--- Abstract ---

Relational program logics - formalisms for specifying and verifying properties about two programs or two runs of the same program.

Examples of these properties: correctness of compiler optimizations, equivalence between two implementations of an abstract data type, non-interference and determinism.

Non-interference – a strict multilevel security policy model: if a low user is working on the machine, it will respond in exactly the same manner (on the low outputs) whether or not a high user is working with sensitive data. **In this context**: given a program c and a set of public variables x1, . . . , xk, the property ensures that two terminating runs of c starting in states with equal public variables, end in states with equal public variables.

General notion of **product program** that supports a **direct reduction of relational verification to standard verification**.

--- Section 1 - Introduction ---

Relational reasoning allows us to establish that: the same program behaves similarly on two different runs **or** two programs execute in a related fashion.

Goal: find c, ϕ and ψ such that

$$\models \{\bar{\phi}\}\ c\ \{\bar{\psi}\} \Rightarrow \models \{\phi\}\ c1 \sim c2\ \{\psi\}$$

where ϕ and ψ are relations on the states of the command c1 and the states of the command c2; $\bar{\phi}$ 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.

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 asynchronous 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.

--- Section 2 – Motivating examples ---

Continuity (relational property) – a program is continuous if small variations on its inputs only causes small variations on its output. Continuity can be often derived from the stronger notion of 1-sensitivity.

1-sensitivity – a program is 1-sensitive if the variation of the outputs of two different runs is upper bounded by the variation of the corresponding inputs.

Dúvida: como interpretar?

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

A first intuition on the construction of products from structurally dissimilar components is shown in the following basic example (N >= 0):

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 maximizes synchronization instead of relying only on self-composition, which would require invariants much more complex than what is needed by the construction of the product: the loop invariant *i = j ^ x = y* is sufficient to verify that the two programs above satisfy the pre and post-relation x = y.

--- Section 3 – Program Products ---

The reduction of relational verification into standard verification is based on the capacity of constructing a product program *c* that simulates the execution steps of a pair of its constituents, c1 and c2.

Separable commands – two commands c1 and c2 are separable if they have disjoint set of variables: **var**(c1) ∩ **var**(c2) = ∅. Also, two states are separable if they have disjoint domains.

Valid relational specifications:

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 $\sigma_1' \uplus \sigma_2' \in [\![\psi]\!]$.*

Valid Hoare triple (stronger - nonstuck):

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 $\sigma' \in [\![\psi]\!]$.*

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 discarded during the program verification phase.

For example, in the rule that synchronizes two loop statements, the insertion of the statement **assert**(b1 ⇔ b2) just before the evaluation of the loop guards b1 and b2 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 c1 and c2, then the validity of a relational judgment between c1 and c2 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 $\models \{\varphi\}\, c\, \{\psi\}$ then $\models \{\varphi\}\, c_1 \sim c_2\, \{\psi\}$.*

Next in this section is 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, in the sense that every execution of *c* is an execution of *c'* except when *c'* gets stuck. It is denoted with a judgment of the form *c* >= *c'*.

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

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 $\models \{\varphi\}\, c_1 \sim c_2\, \{\psi\}$.*

Reduces the problem of proving the validity of a relational judgment into 2 steps: the construction of the corresponding program product and a standard verification over the program product.

--- Section 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.

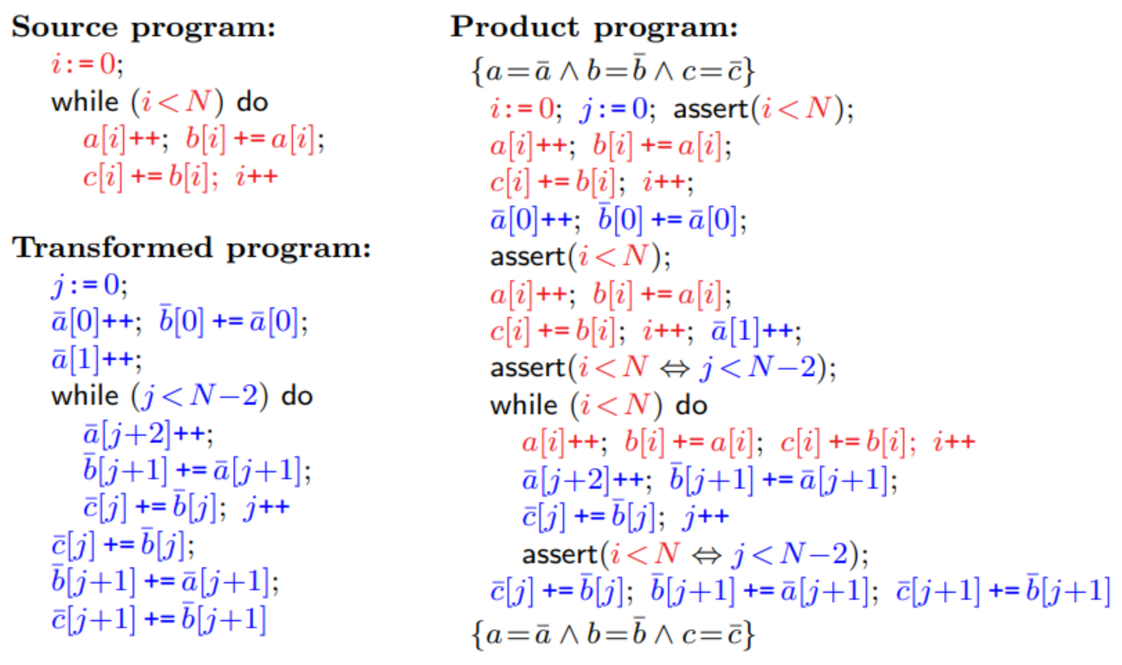
Example	SMT/P.O.
Non-interference	42/42
Loop alignment	49/49
Loop pipelining	73/73
Loop unswitching	123/123
Code sinking	435/435
Static caching	162/176

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

P.O. - number of proof obligations generated **SMT –** proofs automatically discharged by SMT solvers

Loop pipelining is an optimization that reduces the proximity of memory reference inside a loop, 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

Dúvida: como o transformed program reduz a proximidade?