Projecting Programs on Specifications: Definition and Implications

Jules Desharnais

Département d'informatique et de génie logiciel, Université Laval, Québec, Canada
Jules.Desharnais@ift.ulaval.ca

Nafi Diallo

New Jersey Institute of Technology, Newark, NJ
ncd8@njit.edu

Wided Ghardallou

Faculté des Sciences de Tunis, El Manar, Tunisia
wided.ghardallou@gmail.com

Ali Mili

New Jersey Institute of Technology, Newark, NJ
mili@cis.njit.edu

Abstract

Given a specification R, it is common for a candidate program P to be doing more than R requires; this is not necessarily bad, and is often unavoidable, due to programming language constraints or to otherwise sensible design decisions. In this paper, we introduce a relational operator that captures, for a given specification R and candidate program P, the functionality delivered by P that is relevant to R. This operator, which we call the *projection* of P over R (for reasons we explain), has a number of interesting properties, which we explore in this paper.

Keywords: absolute correctness, relative correctness, refinement, projection, lattices.

1. Projecting Programs on Specifications

Given a specification R and a program P that is written to satisfy R, we refer to P as a *candidate* program for specification R, and we refer to R as the *target* specification of program P, regardless of whether P does or does not satisfy specification R. Given a specification R and a candidate program P, the program P could well be falling short of some of the requirements of R, while at the same time exceeding (i.e. doing more than needed) on some other

requirements. There is no shortage of reasons why a program may fall short of the requirements mandated by the specification, but there are also ample reasons why a program may do more than required: these include cases where excess functionality is a byproduct of normal design decisions, cases where it stems from programming language constructs, and more generally the need to bridge the gap between a potentially non-deterministic specification and a deterministic program. Consider for example the following relational specification on space S defined by integer variables x and y:

$$R = \{(\langle x, y\rangle, \langle x', y'\rangle)|x' = x + y\},$$

and consider the following program on the same space:

```
while (y!=0) {x:=x+1; y:=y-1}.
```

For non-negative values of y this program computes the sum of x and y in x while placing 0 in y; for negative values of y, it fails to terminate. Hence its function can be written as:

$$P = \{(\langle x, y\rangle, \langle x', y'\rangle)|y \geq 0 \wedge x' = x + y \wedge y' = 0\}.$$

This program does not do everything that specification R requires, since it fails to compute the sum of x and y into x for negative values of y; on the other hand, it puts 0 in y even though the specification does not ask for it (but this is a side effect of the algorithm we have chosen to satisfy the specification). Hence looking at specification R and program P, we would like to think that the functionality of P that is relevant to (mandated by) R is captured by the following relation:

$$\pi = \{(\langle x, y\rangle, \langle x', y'\rangle)|y \geq 0 \wedge x' = x + y\}.$$

Whatever else P does (e.g. it sets y to 0) is not relevant to specification R; on the other hand, whatever else the specification mandates (computing the sum of x and y into x for negative values of y), program P is not delivering. In other words, relation π fails to capture the condition $y' = 0$ because that is not mandated by the specification, and it fails to capture the condition the case $y < 0$ because that is not delivered by the candidate program P.

In this paper, we introduce a relational operator that captures, for a given specification R and program P, the functionality delivered by P that is relevant to (i.e. mandated by) R; we refer to this operator as the *projection of P over R*, for reasons that we elucidate in our subsequent discussions. We define this operator in section 4, and we discuss its properties and implications in section 5. The concept of projection of a program on a specification comes about as a byproduct of the definition of relative correctness, i.e. the property of a program to be more-correct than another with respect to a specification. Relative correctness, due to [3], is discussed in section 3. In section 2 we introduce the mathematical background that we use throughout this paper. In section 6 we briefly discuss related work; because no other authors we know of have written about projections, the closest work to ours pertains to relative correctness. Finally, in section 7 we summarize and assess our work, and we discuss possible future directions of research.

2. Mathematical Background

2.1. Relational Notations

In this section, we introduce some elements of relational mathematics that we use in the remainder of the paper to carry out our discussions. We assume the reader familiar with relational algebra, and we generally adhere to the definitions of [2, 19]. Dealing with programs, we represent sets using a programming-like notation, by introducing variable names and associated data types (sets of values). For example, if we represent set S by the variable declarations

$x : X;\ y : Y;\ z : Z,$

then S is the Cartesian product $X \times Y \times Z$. Elements of S are denoted in lower case s, and are triplets of elements of X, Y, and Z. Given an element s of S, we represent its X-component by $x(s)$, its Y-component by $y(s)$, and its Z-component by $z(s)$. When no risk of ambiguity exists, we may write x to represent $x(s)$, and x' to represent $x(s')$.

A relation on S is a subset of the Cartesian product $S \times S$; given a pair (s, s') in R, we say that s' is an *image* of s by R. Special relations on S include the *universal* relation $L = S \times S$, the *identity* relation $I = \{(s, s')|s' = s\}$, and the *empty* relation $\phi = \{\}$. Operations on relations (say, R and R') include the set-theoretic operations of *union* ($R \cup R'$), *intersection* ($R \cap R'$), *difference* ($R \setminus R'$) and *complement* ($\overline{R}$). They also include the *relational product*, denoted by $R \circ R'$, or (RR', for short) and defined by:

$$RR' = \{(s, s')|\exists s'' : (s, s'') \in R \wedge (s'', s') \in R'\}.$$

The *power* of relation R is denoted by R^n, for a natural number n, and defined by $R^0 = I$, and for $n > 0$, $R^n = R \circ R^{n-1}$. The *reflexive transitive closure* of relation R is denoted by R^* and defined by $R^* = \{(s, s')|\exists n \geq 0 : (s, s') \in R^n\}$. The *converse* of relation R is the relation denoted by $\widehat{R}$ and defined by

$$\widehat{R} = \{(s, s')|(s', s) \in R\}.$$

Finally, the *domain* of a relation R is defined as the set $dom(R) = \{s|\exists s' : (s, s') \in R\}$, and the *range* of relation R is defined as the domain of $\widehat{R}$. Operator precedence is adopted as follows: unary operators apply first, followed by product, then intersection, then union.

A relation R is said to be *reflexive* if and only if $I \subseteq R$, *symmetric* if and only if $R = \widehat{R}$, *antisymmetric* if and only if $R \cap \widehat{R} \subseteq I$, *asymmetric* if and only if $R \cap \widehat{R} = \phi$, and *transitive* if and only if $RR \subseteq R$. A relation is said to be a *partial ordering* if and only if it is reflexive, antisymmetric, and transitive. Also, a relation R is said to be *total* if and only if $I \subseteq R\widehat{R}$, and a relation R is said to be *deterministic* (or: a *function*) if and only if $\widehat{R}R \subseteq I$. In this paper we use a property to the effect that two functions f and f' are identical if and only if $f \subseteq f'$ and $f'L \subseteq fL$. A relation R is said to be a *vector* if and only if $RL = R$; a vector on space S is a relation of the form $R = A \times S$, for some subset A of S; we use vectors to represent subsets of S; in particular, we use the product RL as a relational representation of the domain of R.

The following laws of relation algebra are used in the sequel. The first two are laws of Boolean algebra; other laws of Boolean algebra will be invoked by the simple mention "Boolean algebra".

$$Q \subseteq R \Leftrightarrow Q \cap R = Q \Leftrightarrow Q \cup R = R \tag{1}$$

$$P \cap Q \subseteq P \cap R \Leftrightarrow P \cap Q \subseteq R \tag{2}$$

$$R\phi = \phi R = \phi \tag{3}$$

$$RI = IR = R \tag{4}$$

$$LL = L \tag{5}$$

$$RL \cap R = R \tag{6}$$

$$QL \subseteq RL \Leftrightarrow Q \subseteq RL \tag{7}$$

$$\widehat{L} = L \tag{8}$$

$$\widehat{Q \cap R} = \widehat{Q} \cap \widehat{R} \tag{9}$$

$$(P \cup Q)R = PR \cup QR \qquad P(Q \cup R) = PQ \cup PR \tag{10}$$

$$(PL \cap Q)R = PL \cap QR \qquad P(LQ \cap R) = LQ \cap PR \tag{11}$$

$$PQ \cap R \subseteq P(Q \cap \widehat{P}R) \qquad PQ \cap R \subseteq (P \cap R\widehat{Q})Q \tag{12}$$

In addition, the following laws will be referred to as "monotonicity of product":

$$P \subseteq Q \Rightarrow PR \subseteq QR,$$
$$P \subseteq Q \Rightarrow RP \subseteq RQ.$$

2.2. A Refinement Calculus

Throughout this paper, we interpret relations as program specifications or as programs. A key concept in any study of program correctness is the refinement ordering; the following definition gives our version of this ordering.

Definition 1. *Given two relations R and R', we say that R* is refined by *R' (abbrev: $R \sqsubseteq R'$ or $R' \sqsupseteq R$) if and only if $RL \cap R'L \cap (R \cup R') = R$.*

The following proposition provides an alternative characterization of refinement, which we may use in our proofs throughout this paper, as well as an intuitive interpretation of the refinement ordering.

Proposition 1. *Given two relations R and R', R' refines R if and only if $RL \subseteq R'L \wedge RL \cap R' \subseteq R$.*

Proof. *Proof of Sufficiency.* Let R and R' satisfy the conditions $RL \subseteq R'L$ and $RL \cap R' \subseteq R$. We compute:

$RL \cap R'L \cap (R \cup R')$

$=$ {hypothesis $RL \subseteq R'L$}

$RL \cap (R \cup R')$

$=$ {distribution of $\cap$ over $\cup$}

$RL \cap R \cup RL \cap R'$

$=$ $\{(6)$ and hypothesis $RL \cap R' \subseteq R\}$

$R.$

Hence R' refines R.

Proof of Necessity. Let R' refine R; we must prove $RL \subseteq R'L$ and $RL \cap R' \subseteq R$. For the first clause, we proceed as follows:

RL

$=$ {hypothesis and Definition 1}

$(RL \cap R'L \cap (R \cup R'))L$

$=$ $\{(10)\}$

$RL \cap R'L \cap (RL \cup R'L)$

$=$ $\{RL \cap R'L \subseteq RL \cup R'L\}$

$RL \cap R'L.$

Hence (by set theory) $RL \subseteq R'L$. For the second clause, we proceed as follows:

$RL \cap R'$

$\subseteq$ {monotonicity of product}

$RL \cap (R \cup R')$

$=$ {since $RL \subseteq R'L$}

$RL \cap R'L \cap (R \cup R')$

$=$ {hypothesis and Definition 1}

$R.$ **qed**

According to proposition 1, R' refines R if and only if it has a larger domain and assigns fewer images to elements in the domain of R.

We find in [1] that the refinement relation $\sqsupseteq$ is reflexive, antisymmetric and transitive (and thus is indeed an ordering), and that it has the following lattice-like properties:

- Any two relations R and R' have a greatest lower bound, which we denote by $R \sqcap R'$ and to which we refer as the *meet* of R and R'. Also, we find: $R \sqcap R' = RL \cap R'L \cap (R \cup R')$.

- Two relations R and R' admit a least upper bound if and only if they satisfy the condition $RL \cap R'L = (R \cap R')L$, which we call the *consistency condition*; we may also say that they are *consistent*.

- Any two relations that are consistent admit a least upper bound, which we denote by $R \sqcup R'$ and to which we refer as the *join* of R and R'. Also, we find:

$$R \sqcup R' = (\overline{RL} \cap R') \cup (\overline{R'L} \cap R) \cup (R \cap R').$$

  The join of two specifications (or programs) R and R' is very important, because it captures the specification that represents all the functional attributes of R, all the functional attributes of R', and nothing else. It is possible to capture all the functional attributes of R and R' only if R and R' do not contradict each other: this is what the consistency condition represents.

- Two relations R and R' have a least upper bound if and only if they have an upper bound. Consequently, if R and R' are consistent and R refines Q and R' refines Q' then a fortiori Q and Q' are consistent. Conversely, as we refine R and R', we make it less and less likely that they are consistent.
- The empty relation is the universal lower bound of this ordering.
- This ordering admits no universal upper bound. Total deterministic relations are the maximal elements of this ordering.

The way we model specifications, programs, refinement and correctness differ from traditional models, such as Hoare's *Unified Theory of Programming* [9], Hehner's *Practical Theory of Programming* [7], Morgan's *Programming from Specifications* [17], and others. The simplest and most concise, though not necessarily the most precise, way to characterize our programming model by contrast with traditional models is to consider how a (pre-condition/post-condition) pair is mapped into a relational specification. We consider a precondition under the forms of a unary predicate $P(s)$ on S and a post-condition under the form of a binary predicate $Q(s, s')$ on S. Whereas traditional models interpret this $(P(s), Q(s, s'))$ pair as the relation

$$\{(s, s')|P(s) \Rightarrow Q(s, s')\},$$

we interpret it instead as the relation

$$R = \{(s, s')|P(s) \wedge Q(s, s')\}.$$

Consequently, in our model $P(s)$ is actually redundant: if we redefine the post-condition as $Q'(s, s') \equiv P(s) \wedge Q(s, s')$ then the precondition is merely the characteristic predicate of the domain of R.

This distinction is important because, not surprisingly, it has an impact on the way that refinement is defined. Any definition of refinement must acknowledge the contravariant role of the pre-condition and the post-condition: Indeed, one makes a specification more-refined by weakening the pre-condition and(or) by strengthening the post-condition. In traditional models, the contravariance is built into the relational form of the specification (by the $\Rightarrow$ connective); hence refinement takes the simple form of logical implication. By contrast, in our model the contravariance is expressed by the definition of refinement: In fact the two terms of the formula $RL \subseteq R'L \wedge RL \cap R' \subseteq R$ express exactly the property that the pre-condition of R' is weaker and the post-condition is stronger.

The absence of a miracle specification stems from the same decision as above: Because we map a pre/post specification $(P(s), Q(s, s'))$ into the relation $\{(s, s')|P(s) \wedge Q(s, s')\}$, we have no way to distinguish between the miracle specification $(\mathbf{true}, \mathbf{false})$ and the trivial/ bottom specification $(\mathbf{false}, \mathbf{true})$. In our model, the bottom specification is the empty relation, but there is no top/ miracle specification, as no relation can be total (i.e. have S for a domain) and yet assign no image to the elements of its domain. Maximal elements in our refinement ordering are total deterministic relations.

3. Absolute Correctness and Relative Correctness

Whereas absolute correctness characterizes the relationship between a specification and a candidate program, relative correctness ranks two candidate programs with respect to a specification; in order to discuss the latter, it helps to review the former, to see how it is defined in our notation.

3.1. Program Functions

Given a program p on space S, we denote by $[p]$ the function that p defines on its space, i.e.

$[p] = \{(s, s')|\text{if program } p \text{ executes on state } s \text{ then it terminates in state } s'\}$.

We represent program spaces by means of C-like variable declarations and we represent programs by means of a few simple C-like programming constructs, which we present below along with their semantic definitions:

- *Abort:* $[\texttt{abort}] = \phi$.
- *Skip:* $[\texttt{skip}] = I$.
- *Assignment:* $[s\texttt{:=}E(s)] = \{(s, s')|s \in \delta(E) \wedge s' = E(s)\}$, where $\delta(E)$ is the set of states for which expression E can be evaluated.
- *Sequence:* $[p_1; p_2] = [p_1] \circ [p_2]$.
- *Conditional:* $[\texttt{if (}t\texttt{) }\{p\}] = T \cap [p] \cup \overline{T} \cap I$, where T is the vector defined as: $T = \{(s, s')|t(s)\}$.
- *Alternation:* $[\texttt{if (}t\texttt{) }\{p\}\texttt{ else }\{q\}] = T \cap [p] \cup \overline{T} \cap [q]$, where T is defined as above.
- *Iteration:* $[\texttt{while (}t\texttt{) }\{b\}] = (T \cap [b])^* \cap \widehat{\overline{T}}$, where T is defined as above.
- *Block:* $[\{x : X; p\}] = \{(s, s')|\exists x, x' \in X : (\langle s, x\rangle, \langle s', x'\rangle) \in [p]\}$.

Rather than use the notation $[p]$ to denote the function of program p, we will usually use upper case P as a shorthand for $[p]$. By abuse of notation, we may, when it causes no confusion, refer to a program and its function by the same name (which we represent by an upper case letter).

3.2. Absolute Correctness

Definition 2. *Let p be a program on space S and let R be a specification on S.*

- *We say that program p is* correct *with respect to R if and only if P refines R.*
- *We say that program p is* partially correct *with respect to specification R if and only if P refines $R \cap PL$.*

This definition is consistent with traditional definitions of partial and total correctness [4, 5, 6, 8, 14]. Whenever we want to contrast correctness with partial correctness, we may refer to it as *total correctness*. The definition of partial correctness in Definition 2 links it to total correctness. The traditional definition of partial correctness is that program p is partially correct with respect to specification R if and only if $RL \cap P \subseteq R$. Proposition 1 then displays the link in a different manner by showing that total correctness is partial correctness plus the constraint that the domain of the program is at least as large as that of the specification.

The following proposition, due to [16], gives a simple characterization of correctness for deterministic programs.

Proposition 2. *Program p is correct with respect to specification R if and only if $(R \cap P)L = RL$.*

Note that because $(R \cap P)L \subseteq RL$ is a tautology (by Boolean algebra and monotonicity of product), the condition above can be written as: $RL \subseteq (R \cap P)L$; by (7), this condition can be written as $R \subseteq (R \cap P)L$. Note also that the condition set forth in this proposition holds only because P is deterministic; in general, $(R' \cap R)L = RL$ is not equivalent to $R' \sqsupseteq R$. From Definition 2 and Proposition 2 we infer that program p is partially correct with respect to specification R if and only if $(R \cap P)L = RL \cap PL$. Interestingly, this condition is equivalent to: P and R are consistent.

3.3. Relative Correctness: Deterministic Programs

Definition 3. *Let R be a specification on space S and let p and p' be two deterministic programs on space S whose functions are respectively P and P'.*

- *We say that program p' is* more-correct *than program p with respect to specification R (denoted by: $P' \sqsupseteq_R P$) if and only if: $(R \cap P')L \supseteq (R \cap P)L$.*
- *Also, we say that program p' is* strictly more-correct *than program p with respect to specification R (denoted by: $P' \sqsupset_R P$) if and only if $(R \cap P')L \supset (R \cap P)L$.*

Interpretation: $(R \cap P)L$ represents (in relational form) the set of initial states on which the behavior of P satisfies specification R. We refer to this set as the *competence domain* of program P with respect to R . Relative correctness of P' over P with respect to specification R simply means that P' has a larger competence domain than P. Whenever we want to contrast correctness (given in Definition 2) with relative correctness, we may refer to it as *absolute correctness*. Note that when we say *more-correct* we really mean *more-correct or as-correct-as*; we use the shorthand, however, for convenience. Note also that in order for program p' to be more-correct than program p, it does not need to duplicate the behavior of p over the competence domain of p; see Figure 1. In the example shown in this figure, we have:

$(R \cap P)L = \{1, 2, 3, 4\} \times S$,

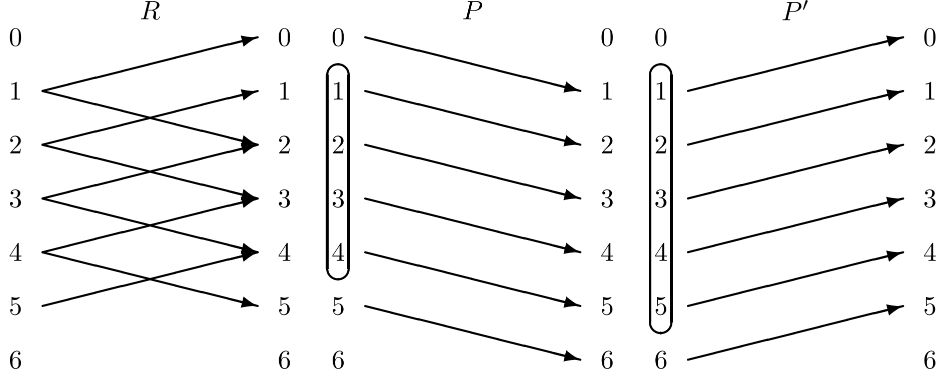

Figure 1: Enhancing correctness without duplicating behavior: $P' \sqsupseteq_R P$

$(R \cap P')L = \{1, 2, 3, 4, 5\} \times S$,
where $S = \{0, 1, 2, 3, 4, 5, 6\}$. Hence p' is more-correct than p with respect to R.

In order to highlight the contrast between relative correctness and absolute correctness, we consider the specification R on the space S of natural numbers

$$R = \{(s, s')|s^2 \leq s' \leq s^3\},$$

and we consider the following programs, where along with each program we indicate its function, then its competence domain with respect to R:

p_0: `abort`. $P_0 = \phi$. $CD_0 = \emptyset$.

p_1: `s:=0`. $P_1 = \{(s, s')|s' = 0\}$. $CD_1 = \{0\}$.

p_2: `s:=1`. $P_2 = \{(s, s')|s' = 1\}$. $CD_2 = \{1\}$.

p_3: `s:=2*s**3-8`. $P_3 = \{(s, s')|s' = 2s^3 - 8\}$. $CD_3 = \{2\}$.

p_4: `skip`. $P_4 = I$. $CD_4 = \{0, 1\}$.

p_5: `s:=2*s**3-3*s**2+2`. $P_5 = \{(s, s')|s' = 2s^3 - 3s^2 + 2\}$. $CD_5 = \{1, 2\}$.

p_6: `s:=s**4-5*s`. $P_6 = \{(s, s')|s' = s^4 - 5s\}$. $CD_6 = \{0, 2\}$.

p_7: `s:=s**2`. $P_7 = \{(s, s')|s' = s^2\}$. $CD_7 = S$.

p_8: `s:=s**3`. $P_8 = \{(s, s')|s' = s^3\}$. $CD_8 = S$.

p_9: `s:=(s**2+s**3)/2`. $P_9 = \{(s, s')|s' = \frac{s^2+s^3}{2}\}$. $CD_9 = S$.

Figure 2 shows how these ten programs are ordered according to their relative correctness with respect to R; in this sample, programs P_7, P_8, P_9 are (absolutely) correct while programs $P_0, P_1, P_2, P_3, P_4, P_5, P_6$ are incorrect because their competence domain is strictly smaller than the domain of R, which is S. Of the latter, only P_0 is partially correct.

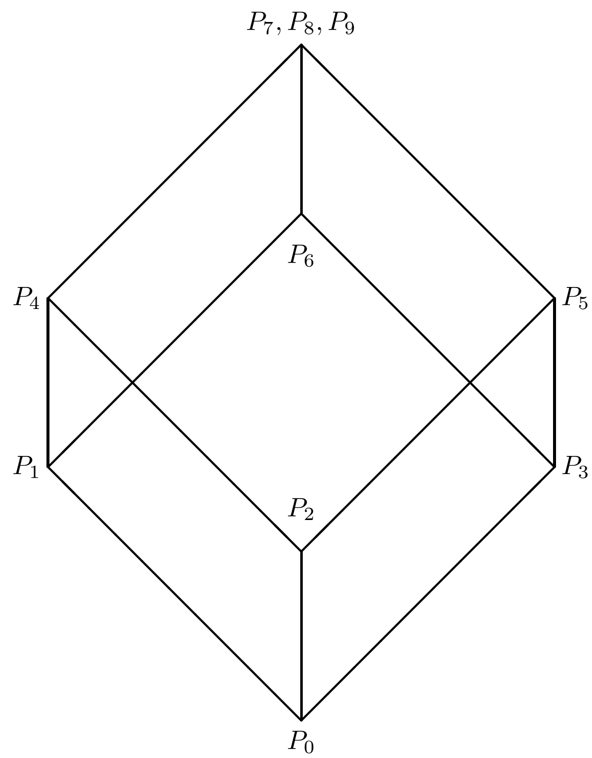

Figure 2: Ordering Candidate Programs by Relative Correctness

3.4. Relative Correctness: Non-Deterministic Programs

In this section we extend the definition of relative correctness to non-deterministic programs. There are several reasons why this is important/useful:

- Non-determinacy is a convenient means to model deterministic programs whose detailed behavior is difficult to capture, unknown, or irrelevant to a particular analysis.
- We may want to reason about the relative correctness of deterministic programs without having to compute their function in all its minute details.
- We may want to apply relative correctness, not only to finished software products, but also to partially defined intermediate artifacts, such as designs.
- Some programs (written in some languages, with specific memory allocation policies) may be observationally non-deterministic due to sensitivity to memory allocation.

We submit the following definition.

Definition 4. *Let R be a specification on set S and let P and P' be (possibly non-deterministic) programs on space S. We say that P' is more-correct than P with respect to R (abbrev: $P' \sqsupseteq_R P$) if and only if*

$$(R \cap P)L \subseteq (R \cap P')L$$

and

$$(R \cap P)L \cap \overline{R} \cap P' \subseteq P.$$

Interpretation: P' is more-correct than P with respect to R if and only if it has a larger competence domain, and for the elements in the competence domain of P, the set of images by P' that violate R is a subset of the set of images by P that violate R. As an illustration, we consider the set $S = \{0, 1, 2, 3, 4, 5, 6, 7\}$ and we let R, P and P' be defined as follows:

$R = \{(0,0), (0,1), (1,0), (1,1), (1,2), (2,1), (2,2), (2,3), (3,2), (3,3), (3,4), (4,3), (4,4), (4,5), (5,4), (5,5)\}$

$P = \{(0,2), (0,3), (1,3), (1,4), (2,0), (2,1), (3,1), (3,2), (4,1), (4,2), (5,2), (5,3)\}$

$P' = \{(0,2), (0,3), (1,2), (1,3), (2,0), (2,3), (3,1), (3,4), (4,2), (4,5), (5,2), (5,3)\}$.

From these definitions, we compute:

$R \cap P = \{(2,1), (3,2)\}$,

$(R \cap P)L = \{2, 3\} \times S$,

$R \cap P' = \{(1,2), (2,3), (3,4), (4,5)\}$,

$(R \cap P')L = \{1, 2, 3, 4\} \times S$,

$(R \cap P)L \cap P' = \{(2,0), (2,3), (3,1), (3,4)\}$,

$(R \cap P)L \cap \overline{R} \cap P' = \{(2,0), (3,1)\}$.

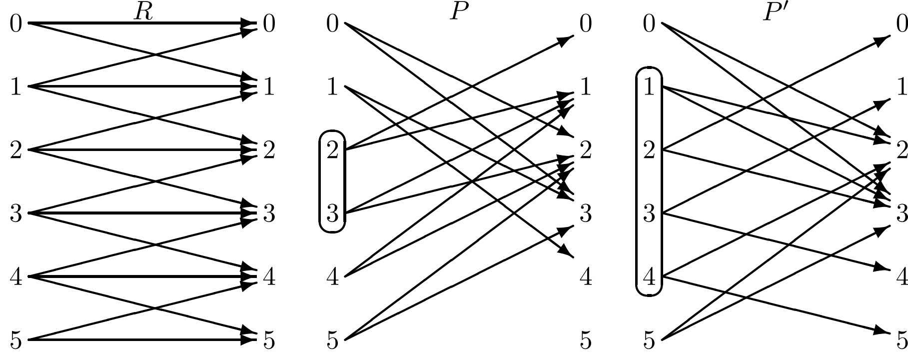

Figure 3: Relative Correctness for Non-Deterministic Programs: $P' \sqsupseteq_R P$.

By inspection, we do find that $(R \cap P)L = \{2,3\} \times S$ is indeed a subset of $(R \cap P')L = \{1,2,3,4\} \times S$. Also, we find that $(R \cap P)L \cap \overline{R} \cap P' = \{(2,0),(3,1)\}$ is a subset of P. Hence the two clauses of Definition 4 are satisfied. Figure 3 represents relations R, P and P' on space S. Program P' is more-correct than program P with respect to R because it has a larger competence domain ($\{2,3\}$ vs. $\{1,2,3,4\}$, highlighted in Figure 3) and because on the competence domain of P ($=\{2,3\}$), program P' generates no incorrect output unless P also generates it ($\{(2,0),(3,1)\}$).

We show in [3] that if P' is deterministic, then the conditions $(R \cap P)L \subseteq (R \cap P')L$ and $P' \sqsupseteq_R P$ are logically equivalent, which means that Definition 4 can be used in general for relative correctness.

3.5. Equal Correctness

Proposition 3. *The relative correctness relation with respect to a given specification is reflexive and transitive.*

Proof. Reflexivity is trivial. To prove transitivity, we consider relations R, P, P' and P'', and we assume that P' is more-correct than P with respect to R, and that P'' is more-correct than P' with respect to R. The condition $(R \cap P)L \subseteq (R \cap P'')L$ stems readily from the hypothesis. We focus on the condition $(R \cap P)L \cap \overline{R} \cap P'' \subseteq P$, which we prove as follows:

$(R \cap P)L \cap \overline{R} \cap P'' \subseteq P$

$\Leftrightarrow$ {hypothesis $P' \sqsupseteq_R P$, hence $(R \cap P)L \subseteq (R \cap P')L$}

$(R \cap P)L \cap (R \cap P')L \cap \overline{R} \cap P'' \subseteq P$

$\Leftrightarrow$ {hypothesis $P'' \sqsupseteq_R P'$, hence $(R \cap P')L \cap \overline{R} \cap P'' \subseteq P'$}

$(R \cap P)L \cap (R \cap P')L \cap \overline{R} \cap P'' \cap P' \subseteq P$

$\Leftarrow$ {Boolean algebra}

$(R \cap P)L \cap \overline{R} \cap P' \subseteq P$,

which holds, by hypothesis. **qed**

Since $\sqsupseteq_R$ is reflexive and transitive, it is a preorder; we use this preorder to define an equivalence relation, as follows:

Definition 5. *Two programs P and P' are said to be* equally correct *with respect to specification R (abbrev: $P \equiv_R P'$) if and only if $P \sqsupseteq_R P'$ and $P' \sqsupseteq_R P$.*

For deterministic relations P and P', equal correctness simply means having the same competence domain; the following proposition characterizes equal correctness for arbitrary (not necessarily deterministic) programs.

Proposition 4. *Let R be a specification on space S, and let P and P' be candidate programs on S. Then*

$$(P \equiv_R P') \Leftrightarrow (R \cap P)L = (R \cap P')L \wedge (R \cap P)L \cap \overline{R} \cap P = (R \cap P')L \cap \overline{R} \cap P'.$$

Proof. From $P \equiv_R P'$ we infer readily that P and P' have the same competence domain with respect to R. Also, from $(R \cap P)L \cap \overline{R} \cap P' \subseteq P$ and $(R \cap P)L = (R \cap P')L$ we infer:

$$(R \cap P')L \cap \overline{R} \cap P' \subseteq (R \cap P)L \cap \overline{R} \cap P.$$

By interchanging P and P' and combining the two results, we find:

$$(R \cap P')L \cap \overline{R} \cap P' = (R \cap P)L \cap \overline{R} \cap P.$$

The converse implication is trivial, if we replace equality by inclusion, and note that an intersection is a subset of its terms. **qed**

4. Projections: Definition and Properties

4.1. Definition

Definition 6. *Given a specification R and a program P, the* projection *of P on R is the relation denoted by $\Pi_R(P)$ and defined by:*

$$\Pi_R(P) = (R \cap P)L \cap (R \cup P).$$

By commutativity of $\cap$ and $\cup$, the formula of $\Pi_R(P)$ is symmetric in R and P, so

$$\Pi_R(P) = \Pi_P(R). \tag{13}$$

Despite this symmetry, we refer to projection in asymmetric terms, since the specification R and the program P play distinct roles. The domain of $\pi_R(P)$ is the competence domain of P with respect to R:

$$\Pi_R(P)L = (R \cap P)L. \tag{14}$$

This is easily obtained by using (11), Boolean algebra and monotonicity of the product:

$$\Pi_R(P)L = ((R \cap P)L \cap (R \cup P))L = (R \cap P)L \cap (R \cup P)L = (R \cap P)L.$$

The first property we explore about projection is idempotence (just like projections in geometry).

Proposition 5. *The projection of a program P on a specification R (viewed as a function of P) is idempotent, i.e.* $\Pi_R(\Pi_R(P)) = \Pi_R(P)$.

Proof.

$\Pi_R(\Pi_R(P))$
$=$ {definition of Π_R}
$(R \cap \Pi_R(P))L \cap (R \cup \Pi_R(P))$
$=$ {definition of Π_R}
$(R \cap (R \cap P)L \cap (R \cup P))L \cap (R \cup (R \cap P)L \cap (R \cup P))$
$=$ {Boolean algebra}
$((R \cap P)L \cap R)L \cap (R \cup (R \cap P)L \cap P)$
$=$ {(11)}
$(R \cap P)L \cap RL \cap (R \cup (R \cap P)L \cap P)$
$=$ {monotonicity of product and Boolean algebra}
$(R \cap P)L \cap (R \cup (R \cap P)L \cap P)$
$=$ {Boolean algebra}
$(R \cap P)L \cap (R \cup P)$
$=$ {definition of Π_R}
$\Pi_R(P)$ **qed**

Proposition 6. *Given a specification R and a program P, specification R and program P both refine the projection of program P over R.*

Proof. According to Proposition 1, we have to prove $\Pi_R(P)L \subseteq RL$ and $\Pi_R(P)L \cap R \subseteq \Pi_R(P)$. The first clause stems readily from (14) by monotonicity of product. The second clause can be proved as follows:

$\Pi_R(P)L \cap R$
$=$ {(14)}
$(R \cap P)L \cap R$
$\subseteq$ {monotonicity of $\cap$}
$(R \cap P)L \cap (R \cup P)$
$=$ {substitution}
$\Pi_R(P)$.

Switching R and P yields $P \sqsupseteq \Pi_P(R)$. By (13), this is equivalent to $P \sqsupseteq \Pi_R(P)$. **qed**

Corollary 1. *Given a specification R and two candidate programs P and P', the projections of P and P' on R are consistent.*

Proof. By Proposition 6, $\Pi_R(P)$ and $\Pi_R(P')$ have an upper bound. By the results provided in section 2.2, $\Pi_R(P)$ and $\Pi_R(P')$ have a least upper bound, which means that they are consistent. **qed**

Whereas P and P' may fail to have a join (due to divergent designs), their projections on R do not (since they only reflect subspecifications of R).

Corollary 2. *Given a specification R and a candidate program P, the projection of P over R is refined by the meet of P and R, i.e. $R \sqcap P \sqsupseteq \Pi_R(P)$. In addition, if P and R are consistent, then $R \sqcap P = \Pi_R(P)$.*

Proof. By Proposition 6, R and P both refine $\Pi_R(P)$. Hence $\Pi_R(P)$ is a lower bound of R and P, whence it is refined by the greatest lower bound of R and P. We write: $R \sqcap P \sqsupseteq \Pi_R(P)$.

If R and P are consistent then $(R \cap P)L = RL \cap PL$; hence $R \sqcap P = RL \cap PL \cap (R \cup P) = (R \cap P)L \cap (R \cup P) = \Pi_R(P)$. **qed**

4.2. Projection and Equal Correctness

The following propositions elucidate the relationship between the projection of a program on a specification and its degree of relative correctness.

Proposition 7. *Let R be a specification on space S and P be a candidate program on S. Then the projection of P over R is in the same equivalence class of $\equiv_R$ as P.*

Proof. We check that P and $\Pi_R(P)$ are equally correct.

$P \equiv_R \Pi_R(P)$

$\Leftrightarrow$ {Proposition 4}

$(R \cap P)L = (R \cap (R \cap P)L \cap (R \cup P))L \wedge$
$(R \cap P)L \cap \overline{R} \cap P = (R \cap (R \cap P)L \cap (R \cup P))L \cap \overline{R} \cap (R \cap P)L \cap (R \cup P)$

$\Leftrightarrow$ {Boolean algebra}

$(R \cap P)L = ((R \cap P)L \cap R)L \wedge$
$(R \cap P)L \cap \overline{R} \cap P = ((R \cap P)L \cap R)L \cap \overline{R} \cap (R \cap P)L \cap P$

$\Leftrightarrow$ {(11)}

$(R \cap P)L = (R \cap P)L \cap RL \wedge$
$(R \cap P)L \cap \overline{R} \cap P = (R \cap P)L \cap RL \cap \overline{R} \cap (R \cap P)L \cap P$

$\Leftrightarrow$ {Boolean algebra and monotonicity of product}

true **qed**

Proposition 8. *Let R be a specification on space S and P be a candidate program on S. Then the projection of P on R is the least-refined element of the equivalence class of P modulo $\equiv_R$.*

Proof. We let P' be an element in the equivalence class of P, and we show that P' refines $\Pi_R(P)$.

$P' \equiv_R P$

$\Leftrightarrow$ {Proposition 4}

$(R \cap P)L = (R \cap P')L \wedge (R \cap P)L \cap \overline{R} \cap P = (R \cap P')L \cap \overline{R} \cap P'$

$\Rightarrow$ {Boolean algebra}

$(R \cap P)L = (R \cap P')L \wedge (R \cap P')L \cap \overline{R} \cap P' \subseteq P$

$\Leftrightarrow$ {shunting}

$(R \cap P)L = (R \cap P')L \wedge (R \cap P')L \cap P' \subseteq R \cup P$
$\Rightarrow$ $\{(R \cap P)L = (R \cap P')L$ and Boolean algebra$\}$
$(R \cap P)L \subseteq P'L \wedge (R \cap P)L \cap P' \subseteq (R \cap P)L \cap (R \cup P)$
$\Leftrightarrow$ $\{(11)$ and definition of $\Pi_R\}$
$\Pi_R(P)L \subseteq P'L \wedge \Pi_R(P)L \cap P' \subseteq \Pi_R(P)$
$\Leftrightarrow$ $\{$definition of refinement$\}$
$P' \sqsupseteq \Pi_R(P)$. **qed**

Corollary 3. *Let R be a specification on space S and P and P' be (candidate) programs on S. Then P and P' are equally correct if and only if they have the same projection on R.*

Proof. *Proof of Necessity.* By Proposition 7, P and $\Pi_R(P)$ are in the same equivalence class, and so are P' and $\Pi_R(P')$. If P and P' are in the same equivalence class, then (by symmetry and transitivity), so are $\Pi_R(P)$ and $\Pi_R(P')$. Then, by Proposition 8, $\Pi_R(P) \sqsupseteq \Pi_R(P')$ and $\Pi_R(P') \sqsupseteq \Pi_R(P)$. By antisymmetry of refinement, we conclude $\Pi_R(P) = \Pi_R(P')$.

Proof of Sufficiency. If P and P' have the same projection, say π, then by Proposition 7, $P \equiv_R \pi$ and $P' \equiv_R \pi$. By symmetry and transitivity, we infer $P \equiv_R P'$. **qed**

Figure 4 shows an example of a specification R, two equally correct programs with respect to R, P and P', and their common projection on R.

4.3. Projections and Relative Correctness

Because all the programs that are equally correct map to the same projection, one would expect that relative correctness is determined by considering projections. This is confirmed by the following proposition.

Proposition 9. *Let R be a specification on space S, and let P and P' be candidate programs on S. Then P' is more-correct than P with respect to R if and only if $\Pi_R(P')$ refines $\Pi_R(P)$.*

Proof. We proceed by equivalences:
$\Pi_R(P') \sqsupseteq \Pi_R(P)$
$\Leftrightarrow$ $\{$definition of refinement$\}$
$\Pi_R(P)L \subseteq \Pi_R(P')L \wedge \Pi_R(P)L \cap \Pi_R(P') \subseteq \Pi_R(P)$
$\Leftrightarrow$ $\{(11)$ and definition of $\Pi_R\}$
$(R \cap P)L \subseteq (R \cap P')L \wedge (R \cap P)L \cap (R \cap P')L \cap (R \cup P') \subseteq (R \cap P)L \cap (R \cup P)$
$\Leftrightarrow$ $\{(1)\}$
$(R \cap P)L \subseteq (R \cap P')L \wedge (R \cap P)L \cap (R \cup P') \subseteq (R \cap P)L \cap (R \cup P)$
$\Leftrightarrow$ $\{(2)\}$
$(R \cap P)L \subseteq (R \cap P')L \wedge (R \cap P)L \cap (R \cup P') \subseteq R \cup P$
$\Leftrightarrow$ $\{$shunting and Boolean algebra$\}$
$(R \cap P)L \subseteq (R \cap P')L \wedge (R \cap P)L \cap P' \cap \overline{R} \subseteq P$

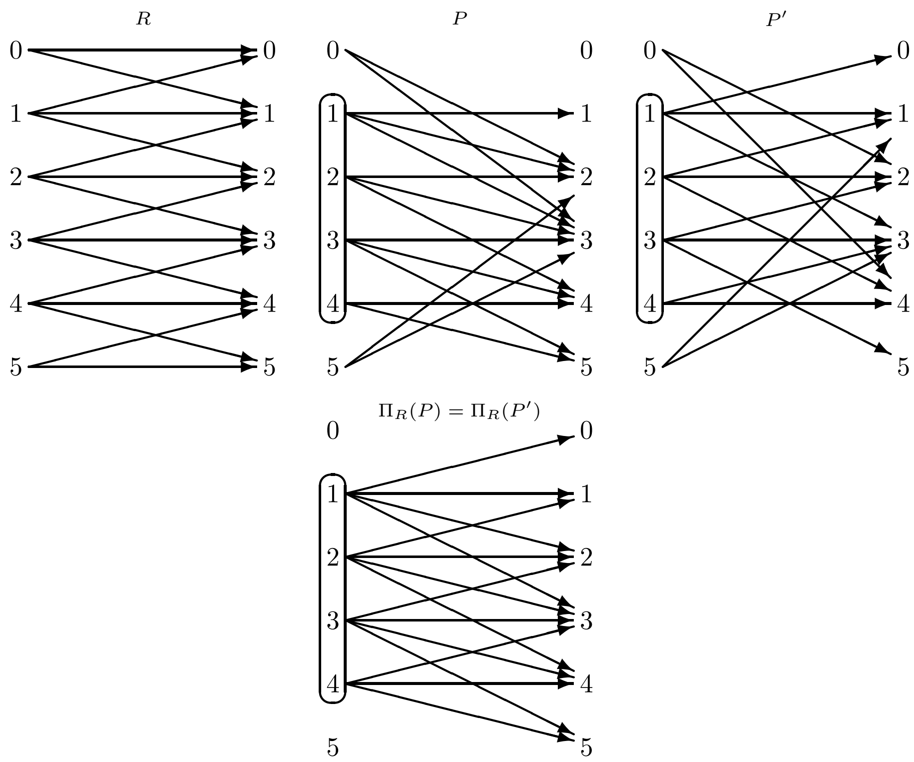

Figure 4: Equal Correctness of Non-Deterministic Programs: $P' \equiv_R P$.

$\Leftrightarrow$ {definition of $\sqsupseteq_R$}

$P' \sqsupseteq_R P$. **qed**

According to this proposition, the more-refined the projection of a program, the more-correct the program. This raises the question: under what condition is a candidate program absolutely correct? The answer is given below.

4.4. Projections and Absolute Correctness

Proposition 10. *A program P is correct with respect to a specification R if and only if the projection of P on R is R.*

Proof. *Proof of Sufficiency.* Assume $\Pi_R(P) = R$. Then by Proposition 6 $P \sqsupseteq \Pi_R(P) = R$.

Proof of Necessity. Assume $P \sqsupseteq R$, i.e. $RL \subseteq PL$ and $RL \cap P \subseteq R$. We first prove $RL = (R \cap P)L$.

true

$\Leftrightarrow$ {assumption}

$RL \cap P \subseteq R$

$\Leftrightarrow$ {Boolean algebra}

$RL \cap P \subseteq R \cap P$

$\Rightarrow$ {monotonicity of product}

$(RL \cap P)L \subseteq (R \cap P)L$

$\Leftrightarrow$ {(11)}

$RL \cap PL \subseteq (R \cap P)L$

$\Leftrightarrow$ {assumption $RL \subseteq PL$ and Boolean algebra}

$RL \subseteq (R \cap P)L$

$\Leftrightarrow$ {Boolean algebra and monotonicity of product}

$RL = (R \cap P)L$

And now the proof of $\Pi_R(P) = R$.

$\Pi_R(P)$

$=$ {definition of Π_R}

$(R \cap P)L \cap (R \cup P)$

$=$ {previous proof}

$RL \cap (R \cup P)$

$=$ {Boolean algebra and $R \subseteq RL$}

$R \cup RL \cap P$

$=$ {assumption $RL \cap P \subseteq R$ and Boolean algebra}

R **qed**

Figure 5 captures the result of Proposition 10, along with that of Corollary 2 in graphic form. The chart on the left shows the configuration of a specification R and a candidate program P when they are unrelated; in that case, according to Corollary 2, the meet of R and P refines the projection of P on R. The chart in the middle shows the configuration of R and P when they are consistent; for a deterministic program P, the consistency condition means that P is partially

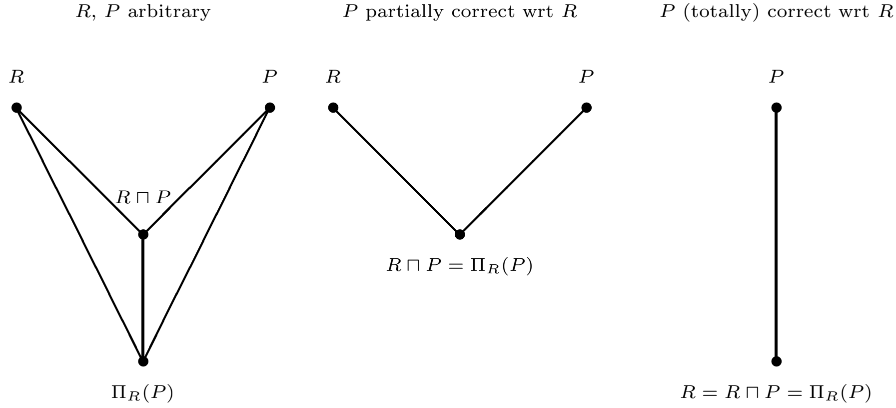

Figure 5: The Meet (greatest lower bound) and the Projection

correct with respect to R; in that case, according to Corollary 2, the meet of R and P and the projection of P over R are identical. The chart on the right represents the configuration of a specification R and a program P where the program P is correct with respect to the specification; in that case, according to Proposition 10, the projection of P over R is equal to R, and according to the definition of refinement, the meet of P and R is equal to R. In this figure, refinement is represented by height (higher relations are more-refined than lower relations).

To summarize, then: the projection of a program on a specification determines the level of relative correctness of a program with respect to the specification; all the programs that are equally correct have the same projection; a candidate program is all the more-correct that its projection is more-refined; all projections are less-refined than or as refined as the specification; correct programs are those whose projection is R.

We present herein a simple graphic interpretation of projection, which reflects all the properties listed above and justifies why we called it thus. Imagine a horseboating arrangement whereby we want to pull a barge along a canal, and to this effect we enlist a horse to walk on a path along the canal; let R be the acceleration that we want to apply to the barge and let P be the acceleration that the horse applies to the cable that connects it to the barge. Then P can be decomposed into two components: the projection of P over the axis of the canal, which represents the useful acceleration that will drive the barge; and the orthogonal acceleration, which serves no purpose as far as accelerating the barge, but is a byproduct of the horseboating design. We argue that the projection of a program on a specification represents the same concept, namely the functionality of P that is relevant to R (i.e. that contributes to meeting requirements of R). See Figure 6; vector E represents the excess function of P (i.e. functional features offered by P but irrelevant for the purpose of R) and vector D represents the functional deficit of P (functionality that is required by R but

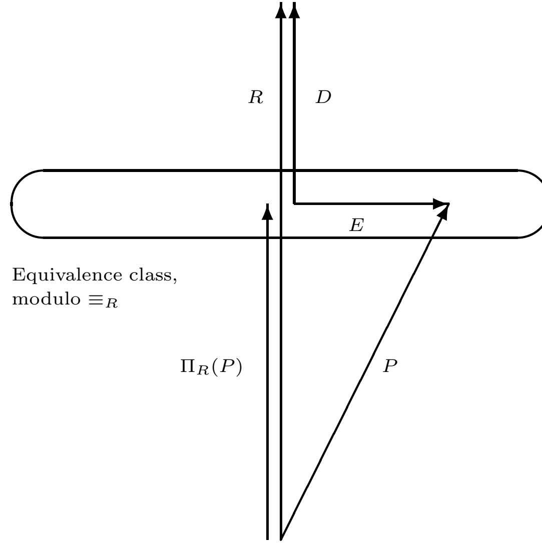

Figure 6: Projecting a Program on a Specification

not fulfilled by P). The oval in this figure represents programs that are equally correct, as they all have the same projection on R; the least-refined element of each oval is the projection (Proposition 8); farther elements may be refinements of the projection, but they are not more-correct than the projection.

4.5. Illustrative Example

We now consider some concrete (yet simple) programs and specifications, and discuss in what way the projection of a program on a specification is consistent with the interpretation we make of it. We let S be the space defined by three variables x, y and z of type integer and we let R be the following relation/specification on S:

$$\{(s, s')|x' = x + y \wedge z < z'\}.$$

We consider the following program on S,

```
z:=z+1; while (y!=0) {x:=x+1; y:=y-1}
```

The function of this program is:

$$P_0 = \{(s, s')|y \geq 0 \wedge x' = x + y \wedge y' = 0 \wedge z' = z + 1\}.$$

The projection of P_0 can be computed as follows:

$(R \cap P_0)L \cap (R \cup P_0)$

$=$ {substitution and simplification}

$\{(s, s')|y \geq 0 \wedge x' = x + y \wedge y' = 0 \wedge z' = z + 1\} \circ L$

$\cap \{(s,s')|x'=x+y \wedge z<z'\}$
$=$ {simplification}
$\{(s,s')|y \geq 0 \wedge x'=x+y \wedge z<z'\}$.

Indeed, program P_0 terminates only when $y \geq 0$, and when it does, it places $x+y$ in x and increases z; we know that P_0 actually increments z by exactly 1, but the projection of P_0 on R keeps no record of that, as R only cares that z has increased, and does not care by how much.

We consider another program on space S,

```
z:=z+1; while (y>0) {x:=x+1; y:=y-1}
```

The function of this program is:

$$P_1 = \{(s,s')|y \geq 0 \wedge x'=x+y \wedge y'=0 \wedge z'=z+1\}$$
$$\cup \{(s,s')|y<0 \wedge x'=x \wedge y'=y \wedge z'=z+1\}.$$

The projection of P_1 on R can be computed as follows:
$(R \cap P_1)L \cap (R \cup P_1)$
$=$ {substitution and simplification}
$(\{(s,s')|y \geq 0 \wedge x'=x+y \wedge y'=0 \wedge z'=z+1\}$
$\cup \{(s,s')|y<0 \wedge x'=x+y \wedge x'=x \wedge y'=y \wedge z'=z+1\}) \circ L$
$\cap (R \cup P_1)$
$=$ $\{x'=x+y \wedge x'=x \Rightarrow y=0$, which contradicts $y<0$ $\}$
$\{(s,s')|y \geq 0 \wedge x'=x+y \wedge y'=0 \wedge z'=z+1\} \circ L \cap (R \cup P_1)$
$=$ {computing the domain}
$\{(s,s')|y \geq 0\} \cap (R \cup P_1)$
$=$ {distribution of $\cap$ over $\cup$, substitution and simplification}
$\{(s,s')|y \geq 0 \wedge x'=x+y \wedge z<z'\}$.

Interestingly, even though P_1 refines P_0, P_1 is not more-correct than P_0 with respect to R, since they have the same projection on R. In other words, P_1 refines P_0 in a way that is orthogonal to R, hence it does not enhance relative correctness with respect to R. See Figure 7.

Next, we consider the following program:

```
x:=x+y; y:=0; z:=z+1
```

The function of this program is:

$$P_2 = \{(s,s')|x'=x+y \wedge y'=0 \wedge z'=z+1\}.$$

The projection of P_2 on R is:
$(R \cap P_2)L \cap (R \cup P_2)$
$=$ {substitution and simplification}
$\{(s,s')|x'=x+y \wedge y'=0 \wedge z'=z+1\} \circ L$
$\cap \{(s,s')|(x'=x+y \wedge z<z') \vee (x'=x+y \wedge y'=0 \wedge z'=z+1)\}$
$=$ {computing the domain and simplification}

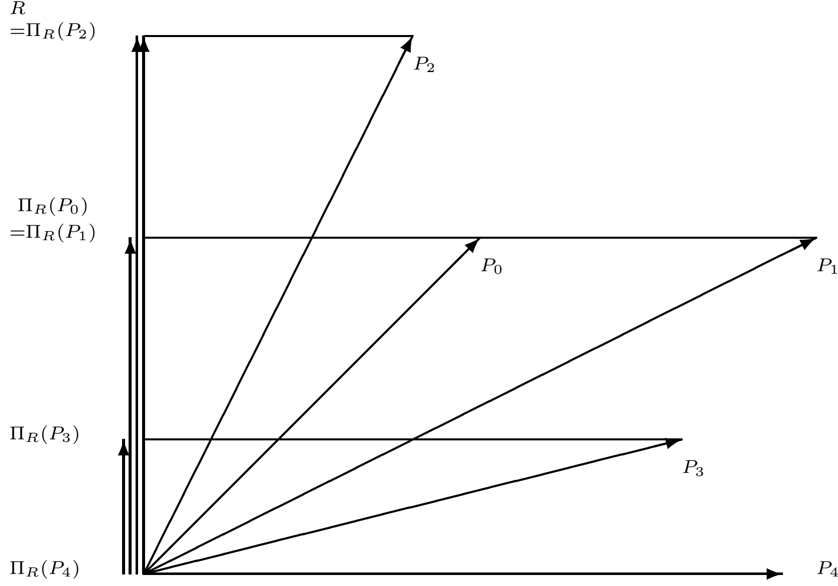

Figure 7: Projecting Candidate Programs on a Specification

$\{(s, s')|x' = x + y \wedge z < z'\}$
$=$ {substitution}
R.

Since the projection of P_2 on R is equal to R, we conclude that P_2 is correct with respect to R. See Figure 7. Note that P_2 refines P_0, as does P_1; but while P_2 is more-correct than P_0 with respect to R, P_1 is not. Intuitively, this is because P_2 refines P_0 in the direction of R (so to speak), whereas P_1 refines it orthogonally to R; see Figure 7.

Next, we consider the following program:

```
z:=z+3; x:=x+10; while (y!=10){y:=y-1; x:=x+1}
```

The function of this program is:

$$P_3 = \{(s, s')|y \geq 10 \wedge x' = x + y \wedge y' = 10 \wedge z' = z + 3\}.$$

The projection of P_3 on R is:
$(R \cap P_3)L \cap (R \cup P_3)$
$=$ {substitution and simplification}
$\{(s, s')|y \geq 10 \wedge x' = x + y \wedge y' = 10 \wedge z' = z + 3\} \circ L$
$\cap \{(s, s')|x' = x + y \wedge z < z'\}$
$=$ {simplification}
$\{(s, s')|y \geq 10 \wedge x' = x + y \wedge z < z'\}$.

We find that the projection of P_3 on R is less-refined than the projection of P_0 and P_1 on R, hence we conclude that P_0 and P_1 are both more-correct than P_3 with respect to R. See Figure 7.

Finally, we consider the following program:

```
while (y!=0) {x:=x+1; y:=y-1}
```

The function of this program is:

$$P_4 = \{(s, s')|y \geq 0 \land x' = x + y \land y' = 0 \land z' = z\}.$$

The projection of P_4 on R is then:

$(R \cap P_4)L \cap (R \cup P_4)$

$=$ $\{$substitution and simplification$\}$

$\{(s, s')|y \geq 0 \land x' = x + y \land y' = 0 \land z < z' \land z' = z\} \circ L \cap (R \cup P_4)$

$=$ $\{$the first term is empty, since $z < z' \land z' = z$ is false$\}$

ϕ.

Figure 7 gives a geometric representation of the five candidate programs P_0 to P_4, and serves to illustrate the definition of projection, as well as the property that relative correctness can be judged by looking exclusively at the projections of candidate programs on the target specification.

Consider program P_4: even though it does some of what the specification mandates, which is to add x and y into x, our calculations show its projection to be the empty specification, hence P_4 is as good as `abort`, as far as this specification is concerned. This may sound counter-intuitive, but it is actually consistent with the fact that our definition of relative correctness does not reflect how many clauses of the specification are satisfied; rather it reflects how many initial states the program is handling correctly; and in this situation, because the specification mandates to increase z and the program does not, there is not a single initial state for which the program satisfies the specification. It is possible to (re)define relative correctness in such a way as to reflect the extent of requirements clauses that are satisfied rather than the extent of the input domain for which the specification (as an indivisible monolith) is satisfied; this matter is beyond the scope of this paper.

5. Implications and Applications

The purpose of this section is to summarily explore some implications and/or applications of projections in software engineering; our only goal is to draw attention to the fact that projections may be used to model several software engineering processes. But first, we need to introduce a proposition, which we need in our subsequent discussions; this is the subject of section 5.1.

5.1. Refining a Projection

Proposition 9 provides that a program P' is more-correct than a program P with respect to some specification R if and only if the projection of P' on R refines the projection of P on R. But in practice, we find it convenient to reason about relative correctness on the basis of a lower bound of P', rather than a lower bound of the projection of P'. The following proposition provides this lower bound.

Proposition 11. *Given a specification R and two candidate programs P and P', such that P is deterministic. Then $P' \sqsupseteq_R P$ if and only if $P' \sqsupseteq \Pi_R(P)$.*

Proof. *Proof of Necessity.* If P' is more-correct than P with respect to R then, by Proposition 9, $\Pi_R(P')$ refines $\Pi_R(P)$. On the other hand, by Proposition 8, P' refines $\Pi_R(P')$. By transitivity, we infer that P' refines $\Pi_R(P)$.

Proof of Sufficiency. Assume that P is deterministic. We first show that $(R \cap P)L \cap P \subseteq R$:

$$(R \cap P)L \cap P \subseteq (R \cap P)(L \cap (\widehat{R} \cap \widehat{P})P) \subseteq (R \cap P)\widehat{P}P \subseteq R \cap P \subseteq R,$$

where the fist step uses (12) and (9), the second one Boolean algebra and monotonicity of product, the third one the assumption of determinacy of P and monotonicity of product, and the last one Boolean algebra. Thus, by Boolean algebra,

$$(R \cap P)L \cap (R \cup P) = (R \cap P)L \cap R \cup (R \cap P)L \cap P \subseteq R \cup R = R.$$

And now the main proof.

$P' \sqsupseteq \Pi_R(P)$
$\Leftrightarrow$ {definition of refinement}
$\Pi_R(P)L \subseteq P'L \ \wedge\ \Pi_R(P)L \cap P' \subseteq \Pi_R(P)$
$\Leftrightarrow$ {(14) and definition of Π_R}
$(R \cap P)L \subseteq P'L \ \wedge\ (R \cap P)L \cap P' \subseteq (R \cap P)L \cap (R \cup P)$
$\Rightarrow$ {last result above}
$(R \cap P)L \subseteq P'L \ \wedge\ (R \cap P)L \cap P' \subseteq R$
$\Rightarrow$ {Shunting and (2)}
$(R \cap P)L \subseteq P'L \ \wedge\ (R \cap P)L \cap \overline{R} \cap P' \subseteq \phi \ \wedge\ (R \cap P)L \cap P' \subseteq R \cap P'$
$\Rightarrow$ {Boolean algebra, multiplying on the right by L and (11)}
$(R \cap P)L \subseteq P'L \ \wedge\ (R \cap P)L \cap \overline{R} \cap P' \subseteq P \ \wedge\ (R \cap P)L \cap P'L \subseteq (R \cap P')L$
$\Rightarrow$ {Using first conjunct in third one and Boolean algebra}
$(R \cap P)L \cap \overline{R} \cap P' \subseteq P \ \wedge\ (R \cap P)L \subseteq (R \cap P')L$
$\Leftrightarrow$ {definition of $\sqsupseteq_R$}
$P' \sqsupseteq_R P$ **qed**

Hence in subsequent discussions, when we want a program P' to be more-correct than a deterministic program P, we do not need to mandate that the projection of P' refine the projection of P; rather it suffices that P' itself refine the projection of P.

5.2. Program Merger

Given a specification R and two candidate programs P and P', imagine that each of P and P' delivers some (but not all) of the functionality mandated by R, and we are interested to derive a program that delivers all of the relevant functionality of P and all of the relevant functionality of P'; we refer to this composite program as the *merger* of P and P'.

Definition 7. *Given a specification R and two candidate programs P and P', a* merger *of programs P and P' is a program P'' that is more-correct than P and more-correct than P' with respect to R.*

The first step in deriving the merger P'' of P and P' is to compute the specification that P'' must satisfy. A naive approach would be to let the specification of the merger be the join of P and P'. But this approach ought to be ruled out for several reasons:

- The join of P and P' may not exist; we show an example below of such a situation.
- When the join of P and P' does not exist, it is usually because of an incompatibility between the design of P and the design of P'; it is never due to an incompatibility between specification-relevant behaviors of P and P'.

A more interesting approach to the merger is to let the specification of the merger be the join of the projections of P and P' over R; this approach has the following advantages:

- According to Corollary 1, the projections of P and P' over R are consistent, hence they necessarily admit a join.
- According to Proposition 6, the projections of P and P' over R are less-refined than, respectively, P and P'. Less-refined specifications are easier to satisfy because they impose fewer/weaker requirements.
- The projections of P and P' on R only capture requirements information that is relevant to R, and exclude any information that may stem from design decisions that have been taken in the derivation of P and P'.
- According to Proposition 11, if P'' refines the projections of P and P' on R, then it is more-correct than P and P'.
- According to lattice theory, in order for P'' to refine the projections of P and P' on R, it suffices that P'' refines their join.

We consider the following example: Let S be the space defined by variables x, y and z of type integer, and let R be the following specification on S:

$$R = \{(s, s')|x' = x + y \land z \neq z'\}.$$

We let p and p' be the following candidate programs on S:

p : `z:=z+1; while (y!=0)` {`x:=x+1; y:=y-1`},

p' : `z:=z-1; while (y!=0)` {`y:=y+1; x:=x-1`}.

The functions of p and p' are, respectively:

$P = \{(s, s')|y \geq 0 \wedge x' = x + y \wedge y' = 0 \wedge z' = z + 1\}$

$P' = \{(s, s')|y \leq 0 \wedge x' = x + y \wedge y' = 0 \wedge z' = z - 1\}$

We compute the projections of P and P' on specification R. We find:

$\Pi_R(P)$

$=$ {definition of Π_R, substitution and simplification}

$\{(s, s')|y \geq 0 \wedge x' = x + y \wedge y' = 0 \wedge z' = z + 1\} \circ L \cap (R \cup P)$

$=$ {computing the domain}

$\{(s, s')|y \geq 0\} \cap (R \cup P)$

$=$ {since $P \subseteq R$}

$\{(s, s')|y \geq 0\} \cap R$

$=$ {substitution}

$\{(s, s')|y \geq 0 \wedge x' = x + y \wedge z' \neq z\}$.

Likewise, we find:

$\Pi_R(P')$

$=$ {definition of Π_R, substitution and simplification}

$\{(s, s')|y \leq 0 \wedge x' = x + y \wedge y' = 0 \wedge z' = z - 1\} \circ L \cap (R \cup P')$

$=$ {computing the domain}

$\{(s, s')|y \leq 0\} \cap (R \cup P')$

$=$ {since $P' \subseteq R$}

$\{(s, s')|y \leq 0\} \cap R$

$=$ {substitution}

$\{(s, s')|y \leq 0 \wedge x' = x + y \wedge z' \neq z\}$.

We know, by Corollary 1, that these two projections are consistent, hence we need not check it; we proceed directly with computing their join. We compute in turn the three terms of the join:

$\Pi_R(P) \cap \overline{\Pi_R(P') \circ L}$

$=$ {substitution, computing the domain and simplification}

$\{(s, s')|y > 0 \wedge x' = x + y \wedge z' \neq z\}$.

The second term:

$\Pi_R(P') \cap \overline{\Pi_R(P) \circ L}$

$=$ {substitution, computing the domain and simplification}

$\{(s, s')|y < 0 \wedge x' = x + y \wedge z' \neq z\}$.

The third term:

$\Pi_R(P) \cap \Pi_R(P')$

$=$ {substitution and simplification}

$\{(s, s')|y = 0 \wedge x' = x \wedge z' \neq z\}$.

To satisfy the union of these three terms, we write the following program:

```
z:=z+1; // from P, contradicts P',
        // but compatible with projection of P'
if (y>0) {while (y!=0) {x:=x+1; y:=y-1}} // from P
else {while (y!=0) {y:=y+1; x:=x-1}} // from P'.
```

This program delivers all the functionality of P and all the functionality of P', and while it differs from P' in its effect on variable z, it is compatible with the projection of P' over R, which is sufficient; it constitutes a merger of P and P', i.e. a program that is more-correct than P and more-correct than P' with respect to R.

Note that if we had tried to compute the join of P and P' (rather than their projections on R), we would have failed: To check the consistency condition between P and P', we proceed as follows:

$PL = \{(s,s')|y \geq 0\}$. $P'L = \{(s,s')|y \leq 0\}$.

Hence

$PL \cap P'L = \{(s,s')|y = 0\}$.

On the other hand,

$$
\begin{aligned}
& (P \cap P')L \\
= & \quad \{\text{substitution}\} \\
& \{(s,s')|y = 0 \wedge x' = x + y \wedge y' = 0 \wedge z' = z + 1 \wedge z' = z - 1\} \circ L \\
= & \quad \{\text{contradiction and simplification}\} \\
& \phi L \\
= & \quad \{(3)\} \\
& \phi.
\end{aligned}
$$

Because this is different from $PL \cap P'L$, we conclude that P and P' have no join. But we notice that the failure to satisfy the consistency condition does not stem from requirements mandated by R; rather it stems from design decisions taken in P and P' that turned out to be irreconcilable (whereas program P satisfies specification $z \neq z'$ by incrementing z, program P' satisfies it by decrementing z). So that by trying to take the join of P and P', we are trying to combine, not only specification-relevant features of P and P', but also design-based features that stem from arbitrary design decisions taken in the process of deriving P and P'. Such design decisions are not only irrelevant for our purposes, they also run the risk of creating inconsistencies between P and P' that preclude us from taking their join.

We summarize:

> The merger of two candidate programs P and P' for specification R is a program that is more-correct than P and more-correct than P' with respect to R. Its specification is obtained by taking the join of $\Pi_R(P)$ and $\Pi_R(P')$.

5.3. Program Upgrade

Let R be a specification on space S and let P be a candidate program for R. Let Q be the specification of a feature that we want to add to program P.

Definition 8. *Given a specification R, a candidate program P and a feature specification Q. An upgrade of program P by feature Q is a program, say P', that is correct with respect to Q and more-correct than P with respect to R.*

Typically, R is a massive/complex specification (e.g. the specification of an enterprise data processing application) and Q is a small feature that we want to add to P (e.g. a new report format); the upgrade of P by Q consists in fulfilling requirements Q without degrading the relative correctness of P with respect to R. According to Proposition 11, P' is more-correct than P with respect to R if and only if P' refines the projection of P over R. According to lattice theory, program P' refines Q and $\Pi_R(P)$ if and only if it refines their join. On the other hand, the join of Q and $\Pi_R(P)$ exists if and only if Q and $\Pi_R(P)$ are consistent. If they are not consistent, then no program can satisfy requirement Q while preserving (relative) correctness with respect to R; in other words, any attempt to make P correct with respect to Q will necessarily make it less-correct with respect to R. We summarize this as follows:

> Given a specification R and a candidate program P, and given a specification Q of a feature we wish to add to P. If Q and $\Pi_R(P)$ are consistent, then any program that refines $Q \sqcup \Pi_R(P)$ represents an upgrade of P with feature Q. If Q and $\Pi_R(P)$ are not consistent, then it is impossible to satisfy specification Q while preserving or enhancing the relative correctness of P.

Note that if R and Q are consistent, then so do $\Pi_R(P)$ and Q, since R refines $\Pi_R(P)$ for any P. Note also that Q and $\Pi_R(P)$ may be consistant while P and Q are not, as we show in the illustrative example below.

We consider space S defined by two variables x and y of type integer and specification R defined as follows:

$$R = \{(s, s')|x' = x + y\}.$$

We let p be the following program on space S:

```
while (y!=0) {y:=y-1; x:=x+1}.
```

Hence $P = \{(s, s')|y \geq 0 \wedge x' = x + y \wedge y' = 0\}$. Note that program P is not correct with respect to R, since it fails to compute the sum of x and y into x for negative y. We now consider the specification Q defined by:

$$Q = \{(s, s')|y > y'\}.$$

We want to upgrade program P to satisfy specification Q without making P less-correct with respect to R in the process. To check whether this is possible, we need to check whether Q and $\Pi_R(P)$ are consistant. To this effect, it is sufficient, though not necessary, to prove that Q and R are consistent, which we can easily see, since RL, QL and $(R \cap Q)L$ are all equal to L.

We now compute $\Pi_R(P)$, then the join of $\Pi_R(P)$ with Q.

$\Pi_R(P)$

$=$ {definition of $\Pi_R(P)$ and substitution}

$\{(s, s')|y \geq 0 \wedge x' = x + y \wedge y' = 0\} \circ L \cap (R \cup P)$

$=$ {taking the domain in the first term, simplifying the second}
$\{(s,s')|y \geq 0 \wedge x' = x + y\}$.

To check the consistency condition between Q and $\Pi_R(P)$, we compute the intersection of their domains, then the domain of their intersection.

$QL = L$.

$\Pi_R(P) \circ L = \{(s,s')|y \geq 0\}$.

$(Q \cap \Pi_R(P)) \circ L = \{(s,s')|y \geq 0 \wedge x' = x + y \wedge y > y'\} \circ L = \{(s,s')|y \geq 0\}$.

The consistency condition is satisfied, hence we now compute the join of Q and $\Pi_R(P)$, and we find, for the first term:

$\overline{QL} \cap \Pi_R(P) = \overline{L} \cap \Pi_R(P) = \phi$.

The second term:

$\overline{\Pi_R(P)L} \cap Q = \overline{\{(s,s')|y \geq 0\}} \cap Q = \{(s,s')|y < 0 \wedge y > y'\}$.

The third term:

$Q \cap \Pi_R(P) = \{(s,s')|y \geq 0 \wedge x' = x + y \wedge y > y'\}$.

The join can then be written as:

$$Q \sqcup \Pi_R(P) = \{(s,s')|y < 0 \wedge y > y'\} \cup \{(s,s')|y \geq 0 \wedge x' = x + y \wedge y > y'\}.$$

The following program, say P', satisfies this specification:

```
if (y<=0) {y:=y-1}
else {while (y!=0) {y:=y-1; x:=x+1}}
```

Note that this program is not correct with respect to R, but neither is P; It is, however correct with respect to Q, and it is more-correct (though not strictly more-correct) than P with respect to R. Note however that this program does not duplicate the behaviour of P: for $y = 0$, program P preserves x and y whereas program P' preserves x but decrements y (as mandated by Q).

While Q and $\Pi_R(P)$ are consistent, Q and P are not, as we see below:

$QL = L$.

$PL = \{(s,s')|y \geq 0 \wedge x' = x + y \wedge y' = 0\} \circ L = \{(s,s')|y \geq 0\}$.

Hence $QL \cap PL = \{(s,s')|y \geq 0\}$. On the other hand,

$(Q \cap P)L$
$=$ {substitution}
$\{(s,s')|y \geq 0 \wedge x' = x + y \wedge y' = 0 \wedge y > y'\} \circ L$
$=$ {computing the domain}
$\{(s,s')|y > 0\}$.

Since $QL \cap PL$ is not identical to $(Q \cap P)L$, specifications Q and P are not consistent, hence they admit no join. In other words, it is impossible to satisfy specification Q while preserving the function of P; what we did above is to satisfy Q while preserving the projection of P over R (so as to preserve the relative correctness of P with respect to R).

5.4. Program Documentation

Documentation is an important aspect of software engineering practice; professional standards of software engineering dictate that one must document software products by commenting on design features and/or by providing information about the programmer's intent; traditionally, this has meant documenting

the function of the program. The trouble with documenting the function of the program is that typically, the function of the program reflects two separate sources of information: the requirements that stem from the specification and the functional features that stem from the design of the product. We argue that it is important to distinguish between these, which the projection does, since it captures only the former. We illustrate this contrast by a very simple example: consider a space S defined by a single integer variable x, and consider the specification

$$R = \{(x, x')|x < x'\}.$$

Let p be a candidate program (i.e. a program written to satisfy R) that raises x to power 2. We give below the function of P of p and the projection of P over R:

- $P = \{(x, x')|x' = x^2\}$.
- $\Pi_R(P)$
  $=$ $\{\text{definition of } \Pi_R\}$
  $(R \cap P)L \cap (R \cup P)$
  $=$ $\{\text{substitution}\}$
  $\{(x, x')|x < x' \wedge x' = x^2\} \circ L \cap \{(x, x')|x < x' \vee x' = x^2\}$
  $=$ $\{\text{computing the domain}\}$
  $\{(x, x')|x < 0 \vee x > 1\} \cap \{(x, x')|x < x' \vee x' = x^2\}$
  $=$ $\{\text{since } (x < 0 \vee x > 1) \Rightarrow x < x^2\}$
  $\{(x, x')|(x < 0 \vee x > 1) \wedge x < x'\}$.

The question we discuss is: what is an appropriate comment for program P?

- We could comment the program with the program function P: `x:=x*x; // x'=x*x`. But this comment reflects not only the specification that the program is intended to satisfy, but also the design decisions that the programmer took in order to satisfy the specification. Yet in practice, we may want to distinguish between these two aspects: in the context of corrective maintenance, for example, it is important to distinguish between what stems from the specification (which must, therefore, be preserved) and what stems from the design (which may be modified if that is needed to remove the fault).
- We could, alternatively, comment the program with the specification R that the program is intended to satisfy: `x:=x*x; // x<x'`. The trouble with this comment is that it does not hold, actually, since the program does not increase x for x between 0 and 1.
- The third alternative is to comment this program with the projection of P over R: `x=x*x; // (x<0 || x>1) && x<x'`. This comment captures the essence of what program P does to satisfy specification R; it captures neither the functional deficit of P (the fact that P does not satisfy R for x between 0 and 1) nor the functional excess of P (the fact that P satisfies R by raising x to power 2).

In light of this discussion, we argue that while documenting the projection of a program on a specification is not necessarily always the best way to comment on a piece of code, it is nevertheless a competitive option. Indeed, commenting on a software product by means of its function may be unnecessary (i.e. provide more information than needed) whereas commenting on it by means of its target specification may be insufficient (i.e. recording what a program must do, rather than what it actually does). We summarize our finding as follows:

> When we comment on a program by documenting its projection on the target specification, we capture all the functionality that the program delivers to satisfy the specification, and nothing else (i.e. no functionality that is not mandated by the specification, and no mandated functionality that is not delivered by the program).

6. Related Work

We are not aware of any other researchers who have studied program projections; therefore, we compare our work to other researchers who have studied various forms of relative correctness, which is the closest topic to what we are discussing in this paper.

In [13] Logozzo et al. introduce a technique for extracting and maintaining semantic information across program versions. The focus of their analysis is the condition under which programs P and P' can execute without causing an abort. They implement their approach in a system called VMV (*Verification Modulo Versions*) whose goal is to exploit semantic information about P in the analysis of P', and to ensure that the transition from P to P' happens without regression; in that case, they say that P' is *correct relative to* P. The definition of relative correctness of Logozzo et al. [13] is different from ours, for several reasons: whereas [13] talk about relative correctness between an original program and a subsequent version in the context of adaptive maintenance, we talk about relative correctness between an original (faulty) software product and a revised version of the program (possibly still faulty yet more-correct) in the context of corrective maintenance with respect to a fixed requirements specification; whereas [13] use a set of assertions inserted throughout the program as a specification, we use a relation that maps initial states to final states; whereas [13] represent program executions by execution traces, we represent program executions by functions mapping initial states to final states; finally, whereas Logozzo et al. define a successful execution as a trace that satisfies all the relevant assertions, we define it as an initial state/final state pair that satisfies the relational specification.

In [11] Lahiri et al. introduce a technique called *Differential Assertion Checking* for verifying the relative correctness of a program with respect to a previous version of the program. Lahiri et al. explore applications of this technique as a tradeoff between soundness (which they concede) and lower costs (which they

hope to achieve). Also, they define relative correctness between programs P and P' as the property that P' has a larger set of successful traces and a smaller set of unsuccessful traces than P; and they introduce relative specifications as specifications that capture functionality of P' that P does not have. By contrast, we use input/output (or initial state/final state) relations as specifications, we represent program executions by functions from initial states to final states, we characterize correct executions by initial state/final state pairs that belong to the specification, and we make no distinction between abort-freedom (a.k.a. safety, in [11]) and normal functional properties. Another important distinction with [11] is that we do not view relative correctness as a compromise that we accept as a substitute for absolute correctness; rather we argue that in many cases, we ought to test programs for relative correctness rather than absolute correctness, regardless of cost. In other words, whereas Lahiri et al. use relative correctness as a compromise solution whenever absolute correctness is too expensive/ impractical, we use relative correctness in specific contexts because that is the exact property that we want to consider.

In [12], Logozzo and Ball introduce a definition of relative correctness whereby a program P' is correct relative to P (*an improvement* over P) if and only if P' has more good traces and fewer bad traces than P. Programs are modeled with trace semantics, and execution traces are compared in terms of executable assertions inserted into P and P'; in order for the comparison to make sense, programs P and P' have to have the same (or similar) structure and/or there must be a mapping from traces of P to traces of P'. When P' is obtained from P by a transformation, and when P' is provably correct relative to P, the transformation in question is called a *verified repair*. Like the work cited above ([11, 13]), Logozzo and Ball model programs by execution traces and distinguish between two types of failures: contract violations, when functional properties are not satisfied; and run-time errors, when the execution causes an abort; for the reasons we discuss above, we do not make this distinction, and model the two aspects with the same relational framework.

Most of the work cited in this section is due to the software research group at Microsoft Research; their main motivation for introducing and discussing relative correctness is different from ours. Whereas we introduced relative correctness as a way to define software faults [15], they introduce it in the context of implementing practical automatic verification tools, as part of a tool chain that already uses trace and assertion-based semantics. This explains, in particular, the contrasts that we have alluded to throughout this section between our work and theirs.

7. Conclusion

Given a specification R and a candidate program P, it is very common for program P to feature functional properties that are not mandated by the specification, even while falling short of fulfilling the functional properties that the specification does mandate. In this paper, we introduce a relational operator that captures the functional properties of a candidate program that are directly

relevant to the requirements of the specification, and we analyze its properties and its applications. We refer to this operator as the projection of program P on specification R, and we liken it to a horseboating arrangement whereby a horse pulls a barge along a canal by walking on a path that runs alongside the canal; the acceleration applied by the horse to the cable that connects it to the barge is composed of two vectors: the projection of the acceleration on the axis of the canal, which is the useful acceleration; and the orthogonal acceleration which serves no useful purpose but is a byproduct of the system's design. Among the most salient properties we have found for this operator, we cite:

- The projection is idempotent.
- All the candidate programs that are equally correct with respect to specification R have the same projection on R; the projection of P on R is as-correct as P; and the projection is the least-refined element of its equivalence class.
- A candidate program is all the more-correct with respect to R that its projection on R is more-refined.
- All the projections of candidate programs on specification R form a lattice, in which `abort` is the minimal element and R is the maximal element.
- Candidate Programs whose projection on R is R are correct with respect to R.

We tentatively analyze possible applications of program projections, and we find the following:

- *Documentation.* In general, when we analyze the functional attributes of a program, we are looking at two sources of information: information that stems from the specification; and information that stems from design decisions that were taken in the process of deriving the program from its specification. To the extent that the projection of the program captures the former information and abstracts away the latter, we feel that it may be useful to document programs by describing their projection. Also, by documenting a program with its projection on the specification, we record not only what the program does, but also why it does it.
- *Program Merger.* When we want to merge two candidate programs so as to cumulate their functional attributes, it is sensible not to try to combine all the functional attributes of each, but rather to combine the projections of the programs on the specification. By focusing on projections rather than on the original program, we ensure that we only deal with relevant information (i.e. information that pertains to the specification, rather than information that stems from design decisions), we ensure that we deal with minimal specifications (that are easier to solve), and we ensure that the programs can be combined (the join of the projections is defined).

- *Program Upgrade*. Given a specification R and a candidate program P, and given a new feature Q, we model the upgrade of P by Q as the problem of producing a program that is correct with respect to Q and more-correct than P with respect to R. We find that the upgrade can be obtained by refining the join of Q with the projection of P on R; we also find that if the join does not exist, then the upgrade is not possible without degrading the relative correctness of P with respect to R.

We are currently exploring the use of this concept to model such software engineering processes as: software design, software evolution, agile programming, and test driven software design [18, 10].

Acknowledgements

The authors are very impressed by the rigor, precision and insightfulness of the reviewers' comments, and are very grateful to the reviewers for the care and meticulousness with which they have read, analyzed and assessed this paper.

References

[1] Boudriga, N., Elloumi, F., Mili, A., 1992. The lattice of specifications: Applications to a specification methodology. Formal Aspects of Computing 4, 544–571.

[2] Brink, C., Kahl, W., Schmidt, G., 1997. Relational Methods in Computer Science. Springer Verlag.

[3] Desharnais, J., Diallo, N., Ghardallou, W., Frias, M.F., Jaoua, A., Mili, A., 2015. Relational mathematics for relative correctness, in: Relational and Algebraic Methods in Computer Science, 2015, Springer Verlag, Lisbon, Portugal. pp. 191–208. doi:10.1007/978-3-319-24704-5_12.

[4] Dijkstra, E., 1976. A Discipline of Programming. Prentice Hall.

[5] Gries, D., 1981. The Science of Programming. Springer Verlag. doi:10.1007/978-1-4612-5983-1.

[6] Hehner, E., 1993. A Practical Theory of Programming. Springer-Verlag.

[7] Hehner, E.C., 1992. A Practical Theory of Programming. Prentice Hall. doi:10.1007/978-1-4419-8596-5.

[8] Hoare, C., 1969. An axiomatic basis for computer programming. Communications of the ACM 12, 576–583. doi:10.1145/363235.363259.

[9] Hoare, T., 1997. Unified theories of programming, in: Mathematical Methods in Program Development, Springer-Verlag. pp. 313–367.

[10] Janzen, D., Saiedian, H., 2005. Test driven development: Concepts, taxonomy and future direction. IEEE Computer 38, 43–50.

[11] Lahiri, S.K., McMillan, K.L., Sharma, R., Hawblitzel, C., 2013. Differential assertion checking, in: Proceedings, ESEC/FSE, pp. 345–355. doi:10.1145/2491411.2491452.

[12] Logozzo, F., Ball, T., 2012. Modular and verified automatic program repair, in: Proceedings, OOPSLA, pp. 133–146. doi:10.1145/2384616.2384626.

[13] Logozzo, F., Lahiri, S., Faehndrich, M., Blackshear, S., 2014. Verification modulo versions: Towards usable verification, in: Proceedings, PLDI, pp. 294–304. doi:10.1145/2594291.2594326.

[14] Manna, Z., 1974. A Mathematical Theory of Computation. McGraw-Hill.

[15] Mili, A., Frias, M., Jaoua, A., 2014. On faults and faulty programs, in: Hoefner, P., Jipsen, P., Kahl, W., Mueller, M.E. (Eds.), Proceedings, RAMICS: 14th International Conference on Relational and Algebraic Methods in Computer Science, Springer, Marienstatt, Germany. pp. 191–207. doi:10.1007/978-3-319-06251-8_12.

[16] Mills, H.D., Basili, V.R., Gannon, J.D., Hamlet, R.G., 1987. Principles of Structured Programming: A Mathematical Approach. Allyn and Bacon, Boston, Ma.

[17] Morgan, C.C., 1998. Programming from Specifications, Second Edition. International Series in Computer Sciences, Prentice Hall, London, UK.

[18] Perelman, D., Gulwani, S., Grossman, D., Provost, P., 2014. Test driven synthesis, in: Proceedings, 35th ACM SIGPLAN Conference, PLDI, Edinburgh, UK. pp. 408–418.

[19] Schmidt, G., Ströhlein, T., 1993. Relations and Graphs. Springer Verlag.