# Beyond 2-Safety: Asymmetric Product Programs for Relational Program Verification

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

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

**Abstract.** Relational Hoare Logic is a generalization of Hoare logic that allows reasoning about executions of two programs, or two executions of the same program. It can be used to verify that a program is robust or (information flow) secure, and that two programs are observationally equivalent. Product programs provide a means to reduce verification of relational judgments to the verification of a (standard) Hoare judgment, and open the possibility of applying standard verification tools to relational properties. However, previous notions of product programs are defined for deterministic and structured programs. Moreover, these notions are symmetric, and cannot be applied to properties such as refinement, which are asymmetric and involve universal quantification on the traces of the first program and existential quantification on the traces of the second program.

Asymmetric products generalize previous notions of products in three directions: they are based on a control-flow graph representation of programs, they are applicable to non-deterministic languages, and they are by construction asymmetric. Thanks to these characteristics, asymmetric products allow to validate abstraction/refinement relations between two programs, and to prove the correctness of advanced loop optimizations that could not be handled by our previous work. We validate their effectiveness by applying a prototype implementation to verify representative examples from translation validation and predicate abstraction.

## 1 Introduction

Program verification tools provide an effective means to verify trace properties of programs. However, many properties of interest are 2-properties, i.e. consider pairs of traces, rather than traces; examples include non-interference and robustness, which consider two executions of the same program, and abstraction/equivalence/refinement properties, which relate executions of two programs. Relational Hoare logic [8] generalizes Hoare logic by allowing to reason about two programs, and provides an elegant theoretical framework to reason about 2-properties. However, relational Hoare logic is confined to reason about universally quantified statements over traces, and only relates programs with the same termination behavior. Thus, relational Hoare logic cannot capture notions of refinement, and more generally properties that involve an alternation of existential and universal quantification. Moreover, relational Hoare logic is not tool supported.

Product programs [20,4] provide a means to reduce verification of relational Hoare logic quadruples to verification of standard Hoare triples. Informally, the product program construction transforms two programs $P_1$ and $P_2$ into a single program $P$ that soundly abstracts the behavior of $P_1$ and $P_2$, so that relational verification over $P_1$ and $P_2$ can be reduced to verification of $P$. Product programs are attractive, because they allow reusing existing verification tools for relational properties. However, like relational Hoare logic, the current definition of product program is only applicable to universally quantified statements over traces. Moreover, the construction of product programs has been confined to structured and deterministic programs written in a single language. This article introduces asymmetric (left or right) product programs, which generalize symmetric products from [20,4], and allow showing abstraction/refinement properties, which are typically of the form: for all execution of the first program, there is a related execution of the second program. Furthermore, asymmetric product are based on a flow-graph representation of programs, which provides significant advantages over previous works. In particular, asymmetric products can relate programs: 1. with different termination behaviors; 2. including non-deterministic statements; 3. written in two different languages (provided they support a control flow graph representation). Finally, asymmetric products allow justifying some loop transformations that where out of reach of our previous work on translation validation. We evaluate our method on representative examples, using a prototype implementation that builds product programs and sends the verification task to the Why platform.

Section 2 motivates left products with examples of predicate abstraction and translation validation. Sections 3 and 4 introduce the notion of left product and show how they can be used to reduce relational verification to functional verification. Section 5 introduces full products, a symmetric variant of left products that is used to validate examples of translation validation that were not covered by [4]. Section 6 presents an overview of our implementation.

## 2 Motivating Examples

In this section we illustrate our technique through some examples. The first two are abstraction validation examples and for their verification we use the asymmetric framework, while for the verification of the loop optimization, we use a stronger version of the method, introduced in Section 5.

For both domains of application, we first provide an informal overview of the verification technique. Throughout the rest of the paper, we refer back to these examples in order to illustrate the technical concepts and results.

### 2.1 Abstraction Validation

The correctness of the verification methods based on program abstraction relies on the soundness of its abstraction mechanism. Since such abstraction mechanisms are increasingly complex it becomes desirable to perform a posteriori, independent validation of their results.

In general, abstractions induce some loss of information, represented in the abstract programs as non-deterministic statements. The extensions presented in this paper enable our framework to cope with non-determinism.

*Predicate Abstraction.* Predicate abstraction [1,12] reduces complexity of a program verification to the analysis of a bounded-state program, by representing infinite-state systems in terms of a finite set of user-provided predicates. The program on the left of Figure 1, drawn from [1], partitions a singly linked list of integers into two lists: one containing the elements with value greater than the parameter $v$ and the other one containing the cells with value less than or equal to $v$. The program on the right represents the predicate abstraction of the program on the left, w.r.t. a set of user-provided boolean predicates: $\{curr = \mathsf{null}, prev = \mathsf{null}, curr{\mapsto}val > v, prev{\mapsto}val < v\}$. The abstraction is performed by representing each boolean predicate with a boolean variable: e.g., $\overline{curr}$ represents the condition $curr = \mathsf{null}$. The effect of the instructions of the original program is captured by assignments and assert statements involving the boolean variables of the abstraction: e.g. the effect of the assignment $prev := \mathsf{null}$ on the predicate $prev = \mathsf{null}$ is reflected by the assignment $\overline{prev} := \mathsf{true}$ on the right program. Note that some of the abstract predicates will have an unknown value after some of the concrete instructions, as is the case with the predicate $curr = \mathsf{null}$ after the assignment $curr := *\, l$, reflected by the non-deterministic assignment $\overline{curr} := ?$.

We consider the problem of automatically validating abstractions that are expressed as non-deterministic programs in some variant of the original programming language. Our goal is to verify that the program on the right soundly abstracts the original one, i.e. any execution path of the original program can be simulated by an execution path of the abstracted program. In order to establish the correctness of the program abstraction, we must verify a simulation relation between the execution traces of both programs. This simulation is captured by a new program constructed from the original and abstract programs, shown in Figure 4, providing a fixed control flow for the simulation relation.

The validation of the abstraction is carried over the product program in Fig. 4 by two independent verification steps. One must first verify that the product program captures correctly the synchronous executions of the original and abstract programs, i.e., that for any trace on the left program there exists a trace on the right program. We say then that the graph is a *left product* and it satisfies the properties stated in Lemma 2. In a second step, one must check that the product program satisfies the given refinement relation, stated as a relational invariant specification: $(curr = \mathsf{null} \Leftrightarrow \overline{curr}) \wedge (prev = \mathsf{null} \Leftrightarrow \overline{prev}) \wedge (curr{\mapsto}val < v \Leftrightarrow currV) \wedge (prev{\mapsto}val < v \Leftrightarrow prevV)$

*Numeric Abstraction.* Numeric abstraction [14] is a similar program abstraction strategy based on a shape analysis defined from user-provided size abstractions. The output of this transformation is not necessarily a bounded-state program, but it can be used to establish some properties of the original program, e.g., termination behavior, or resource consumption.

Figure 2 shows an example of a source and an abstract programs, drawn from [14]. The program on the left performs left to right, depth first traversal of the binary tree pointed by its argument. It maintains a stack of nodes to be processed. On each iteration, the top of the stack is removed and its children (if any) are added. The program on the right of the figure is a numeric abstraction of the source program that explicitly keeps track of the changes in data structure sizes. In the abstract program, $tsizeroot$ represents the number of nodes in the tree, $slen$ the length of the list representing the stack and $ssize$ the number of nodes contained in the trees held within the stack. More

```
$curr := *l$; $prev :=$ null;
$newl :=$ null;
while ($curr \neq$ null) do
  $nextCurr := curr \mapsto next$;
  if ($curr \mapsto val > v$) then
    if ($prev \neq$ null) then
      $prev \mapsto next := nextCurr$;
    if ($curr = *l$) then $*l := nextCurr$;
    $curr \mapsto next := newl$; $newl := curr$;
  else
    $prev := curr$;
  $curr := nextCurr$;
```

```
$\overline{\mathbf{curr}} := ?$; $\overline{\mathbf{prev}} :=$ true;
$\mathbf{currV} := ?$; $\mathbf{prevV} := ?$;
while ($*$) do
  assert($\neg\overline{\mathbf{curr}}$);
  if ($*$) then
    assert($\mathbf{currV}$);
    if ($*$) then
      assert($\neg\overline{\mathbf{prev}}$);
  else
    assert($\mathbf{currV} = \mathbf{false}$);
    $\overline{\mathbf{prev}} := \overline{\mathbf{curr}}$;
    $\mathbf{prevV} := \mathbf{currV}$;
  $\overline{\mathbf{curr}} := ?$; $\mathbf{currV} := ?$;
assert($\overline{\mathbf{curr}}$);
```

**Fig. 1.** Predicate abstraction

precisely, the user-provided abstractions are defined as inductive predicates over acyclic heap structures, e.g.:

$$\frac{}{\mathsf{ListLength}(\mathsf{null}, 0)} \qquad \frac{\mathsf{ListLength}(ls \mapsto tail, n)}{\mathsf{ListLength}(ls, n+1)}\; ls \neq \mathsf{null}$$

$$\frac{}{\mathsf{TreeSize}(\mathsf{null}, 0)} \qquad \frac{\mathsf{TreeSize}(t \mapsto left, n_l) \qquad \mathsf{TreeSize}(t \mapsto right, n_r)}{\mathsf{TreeSize}(t, n_l+n_r+1)}\; t \neq \mathsf{null}$$

Note that upon entering the loop, we do not have information on the size of the first tree contained in the stack, nor of the size of the trees in the rest of the stack. This is represented in the abstraction by a non-deterministic assignment.

As in the previous example, we can verify a posteriori that the numeric program soundly abstracts a heap manipulating program by constructing a product program that fixes the control flow of the simulation to be verified. The product program shown in Figure 5 is totally synchronized, in the sense that every program edge represents a simultaneous execution of the program components.

The simulation relation is defined in terms of the user-provided size abstractions. This relational specification makes explicit the correspondence between the abstract numeric variables and the size predicates over the original data structures; these size relations include, e.g., $\mathsf{ListLength}(st, slen)$ and $\mathsf{TreeSize}(root, tsizeroot)$, which must hold whenever the variables are in scope.

We develop the notion of left product used for abstraction validation in Section 3.

## 2.2 Translation Validation

Translation validation [3,16] 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. In previous work, we have used a notion of program products to formally verify the correctness of several program optimizations [4]. An important limitation of our previous notion of program

```
st := push(root, 0);                      assert(0 ≤ tsizeroot);
                                          slen := 1; ssize := tsizeroot;
while (st ≠ 0) do                         while (slen > 0) do
  tail := st → next;                        tsize := ?; ssizetail := ?;
                                            assert(0 ≤ tsize ∧ 0 ≤ ssizetail);
                                            assert(ssize = tsize + ssizetail);
  if (st → tree = 0) then                   if (tsize = 0) then
    free(st); st := tail;                     slen−−;
  else                                      else
    tail := push(st → tree → right, tail);    tsizel := ?; tsizer := ?;
    tail := push(st → tree → left, tail);     assert(0 ≤ tsizel ∧ 0 ≤ tsizer);
    free(st);                                 assert(tsize = tsizel + tsizer + 1);
    st := tail;                               ssize := tsizel + tsizer + ssizetail;
                                              slen++;
```

**Fig. 2.** Numeric abstraction

```
a: x := 0;                 0: i := 0;
b: while (x < NM) do       1: while (i < N) do
     a[x] := f(x);               j := 0;
     x++                   2:    while (j < M) do
                                   A[i, j] := f(iM + j); j++;
                                 i++
```

**Fig. 3.** Loop tiling example

products is that they are required to be representable syntactically as structured code. The extension provided in this work enables the verification of more complex loop optimizations that were not considered in previous work.

Loop tiling is an optimization that splits the execution of a loop into smaller blocks, improving the cache performance. If the loop accesses a block of contiguous data during its execution, splitting the block in fragments that fits the cache size can help avoiding cache misses, depending on the target architecture. The program at the right of Fig. 3 shows the result of applying a loop tiling transformation to the code at the left. The traversal of a block of size $NM$ is split into $N$ iterations accessing smaller blocks of size $M$, by the introduction of an inner loop and new iteration variables $i$ and $j$. It is not hard to see that the iteration space of the outermost loops are equal and that the relational invariant $x = iM + j$ holds.

The structural dissimilarity of the original and transformed loop is a main obstacle for the application of our previous relational verification method. However, the relaxed notion of program product presented in this article can be used to validate this transformation. Figure 6 shows a possible product of the two programs in Fig. 3. The loop bodies (i.e., edges $\langle 2, 2\rangle$ and $\langle b, b\rangle$) are executed synchronously, represented by edge $\langle (b, 2), (b, 2)\rangle$. Notice that asynchronous edges represents the transitions of the right program that cannot be matched with transitions on the left program. We develop the notion of full products for the validation of compiler optimizations in Section 5.

## 3 Simulation by Left Products

We define a general notion of product program and prove that under mild conditions they mimic the behavior of their constituents. We adopt a representation of programs based on labeled directed graphs. Nodes correspond to program points, and include an initial and a final node; for simplicity, we assume their unicity. Edges are labeled with statements from the set $\mathsf{Stmt}$.

**Definition 1 (Program).** *A program $P$ is a tuple $\langle \mathcal{N}, \mathcal{E}, G\rangle$, where $\langle \mathcal{N}, \mathcal{E}\rangle$ is a directed graph with unique source* $\mathsf{in} \in \mathcal{N}$ *and sink* $\mathsf{out} \in \mathcal{N}$*, and $G : \mathcal{E} \to \mathsf{Stmt}$ maps edges to statements.*

The semantics of statements is given by a mapping $[\![.]\!] : \mathsf{Stmt} \to \mathcal{P}(\mathcal{S} \times \mathcal{S})$, where $\mathcal{S}$ is a set of states. A configuration is a pair $\langle l, \sigma\rangle$, where $l \in \mathcal{N}$ and $\sigma \in \mathcal{S}$; we let $\langle l, \sigma\rangle \rightsquigarrow \langle l', \sigma'\rangle$ stand for $(\sigma, \sigma') \in [\![G\langle l, l'\rangle]\!]$. A trace is a sequence of configurations s.t. the first configuration is of the form $\langle \mathsf{in}, \sigma\rangle$, and $(\sigma, \sigma') \in [\![G\langle l, l'\rangle]\!]$ for any two consecutive elements $\langle l, \sigma\rangle$ and $\langle l', \sigma'\rangle$ of the sequence; we let $\mathsf{Tr}(P)$ denote the set of traces of $P$. Moreover, an execution is a trace whose last configuration is of the form $\langle \mathsf{out}, \sigma\rangle$; we let $\mathsf{Ex}(P) \subseteq \mathsf{Tr}(P)$ denote the set of executions of $P$. Finally, we write $(\sigma, \sigma') \in [\![P]\!]$ if there exists an execution of $P$ with initial state $\sigma$ and final state $\sigma'$; and we say that $P$ is strongly terminating, written $P \Downarrow^{\star}$, iff for every $t \in \mathsf{Tr}(P)$ there exists $t' \in \mathsf{Ex}(P)$ such that $t$ is a prefix of $t'$. For example, the abstract program in the right of Fig. 1 is strongly terminating, since every execution trace can be extended to a terminating trace by suitable choices when evaluating the non-deterministic guards.

### 3.1 Synchronized Products

Informally, a product of two programs is a program that combines their effects. We begin with a weaker definition (Def. 3) which only guarantees that the behavior of products is included in the behavior of their constituents. Then, we provide a sufficient condition (Def. 4) for the behavior of products to coincide with the behavior of its constituents.

One practical goal of this article is to be able to perform relational reasoning about programs that are written in the same language, by using off-the-shelf verification tools for this language. The embedding relies on separability; our conditions are inspired from self-composition [5], and are reminiscent of the monotonicity and frame properties of separation logic [19].

Assume given two functions $\pi_1, \pi_2 : \mathcal{S} \to \mathcal{S}$ s.t. for all $\sigma, \sigma' \in \mathcal{S}$, $\sigma = \sigma'$ iff $\pi_1(\sigma) = \pi_1(\sigma')$ and $\pi_2(\sigma) = \pi_2(\sigma')$. Given two states $\sigma_1, \sigma_2 \in \mathcal{S}$, we define $\sigma_1 \uplus \sigma_2 \in \mathcal{S}$ to be the unique, if it exists, state $\sigma$ s.t. $\pi_1(\sigma) = \sigma_1$ and $\pi_2(\sigma) = \sigma_2$.

**Definition 2 (Separable statements).** *A statement $c$ is a left statement iff for all $\sigma_1, \sigma_2$ in $\mathcal{S}$ s.t. $\sigma_1 \uplus \sigma_2$ is defined:*

1. *for all $\sigma_1' \in \mathcal{S}$, if $(\sigma_1, \sigma_1') \in [\![c]\!]$, then $\sigma_1' \uplus \sigma_2$ is defined and $(\sigma_1 \uplus \sigma_2, \sigma_1' \uplus \sigma_2) \in [\![c]\!]$;*
2. *for all $\sigma' \in \mathcal{S}$, if $(\sigma_1 \uplus \sigma_2, \sigma') \in [\![c]\!]$, then there exists $\sigma_1' \in \mathcal{S}$ s.t. $(\sigma_1, \sigma_1') \in [\![c]\!]$ and $\sigma_1' \uplus \sigma_2 = \sigma'$.*

Right statements are defined symmetrically. Two statements $c_1$ and $c_2$ are separable iff $c_1$ is a left statement and $c_2$ is a right statement. Finally, two programs $P_1$ and $P_2$ are separable iff $P_1$ is a left program, i.e. it only contains left statements, and $P_2$ is a right program, i.e. it only contains right statements. In this section, we let $P_1 = \langle \mathcal{N}_1, \mathcal{E}_1, G_1 \rangle$ and $P_2 = \langle \mathcal{N}_2, \mathcal{E}_2, G_2 \rangle$ be separable programs.

*Example 1.* The programs in Fig. 1 manipulate disjoint fragments of scalar state, thus they are clearly separable. Dynamic memory manipulation may break separability if both the left and right programs invoke a non-deterministic allocator. However, in this particular example one of the product components does not manipulate the heap.

**Definition 3 (Product).** *Let $P = \langle \mathcal{N}, \mathcal{E}, G \rangle$ be a program with statements in* $\mathsf{Stmt}$. *$P$ is a product of $P_1$ and $P_2$, written $P \in P_1 \times P_2$, iff $\mathcal{N} \subseteq \mathcal{N}_1 \times \mathcal{N}_2$, and $(\mathsf{in}_1, \mathsf{in}_2) \in \mathcal{N}$ and for all $(l_1, l_2) \in \mathcal{N}$ $l_1 = \mathsf{out}_1$ iff $l_2 = \mathsf{out}_2$, and every edge $e \in \mathcal{E}$ is of one of the forms:*

- left edge: $(l_1, l_2) \overset{\mathsf{l}}{\mapsto} (l_1', l_2)$, *with $\langle l_1, l_1' \rangle$ in $\mathcal{E}_1$, and $[\![G\, e]\!] = [\![G_1\, \langle l_1, l_1' \rangle]\!]$;*
- synchronous edge: $(l_1, l_2) \Mapsto (l_1', l_2')$, *with edges $\langle l_1, l_1' \rangle$ in $\mathcal{E}_1$ and $\langle l_2, l_2' \rangle$ in $\mathcal{E}_2$, and $[\![G\, e]\!] = [\![G_1\, \langle l_1, l_1' \rangle]\!] \circ [\![G_2\, \langle l_2, l_2' \rangle]\!]$; or*
- right edge: $(l_1, l_2) \overset{\mathsf{r}}{\mapsto} (l_1, l_2')$, *with $\langle l_2, l_2' \rangle$ in $\mathcal{E}_2$, and $[\![G\, e]\!] = [\![G_2\, \langle l_2, l_2' \rangle]\!]$.*

For simplicity, the notion of product program is defined for two programs of the same language. However, the definition readily extends to 2-languages products, i.e. products of programs written in two distinct languages. Alternatively, 2-languages products can be encoded in our setting: given two programming languages with statements in $\mathsf{Stmt}_1$ and $\mathsf{Stmt}_2$ respectively, and with state spaces $\mathcal{S}_1$ and $\mathcal{S}_2$ respectively and semantics $[\![.]\!]_1 : \mathsf{Stmt}_1 \to \mathcal{P}(\mathcal{S}_1 \times \mathcal{S}_1)$ and $[\![.]\!]_2 : \mathsf{Stmt}_2 \to \mathcal{P}(\mathcal{S}_2 \times \mathcal{S}_2)$, one can define $\mathsf{Stmt} = \mathsf{Stmt}_1 + \mathsf{Stmt}_2$, and $\mathcal{S} = \mathcal{S}_1 + \mathcal{S}_2$, and $[\![.]\!] = [\![.]\!]_1 + [\![.]\!]_2$. Then, programs of the first and second languages can be embedded in a semantic-preserving manner into the "sum" language, and one can use the notion of product program the usual way.

*Example 2.* The definition of products ensures that every edge in $\mathcal{E}$ represents either an execution step of program $P_1$, an execution step of program $P_2$, or a pair of simultaneous steps of both programs. The program product in Fig. 4 contains both synchronous and left edges. In this particular example, the left edges represent portions of the original program that are sliced out in the abstract program, since they do not have an effect on the validity of the boolean predicates.

Products underapproximate the behavior of their constituents, i.e, every trace of $P \in P_1 \times P_2$ is a combination of a trace of $P_1$ and a trace of $P_2$. We formalize this fact using left and right projections of traces. The left projection of an execution step in $P$ is defined by case analysis: 1. if $\langle (l_1, l_2), \sigma \rangle \rightsquigarrow \langle (l_1', l_2'), \sigma' \rangle$ and either $(l_1, l_2) \overset{\mathsf{l}}{\mapsto} (l_1', l_2')$ or $(l_1, l_2) \Mapsto (l_1', l_2')$, then the left projection is defined as $\langle l_1, \pi_1(\sigma) \rangle \rightsquigarrow \langle l_1', \pi_1(\sigma') \rangle$; 2. otherwise, the left projection is undefined. The left projection $\pi_1(t)$ of a trace $t$ is then defined as the concatenation of the left projections of its steps (steps with undefined projections are omitted). The right projection $\pi_2(t)$ of a trace $t$ is defined in a similar way.

**Lemma 1.** *Let $P \in P_1 \times P_2$. For all $t \in \mathsf{Tr}(P)$, $\pi_1(t) \in \mathsf{Tr}(P_1)$ and $\pi_2(t) \in \mathsf{Tr}(P_2)$.*

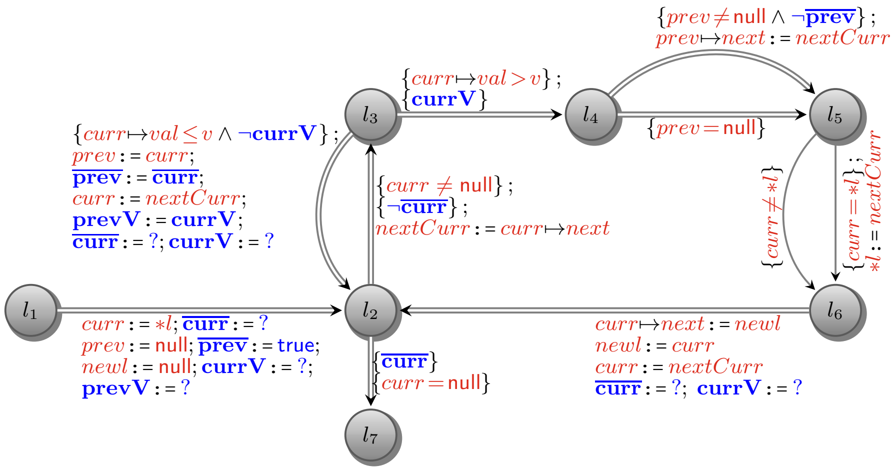

$l_1$
$l_2$
$l_3$
$l_4$
$l_5$
$l_6$
$l_7$
$\{prev \neq \text{null} \wedge \neg\overline{\mathbf{prev}}\}$ ; $prev{\mapsto}next := nextCurr$
$\{curr{\mapsto}val > v\}$ ; $\{\mathbf{currV}\}$
$\{prev = \text{null}\}$
$\{curr{\mapsto}val \leq v \wedge \neg\mathbf{currV}\}$ ; $prev := curr$; $\overline{\mathbf{prev}} := \overline{\mathbf{curr}}$; $curr := nextCurr$; $\mathbf{prevV} := \mathbf{currV}$; $\overline{\mathbf{curr}} := ?$; $\mathbf{currV} := ?$
$\{curr \neq \text{null}\}$ ; $\{\neg\overline{\mathbf{curr}}\}$ ; $nextCurr := curr{\mapsto}next$
$\{curr \neq *l\}$
$\{curr = *l\}$ ; $*l := nextCurr$
$curr := *l$; $\overline{\mathbf{curr}} := ?$ $prev := \text{null}$; $\overline{\mathbf{prev}} := \text{true}$; $newl := \text{null}$; $\mathbf{currV} := ?$; $\mathbf{prevV} := ?$
$\{\overline{\mathbf{curr}}\}$ $\{curr = \text{null}\}$
$curr{\mapsto}next := newl$ $newl := curr$ $curr := nextCurr$ $\overline{\mathbf{curr}} := ?$; $\mathbf{currV} := ?$

**Fig. 4.** Predicate abstraction example — Product program

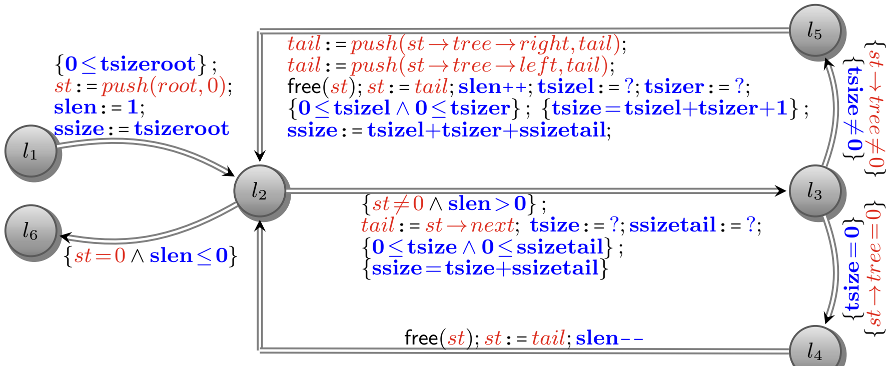

$l_1$
$l_2$
$l_3$
$l_4$
$l_5$
$l_6$
$\{0 \leq \mathbf{tsizeroot}\}$ ; $st := push(root, 0)$; $\mathbf{slen} := \mathbf{1}$; $\mathbf{ssize} := \mathbf{tsizeroot}$
$tail := push(st \to tree \to right, tail)$; $tail := push(st \to tree \to left, tail)$; free$(st)$; $st := tail$; $\mathbf{slen}{+}{+}$; $\mathbf{tsizel} := ?$; $\mathbf{tsizer} := ?$; $\{0 \leq \mathbf{tsizel} \wedge 0 \leq \mathbf{tsizer}\}$ ; $\{\mathbf{tsize} = \mathbf{tsizel} + \mathbf{tsizer} + \mathbf{1}\}$ ; $\mathbf{ssize} := \mathbf{tsizel} + \mathbf{tsizer} + \mathbf{ssizetail}$;
$\{st \to tree \neq 0\}$ $\{\mathbf{tsize} \neq \mathbf{0}\}$
$\{st \neq 0 \wedge \mathbf{slen} > \mathbf{0}\}$ ; $tail := st \to next$; $\mathbf{tsize} := ?$; $\mathbf{ssizetail} := ?$; $\{0 \leq \mathbf{tsize} \wedge 0 \leq \mathbf{ssizetail}\}$ ; $\{\mathbf{ssize} = \mathbf{tsize} + \mathbf{ssizetail}\}$
$\{st \to tree = 0\}$ $\{\mathbf{tsize} = \mathbf{0}\}$
$\{st = 0 \wedge \mathbf{slen} \leq \mathbf{0}\}$
free$(st)$; $st := tail$; $\mathbf{slen}{-}{-}$

**Fig. 5.** Numeric abstraction — Product Program

The notion of left product guarantees that some converse of Lemma 1 holds. Informally, a program $P$ is a left product of $P_1$ and $P_2$ if $P$ can progress at any program point where $P_1$ can progress. More precisely, we informally want that for every node $(l_1, l_2)$ such that $P_1$ can progress from $l_1$ to $l_1'$ and $P_2$ is not stuck, there exists a left or synchronous edge from $(l_1, l_2)$ to $(l_1', l_2')$. Since it would be clearly too strong to require this progress property for arbitrary states, the definition is parametrized by a precondition.

**Definition 4 (Left product).** *$P \in P_1 \times P_2$ is a left product w.r.t. a precondition $\varphi$, written $P \in P_1 \ltimes_\varphi P_2$, iff for every trace $t :: \langle (l_1, l_2), \sigma_1 \uplus \sigma_2 \rangle \in \mathsf{Tr}(P)$ with initial state $\sigma$ such that $\varphi(\sigma)$, and for every nodes $l_1' \in \mathcal{N}_1$ and $l_2' \in \mathcal{N}_2$ such that $\sigma_1 \in$ $\mathsf{dom}([\![G_1\langle l_1, l_1'\rangle]\!])$ and $\sigma_2 \in \mathsf{dom}([\![G_2\langle l_2, l_2'\rangle]\!])$, one of the following holds:*

1. *$(l_1, l_2) \overset{\mathsf{l}}{\mapsto} (l_1', l_2)$ or $(l_1, l_2) \overset{\mathsf{r}}{\mapsto} (l_1, l_2')$ belongs to $P$;*
2. *there exists an edge $(l_1, l_2) \Mapsto (l_1', l_2'')$ in $P$ s.t. $\sigma_2 \in \mathsf{dom}([\![G_2\langle l_2, l_2''\rangle]\!])$;*

*Example 3.* One can verify that the product examples shown in Section 2 are left products. For the product in Fig. 4, one can deduce at node $l_2$ the validity of the invariant $curr = \mathsf{null} \Leftrightarrow \overline{curr}$. In order to verify the leftness condition at node $l_2$ one must check that every feasible transition on the left program is eventually feasible in the product program. In this particular case, the equivalent of the boolean guards holds by the invariant above.

**Lemma 2 (Lifting left products).** *Assume $P \in P_1 \ltimes_\varphi P_2$ with $P_2 \Downarrow^\star$. Let $t_1 \in \mathsf{Tr}(P_1)$ with initial state $\sigma_1$, and let $\sigma_2 \in \mathcal{S}$ s.t. $\varphi\,(\sigma_1 \uplus \sigma_2)$. Then there exists a trace $t \in \mathsf{Tr}(P)$ with initial state $\sigma_1 \uplus \sigma_2$ s.t. $\pi_1(t) = t_1$.*

The result above requires in general proving strong-termination of the right component $P_2$. However, it is often sufficient to perform a syntactic check over a program product $P \in P_1 \ltimes_\varphi P_2$ as suggested by the following result.

**Lemma 3.** *Assume $P \in P_1 \ltimes_\varphi P_2$ has no asynchronous right loops, i.e., that for all sequences of edges $l_1 \overset{\mathsf{r}}{\mapsto} l_2, \ldots, l_{n-1} \overset{\mathsf{r}}{\mapsto} l_n$ we have $l_1 \neq l_n$. Then $P_1 \Downarrow^\star$ implies $P_2 \Downarrow^\star$.*

It follows from the lemma above, and the fact that we are interesting in terminating executions of $P_1$, that it is enough to check for the absence of asynchronous right loops in the product $P$.

## 4 Logical Validation

We now show how to check the correctness of product constructions and relational specifications using standard logical frameworks. Assuming that $P_1$ and $P_2$ are separable, we cast the correctness of two programs $P_1$ and $P_2$ w.r.t. a relational specification $\Phi$, in terms of the functional correctness of a left product $P \in P_1 \ltimes_{\Phi(\mathsf{in})} P_2$. If the statement languages of $P_1$ and $P_2$ are amenable to verification condition generation, one can generate from a product program $P$ a set of verification conditions that ensure that $P$ is a left product of $P_1$ and $P_2$, and that $P_1$ and $P_2$ are correct w.r.t. a relational specification $\Phi$. For clarity, we instantiate this section to the programming model used for the examples in Section 2, and a weakest precondition calculus over first-order formulae.

Program correctness is usually expressed by a judgment of the form $\{\varphi\}\, P\, \{\psi\}$, where $P = \langle \mathcal{N}, \mathcal{E}, G \rangle$ is a program, and $\varphi, \psi$ are assertions. A judgment is valid, written $\models \{\varphi\}\, P\, \{\psi\}$, iff for all states $\sigma, \sigma' \in \mathcal{S}$ s.t. $(\sigma, \sigma') \in [\![P]\!]$, if $\varphi\,\sigma$ then $\psi\,\sigma'$. One can prove the validity of triples using a variant of Hoare logic [2], or working with a compositional flow logic [17]. However, the prominent means to prove that $\{\varphi\}\, P\, \{\psi\}$ is valid is to exhibit a partial specification $\Phi : \mathcal{N} \rightharpoonup \phi$ s.t. all cycles in the graph of $P$ go through an annotated node, i.e. a node in $\mathsf{dom}(\Phi)$; and $\mathsf{in}, \mathsf{out} \in \mathsf{dom}(\Phi)$ with $\varphi = \Phi(\mathsf{in})$ and $\psi = \Phi(\mathsf{out})$.

We adopt a simplified version of the memory model of Leroy and Blazy [13]—locations are interpreted as integer values and field accesses as pointer offsets. We introduce to the assertion language a variable $\mathsf{h}$, and the non-interpreted functions $\mathsf{load}$, $\mathsf{store}$, $\mathsf{alloc}$, and $\mathsf{free}$, and the predicate $\mathsf{Valid}$. We also introduce a suitable set of axioms, including for instance:

$$\begin{aligned}
\mathsf{Valid}(\mathsf{h},l) &\implies \mathsf{load}(\mathsf{store}(\mathsf{h},l,v),l) = v\\
\mathsf{Valid}(\mathsf{h},l) \land \mathsf{Valid}(\mathsf{h},l') \land l \neq l' &\implies \mathsf{load}(\mathsf{store}(\mathsf{h},l',v),l) = \mathsf{load}(\mathsf{h},l)\\
\mathsf{alloc}(\mathsf{h}) = (\mathsf{h}',l) &\implies \mathsf{Valid}(\mathsf{h}',l)\\
\mathsf{Valid}(\mathsf{h},l) \land l \neq l' \land \mathsf{free}(\mathsf{h},l) = \mathsf{h}' &\implies \mathsf{Valid}(\mathsf{h}',l')
\end{aligned}$$

The weakest precondition calculus is standard, with the exception perhaps of heap operations:

$$\begin{aligned}
\mathsf{wp}(x := [l], \phi) &\doteq \phi[\mathsf{load}(\mathsf{h},l)/x] & \mathsf{wp}([l] := x, \phi) &\doteq \phi[\mathsf{store}(\mathsf{h},l,x)/\mathsf{h}]\\
\mathsf{wp}(\mathsf{free}(l), \phi) &\doteq \phi[\mathsf{free}(\mathsf{h},l)/\mathsf{h}] & \mathsf{wp}(l := \mathsf{alloc}, \phi) &\doteq \phi[\mathsf{h}^\star/\mathsf{h}] \land (\mathsf{h}^\star, l) = \mathsf{alloc}(\mathsf{h})
\end{aligned}$$

where $\mathsf{h}^\star$ stands for a fresh variable. One can use the weakest preconditions to generate a specification $\Phi^\natural$ that extends $\Phi$ to all nodes. Using the well-founded induction principle attached to partial specifications, see e.g. [7], we set

$$\Phi^\natural(l) \doteq \bigwedge_{\langle l,l'\rangle \in \mathcal{E}} \mathsf{wp}(G\langle l,l'\rangle, \Phi^\natural(l')) \qquad \text{for all } l \notin \mathsf{dom}(\Phi)$$

The logical judgement $\vdash \{\varphi\}\, P\, \{\psi\}$ is verifiable if there is a specification $\Phi^\natural$ with $\varphi \doteq \gamma(\Gamma(\mathsf{in}))$ and $\psi \doteq \gamma(\Gamma(\mathsf{out}))$ such that the verification conditions $\Phi(l) \Rightarrow \mathsf{wp}(G\langle l,l'\rangle, \Phi^\natural(l'))$ are valid for all $\langle l,l'\rangle \in \mathcal{E}$ and $l \in \mathsf{dom}(\Phi)$.

The leftness of a product can also be checked by logical means. We use a simple form of path condition, which we call edge condition, to express leftness. Formally, the edge condition $\mathsf{ec}(c)$ for a statement $c$ is, if it exists, the unique (up to logical equivalence) formula $\phi$ s.t. for all states $\sigma \in \mathcal{S}$, $\sigma \in [\![\phi]\!]$ iff $\sigma \in \mathsf{dom}([\![c]\!])$. We define for every node $(l_1,l_2) \in \mathcal{N}$ and edges $\langle l_1,l_1'\rangle \in \mathcal{E}_1$ and $\langle l_2,l_2'\rangle \in \mathcal{E}_2$ s.t. $(l_1,l_2) \overset{\rfloor}{\not\mapsto} (l_1',l_2)$ and $(l_1,l_2) \overset{\lceil}{\not\mapsto} (l_1,l_2')$ the $\Phi$-leftness condition as

$$\Phi(l_1,l_2) \land \mathsf{ec}(G_1\langle l_1,l_1'\rangle) \land \mathsf{ec}(G_2\langle l_2,l_2'\rangle) \Rightarrow \bigvee_{l_2'' : (l_1,l_2) \mapsto (l_1',l_2'')} \mathsf{ec}(G_2\langle l_2,l_2''\rangle)$$

and say that $P$ is $\Phi$-left iff all its $\Phi$-leftness conditions are valid.

Weakest preconditions can be used to compute edge conditions. For instance one can define $\mathsf{ec}(c)$ by the clauses:

$$\begin{aligned}
\mathsf{ec}(\mathsf{skip}) &\doteq \mathsf{true} & \mathsf{ec}(x := e) &\doteq \mathsf{true}\\
\mathsf{ec}(\{b\}) &\doteq b & \mathsf{ec}(c_1; c_2) &\doteq \mathsf{ec}(c_1) \land \mathsf{wp}(c_1, \mathsf{ec}(c_2))\\
\mathsf{ec}([l] := x) &\doteq \mathsf{Valid}(\mathsf{h},l) & \mathsf{ec}(x := [l]) &\doteq \mathsf{Valid}(\mathsf{h},l)\\
\mathsf{ec}(\mathsf{free}(l)) &\doteq \mathsf{Valid}(\mathsf{h},l) & \mathsf{ec}(l := \mathsf{alloc}) &\doteq \mathsf{true}
\end{aligned}$$

*Example 4.* In order to verify the leftness of the product program in Figure 4 it is sufficient to check for every synchronous edge $(l_1,l_2) \mapsto (l_1',l_2')$ that $\mathsf{ec}(G_1\langle l_1,l_1'\rangle)$ implies $\mathsf{ec}(G_2\langle l_2,l_2'\rangle)$. Consider for instance the product edge $\langle l_2,l_3\rangle$. The edge condition of the corresponding left edge is $curr \neq \mathsf{null}$ whereas the edge condition of the corresponding right edge is $\neg\overline{curr}$. The validity of $curr \neq \mathsf{null} \Rightarrow \neg\overline{curr}$ follows trivially from the strong invariant $curr = \mathsf{null} \Leftrightarrow \overline{curr}$.

Let $\mathcal{S}_1, \mathcal{S}_2 \subseteq \mathcal{S}$ be sets of pairwise separable[1] states. From the separability hypothesis, one can embed relational assertions on $\mathcal{P}(\mathcal{S}_1 \times \mathcal{S}_2)$ as assertions on the set $\{\sigma_1 \uplus \sigma_2 \mid$

[1] Two states $\sigma_1$ and $\sigma_2$ are separable if $\sigma_1 \uplus \sigma_2$ is defined.

$\sigma_1 \in \mathcal{S}_1, \sigma_2 \in \mathcal{S}_2\}$. Relational program correctness is formalized by refinement quadruples of the form $\models \{\varphi\}P_1 \mapsto P_2\{\psi\}$, where $P_1, P_2$ are programs, and $\varphi, \psi$ are assertions. Such refinement judgment is valid iff for all $t_1 \in \mathsf{Ex}(P_1)$ with initial state $\sigma_1$ and final state $\sigma_1'$, and $\sigma_2$ s.t. $\varphi\,(\sigma_1 \uplus \sigma_2)$ and $P_2 \Downarrow^\star$, there exists $t_2 \in \mathsf{Ex}(P_2)$ with initial state $\sigma_2$ and final state $\sigma_2'$ s.t. $\psi\,(\sigma_1' \uplus \sigma_2')$.

**Theorem 1.** *Let $P_1, P_2$ be separable programs and let $\varphi, \psi$ be assertions. Then the judgement $\models \{\varphi\}P_1 \mapsto P_2\{\psi\}$ holds, provided there is a partial specification $\Phi$ s.t. $\varphi = \Phi(\mathsf{in}_1, \mathsf{in}_2)$ and $\psi = \Phi(\mathsf{out}_1, \mathsf{out}_2)$, and a product program $P \in P_1 \times P_2$ that is $\Phi$-left and correct w.r.t. $\Phi$.*

Theorem 1 provides direct proofs of correctness for many common refinement steps, e.g. replacing a non-deterministic assignment by an assignment (or a non-deterministic choice by one of its substatements). Observe that by Lemma 3 it is enough to check alternatively for the absence of right loops in the product program instead of requiring the strong-termination of $P_2$.

## 4.1 Completeness of Abstraction Validation

We briefly show that abstraction validation is relatively complete under a soundness assumption of the program abstraction procedure. To this end, we use the framework of abstract interpretation [11] to characterize sound program abstractions. Then we show that the correctness of the abstract semantics w.r.t. the verification calculus implies the verifiability of the resulting program abstraction using left products. For brevity, we only consider forward abstract semantics; the adaptation to backward semantics is straightforward.

In the rest of this section we let $I = \langle A, [\![.]\!]^\sharp\rangle$ be an abstract semantics composed of

- an abstract domain $A$, that can be interpreted as assertions over states;
- an abstract interpretation function $[\![.]\!]^\sharp : \mathsf{Stmt} \to A \to A$ for statements: $[\![c]\!]^\sharp$ approximates the execution of statement $c$ in the abstract domain;

We assume the existence of a concretization function $\gamma$ from abstract values in $A$ to first order formulae. We need to assume also the soundness of the abstract semantics $I = \langle A, [\![.]\!]^\sharp\rangle$ w.r.t. the $\mathsf{wp}$ calculus, i.e. that for all $c \in \mathsf{Stmt}$ and $a \in A$, $\Phi \doteq \gamma(a) \Rightarrow \mathsf{wp}(c, \gamma([\![c]\!]^\sharp a))$ is a verifiable formula. We also assume a standard characterization of valid post-fixpoints: a labeling $\Gamma : \mathcal{N} \to A$ is a post-fixpoint of the abstract semantics $I$ if for all $\langle l, l'\rangle \in \mathcal{E}$ $[\![G\langle l, l'\rangle]\!]\Gamma(l) \sqsubseteq \Gamma(l')$.

Let $P = \langle \mathcal{N}, \mathcal{E}, G\rangle$ and $\hat{P} = \langle \mathcal{N}, \mathcal{E}, \hat{G}\rangle$ be separable programs, and assume that the abstract domain $A$ represents relations between the disjoint memories of $P$ and $\hat{P}$. We say that a $\hat{P}$ is a sound abstraction of $P$ w.r.t. a labeling $\Gamma : \mathcal{N} \to A$ if for all $e = \langle l, l'\rangle \in \mathcal{E}$ we have

$$[\![G\,e; \hat{G}\,e]\!]^\sharp \Gamma(l) \sqsubseteq \Gamma(l')$$

**Lemma 4.** *Let $P$ be a program, $I = \langle A, [\![.]\!]^\sharp\rangle$ an abstract semantics, and $P'$ a sound abstraction of $P$ w.r.t. a post-fixpoint $\Gamma : \mathcal{N} \to A$. Assume that $\gamma\, a \Rightarrow \gamma\, a'$ is verifiable* *for all $a, a' \in A$ s.t. $a \sqsubseteq a'$. If $I$ is sound w.r.t the* wp *calculus then there exists $Q \in P \ltimes_\varphi P'$ s.t. $\vdash \{\varphi\}\, Q\, \{\psi\}$ is a verifiable judgement, where $\varphi \doteq \gamma(\Gamma(\mathsf{in}))$ and $\psi \doteq \gamma(\Gamma(\mathsf{out}))$.*

It follows from the lemma above that, under mild conditions, if $P'$ is an abstract program computed from $P$ using a sound abstract semantics, then one can verify that $P$ is correctly abstracted by $P'$. Besides, in settings in which the abstract semantics is defined as a strongest postcondition calculus, as in e.g. predicate abstraction, abstraction validation is decidable. Indeed, it is sufficient that the decision procedure used for program verification is as complete as the one used by the program abstraction algorithm.

## 5 Full Products

We introduce a symmetric variant of the notion of left product of Section 3, which allows verifying one-to-one correspondences between traces of a source and transformed program, as required by translation validation.

**Definition 5 (Full product).** *$P \in P_1 \times P_2$ is a full product w.r.t. a precondition $\varphi$, written $P \in P_1 \bowtie_\varphi P_2$, iff for every trace $t :: \langle (l_1, l_2), \sigma_1 \uplus \sigma_2 \rangle \in \mathsf{Tr}(P)$ with initial state $\sigma$ such that $\varphi(\sigma)$, and for every nodes $l'_1 \in \mathcal{N}_1$ and $l'_2 \in \mathcal{N}_2$ such that $\sigma_1 \in \mathsf{dom}([\![G_1\langle l_1, l'_1\rangle]\!])$ and $\sigma_2 \in \mathsf{dom}([\![G_2\langle l_2, l'_2\rangle]\!])$, one of the edges $(l_1, l_2) \overset{\mathsf{l}}{\mapsto} (l'_1, l_2)$, $(l_1, l_2) \overset{\mathsf{r}}{\mapsto} (l_1, l'_2)$, or $(l_1, l_2) \mapsto (l'_1, l'_2)$ belongs to P;*

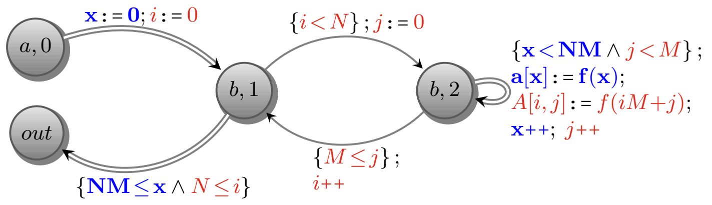
Product fullness is a stronger property than being both left and right. Indeed, requiring the existence of the edge $(l_1, l_2) \mapsto (l'_1, l'_2)$ is stronger that requiring the existence of $(l_1, l_2) \mapsto (l'_1, l''_2)$ or $(l_1, l_2) \mapsto (l''_1, l'_2)$ for some $l''_1$ or $l''_2$. In a deterministic setting, however, a product program $P$ is full iff $P$ is left and right. Moreover, for deterministic programs, left products and full products coincide: assume that $P_2$ is deterministic, i.e. if $\sigma \in \mathsf{dom}([\![G\langle l, l'\rangle]\!])$ and $\sigma \in \mathsf{dom}([\![G\langle l, l''\rangle]\!])$ then $l' = l''$. Then $P \in P_1 \ltimes_\varphi P_2$ iff $P \in P_1 \bowtie_\varphi P_2$. This has practical advantages when verifying deterministic programs, since it is sufficient to discharge verification conditions for leftness to formally verify the fullness of a program product.

Relational correctness is formalized by judgments of the form $\{\varphi\}\, P_1 \sim P_2\, \{\psi\}$, where $P_1, P_2$ are separable programs, and $\varphi, \psi$ are assertions. A relational judgment is valid, written $\models \{\varphi\}\, P_1 \sim P_2\, \{\psi\}$, iff for all $t_1 \in \mathsf{Ex}(P_1)$ and $t_2 \in \mathsf{Ex}(P_2)$ with initial states $\sigma_1$ and $\sigma_2$, and final states $\sigma'_1$ and $\sigma'_2$, $\varphi\ (\sigma_1 \uplus \sigma_2)$ imply $\psi\ (\sigma'_1 \uplus \sigma'_2)$. Full products yield a symmetric variant of Theorem 1.

**Theorem 2.** *Let $P_1, P_2$ be deterministic separable programs and let $\varphi, \psi$ be assertions. Then $\models \{\varphi\}\, P_1 \sim P_2\, \{\psi\}$, provided there is a partial specification $\Phi$ and a product program $P \in P_1 \times P_2$ s.t. $\varphi = \Phi(\mathsf{in}_1, \mathsf{in}_2)$, $\psi = \Phi(\mathsf{out}_1, \mathsf{out}_2)$, and $P$ is $\Phi$-left and correct w.r.t. $\Phi$.*

*Example 5.* Figure 6 shows a full product for the validation of the loop tiling example in Fig. 3. From Theorem 2, one can show that the product is full by proving that is $\Phi$-left, where $\Phi$ is shown in the figure. E.g. $\Phi$-leftness at the node $(b, 2)$ and for the edges $\langle b, b\rangle$ and $\langle 2, 2\rangle$ reduces to showing that $\Phi(b, 2) \wedge \mathsf{ec}(2, 2) \wedge \mathsf{ec}(b, b)$ implies $\mathsf{ec}((b, 2) \mapsto (b, 2))$.

**Product**



$a,0$
$\mathbf{x} := \mathbf{0}; i := 0$
$b,1$
$\{i < N\}; j := 0$
$b,2$
$\{\mathbf{x} < \mathbf{NM} \wedge j < M\};$
$\mathbf{a}[\mathbf{x}] := \mathbf{f}(\mathbf{x});$
$A[i,j] := f(iM+j);$
$\mathbf{x}{+}{+};\ j{+}{+}$
$\{M \leq j\};$
$i{+}{+}$
$out$
$\{\mathbf{NM} \leq \mathbf{x} \wedge N \leq i\}$

**Specification**

$\Phi(a,0) \doteq$ true

$\Phi(b,2) \doteq x = iM+j \wedge i < N \wedge j \leq M \wedge \varphi(i) \wedge \forall r.\ 0 \leq r < j \Rightarrow A[i,r] = a[iM+r]$

$\Phi(out) \doteq \varphi(N)$

where $\varphi(i) \doteq \forall l, r.\ 0 \leq l < i \wedge 0 \leq r < M \Rightarrow A[l,r] = a[lM+r]$

**Fig. 6.** Loop tiling example — Product program

## 6 Implementation

We have implemented a proof of concept verification plugin in the Frama-C environment. We have used our this plugin to validate abstraction examples for list traversing algorithms.

The plugin receives as input a file with a program, its abstraction, and a predicate that describes the relation between the abstract and concrete states, using the ANSI C Specification Language (ACSL). A product of the supplied programs is constructed by following the program graphs and deciding at each branch statement whether to introduce a right, left or synchronized edge, and generating additional program annotations. Non-deterministic assignments are modeled in abstract programs with the use of undefined functions, and assert statements were added to introduce hypotheses regarding the non-deterministic output values. In order to deal with the weakness of the alias analysis, we added some memory disjointness annotations manually.

The final annotated product program is fed into the Frama-C Jessie plugin, which translates the C product program into Why's intermediate language and discharges the verification conditions using the available SMT solvers (AltErgo, Simplify, Z3, etc.). Figure 7 depicts the interaction of the plugin with other components of the framework.

## 7 Related Work

Our technique builds upon earlier work on relational verification using product programs [4,20], and is closely related to relational logics [8,18] used to reason about compiler correctness and program equivalence. Furthermore, there exist strong connections between abstraction validation and refinement proofs—refinement can be viewed as a form of contextual approximation. In particular, developing connections between our method and proof methods for program refinement, such as the refinement calculus [15], or refinement with angelic non-determinism [10] is left for future work.

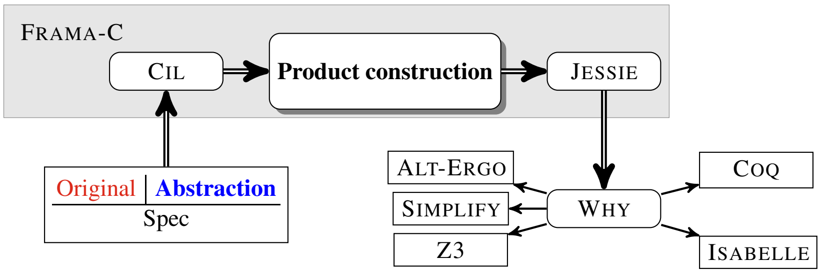


**Fig. 7.** Tool architecture

Abstraction validation may be seen as an instance of result checking, i.e. of the a posteriori validation of a computed result, in the context of program analysis and program transformations algorithms. In this sense, it is closely related to translation validation [21] and abstraction checking for embedded systems [9].

## 8 Conclusion

Asymmetric products provide a convenient means to validate relational properties using standard verification technology. They provide an automated method for reducing a refinement task to a functional verification task, and allow the validation of a broad set of program optimizations. Their applicability has been illustrated with the implementation a product construction prototype. In the future, we intend to used asymmetric products for performing a certified complexity analysis of cryptographic games [6]. Another target for future work is to broaden the scope of relational validation to object-oriented and concurrent programs.

**Acknowledgments.** 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 the Spanish Juan de la Cierva programme (JCI-2010-08550). Juan Manuel Crespo is funded by FPI Spanish programme (BES-2010-031271).

## References

1. Ball, T., Majumdar, R., Millstein, T.D., Rajamani, S.K.: Automatic predicate abstraction of C programs. In: Programming Languages Design and Implementation, pp. 203–213 (2001)
2. Bannwart, F.Y., Müller, P.: A program logic for bytecode. Electronic Notes in Theoretical Computer Science 141, 255–273 (2005)
3. 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)
4. Barthe, G., Crespo, J.M., Kunz, C.: Relational Verification Using Product Programs. In: Butler, M., Schulte, W. (eds.) FM 2011. LNCS, vol. 6664, pp. 200–214. Springer, Heidelberg (2011)

5. 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 (2004)
6. Barthe, G., Grégoire, B., Heraud, S., Béguelin, S.Z.: Computer-Aided Security Proofs for the Working Cryptographer. In: Rogaway, P. (ed.) CRYPTO 2011. LNCS, vol. 6841, pp. 71–90. Springer, Heidelberg (2011)
7. Barthe, G., Kunz, C.: Certificate Translation in Abstract Interpretation. In: Drossopoulou, S. (ed.) ESOP 2008. LNCS, vol. 4960, pp. 368–382. Springer, Heidelberg (2008)
8. 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 (2004)
9. Blech, J.O., Schaefer, I., Poetzsch-Heffter, A.: Translation Validation of System Abstractions. In: Sokolsky, O., Taşıran, S. (eds.) RV 2007. LNCS, vol. 4839, pp. 139–150. Springer, Heidelberg (2007)
10. Bodík, R., Chandra, S., Galenson, J., Kimelman, D., Tung, N., Barman, S., Rodarmor, C.: Programming with angelic nondeterminism. In: Principles of Programming Languages, pp. 339–352 (2010)
11. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Principles of Programming Languages, pp. 238–252 (1977)
12. Graf, S., Saïdi, H.: Construction of Abstract State Graphs with PVS. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 72–83. Springer, Heidelberg (1997)
13. Leroy, X., Blazy, S.: Formal verification of a C-like memory model and its uses for verifying program transformations. J. Autom. Reasoning 41(1), 1–31 (2008)
14. Magill, S., Tsai, M.-H., Lee, P., Tsay, Y.-K.: Automatic numeric abstractions for heap-manipulating programs. In: Hermenegildo, M., Palsberg, J. (eds.) Principles of Programming Languages, pp. 211–222. ACM (2010)
15. Morgan, C.: Programming from specifications. Prentice-Hall International Series in Computer Science. Prentice-Hall, Inc. (June 1990)
16. Pnueli, A., Siegel, M., Singerman, E.: Translation Validation. In: Steffen, B. (ed.) TACAS 1998. LNCS, vol. 1384, pp. 151–166. Springer, Heidelberg (1998)
17. Tan, G., Appel, A.W.: A Compositional Logic for Control Flow. In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855, pp. 80–94. Springer, Heidelberg (2006)
18. Yang, H.: Relational separation logic. Theoretical Computer Science 375(1-3), 308–334 (2007)
19. Yang, H., O'Hearn, P.W.: A Semantic Basis for Local Reasoning. In: Nielsen, M., Engberg, U. (eds.) FOSSACS 2002. LNCS, vol. 2303, pp. 402–416. Springer, Heidelberg (2002)
20. 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)
21. Zuck, L.D., Pnueli, A., Goldberg, B.: Voc: A methodology for the translation validation of optimizing compilers. J. UCS 9(3), 223–247 (2003)