Relational verification using product programs

Gilles Barthe, Juan Manuel Crespo, and César Kunz

IMDEA Software, Madrid, Spain

Abstract. Relational program logics are formalisms for specifying and verifying properties about two programs or two runs of the same program. These properties range from correctness of compiler optimizations or equivalence between two implementations of an abstract data type, to properties like non-interference or determinism. Yet the current technology for relational verification remains underdeveloped. We provide a general notion of product program that supports a direct reduction of relational verification to standard verification. We illustrate the benefits of our method with selected examples, including non-interference, standard loop optimizations, and a state-of-the-art optimization for incremental computation. All examples have been verified using the Why tool.

1 Introduction

Relational reasoning provides an effective mean to understand program behavior: in particular, it allows to establish that the same program behaves similarly on two different runs, or that two programs execute in a related fashion. Prime examples of relational properties include notions of simulation and observational equivalence, and 2-properties, such as non-interference and continuity. In the former, the property considers two programs, possibly written in different languages and having different notions of states, and establishes a relationship between their execution traces, whereas in the latter only one program is considered, and the relationship considers two executions of that program.

In spite of its important role, and of the wide range of properties it covers, there is a lack of applicable program logics and tools for relational reasoning. Indeed, existing logics [7, 23] are confined to reasoning about structurally equal programs, and are not implemented. This is in sharp contrast with the more traditional program logics for which robust tool support is available. Thus, one natural approach to bring relational verification to a status similar to standard verification is to devise methods that soundly transform relational verification tasks into standard ones. More specifically for specifications expressed using pre and post-conditions, one would aim at developing methods to transform Hoare quadruples of the form $\{\varphi\}\, c_1 \sim c_2\, \{\psi\}$, where φ and ψ are relations on the states of the command c_1 and the states of the command c_2, into Hoare triples of the form $\{\bar{\varphi}\}\, c\, \{\bar{\psi}\}$, where $\bar{\varphi}$ and $\bar{\psi}$ are predicates on the states of the command c, and such that the validity of the Hoare triple entails the validity of the original Hoare quadruple; using $\models$ to denote validity, the goal is to find c, $\bar{\varphi}$ and $\bar{\psi}$ such that

$$\models \{\bar{\varphi}\}\, c\, \{\bar{\psi}\} \quad \Rightarrow \quad \models \{\varphi\}\, c_1 \sim c_2\, \{\psi\}$$

Consider two simple imperative programs c_1 and c_2 and assume that they are separable, i.e. operate on disjoint variables. Then we can let assertions be first-order formulae over the variables of the two programs, and achieve the desired effect by setting $c \equiv c_1; c_2$, $\bar{\varphi} \equiv \varphi$ and $\bar{\psi} \equiv \psi$. This method, coined self-composition by Barthe, D'Argenio and Rezk [4], is sound and relatively complete, but it is also impractical [22]. In a recent article, Zaks and Pnueli [24] develop another construction, called cross-product, that performs execution of c_1 and c_2 in lockstep and use it for translation validation [25], a general method for proving the correctness of compiler optimizations. Cross-products, when they exist, meet the required property; however their existence is confined to structurally equivalent programs and hence they cannot be used to validate loop optimizations that modify the control flow of programs, nor to reason about 2-properties such as non-interference, because such properties consider runs of the program that do not follow the same control flow.

The challenge addressed in this paper is to provide a general notion of product programs which allows transforming relational verification tasks into standard ones, without the setbacks of cross-products or self-composition. In our setting, a product between two programs c_1 and c_2 is a program c which combines synchronous steps, in which instructions from c_1 and c_2 are executed in lockstep, with asynchronous steps, in which instructions from c_1 or c_2 are executed separately. Products combine the best of cross-products and self-composition: the ability of performing asynchronous steps recovers the flexibility and generality of self-composition, and make them applicable to programs with different control structures, whereas the ability of performing synchronous steps is the key to make the verification of c as effective as the verification of cross-products and significantly easier than the verification of the programs obtained by self-composition. Concretely, we demonstrate how product programs can be combined with off-the-shelf verification tools to carry relational reasoning on a wide range of examples, including: various forms of loop optimizations, static caching for incremental computation, SSE transformations for increasing performance on multi-core platforms, information flow and continuity analyses. All examples have been formally verified using the Why framework with its SMT back-end; in one case, involving complex summations on arrays, we used a combination of the SMT back-end and the Coq proof assistant back-end—however it is conceivable that the proof obligations could be discharged automatically by declaring suitable axioms in the SMT solver.

Contents. The paper is organized as follows: Section 2 introduces the product construction and shows the need of a generalization of cross-product and self-composition. Section 3 defines product programs and shows how they enable reducing relational verification to existing standard logics. Section 4 illustrates the usefulness of our method through examples drawn from several settings including non-interference and translation validation of loop optimizations [3]. In particular, we provide a formal proof of Static Caching [17], a challenging optimization used for incremental computation e.g. in image processing or computational geometry.

2 Motivating examples

Continuity is a relational property that measures the robustness of programs under changes: informally, a program is continuous if small variations on its inputs only causes small variations on its output. While program continuity is formalized by a formula of the form $\forall\,\epsilon > 0.\,\exists\,\delta > 0.\; P$, see e.g. [9], continuity can be often derived from the stronger notion of 1-sensitivity, see e.g. [21]. Informally, a program is 1-sensitive if it does not make the distance grow, i.e. the variation of the outputs of two different runs is upper bounded by the variation of the corresponding inputs.

Consider the standard bubble-sort algorithm shown at the left of Figure 1. Suppose that instead of the expected array a the algorithm is fed with an array a' satisfying the following relation: $|a[i] - a'[i]| < \epsilon$ for all i in the range of a and a' and for an infinitesimally small positive value ϵ. Clearly, the permutations performed by the sorting algorithm over a and a' can differ, as the variation ϵ may affect the validity of the guard $a[j-1] > a[j]$ that triggers the permutations. Fortunately, this small variation on the input data can at most cause a small variation in the final result. Indeed, one can verify the validity of the following relational judgment:

$$\vDash \{\forall i.\, |a[i]-a'[i]| < \epsilon\}\; c \sim c'\; \{\forall i.\, |a[i]-a'[i]| < \epsilon\}$$

where c stands for the sorting algorithm in Figure 1 and c' for the result of substituting every variable v in c by its primed version v'. Instead of relying on a special purpose logic to reason about program continuity, we suggest to construct a product program that performs the execution steps of c and c' synchronously. Since c and c' have the same structure, it is immediate to build the program d, shown at the left of Figure 1, that weaves the instructions of c and c'. The algorithm d simulates every pair of executions of c and c' synchronously, capturing all executions of c and c'. Notice that the program product synchronizes the loops iterations of its components, as their loop guards are equivalent and thus perform the same number of iterations. This is not the case with the conditional statements inside the loop body, as the small variations on the contents of the array a w.r.t. a' may break the equivalence of the guards $a[j-1] > a[j]$ and $a'[j'-1] > a'[j']$.

One can use a standard program logic to verify the validity of the non relational judgment $\vDash \{\forall i.\;\, |a[i]-a'[i]| < \epsilon\}\, d\, \{\forall i.\;\, |a[i]-a'[i]| < \epsilon\}$. If the program product is a correct representation of its components, the validity of this judgment over d is enough to establish the validity of the relational judgment over c and c'.

It is not difficult to build a program product from structurally equivalent programs by a total synchronization of the loops, as in the example above. Structural equivalence is, however, a significant constraint as it rules out many interesting cases of relational reasoning, including the translation validation examples in Section 4. Consider the case of the loop pipelining optimization shown in Figure 6. The source and transformed programs have a similar structure: a

Source code:

```
$i := 0$;
while $(i < N)$ do
  $j := N-1$;
  while $(j > i)$ do
    if $(a[j-1] > a[j])$ then
      $swap(a, j, j-1)$;
    $j$--
  $i$++
```

Program product (simplified):

```
$i := 0$; $i' := 0$;
while $(i < N)$ do
  $j := N-1$; $j' := N-1$;
  while $(j > i)$ do
    if $(a[j-1] > a[j])$ then
      $swap(a, j, j-1)$;
    if $(a'[j'-1] > a'[j'])$ then
      $swap(a', j', j'-1)$;
    $j$--; $j'$--
  $i$++; $i'$++
```

Fig. 1. Continuity of bubble-sort algorithm

loop statement plus some initialization and clean-up code. However, both programs cannot be synchronized a priori, since the number of loop iterations in the source and transformed program do not coincide. A more difficult situation arises when verifying the correctness of static-caching, shown in Figure 11, since it involves synchronizing two nested loops with different depths.

A first intuition on the construction of products from structurally dissimilar components is shown in the following basic example (assume $0 \leq N$):

Source code:

```
$i := 0$;
while $(i \leq N)$ do
  $x$ += $i$;
  $i$++
```

Transformed code:

```
$j := 1$;
while $(j \leq N)$ do
  $y$ += $j$;
  $j$++
```

Program product (simplified):

```
$i := 0$; $x$ += $i$; $i$++; $j := 1$;
while $(i \leq N)$ do
  $y$ += $j$; $j$++;
  $x$ += $i$; $i$++;
```

To build the product program, the first loop iteration of the source code is unrolled before synchronizing the loop statements. This simple idea maximizes synchronization instead of relying plainly on self-composition, avoiding a functional interpretation of the product components. Indeed, a sequential composition would require providing invariants of the form $x = X + \frac{i(i-1)}{2}$ and $y = Y + \frac{j(j-1)}{2}$, respectively (under the preconditions $x = X$ and $y = Y$). In contrast, by the construction of the product, the trivial loop invariant $i = j \wedge x = y$ is sufficient to verify that the two programs above satisfy the pre and post-relation $x = y$.

In the rest of the paper, we develop a more flexible notion of program products, extending the construction of products from components that are not structurally equal or with a different number of loop iterations.

3 Program products

Our reduction of relational verification into standard verification relies on the ability of constructing, for any pair of programs c_1 and c_2, a product program c that simulates the execution steps of its constituents. We first introduce a basic

$$\frac{}{\langle \mathsf{assert}(b), \sigma\rangle \rightsquigarrow \langle \mathsf{skip}, \sigma\rangle}\ [\![b]\!]\sigma$$

$$\frac{\langle c_1, \sigma\rangle \rightsquigarrow \langle c_1', \sigma'\rangle}{\langle c_1; c_2, \sigma\rangle \rightsquigarrow \langle c_1'; c_2, \sigma'\rangle} \qquad \frac{\langle c_1, \sigma\rangle \rightsquigarrow \langle \mathsf{skip}, \sigma'\rangle}{\langle c_1; c_2, \sigma\rangle \rightsquigarrow \langle c_2, \sigma'\rangle}$$

$$\frac{}{\langle \mathsf{while}\ b\ \mathsf{do}\ c, \sigma\rangle \rightsquigarrow \langle c; \mathsf{while}\ b\ \mathsf{do}\ c, \sigma\rangle}\ [\![b]\!]\sigma \qquad \frac{}{\langle \mathsf{while}\ b\ \mathsf{do}\ c, \sigma\rangle \rightsquigarrow \langle \mathsf{skip}, \sigma\rangle}\ [\![\neg b]\!]\sigma$$

Fig. 2. Program semantics (excerpt)

program setting that will serve to formalize the ideas exposed in this article, and then provide a formalization of product program.

3.1 Programming model

Commands are defined by the following grammar rule:

$$c \quad ::= \quad x := e \mid a[e] := e \mid \mathsf{skip} \mid \mathsf{assert}(b) \mid c; c \mid \mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \mid \mathsf{while}\ b\ \mathsf{do}\ c$$

in which x ranges over a set of integer variables $\mathcal{V}_i$, a ranges over a set of array variables $\mathcal{V}_a$ (we assume $\mathcal{V}_i \cap \mathcal{V}_a = \emptyset$ and let $\mathcal{V}$ denote $\mathcal{V}_i \cup \mathcal{V}_a$), and $e \in \mathsf{AExp}$ and $b \in \mathsf{BExp}$ range over integer and boolean expressions. Execution states are represented as $\mathcal{S} = (\mathcal{V}_i \rightharpoonup \mathbb{Z}) \times (\mathcal{V}_a \rightharpoonup (\mathbb{Z} \rightharpoonup \mathbb{Z}))$. The semantics of integer and boolean expressions are given by $([\![e]\!])_{e\in\mathsf{AExp}} : \mathcal{S} \to \mathbb{Z}$ and $([\![b]\!])_{b\in\mathsf{BExp}} : \mathcal{S} \to \mathbb{B}$, respectively. The semantics of commands is standard, deterministic, and defined by a relation $\langle c, \sigma\rangle \rightsquigarrow \langle c', \sigma'\rangle$ in Figure 2, with $\langle \mathsf{skip}, \sigma\rangle$ denoting final configurations. In particular, notice that a statement of the form $\mathsf{assert}(b)$ blocks a program execution if b is not satisfied. We let $\langle c, \sigma\rangle \Downarrow \sigma'$ denote $\langle c, \sigma\rangle \rightsquigarrow^\star \langle \mathsf{skip}, \sigma'\rangle$.

An assertion ϕ is a first-order formula with variables in $\mathcal{V}$. We let $[\![\phi]\!]$ denote the set of states satisfying ϕ. Finally, we let $\mathsf{var}(c) \subseteq \mathcal{V}$ and $\mathsf{var}(\phi) \subseteq \mathcal{V}$ denote the set of (free) variables of a command c and assertion ϕ, respectively.

In order to simplify the definition of valid relational judgment, we introduce a notion of separable commands: two commands c_1 and c_2 are separable if they have disjoint set of variables: $\mathsf{var}(c_1) \cap \mathsf{var}(c_2) = \emptyset$. Two states are separable if they have disjoint domains. We let $\mu_1 \uplus \mu_2$ denote the union of finite maps

$$(\mu_1 \uplus \mu_2)\, x = \begin{cases} \mu_1\, x & \text{if } x \in \mathsf{dom}(\mu_1) \\ \mu_2\, x & \text{if } x \in \mathsf{dom}(\mu_2) \end{cases}$$

and overload this notation for the union of separable states $(\mu, \nu) \uplus (\mu', \nu')$, defined as $(\mu \uplus \mu', \nu \uplus \nu')$. Under this separability assumption, one can identify assertions as relations on states: $(\sigma_1, \sigma_2) \in [\![\phi]\!]$ iff $\sigma_1 \uplus \sigma_2 \in [\![\phi]\!]$. The formal statement of valid relational specifications is then given by the following definition.

Definition 1. *Two commands c_1 and c_2 satisfy the pre and post-relation φ and ψ, denoted by the judgment $\vDash \{\varphi\}\, c_1 \sim c_2\, \{\psi\}$ if for all states $\sigma_1, \sigma_2, \sigma_1', \sigma_2'$ s.t. $\sigma_1 \uplus \sigma_2 \in [\![\varphi]\!]$ and $\langle c_1, \sigma_1\rangle \Downarrow \sigma_1'$ and $\langle c_2, \sigma_2\rangle \Downarrow \sigma_2'$, we have $\sigma_1' \uplus \sigma_2' \in [\![\psi]\!]$.*

Our goal is to reduce validity of relational judgments to validity of Hoare triples, hence we also define the notion of valid Hoare triple. For technical reasons, we adopt a stronger definition of validity, which requires that the command is non-stuck w.r.t. the precondition of the triple, where a command c is φ-nonstuck if its execution does not block under the precondition φ. That is, for all states σ, σ' and command $c' \neq \mathsf{skip}$ such that $\sigma \in [\![\varphi]\!]$ and $\langle c, \sigma\rangle \rightsquigarrow^{\star} \langle c', \sigma'\rangle$, there exists c'' and σ'' such that $\langle c', \sigma'\rangle \rightsquigarrow \langle c'', \sigma''\rangle$.

Definition 2. *A triple $\{\varphi\}c\{\psi\}$ is* valid*, denoted by the judgment $\vDash \{\varphi\}\, c\, \{\psi\}$, if c is φ-nonstuck and for all $\sigma, \sigma' \in \mathcal{S}$, $\sigma \in [\![\varphi]\!]$ and $\langle c, \sigma\rangle \Downarrow \sigma'$ imply $\sigma' \in [\![\psi]\!]$.*

Such a notion of validity can be established using an extension of Hoare logic with the following rule to deal with assert statements:

$$\frac{}{\vdash \{b \wedge \phi\}\, \mathsf{assert}(b)\, \{\phi\}}$$

3.2 Product construction

We start in this section with a set of rules appropriate for structurally equivalent programs. Then, we extend the set of rules with a structural transformation to deal with structurally dissimilar programs.

Figure 3 provides a set of rules to derive a product construction judgment $c_1 \times c_2 \rightarrow c$. The construction of products introduces assert statements to verify that the resulting program simulates precisely the behavior of its components. This validation constraints are interpreted as local assertions, which are discharged during the program verification phase. For instance, in the rule that synchronizes two loop statements, the insertion of the statement $\mathsf{assert}(b_1 \Leftrightarrow b_2)$ just before the evaluation the loop guards b_1 and b_2 enforces that the number of loop iterations coincide. The resulting product containing assert statements can thus be verified with a standard logic. If a command c is the product of c_1 and c_2, then the validity of a relational judgment between c_1 and c_2 can be deduced from the validity of a standard judgment on c.

Proposition 1. *For all statements c_1 and c_2 and pre and post-relations φ and ψ, if $c_1 \times c_2 \rightarrow c$ and $\vDash \{\varphi\}\, c\, \{\psi\}$ then $\vDash \{\varphi\}\, c_1 \sim c_2\, \{\psi\}$.*

Proof. We prove by induction on the derivation of a judgment $c_1 \times c_2 \rightarrow c$ that if $\langle c_1, \sigma_1\rangle \Downarrow \sigma_1'$, $\langle c_2, \sigma_2\rangle \Downarrow \sigma_2'$ and $\sigma_1 \uplus \sigma_2$ does not make c get stuck, then $\langle c, \sigma_1 \uplus \sigma_2\rangle \Downarrow \sigma_1' \uplus \sigma_2'$. We consider here some selected cases.

Suppose that the last rule applied is

$$\frac{c_1 \succcurlyeq c_1' \qquad c_2 \succcurlyeq c_2' \qquad c_1' \times c_2' \rightarrow c}{c_1 \times c_2 \rightarrow c}$$

and assume $\langle c_1, \sigma_1\rangle \Downarrow \sigma_1'$, $\langle c_2, \sigma_2\rangle \Downarrow \sigma_2'$, and $\vDash \{\varphi\}\, c\, \{\psi\}$. From the construction of products, for all separable commands c_1 and c_2, if c does not get stuck with initial state $\sigma_1 \uplus \sigma_2$ then neither c_1 nor c_2 gets stuck with initial states σ_1 and σ_2.

$$\frac{}{c_1 \times c_2 \rightarrow c_1; c_2} \qquad \frac{c_1 \times c_2 \rightarrow c \qquad c_1' \times c_2' \rightarrow c'}{(c_1; c_1') \times (c_2; c_2') \rightarrow c; c'}$$

$$\frac{c_1 \times c_2 \rightarrow c}{(\mathsf{while}\ b_1\ \mathsf{do}\ c_1) \times (\mathsf{while}\ b_2\ \mathsf{do}\ c_2) \rightarrow \mathsf{assert}(b_1 \Leftrightarrow b_2); \mathsf{while}\ b_1\ \mathsf{do}\ (c; \mathsf{assert}(b_1 \Leftrightarrow b_2))}$$

$$\frac{c_1 \times c_2 \rightarrow c \qquad c_1' \times c_2' \rightarrow c'}{(\mathsf{if}\ b_1\ \mathsf{then}\ c_1\ \mathsf{else}\ c_1') \times (\mathsf{if}\ b_2\ \mathsf{then}\ c_2\ \mathsf{else}\ c_2') \rightarrow \mathsf{assert}(b_1 \Leftrightarrow b_2); \mathsf{if}\ b_1\ \mathsf{then}\ c\ \mathsf{else}\ c'}$$

$$\frac{c_1 \times c \rightarrow c_1' \qquad c_2 \times c \rightarrow c_2'}{(\mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2) \times c \rightarrow \mathsf{if}\ b\ \mathsf{then}\ c_1'\ \mathsf{else}\ c_2'}$$

Fig. 3. Product construction rules

Then, since $\sigma_1 \uplus \sigma_2 \in \varphi$ and $\vDash \{\varphi\}\, c\, \{\psi\}$, and from the definition of refinement we have $\langle c_1', \sigma_1 \rangle \Downarrow \sigma_1'$ and $\langle c_2', \sigma_2 \rangle \Downarrow \sigma_2'$. By I.H. we can conclude $\sigma_1' \uplus \sigma_2' \in [\![\psi]\!]$.

Suppose that c_1, c_2, and c are of the form $\mathsf{while}\ b_1\ \mathsf{do}\ c_1'$, $\mathsf{while}\ b_2\ \mathsf{do}\ c_2'$, *and*

$$\mathsf{assert}(b_1 \Leftrightarrow b_2); \mathsf{while}\ b_1\ \mathsf{do}\ (c'; \mathsf{assert}(b_1 \Leftrightarrow b_2))$$

respectively, and the last rule applied is

$$\frac{c_1' \times c_2' \rightarrow c'}{c_1 \times c_2 \rightarrow c}$$

Consider a state $\sigma_1 \uplus \sigma_2 \in [\![\varphi]\!]$ such that $\langle c_1, \sigma_1 \rangle \Downarrow \sigma_1'$ and $\langle c_2, \sigma_2 \rangle \Downarrow \sigma_2'$. From the hypothesis $\vDash \{\varphi\}\, c\, \{\psi\}$, c is φ-nonstuck and, thus, from the semantics of the assert *statement we have that b_1 and b_2 are equivalent in $\sigma_1 \uplus \sigma_2$. We proceed to prove by induction on $\langle c_1, \sigma_1 \rangle \Downarrow \langle \mathsf{skip}, \sigma_1' \rangle$ that $\langle c, \sigma_1 \uplus \sigma_2 \rangle \Downarrow \langle \mathsf{skip}, \sigma_1' \uplus \sigma_2' \rangle$. The base case is trivial from the equivalence of b_1 and b_2 in $\sigma_1 \uplus \sigma_2$. Suppose that b_1 is valid in σ_1 and we have μ_1 such that $\langle c_1', \sigma_1 \rangle \Downarrow \mu_1$ and $\langle c_1, \mu_1 \rangle \Downarrow \sigma_1'$. Similarly, from the equivalence of b_2, we have a state μ_2 s.t. $\langle c_2', \sigma_2 \rangle \Downarrow \mu_2$ and $\langle c_2, \mu_2 \rangle \Downarrow \sigma_2'$. From the inductive hypothesis on the derivation of $c_1' \times c_2' \rightarrow c'$, $\langle c', \sigma_1 \uplus \sigma_2 \rangle \Downarrow \mu_1 \uplus \mu_2$. The execution of c from the initial state $\sigma_1 \uplus \sigma_2$ does not get stuck, it must be also the case that $b_1 \Leftrightarrow b_2$ is valid in $\mu_1 \uplus \mu_2$. Since c does not get stuck with $\mu_1 \uplus \mu_2$, one can apply I.H. to conclude that $\langle c; \mu_1 \uplus \mu_2 \rangle \Downarrow \sigma_1' \uplus \sigma_2'$.*

A constraint of the product construction rules in Fig. 3 is that two loops with non-equivalent guards must be sequentially composed. In the rest of this section we propose a structural transformation that helps extending the application of relational verification by product construction to non-structurally equivalent programs.

We characterize the structural transformations extending the construction of products as a refinement relation, denoted with a judgment of the form $c \succcurlyeq c'$. It is a refinement relation in the sense that every execution of c is an execution of c' except when c' gets stuck:

$$\frac{}{\vdash \mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \succcurlyeq \mathsf{assert}(b); c_1} \qquad \frac{}{\vdash \mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \succcurlyeq \mathsf{assert}(\neg b); c_2}$$

$$\frac{}{\vdash \mathsf{while}\ b\ \mathsf{do}\ c \succcurlyeq \mathsf{assert}(b); c; \mathsf{while}\ b\ \mathsf{do}\ c}$$

$$\frac{}{\vdash \mathsf{while}\ b\ \mathsf{do}\ c \succcurlyeq \mathsf{while}\ b \wedge b'\ \mathsf{do}\ c; \mathsf{while}\ b\ \mathsf{do}\ c} \quad \frac{}{\vdash \mathsf{while}\ b\ \mathsf{do}\ c \succcurlyeq \mathsf{assert}(b); c; \mathsf{assert}(\neg b)}$$

$$\frac{\vdash c \succcurlyeq c'}{\vdash \mathsf{while}\ b\ \mathsf{do}\ c \succcurlyeq \mathsf{while}\ b\ \mathsf{do}\ c'} \qquad \frac{\vdash c_1 \succcurlyeq c_1' \qquad \vdash c_2 \succcurlyeq c_2'}{\vdash \mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \succcurlyeq \mathsf{if}\ b\ \mathsf{then}\ c_1'\ \mathsf{else}\ c_2'}$$

$$\frac{\vdash c \succcurlyeq c' \qquad \vdash c' \succcurlyeq c''}{\vdash c \succcurlyeq c''} \qquad \frac{}{\vdash c \succcurlyeq c} \qquad \frac{\vdash c_1 \succcurlyeq c_1' \qquad \vdash c_2 \succcurlyeq c_2'}{\vdash c_1; c_2 \succcurlyeq c_1'; c_2'}$$

Fig. 4. Syntactic reduction rules

Definition 3. *A command c' is a refinement of c, if for all states σ, σ':*

1. *if $\langle c', \sigma\rangle \Downarrow \sigma'$ then $\langle c, \sigma\rangle \Downarrow \sigma'$, and*
2. *if $\langle c, \sigma\rangle \Downarrow \sigma'$ then either the execution of c' with initial state σ gets stuck, or $\langle c', \sigma\rangle \Downarrow \sigma'$.*

We provide in Figure 4 a particular set of structural rules defining judgments of the form $\vdash c \succcurlyeq c'$. From the rules given in the figure, one can see that the executions of c and c' coincide for every initial state that makes the introduced assert statements valid. One can prove that the judgment $\vdash c \succcurlyeq c'$ establishes a refinement relation by showing that for every assertion φ, if c' is φ-nonstuck then for all $\sigma \in [\![\varphi]\!]$ such that $\langle c, \sigma\rangle \Downarrow \sigma'$ we have $\langle c', \sigma\rangle \Downarrow \sigma'$.

We enrich the set of rules defining the construction of products by adding an extra rule that introduces a preliminary refinement transformation over the product components:

$$\frac{c_1 \succcurlyeq c_1' \qquad c_2 \succcurlyeq c_2' \qquad c_1' \times c_2' \to c}{c_1 \times c_2 \to c}$$

Proposition 1 remains valid for the extended proof system. The following proposition reduces the problem of proving the validity of a relational judgment into two steps: the construction of the corresponding program product plus a standard verification over the program product.

Proposition 2. *For all statements c_1 and c_2 and pre and post-relations φ and ψ, if $c_1 \times c_2 \to c$ and $\vdash \{\varphi\}\, c\, \{\psi\}$ then $\models \{\varphi\}\, c_1 \sim c_2\, \{\psi\}$.*

4 Case studies

This section illustrates the application of product construction for the verification of relational properties, such as non-interference and the correctness of

Example	SMT/P.O.
Non-interference	42/42
Loop alignment	49/49
Loop pipelining	73/73
Loop unswitching	123/123
Code sinking	435/435
Static caching	162/176

Example	SMT/P.O.'s
Loop reversal	13/13
Strength reduction	5/5
Loop interchange	36/37
Loop fission	15/15
Cyclic hashing	13/13
Bubble sort continuity	62/62

Table 1. Automatic validation of case studies

program transformations. These program transformations include loop optimizations, static-caching, a complex optimization described later in this section, and a transformation step used in the SSE-vectorization of a cyclic hashing algorithm. For each of the examples in this section, a product construction has been verified with the Why tool(and the Frama-C toolwith the Jessie plugin). As shown in Table 1, most of the examples could be automatically verified: the column $\mathsf{P.O.}$ indicates the number of proof obligations generated, and the column SMT those that have been automatically discharged by SMT solvers. For the static-caching and loop interchange examples, the remaining proof obligations have been discharged in the Coq proof assistant.

Logical verification of non-interference

Non-interference is a confidentiality policy defined in terms of two executions of the same program. Given a program c and a set of public variables $x_1, \ldots, x_k$, the property ensures that two terminating runs of c starting in states with equal public variables, end in states with equal public variables:

$$\langle c, \sigma_1\rangle \rightsquigarrow \sigma_2 \;\wedge\; \langle c, \sigma'_1\rangle \rightsquigarrow \sigma'_2 \;\wedge \bigwedge_{x\in\{x_1,\ldots,x_k\}} \sigma_1 x = \sigma'_1 x \implies \bigwedge_{x\in\{x_1,\ldots,x_k\}} \sigma_2 x = \sigma'_2 x.$$

(For simplicity, we express non-interference w.r.t. scalar variables, the extension to array variables is immediate.) Non-interference can thus be formulated as a relational judgment:

$$\vDash \{x_1{=}x'_1 \wedge .. \wedge x_k{=}x'_k\}\; c \sim c'\; \{x_1{=}x'_1 \wedge .. \wedge x_k{=}x'_k\}$$

where c' is the result of replacing every variable v in c by v'.

We illustrate the application of relational verification by product construction for the verification of an example drawn from [12]. Figure 5 shows the construction of the program product (the original program can be obtained by slicing out the statements containing primed variables). This simple algorithm merges a table containing personal information with a table containing salary information. A special field *JoinInd* indicates whether the personal information is private and should not be included as the result of the join operation.

```
{$Pre : es = es' \wedge \forall i : 0 \leq i < N : ps[i].PID = ps'[i].PID \wedge$
  $ps[i].JoinInd = ps'[i].JoinInd \wedge (ps[i].JoinInd \Rightarrow ps[i].salary = ps'[i].salary)$}
  $i := 0$; $i' := 0$; assert($i < N \Leftrightarrow i' < N$);
  while ($i < N$) do
    assert($ps[i].JoinInd \Leftrightarrow ps'[i'].JoinInd$);
    if ($ps[i].JoinInd$) then
      $j := 0$; $j' := 0$; assert($j < M \Leftrightarrow j' < M$);
      while ($j < M$) do
        assert($ps[i].PID = es[j].EID \Leftrightarrow ps'[i'].PID = es'[j'].EID$);
        if ($ps[i].PID = es[j].EID$) then
          $tab[i].employee := es[j]$; $tab'[i'].employee := es'[j]$;
          $tab[i].payroll := ps[i]$; $tab'[i'].payroll := ps'[i]$;
        $j$++; $j'$++; assert($i < N \Leftrightarrow i' < N$);
    $i$++; $i'$++; assert($i < N \Leftrightarrow i' < N$);
{$Post : \forall i : 0 \leq i < N : ps[i].JoinInd \Rightarrow tab[i] = tab'[i]$}
```

Fig. 5. Non-interference product

The pre and postcondition provided in Figure 5 establish that the input data marked as private does not interfere with the final result: if the values stored in the input arrays coincide for the public indices (i.e., for i s.t. $ps[i].JoindInd$ is true), then the return data coincides at the public indices (we let a formula of the form $a = \bar{a}$ stand for $\forall i \in [0, N-1].\ a[i] = \bar{a}[i]$.)

Self-composition is another method that embeds the verification of information-flow policies in non-relational program logics, by reducing it to the logical verification of sequential compositions of the form $\models \{x_1 = x'_1 \wedge .. \wedge x_k = x'_k\}\ c; c'\ \{x_1 = x'_1 \wedge .. \wedge x_k = x'_k\}$. This method based on sequential composition is not amenable for automatic tools, as it requires providing and verifying an intermediate assertion ϕ such that the judgments

$$\models \{x_1 = x'_1 \wedge \ldots \wedge x_k = x'_k\}\ c\ \{\phi\} \quad \text{and} \quad \models \{\phi\}\ c'\ \{x_1 = x'_1 \wedge \ldots \wedge x_k = x'_k\}$$

hold. In practice, this is a significant obstacle, as it may require understanding and verifying a functional specification for the program c. Terauchi and Aiken propose an alternative program composition [22], that can be seen as a particular instance of our product construction, defined in terms of an information-flow type system.

Translation validation of loop optimizations

In this section we consider several non-trivial loop optimizations, and show how they can be verified using product constructions.

Loop pipelining. Loop pipelining is a non-trivial optimization that reduces the proximity of memory references inside a loop, in order to introduce parallelization

Source program:

```
  $i := 0$;
  while ($i < N$) do
    $a[i]$++;  $b[i]$ += $a[i]$;
    $c[i]$ += $b[i]$;  $i$++
```

Transformed program:

```
  $j := 0$;
  $\bar{a}[0]$++;  $\bar{b}[0]$ += $\bar{a}[0]$;
  $\bar{a}[1]$++;
  while ($j < N-2$) do
    $\bar{a}[j+2]$++;
    $\bar{b}[j+1]$ += $\bar{a}[j+1]$;
    $\bar{c}[j]$ += $\bar{b}[j]$;  $j$++
  $\bar{c}[j]$ += $\bar{b}[j]$;
  $\bar{b}[j+1]$ += $\bar{a}[j+1]$;
  $\bar{c}[j+1]$ += $\bar{b}[j+1]$
```

Product program:

```
 $\{a = \bar{a} \wedge b = \bar{b} \wedge c = \bar{c}\}$
  $i := 0$;  $j := 0$;  assert($i < N$);
  $a[i]$++;  $b[i]$ += $a[i]$;
  $c[i]$ += $b[i]$;  $i$++;
  $\bar{a}[0]$++;  $\bar{b}[0]$ += $\bar{a}[0]$;
  assert($i < N$);
  $a[i]$++;  $b[i]$ += $a[i]$;
  $c[i]$ += $b[i]$;  $i$++;  $\bar{a}[1]$++;
  assert($i < N \Leftrightarrow j < N-2$);
  while ($i < N$) do
    $a[i]$++;  $b[i]$ += $a[i]$;  $c[i]$ += $b[i]$;  $i$++
    $\bar{a}[j+2]$++;  $\bar{b}[j+1]$ += $\bar{a}[j+1]$;
    $\bar{c}[j]$ += $\bar{b}[j]$;  $j$++
    assert($i < N \Leftrightarrow j < N-2$);
  $\bar{c}[j]$ += $\bar{b}[j]$;  $\bar{b}[j+1]$ += $\bar{a}[j+1]$;  $\bar{c}[j+1]$ += $\bar{b}[j+1]$
 $\{a = \bar{a} \wedge b = \bar{b} \wedge c = \bar{c}\}$
```

Fig. 6. Loop pipelining

opportunities. Consider the simple example shown in Fig. 6 (drawn from [16].) Assume a, b, and c are arrays of size N, with $2 \le N$.

The program product shown in Fig. 6 pairs the initialization statements over $\bar{a}[0]$, $\bar{b}[0]$, and $\bar{a}[1]$ with the first and second loop iterations of the original program. Similarly, the final assignments to $\bar{b}[N-2]$, $\bar{c}[N-2]$ and $\bar{c}[N-1]$ are executed synchronously with the final loop iteration of the original loop. The remaining $N-2$ loop iterations are synchronized together. In order to verify that $a = \bar{a} \wedge b = \bar{b} \wedge c = \bar{c}$ is a valid pre and post condition, we require a specification that establishes the equalities in b and $\bar{b}$ and c and $\bar{c}$, except for the indices j and $j+1$. In particular, the loop invariant must state that $b[j+1] = \bar{b}[j+1] + a[j+1]$, $c[j] = \bar{c}[j] + b[j]$, and $c[j+1] = \bar{c}[j+1] + b[j+1]$, and $b[i'] = \bar{b}[i']$ and $c[i'] = \bar{c}[i']$ for any other index i'.

Loop alignment. The goal of this transformation is to improve cache performance by increasing the proximity of the memory locations accessed by a loop iteration. Consider the optimization example shown in Fig. 7, and assume $1 \le N$. The array b is accessed twice in the loop body: a write access to the position i, and a read access to the position $i-1$. The optimization aligns the accesses of the array b in the loop body to a same index i. Notice that if the values stored by the array b are not needed after the loop execution, the occurrence of $b[i]$ in the loop body is replaced by a fresh scalar variable.

The product is shown in the same figure. The construction of the product pairs the initialization statement $\bar{d}[1] := \bar{b}[0]$ of the transformed program with the first loop iteration of the source program, and then pairs the remaining $N-1$ loop iterations.

Source program:

```
$i := 1$;
while ($i \le N$) do
    $b[i] := a[i]$;
    $d[i] := b[i-1]$;
    $i$++
```

Transformed program:

```
$j := 1$;  $\bar{d}[1] := \bar{b}[0]$;
while ($j \le N-1$) do
    $\bar{b}[j] := \bar{a}[j]$;
    $\bar{d}[j+1] := \bar{b}[j]$;  $j$++;
$\bar{b}[N] := \bar{a}[N]$
```

Product program:

```
{$a = \bar{a} \wedge b[0] = \bar{b}[0]$}
 $i := 1$;  $j := 1$;
 assert($i \le N$);
 $b[i] := a[i]$;  $d[i] := b[i-1]$;  $i$++;
 $\bar{d}[1] := \bar{b}[0]$;
 assert($i \le N \Leftrightarrow j \le N-1$);
 while ($i \le N$) do
     $b[i] := a[i]$;  $d[i] := b[i-1]$;  $i$++;
     $\bar{b}[j] := \bar{a}[j]$;  $\bar{d}[j+1] := \bar{b}[j]$;  $j$++;
     assert($i \le N \Leftrightarrow j \le N-1$);
 $\bar{b}[N] := \bar{a}[N]$
{$d[1,N] = \bar{d}[1,N]$}
```

Fig. 7. Loop alignment

The program product can be verified w.r.t. the pre and post conditions $a = \bar{a} \wedge b[0] = \bar{b}[0]$ and $d[1,N] = \bar{d}[1,N]$. A suitable loop invariant is

$$d[1,i) = \bar{d}[1,i) \wedge b[j] = a[j] \wedge \bar{b}[i] = b[i] \wedge i = j+1$$

The specification ensures that given equal input arrays a and b, the values stored in the array d coincide after the execution of both product components. The expression $d[1,i) = \bar{d}[1,i)$ means that the values of the arrays d and $\bar{d}$ coincide in the index range $[1,i)$—and similarly with the range $[1,N]$.

Induction variable strength reduction. Program products provide a convenient means to validate the correctness of optimizations that do not modify the control-flow of programs, and simply affect their basic blocks, such as constant propagation or common subexpression elimination. Fig. 8 shows the application of strength reduction to a small fragment of code. In the figure, j is a derived induction variable, defined as a linear function on the induction variable i. The optimization replaces the assignment $j := i*B+C$ by the contextually equivalent and less costly statement j'+= B, swaps the assignments to x and j, and adds an initialization $j' := C$ immediately before the loop header. The program product that simulates the simultaneous execution of the original program and its optimized version in shown at the right of Figure 8. In order to verify the optimization correct, i.e., that $x = x'$ is an invariant of the product program, we require the linear condition $j' = i'B+C$ as part of the loop invariant. The compiler analyzer can provide such information as part of the loop invariant specification, following the techniques suggested in [6].

Loop unswitching. Loop unswitching consists of moving invariant conditional branches outside of a loop. If a conditional statement depends on a guard that

Source code:

```
i := 0;
while (i < N) do
  j := i*B+C;
  x += j; i++
```

Optimized code:

```
i' := 0; j' := C;
while (i' < N) do
  x' += j';
  j' += B; i'++
```

Program product:

```
i := 0;  i' := 0;  j' := C;
while (i < N ∧ i' < N) do
  j := i*B+C;  x += j; i++
  x' += j';  j' += B; i'++
```

Fig. 8. Induction variable strength reduction

is not modified by the loop body, the evaluation of the guard can be performed before entering the loop. Figure 9 shows a simple example of loop unswitching, drawn from [2]. In the source program, the value of the condition $x < 7$ is invariant to the loop, so the conditional statement is pushed outside of the loop in the transformed program, and the loop body is replicated at each of the conditional branches.

The construction of the product is shown at the right of Figure 9. The program product is verified to satisfy the pre and post relation

$$\forall i.\ 0 \leq i < N \implies a[i] = A[i] \wedge b[i] = B[i] \wedge c[i] = C[i]$$

For this simple example, the same formula above has been used as the loop invariant required by the verification tool.

Code Sinking. Code sinking is a program transformation that consists in moving code inside a loop statement. Its main goal is to transform imperfect nested loops into perfect nests, which are amenable to more loop optimizations. Figure 10 shows an example of code sinking [1]. The original algorithm consists of a main loop enclosed by initialization and final statements. In the transformed program, such statements are pushed inside the loop, under appropriate guards to ensure semantics preservation.

The product construction is shown at the right of Figure 10. It consists basically on a preliminary step that unrolls the first and last iteration of the source and transformed program, plus a basic loop synchronization. The program product representing the simultaneous execution of the original and transformed program can be easily verified to satisfy the following pre and post condition

$$i = i' \wedge max = max' \wedge maxi = maxi' \wedge \forall i.\ 0 \leq i \leq N \Rightarrow a[i] = a'[i]$$

which entails the observational equivalence of both product components.

Static caching

In this section we turn our attention to static caching [17], an optimization that has not been considered from the perspective of translation validation. To the best of our knowledge, we provide the first formal validation of such optimization.

Static caching removes redundant computations by exploiting memoized intermediate results. One of its applications is the row summation algorithm in

Source program:

```
$i := 0$;
while $(i < N)$ do
  $a[i] := a[i]+k$;
  if $(x < 7)$ then
    $b[i] := a[i]*c[i]$
  else
    $b[i] := a[i-1]*b[i-1]$;
  $i$++
```

Transformed program:

```
if $(y < 7)$ then
  $j := 0$;
  while $(j < N)$ do
    $A[j] := A[j]+k$;
    $B[j] := A[j]*C[j]$;
    $j$++
else
  $j := 0$;
  while $(j < N)$ do
    $A[j] := A[j]+k$;
    $B[j] := A[j-1]*B[j-1]$;
    $j$++
```

Product program:

```
$i := 0$;
if $(y < 7)$ then
  $j := 0$; assert$(i < N \Leftrightarrow j < N)$;
  while $(i < N)$ do
    $A[j] := A[j]+k$; $a[i] := a[i]+k$;
    assert$(x < 7 \Leftrightarrow y < 7)$;
    if $(x < 7)$ then
      $b[i] := a[i]*c[i]$;
      $B[j] := A[j]*C[j]$;
    else
      $b[i] := a[i-1]*b[i-1]$;
      $B[j] := A[j]*C[j]$;
    $j$++; $i$++; assert$(i < N \Leftrightarrow j < N)$;
else
  $j := 0$; assert$(i < N \Leftrightarrow j < N)$;
  while $(i < N)$ do
    $A[j] := A[j]+k$; $a[i] := a[i]+k$;
    assert$(x < 7 \Leftrightarrow y < 7)$;
    if $(x < 7)$ then
      $b[i] := a[i]*c[i]$;
      $B[j] := A[j-1]*B[j-1]$;
    else
      $b[i] := a[i-1]*b[i-1]$;
      $B[j] := A[j-1]*B[j-1]$;
    $i$++; $i$++; assert$(i < N \Leftrightarrow j < N)$;
```

Fig. 9. Loop unswitching

Fig. 11. The algorithm takes as input an $N \times L$ matrix a and returns an array s of length $N-M+1$ (assume $M \leq N$) such that $s[i] = \sum_{i',j'=i,0}^{M,L} A[i',j']$, for all $i \in [0, N-M]$. The original program performs a significant amount of redundant computation. Let $b[i]$ stand for $\sum_{j=0}^{N} A[i,j]$. One can see that for all i, $s[i]$ differs from $s[i+1]$ on the value $b[i+M]-b[i]$. The computations of the summations $b[i']$ for $i' \in [i+1, i+M-1]$ are thus redundant and can be removed. In the optimized version of the algorithm, the array b of size N is used to store the intermediate computation of row summations. The matrix summations are computed using the computations saved in the array b, and then stored in the array t. As a result, the transformed algorithm has a quadratic complexity, whereas the complexity of the original algorithm is cubic.

Figure 12 shows the product of the original row-summation algorithm and of its optimized version. The specification states that the output arrays s and t coincide in the range $[0, N-M]$ after the synchronous execution of the original and optimized program. The correctness of the product w.r.t. its specification can be verified by simple arithmetic reasoning.

Source program:

```
$max := a[0]$;
$maxi := 0$;
$i := 0$;
while ($i \leq N$) do
  if ($max < a[i]$) then
    $max := a[i]$; $maxi := i$;
  $i$++;
$t := a[N]$;
$a[N] := max$;
$a[maxi] := t$;
```

Transformed program:

```
$j := 0$;
while ($j \leq N$) do
  if ($j = 0$) then
    $max' := a[0]$; $maxi' := 0$;
  if ($max' < a'[j]$) then
    $max' := a'[j]$; $maxi' := j$;
  if ($j = N$) then
    $t' := a'[N]$;
    $a'[N] := max'$;
    $a'[maxi'] := t'$;
  $j$++
```

Product program:

```
$max := a[0]$; $maxi := 0$;
$i := 0$; $j := 0$;
assert($i \leq N$);
if ($max < a[i]$) then $max := a[i]$; $maxi := i$;
$i$++;
assert($j \leq N$); assert($j = 0$);
if ($j = 0$) then $max' := a[0]$; $maxi' := 0$;
if ($max' < a'[j]$) then $max' := a'[j]$; $maxi' := j$;
assert($j \neq N$);
assert($i \leq N \wedge i \neq N \Leftrightarrow j \leq N \wedge j \neq N$);
while ($i \leq N \wedge i \neq N$) do
  assert($j \neq 0$);
  if ($max < a[i]$) then $max := a[i]$; $maxi := i$;
  if ($max' < a'[j]$) then $max' := a'[j]$; $maxi' := j$;
  assert($j \neq N$); $j$++; $i$++;
assert($i = N$); assert($j = N$); assert($j \neq 0$);
if ($max < a[i]$) then $max := a[i]$; $maxi := i$;
if ($max' < a'[j]$) then $max' := a'[j]$; $maxi' := j$;
$i$++;
assert($j = N$);
$t' := a'[N]$; $a'[N] := max'$; $a'[maxi'] := t'$;
$j$++;
$t := a[N]$; $a[N] := max$; $a[maxi] := t$;
```

Fig. 10. Code sinking

5 Related work

Relational logics provide a syntactical counterpart to semantic relational methods, and can be used for similar purposes. To date, relational logics have been applied to prove compiler correctness, program equivalence [5], and non-interference:

Program equivalence. Relational Hoare Logics [7] (RHL) provides a set of elegant and intuitive judgment rules to reason about program equivalence. The main drawback of RHL's core rules is that they can only account for structurally equal programs. This restriction can be lifted by introducing one-sided rules to deal with each particular case; such one-sided rules play a role similar to simulation in our setting. There is a tight connection between relational Hoare logics and products; the details will appear elsewhere. In an independent work, Yang [23] proposes a similar relational framework for a heap-based language with features drawn from separation logic. He applies his logic to the verification of the Schorr-Waite graph marking algorithm. His work also lacks the ability to cope with structurally dissimilar programs.

Source program:

```
$i_1 := 0$;
while ($i_1 \leq N-M$) do
  $s[i_1] := 0$;  $k_1 := 0$;
  while ($k_1 \leq M-1$) do
    $l_1 := 0$;
    while ($l_1 \leq L-1$) do
      $s[i_1]$+=$a[i_1+k_1, l_1]$;  $l_1$++;
    $k_1$++;
  $i_1$++
```

Transformed program:

```
$t[0] := 0$;  $k_2 := 0$;
while ($k_2 \leq M-1$) do
  $b[k_2] := 0$;  $l_2 := 0$;
  while ($l_2 \leq L-1$) do
    $b[k_2]$ += $a[k_2, l_2]$;  $l_2$++;
  $t[0]$ += $b[k_2]$;  $k_2$++;
$i_2 := 1$;
while ($i_2 \leq N-M$) do
  $b[i_2+M-1] := 0$;  $l_2 := 0$;
  while ($l_2 \leq L-1$) do
    $b[i_2+M-1]$+=$a[i_2+M-1, l_2]$;  $l_2$++;
  $z := b[i_2+M-1] - b[i_2-1]$;
  $t[i_2] := t[i_2-1] + z$;  $i_2$++
```

Fig. 11. Static caching: source and optimized code

Compiler correctness. Translation validation [20, 25, 3] is a general method for ensuring the correctness of optimizing compilation by means of a validator which checks after each run of the compiler that the source and target programs are semantically equivalent. Pnueli et al. define Translation Validation for optimizations defined in terms of instruction replacement, reordering of loop iterations, and elimination of loop iterations, handled by the proof rules (Validate), (Permute), and (Reduce), respectively. A drawback of the (Permute) rule is that it can only deal with reordering optimizations, i.e., relating loops with the same number of iterations, disabling the verification of non-consonant loop transformations, such as loop fusion and distribution. In a later work [14], the permute rule is generalized to account for such optimizations.

In an independent line of work, Necula [19] develops a translation validation prototype based on GCC, in terms of a simulation relation between source and transformed program points, and constrained to the validation of structure preserving optimizations. Parametrized equivalence checking [16] lifts the limitations of Necula's relational validation approach to consonant optimizations by combining it with Pnueli et al.'s Permute rule. However, they use a simplified permute rule that restricts reasoning to loops in which every pair of iterations is pair-wise independent, and thus can only account for basic transformations.

A combination of the work in [19] with the Permute rule is provided by Kundu [16] et al. A current deficiency of the correlation inference is the inability to account for asynchronous steps as presented in our work.

Program products. The notion of program product has been previously exploited for the verification of non-interference properties and compiler correctness.

Self-composition [4, 11] provides a sound and complete means to capture non-interference, by traditional verification of the sequential composition of a program with a slightly modified version of itself. Terauchi and Aiken [22] sug-

Product program:

```
{true}
 $i_1 := 0$; assert($i_1 \leq N-M$); $s[i_1] := 0$; $k_1 := 0$; $t[0] := 0$; $k_2 := 0$;
 assert($k_1 \leq M-1 \Leftrightarrow k_2 \leq M-1$);
 while ($k_1 \leq M-1$) {$Inv_1$} do
   $l_1 := 0$; $b[k_2] := 0$; $l_2 := 0$; assert($l_1 \leq L-1 \Leftrightarrow l_2 \leq L-1$);
   while ($l_1 \leq L-1$) {$Inv_2$} do
     $s[i_1]$+=$a[i_1+k_1, l_1]$; $l_1$++; $b[k_2]$ += $a[k_2, l_2]$; $l_2$++;
     assert($l_1 \leq L-1 \Leftrightarrow l_2 \leq L-1$);
   $k_1$++; $t[0]$ += $b[k_2]$; $k_2$++; assert($k_1 \leq M-1 \Leftrightarrow k_2 \leq M-1$);
 $i_1$++; $i_2 := 1$; assert($i_1 \leq N-M \Leftrightarrow i_2 \leq N-M$);
 while ($i_1 \leq N-M$) {$Inv_3$} do
   $b[i_2+M-1] := 0$; $l_2 := 0$;
   while ($l_2 \leq L-1$) {$Inv_4$} do
     $b[i_2+M-1]$+=$a[i_2+M-1, l_2]$; $l_2$++;
   $z := b[i_2+M-1]-b[i_2-1]$; $t[i_2] := t[i_2-1]+z$; $i_2$++;
   $s[i_1] := 0$; $k_1 := 0$;
   while ($k_1 \leq M-1$) {$Inv_5$} do
     $l_1 := 0$;
     while ($l_1 \leq L-1$) {$Inv_6$} do
       $s[i_1]$+=$a[i_1+k_1, l_1]$; $l_1$++;
     $k_1$++;
   $i_1$++
   assert($i_1 \leq N-M \Leftrightarrow i_2 \leq N-M$);
{$\forall i \in [0, N-M].\ s[i] = t[i]$}
```

Fig. 12. Static caching: Program product

gested to improve self-composition by a type directed transformation, a special case of our product construction. Naumann [18] build on Terauchi and Aiken results to encompass the verification of programs with dynamic allocation.

A notion of program products is present in the work of Pnueli and Zack, i.e. *cross-products* [24], for establishing compiler correctness by reducing the relational verification of the original and transformed programs to the analysis of a single program. The restriction of cross-products to structurally equal programs limits the application of the framework to structure preserving transformations.

Other applications of relational methods include regression verification [13], verification of 2-safety properties [22, 10], including determinism [8]. Furthermore, quantitative properties such as continuity [9] or indistinguishability [15] appear as a natural generalization of 2-safety properties.

Clarkson and Schneider provide a general theory of hyperproperties, i.e. set of properties such as non-interference or average response time, which cannot be described as properties, i.e., set of traces. This theory establishes a general classification of policies, but does not serve as a complete verification method.

$$\begin{aligned} Inv_2 \doteq\ & i_1{=}0 \wedge k_1{=}k_2 \wedge l_1{=}l_2 \wedge k_1{\le}M \wedge l_1{\le}L \wedge \\ & s[i_1]=t[0]+b[k_1]=\textstyle\sum_{k'=0}^{k_1-1} b[k']+b[k_1] \wedge \\ & \forall\, k'{\in}[0,k_1).\ b[k']=\textstyle\sum_{l'=0}^{L-1} a[k',l'] \wedge b[k_1]=\textstyle\sum_{l'=0}^{l_1-1} a[k_1,l'] \\ Inv_3 \doteq\ & i_1=i_2 \wedge i_1 \le N{-}M{+}1 \wedge \forall\, i'{\in}[0,i_1) \Rightarrow s[i']=t[i']=\textstyle\sum_{k'=0}^{M-1} b[k'+i'] \wedge \\ & \forall\, i'{\in}[0,i_1{+}M{-}1).\ b[i']=\textstyle\sum_{l'=0}^{L-1} a[i',l'] \\ Inv_4 \doteq\ & Inv_3 \wedge k_1 \le M \wedge l_2 \le L \wedge b[i_2{+}M{-}1]=\textstyle\sum_{l'=0}^{l_2-1} a[i_2{+}M{-}1,l'] \wedge \\ & s[i_1]=\textstyle\sum_{k'=0}^{k_1-1} b[k'{+}i_1] \\ Inv_6 \doteq\ & Inv_3 \wedge k_1 \le M \wedge l_1 \le L \wedge b[i_2{+}M{-}1]=\textstyle\sum_{l'=0}^{L-1} a[i_2{+}M{-}1,l'] \wedge \\ & s[i_1]=\textstyle\sum_{k'=0}^{k_1-1} b[k'{+}i_1]+\textstyle\sum_{l'=0}^{l_1-1} a[i_1{+}k_1,l'] \end{aligned}$$

Fig. 13. Static caching: Loop invariants (excerpt)

6 Further work and conclusions

Relational reasoning provides a mean to enforce a wide range of correctness and security properties, but have lacked methods and tools that are available for traditional program logics. This paper develops a notion of product between programs and reduces verification of relational properties between two programs to verification of functional properties of their product. The notion of product program is general and flexible, and overcomes the limitations of previous approaches.

In this paper, we have concentrated on product programs in the setting of a simple imperative language. However, our constructions extend to products across programs written in two different languages, and also accommodate non-determinism and dynamic allocation. Moreover, we have achieved greater generality by relying on alternative representations of programs, such as flow graphs or their generalizations.

An important goal for further work is to develop methods and tools for building products, and to connect them with off-the-shelf tools to provide a complete framework for relational verification. In a separate line of work, we are investigating applications of products to probabilistic programs, and intend to apply the resulting formalism to provable security [5] and privacy [21].

References

1. T. S. Abdelrahman and R. Sawaya. Improving the structure of loop nests in scientific programs. *Comput. Syst. Sci. Eng.*, 19(1):11–25, 2004.
2. D. F. Bacon, S. L. Graham, and O. J. Sharp. Compiler transformations for high-performance computing. *ACM Comput. Surv.*, 26(4):345–420, 1994.
3. C. W. Barrett, Y. Fang, B. Goldberg, Y. Hu, A. Pnueli, and L. D. Zuck. Tvoc: A translation validator for optimizing compilers. In K. Etessami and S. K. Rajamani, editors, *Computer Aided Verification*, volume 3576 of *Lecture Notes in Computer Science*, pages 291–295. Springer-Verlag, 2005.

4. G. Barthe, P. D'Argenio, and T. Rezk. Secure Information Flow by Self-Composition. In R. Foccardi, editor, *Computer Security Foundations Workshop*, pages 100–114. IEEE Press, 2004.
5. G. Barthe, B. Grégoire, and S. Zanella Béguelin. Formal certification of code-based cryptographic proofs. In Z. Shao and B. C. Pierce, editors, *Principles of Programming Languages*, pages 90–101. ACM Press, 2009.
6. G. Barthe and C. Kunz. Certificate translation in abstract interpretation. In S. Drossopoulou, editor, *European Symposium on Programming*, volume 4960 of *Lecture Notes in Computer Science*, pages 368–382. Springer-Verlag, 2008.
7. N. Benton. Simple relational correctness proofs for static analyses and program transformations. In N. D. Jones and X. Leroy, editors, *Principles of Programming Languages*, pages 14–25. ACM Press, 2004.
8. J. Burnim and K. Sen. Asserting and checking determinism for multithreaded programs. *Communications of the ACM*, 53(6):97–105, 2010.
9. S. Chaudhuri, S. Gulwani, and R. Lublinerman. Continuity analysis of programs. In *Principles of Programming Languages*, pages 57–70, 2010.
10. M. R. Clarkson and F. B. Schneider. Hyperproperties. In *Computer Security Foundations Symposium*, pages 51–65, 2008.
11. A. Darvas, R. Hähnle, and D. Sands. A theorem proving approach to analysis of secure information flow. In D. Hutter and M. Ullmann, editors, *Security in Pervasive Computing*, volume 3450 of *Lecture Notes in Computer Science*, pages 193–209. Springer-Verlag, 2005.
12. G. Dufay, A. P. Felty, and S. Matwin. Privacy-sensitive information flow with JML. In R. Nieuwenhuis, editor, *Conference on Automated Deduction*, volume 3632 of *Lecture Notes in Computer Science*, pages 116–130. Springer-Verlag, 2005.
13. B. Godlin and O. Strichman. Regression verification. In *Design Meets Automation*, pages 466–471. ACM Press, 2009.
14. B. Goldberg, L. D. Zuck, and C. W. Barrett. Into the loops: Practical issues in translation validation for optimizing compilers. *Electr. Notes Theor. Comput. Sci.*, 132(1):53–71, 2005.
15. O. Goldreich. *Foundations of Cryptography.* Cambridge University Press, 2001-2004.
16. S. Kundu, Z. Tatlock, and S. Lerner. Proving optimizations correct using parameterized program equivalence. In *Programming Languages Design and Implementation*, pages 327–337, 2009.
17. Y. A. Liu, S. D. Stoller, and T. Teitelbaum. Static caching for incremental computation. *ACM Transactions on Programming Languages and Systems*, 20(3):546–585, 1998.
18. D. A. Naumann. From coupling relations to mated invariants for checking information flow. In *European Symposium On Research In Computer Security*, number 4189 in Lecture Notes in Computer Science, pages 279–296. Springer-Verlag, 2006.
19. G. C. Necula. Translation validation for an optimizing compiler. *ACM SIGPLAN Notices*, 35(5):83–94, 2000.
20. A. Pnueli, E. Singerman, and M. Siegel. Translation validation. In B. Steffen, editor, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1384 of *Lecture Notes in Computer Science*, pages 151–166. Springer-Verlag, 1998.
21. J. Reed and B. C. Pierce. Distance makes the types grow stronger: a calculus for differential privacy. In P. Hudak and S. Weirich, editors, *ICFP*, pages 157–168. ACM, 2010.

22. T. Terauchi and A. Aiken. Secure information flow as a safety problem. In C. Hankin and I. Siveroni, editors, *Static Analysis Symposium*, volume 3672 of *Lecture Notes in Computer Science*, pages 352–367. Springer-Verlag, 2005.
23. H. Yang. Relational separation logic. *Theoretical Computer Science*, 375(1-3):308–334, 2007.
24. A. Zaks and A. Pnueli. Covac: Compiler validation by program analysis of the cross-product. In *Formal Methods*, pages 35–51, 2008.
25. L. D. Zuck, A. Pnueli, and B. Goldberg. Voc: A methodology for the translation validation of optimizing compilers. *J. UCS*, 9(3):223–247, 2003.