Relational Verification Using Product Programs*

Gilles Barthe[1], Juan Manuel Crespo[1], and César Kunz[1,2]

[1] IMDEA Software Institute
[2] Universidad Politécnica de Madrid

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 [4,20] 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

* Partially funded by European Projects FP7-231620 HATS and FP7-256980 NESSoS, Spanish project TIN2009-14599 DESAFIOS 10, Madrid Regional project S2009TIC-1465 PROMETIDOS. C. Kunz is funded by a Juan de la Cierva Fellowship, MICINN, Spain.

M. Butler and W. Schulte (Eds.): FM 2011, LNCS 6664, pp. 200–214, 2011.

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 $\bar{\psi}$ s.t.

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

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 [2], is sound and relatively complete, but it is also impractical [19]. In a recent article, Zaks and Pnueli [21] develop another construction, called cross-product, that performs execution of c_1 and c_2 in lockstep and use it for translation validation [22], 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 and continuity, 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 for 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 [1]. In particular, we provide a formal proof of Static Caching [14], 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. [6], continuity can be often derived from the stronger notion of 1-sensitivity, see e.g. [18]. 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 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'[1]. 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 $\vDash \{\forall i.\ |a[i]-a'[i]| < \epsilon\}\, d\, \{\forall i.\ |a[i]-a'[i]| < \epsilon\}$. Since 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'.

[1] This introductory section omits the insertion of `assert` statements described in Section 3.

Source code:

```
i := 0;
while (i < N) do
  j := N−1;
  while (j > i) do
    if (a[j−1] > a[j]) then
      x := a[j];
      a[j] := a[j−1];
      a[j−1] := x;
    j−−
  i++
```

Program product:

```
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
      x := a[j]; a[j] := a[j−1]; a[j−1] := x;
    if (a′[j′−1] > a′[j′]) then
      x′ := a′[j′]; a′[j′] := a′[j′−1]; a′[j′−1] := x′;
    j−−; j′−−
  i++; i′++
```

Fig. 1. Continuity of bubble-sort algorithm

As appears from the example above, it is possible 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 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 7, 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 ≤ N) do
  x += i;
  i++
```

Transformed code:

```
j := 1;
while (j ≤ N) do
  y += j;
  j++
```

Program product (simplified):

```
i := 0; x += i; i++; j := 1;
while (i ≤ 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, which requires a greater specification and verification effort. Indeed, the sequential composition of the source and transformed program requires 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 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 $\mathcal{S} = (\mathcal{V}_i + (\mathcal{V}_a \times \mathbb{Z})) \rightharpoonup \mathbb{Z}$, and we let σ be a state in $\mathcal{S}$. The semantics of integer and boolean expressions are given by $(\llbracket e \rrbracket)_{e \in \mathsf{AExp}} : \mathcal{S} \to \mathbb{Z}$ and $(\llbracket b \rrbracket)_{b \in \mathsf{BExp}} : \mathcal{S} \to \mathbb{B}$, 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. Notice that the execution of a statement $\mathsf{assert}(b)$ blocks 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 $\llbracket \phi \rrbracket$ 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. For all separable states σ_1 and σ_2, we define $\sigma_1 \uplus \sigma_2$ as the union of finite maps: $(\sigma_1 \uplus \sigma_2)\, x$ is equal to $\sigma_1\, x$ if $x \in \mathsf{dom}(\sigma_1)$ and equal to $\sigma_2\, x$ if $x \in \mathsf{dom}(\sigma_2)$. Under this separability assumption, one can identify assertions as relations on states: $(\sigma_1, \sigma_2) \in \llbracket \phi \rrbracket$ iff $\sigma_1 \uplus \sigma_2 \in \llbracket \phi \rrbracket$. 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 $\models \{\varphi\}\, c_1 \sim c_2\, \{\psi\}$ if for all states $\sigma_1, \sigma_2, \sigma_1', \sigma_2'$ s.t. $\sigma_1 \uplus \sigma_2 \in \llbracket \varphi \rrbracket$ and $\langle c_1, \sigma_1 \rangle \Downarrow \sigma_1'$ and $\langle c_2, \sigma_2 \rangle \Downarrow \sigma_2'$, we have $\sigma_1' \uplus \sigma_2' \in \llbracket \psi \rrbracket$.*

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,

$$\frac{}{\langle \mathsf{assert}(b), \sigma \rangle \rightsquigarrow \langle \mathsf{skip}, \sigma \rangle}\ \llbracket b \rrbracket \sigma \qquad \frac{\langle c_1, \sigma \rangle \rightsquigarrow \langle c_1', \sigma' \rangle}{\langle c_1; c_2, \sigma \rangle \rightsquigarrow \langle c_1'; c_2, \sigma' \rangle} \qquad \frac{\langle c_1, \sigma \rangle \rightsquigarrow \langle \mathsf{skip}, \sigma' \rangle}{\langle c_1; c_2, \sigma \rangle \rightsquigarrow \langle c_2, \sigma' \rangle}$$

$$\frac{}{\langle \mathsf{while}\ b\ \mathsf{do}\ c, \sigma \rangle \rightsquigarrow \langle c; \mathsf{while}\ b\ \mathsf{do}\ c, \sigma \rangle}\ \llbracket b \rrbracket \sigma \qquad \frac{}{\langle \mathsf{while}\ b\ \mathsf{do}\ c, \sigma \rangle \rightsquigarrow \langle \mathsf{skip}, \sigma \rangle}\ \llbracket \neg b \rrbracket \sigma$$

Fig. 2. Program semantics (excerpt)

we adopt a stronger definition of validity, which requires that the command is non-blocking w.r.t. the precondition of the triple, where a command c is φ-nonblocking if its execution can always progress 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*-nonblocking 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 $c_1 \times c_2 \to c$. The construction of products introduces assert statements to verify that the resulting program simulates precisely the behavior of its components. These 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* $c_1 \times c_2 \to c$ *and* $\vDash \{\varphi\}\, c\, \{\psi\}$ *then* $\vDash \{\varphi\}\, c_1 \sim c_2\, \{\psi\}$*.*

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 extends 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' blocks:

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 *blocks, or* $\langle c', \sigma\rangle \Downarrow \sigma'$*.*

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

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

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

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

Fig. 3. Product construction rules

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

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

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

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

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

Fig. 4. Syntactic reduction rules

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-nonblocking 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 $\vDash \{\varphi\}\, c_1 \sim c_2\, \{\psi\}$.*

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 program transformations. These program transformations include loop optimizations, and static-caching, a complex optimization described later in this section. For each of the examples in this section, a product construction has been verified with the Why tool (and the Frama-C tool with the Jessie plugin). Building a product program from a pair of components is undecidable in general, but feasible in the scenarios we are considering. Products can be constructed in an automatic manner for structure-preserving optimizations, as well as for the verification of non-interference properties, for which a type-system based approach has already been suggested by Terauchi and Aiken. Some complex loop optimizations require involved products, in which case templates can be provided.

As shown in Table 1, most of the examples could be automatically verified: the column 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:

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

(For simplicity, we express non-interference w.r.t. scalar variables, the extension to array variables being 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'.

Table 1. Automatic validation of case studies

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

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

Fig. 5. Non-interference product

We illustrate the application of relational verification by product construction for the verification of an example drawn from [9]. 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.

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 non-interference in standard program logics, by reducing it to the verification of sequential compositions of the form $\vDash \{x_1 = x'_1 \wedge .. \wedge x_k = x'_k\}\ c; c'\ \{x_1 = x'_1 \wedge .. \wedge x_k = x'_k\}$. 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

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

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 [19], 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 Pipelining

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

Source program:

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

Transformed program:

```
j := 0;
ā[0]++; b̄[0] += ā[0];
ā[1]++;
while (j < N−2) do
    ā[j+2]++;
    b̄[j+1] += ā[j+1];
    c̄[j] += b̄[j]; j++
c̄[j] += b̄[j];
b̄[j+1] += ā[j+1];
c̄[j+1] += b̄[j+1]
```

Product program:

```
{a = ā ∧ b = b̄ ∧ c = c̄}
  i := 0; j := 0; assert(i < N);
  a[i]++; b[i] += a[i];
  c[i] += b[i]; i++;
  ā[0]++; b̄[0] += ā[0];
  assert(i < N);
  a[i]++; b[i] += a[i];
  c[i] += b[i]; i++; ā[1]++;
  assert(i < N ⇔ j < N−2);
  while (i < N) do
     a[i]++; b[i] += a[i]; c[i] += b[i]; i++
     ā[j+2]++; b̄[j+1] += ā[j+1];
     c̄[j] += b̄[j]; j++
     assert(i < N ⇔ j < N−2);
  c̄[j] += b̄[j]; b̄[j+1] += ā[j+1]; c̄[j+1] += b̄[j+1]
{a = ā ∧ b = b̄ ∧ c = c̄}
```

Fig. 6. Loop pipelining

Consider the simple example shown in Fig. 6 (drawn from [13].) Assume a, b, and c are arrays of size N, with $2 \leq N$.

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'.

Static Caching

In this section we turn our attention to static caching [14], 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 Fig. 7. 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

Source program:

```
i1 := 0;
while (i1 ≤ N−M) do
  s[i1] := 0; k1 := 0;
  while (k1 ≤ M−1) do
    l1 := 0;
    while (l1 ≤ L−1) do
      s[i1] += a[i1+k1, l1]; l1++;
    k1++;
  i1++
```

Transformed program:

```
t[0] := 0; k2 := 0;
while (k2 ≤ M−1) do
  b[k2] := 0; l2 := 0;
  while (l2 ≤ L−1) do
    b[k2] += a[k2, l2]; l2++;
  t[0] += b[k2]; k2++;
i2 := 1;
while (i2 ≤ N−M) do
  b[i2+M−1] := 0; l2 := 0;
  while (l2 ≤ L−1) do
    b[i2+M−1] += a[i2+M−1, l2]; l2++;
  z := b[i2+M−1] − b[i2−1];
  t[i2] := t[i2−1] + z; i2++
```

Fig. 7. Static caching: source and optimized code

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 8 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.

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 [3], and non-interference:

Program equivalence. Relational Hoare Logics [4] (RHL), and its cousin Relational Separation Logic [20], provide 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: one can isolate a core fragment cRHL of RHL such that derivability in this fragment coincides with derivability of the product in Hoare logic: i.e. $\vdash_{\mathrm{cRHL}} \{\varphi\}\, c_1 \sim c_2\, \{\psi\}$ iff $c_1 \times c_2 \to c$ and $\vdash \{\varphi\} c \{\psi\}$. Moreover, one can define for every refinement relation $\succcurlyeq$ an extension $\mathrm{cRHL}_{\succcurlyeq}$ of the core logic such that $\vdash_{\mathrm{cRHL}_{\succcurlyeq}} \{\varphi\}\, c_1 \sim c_2\, \{\psi\}$ iff $c_1 \times c_2 \to c$ and $\vdash \{\varphi\} c \{\psi\}$.

Product program:

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

Fig. 8. Static caching: Program product

$$Inv_2 \doteq i_1{=}0 \wedge k_1{=}k_2 \wedge l_1{=}l_2 \wedge k_1 \le M \wedge l_1 \le L \wedge s[i_1] = t[0]+b[k_1] = \textstyle\sum_{k'=0}^{k_1-1} b[k'] + b[k_1] \wedge \forall\, k' \in [0, k_1).\ b[k'] = \sum_{l'=0}^{L-1} a[k', l'] \wedge b[k_1] = \sum_{l'=0}^{l_1-1} a[k_1, l']$$

$$Inv_3 \doteq i_1 = i_2 \wedge i_1 \le N-M+1 \wedge \forall\, i' \in [0, i_1) \Rightarrow s[i'] = t[i'] = \textstyle\sum_{k'=0}^{M-1} b[k'+i'] \wedge \forall\, i' \in [0, i_1+M-1).\ b[i'] = \sum_{l'=0}^{L-1} a[i', l']$$

$$Inv_4 \doteq Inv_3 \wedge k_1 \le M \wedge l_2 \le L \wedge b[i_2+M-1] = \textstyle\sum_{l'=0}^{l_2-1} a[i_2+M-1, l'] \wedge s[i_1] = \sum_{k'=0}^{k_1-1} b[k'+i_1]$$

$$Inv_6 \doteq Inv_3 \wedge k_1 \le M \wedge l_1 \le L \wedge b[i_2+M-1] = \textstyle\sum_{l'=0}^{L-1} a[i_2+M-1, l'] \wedge s[i_1] = \sum_{k'=0}^{k_1-1} b[k'+i_1] + \sum_{l'=0}^{l_1-1} a[i_1+k_1, l']$$

Fig. 9. Static caching: Loop invariants (excerpt)

Compiler correctness. Translation validation [17,22,1] 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 [11], the permute rule is generalized to account for such optimizations.

In an independent line of work, Necula [16] 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 [13] 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 [16] with the PERMUTE rule is provided by Kundu et al. [13]. 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 [2,8] 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 [19] suggested to improve self-composition by a type directed transformation, a special case of our product construction. Naumann [15] builds 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* [21], 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.

Beyond properties Other applications of relational methods include regression verification [10], verification of 2-safety properties [19,7], including determinism [5]. Furthermore, quantitative properties such as continuity [6] or indistinguishability [12] appear as a natural generalization of 2-safety properties.

Clarkson and Schneider [7] 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 (intend to) provide a verification method.

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 nondeterminism 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 [3] and privacy [18].

References

1. Barrett, C.W., Fang, Y., Goldberg, B., Hu, Y., Pnueli, A., Zuck, L.D.: TVOC: A translation validator for optimizing compilers. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 291–295. Springer, Heidelberg (2005)
2. Barthe, G., D'Argenio, P., Rezk, T.: Secure Information Flow by Self-Composition. In: Foccardi, R. (ed.) Computer Security Foundations Workshop, pp. 100–114. IEEE Press, Los Alamitos (2004)
3. Barthe, G., Grégoire, B., Zanella Béguelin, S.: Formal certification of code-based cryptographic proofs. In: Shao, Z., Pierce, B.C. (eds.) Principles of Programming Languages, pp. 90–101. ACM Press, New York (2009)
4. Benton, N.: Simple relational correctness proofs for static analyses and program transformations. In: Jones, N.D., Leroy, X. (eds.) Principles of Programming Languages, pp. 14–25. ACM Press, New York (2004)
5. Burnim, J., Sen, K.: Asserting and checking determinism for multithreaded programs. Communications of the ACM 53(6), 97–105 (2010)
6. Chaudhuri, S., Gulwani, S., Lublinerman, R.: Continuity analysis of programs. In: Principles of Programming Languages, pp. 57–70 (2010)
7. Clarkson, M.R., Schneider, F.B.: Hyperproperties. In: Computer Security Foundations Symposium, pp. 51–65 (2008)
8. Darvas, A., Hähnle, R., Sands, D.: A theorem proving approach to analysis of secure information flow. In: Hutter, D., Ullmann, M. (eds.) SPC 2005. LNCS, vol. 3450, pp. 193–209. Springer, Heidelberg (2005)
9. Dufay, G., Felty, A.P., Matwin, S.: Privacy-sensitive information flow with JML. In: Nieuwenhuis, R. (ed.) CADE 2005. LNCS (LNAI), vol. 3632, pp. 116–130. Springer, Heidelberg (2005)
10. Godlin, B., Strichman, O.: Regression verification. In: Design Meets Automation, pp. 466–471. ACM Press, New York (2009)
11. Goldberg, B., Zuck, L.D., Barrett, C.W.: Into the loops: Practical issues in translation validation for optimizing compilers. Electr. Notes Theor. Comput. Sci. 132(1), 53–71 (2005)
12. Goldreich, O.: Foundations of Cryptography. Cambridge University Press, Cambridge (2004)

13. Kundu, S., Tatlock, Z., Lerner, S.: Proving optimizations correct using parameterized program equivalence. In: Programming Languages Design and Implementation, pp. 327–337 (2009)
14. Liu, Y.A., Stoller, S.D., Teitelbaum, T.: Static caching for incremental computation. ACM Transactions on Programming Languages and Systems 20(3), 546–585 (1998)
15. Naumann, D.A.: From coupling relations to mated invariants for checking information flow. In: Gollmann, D., Meier, J., Sabelfeld, A. (eds.) ESORICS 2006. LNCS, vol. 4189, pp. 279–296. Springer, Heidelberg (2006)
16. Necula, G.C.: Translation validation for an optimizing compiler. ACM SIGPLAN Notices 35(5), 83–94 (2000)
17. Pnueli, A., Singerman, E., Siegel, M.: Translation validation. In: Steffen, B. (ed.) TACAS 1998. LNCS, vol. 1384, pp. 151–166. Springer, Heidelberg (1998)
18. Reed, J., Pierce, B.C.: Distance makes the types grow stronger: a calculus for differential privacy. In: Hudak, P., Weirich, S. (eds.) ICFP, pp. 157–168. ACM, New York (2010)
19. Terauchi, T., Aiken, A.: Secure information flow as a safety problem. In: Hankin, C., Siveroni, I. (eds.) SAS 2005. LNCS, vol. 3672, pp. 352–367. Springer, Heidelberg (2005)
20. Yang, H.: Relational separation logic. Theoretical Computer Science 375(1-3), 308–334 (2007)
21. Zaks, A., Pnueli, A.: CoVaC: Compiler validation by program analysis of the cross-product. In: Cuellar, J., Sere, K. (eds.) FM 2008. LNCS, vol. 5014, pp. 35–51. Springer, Heidelberg (2008)
22. Zuck, L.D., Pnueli, A., Goldberg, B.: Voc: A methodology for the translation validation of optimizing compilers. J. UCS 9(3), 223–247 (2003)