrumented execution traces to find cutpoints where the loop bodies are likely to be related by simple invariants. Churchill et al. [9] describe a data-driven approach for proving equivalence between x86 programs by inferring predicates which relate trace elements, then lifting those predicates back to the source code level to construct a product program. Similar to our approach, both works use instrumented execution traces to identify promising alignments in x86 code, although their techniques are specialized to proving equivalence and use specialized equivalence verifier. Rather than identifying predicates over state traces and lifting them to an equivalence proof, our approach scores traces according to a cost function that is used to search a space of alignments encoded in an e-graph, and targets off-the-shelf verifiers as its backend.

[3]We did not adapt KestRel's loop invariant inference pipeline for SeaHorn output, and rely solely on SeaHorn for invariant inference.

E-Graph Extraction. Extracting a desirable term with a heuristic cost function is a core piece of optimization by equality saturation [35]. Using local cost functions to greedily select subterms is a common strategy, and forms the default extraction mechanism of the popular Egg library [38]. Wang et al. [37] 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.

9 CONCLUSION

Many interesting properties, such as observational equivalence and noninterference, are *relational*. That is, they are properties which relate multiple program executions. Reasoning about these properties requires finding alignments between programs so that verifiers are able to exploit their similarities. Without proper alignments, relational verification is often intractable. However, finding good alignments involves overcoming several challenges. The space of possible alignments is combinatorial in the size of the related programs, and many properties of desirable alignments require examining the semantic behavior of the aligned programs.

In this paper, we presented KestRel, a tool for constructing product programs by finding desirable alignments. To do this, KestRel compactly represents a space of possible alignments by embedding terms in an alignment algebra, expressing the embedding in an e-graph, and running equality saturation over rewrite rules from the algebra. It then uses a novel data-driven extraction technique to identify promising alignments from instrumented execution traces. Once a desirable alignment is found, KestRel reifies the algebraic term into a product program which can be verified by off-the-shelf single program verifiers. We have evaluated KestRel on a diverse suite of benchmarks and relational properties taken from the literature, using two off-the-shelf verifiers, Dafny and SeaHorn, to verify the product programs KestRel produces. Our experiments show that KestRel discovers alignments that enable verification to succeed where a naïve alignment strategy would otherwise fail.

REFERENCES

[1] 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.

[2] 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.

[3] Gilles Barthe, Juan Manuel Crespo, and César Kunz. 2011. 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.

[4] Gilles Barthe, Juan Manuel Crespo, and César Kunz. 2013. 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.

[5] 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.

[6] Raven Beutner and Bernd Finkbeiner. 2022. Software Verification of Hyperproperties Beyond k-Safety. In *Computer Aided Verification*, Sharon Shoham and Yakir Vizel (Eds.). Springer International Publishing, Cham, 341–362.

[7] 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

[8] 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.

[9] 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.

[10] 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

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

[12] Thibault Dardinier and Peter Müller. 2023. Hyper Hoare Logic: (Dis-)Proving Program Hyperproperties (extended version). arXiv:2301.10037 [cs.LO]

[13] 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.

[14] 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.

[15] 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.

[16] Robert W. Floyd. 1993. *Assigning Meanings to Programs*. Springer Netherlands, Dordrecht, 65–81. https://doi.org/10.1007/978-94-011-1793-7_4

[17] 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.

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

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

[20] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. RustBelt: Securing the Foundations of the Rust Programming Language. *Proc. ACM Program. Lang.* 2, POPL, Article 66 (dec 2017), 34 pages.

[21] 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.

[22] 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

[23] Ramana Nagasamudram, Anindya Banerjee, and David A. Naumann. 2023. Alignment complete relational Hoare logics for some and all. arXiv:2307.10045 [cs.LO]

[24] Ramana Nagasamudram and David A. Naumann. 2021. Alignment Completeness for Relational Hoare Logics. In *2021 36th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. 1–13. https://doi.org/10.1109/LICS52264.2021.9470690

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

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

[27] Peter W. O'Hearn. 2007. Resources, concurrency, and local reasoning. *Theoretical Computer Science* 375, 1 (2007), 271–307. Festschrift for John C. Reynolds's 70th birthday.

[28] 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

[29] Peter Poetzsch-Heffter, Arndand Müller. 1999. A Programming Logic for Sequential Java. In *Programming Languages and Systems*, S. Doaitse Swierstra (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 162–176.

[30] J.C. Reynolds. 2002. Separation logic: a logic for shared mutable data structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*. 55–74.

[31] 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

[32] 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.

[33] 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*. 57–69.

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

[35] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. 2009. Equality saturation: a new approach to optimization. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 264–276.

[36] 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.

[37] 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).

[38] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. 2021. egg: Fast and extensible equality saturation. *Proceedings of the ACM on Programming Languages* 5, POPL (2021).

[39] 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