Relative Correctness:  
A Bridge Between Testing and Proving

Nafi Diallo  
Monmouth University, West Long Branch, NJ and Amazon Research, New York, NY  
E-mail: ncdiallo@gmail.com

Wided Ghardallou  
University of Tunis El Manar Tunis, Tunisia  
E-mail: wided.ghardallou@gmail.com

Lamia Labed Jilani  
Higher Institute of Management Bardo, Tunisia  
E-mail: lamia.labeled@isg.rnu.tn

Ali Mili  
NJIT, Newark NJ  
E-mail: mili@njit.edu

Keywords  
Absolute correctness, relative correctness, program testing, program proving, debugging without testing, programming without refining, testing for relative correctness, program repair with relative correctness.

1 Blurring Traditional Distinctions

Relative correctness is the property of a program to be more-correct than another with respect to a given specification; intuitively, a program $P'$ is more-correct than a program $P$ with respect to a specification $R$ if and only if $P'$ behaves according to $R$ more often than $P$, and violates $R$ less egregiously (i.e. in fewer ways) than $P$. Whereas traditionally we distinguish between two categories of candidate programs for a given specification $R$, namely correct programs and incorrect programs, relative correctness enables us to arrange candidate programs over a partial ordering structure, whose maximal elements are the correct programs, and all non-maximal elements are incorrect.
Also, traditionally, proving methods and testing methods have been used on different sets of programs:

- Proving methods are deployed on correct programs to prove their correctness; they are of limited use when deployed on incorrect programs because even when a proof fails, we cannot always conclude that the program is incorrect, since we cannot tell whether the proof failed because the program is incorrect or because it was improperly documented (re: invariant assertions, intermediate assertions, etc). Some methods of program analysis can identify sources of faults when the correctness proof fails, but their scope is limited.

- Testing methods are deployed on incorrect programs to detect, locate and remove their faults; they are useless when deployed on correct programs, because no matter how thoroughly we can test a program, we can never be certain of its correctness.

We argue that consideration of relative correctness has the potential to alter the practice of proving methods and testing methods:

- Once we have a formal definition of relative correctness, we can deploy proving methods to an incorrect program to prove that while it may be incorrect, it is still more-correct than another. Given that there are orders of magnitude more incorrect programs than correct programs, the ability to apply proving methods to incorrect programs expands the scope of these methods significantly. This approach is discussed in section 4.

- Program repair is an emerging field of active research whose focus is on altering an incorrect program to remedy observed failures. We argue that it is impossible to study program repair without a concept of relative correctness, as we feel that relative correctness plays for program repair the same role that absolute correctness plays for program derivation: in the same way that absolute correctness is the criterion by which we can judge the validity of a program derivation (is the final program correct with respect to the original specification?), we argue that relative correctness ought to be the criterion by which we can judge the validity of a program repair (is the final program more-correct than the original program with respect to the reference specification?). In section 5 we critique current practice in program repair then briefly discuss how a formal/ vetted definition of relative correctness can enhance the state of the art and the state of the practice.

- It has long been a cornerstone of software engineering wisdom that programs should not be developed then checked for correctness, but should instead be developed hand-in-hand along with their proof, with the proof leading the way [14]; echoing David Gries, Carrol Morgan talks about developing programs by calculation from their specification, in the same way that a mathematician solves an equation by computing its root [32]. The prevailing paradigm for developing programs from specifications is that of refinement, whereby a program is derived from a specification through a sequence of correctness-preserving transformations based on refinement. In section 6 we present an alternative paradigm based on relative correctness, illustrate it with a simple example, and briefly compare it to related work.

In section 2 we introduce the mathematical background that is needed to carry out our discussions, and in section 3 we introduce our definition of relative correctness.
Relative Correctness: A Bridge Between Testing and Proving

for deterministic and non-deterministic programs. Also, we conclude in section 7 by summarizing our findings, discussing related work, and sketching future directions of research.

This paper extends the work presented in [8] by vetting the definition of relative correctness in more detail, by exploring the potential practical applications of relative correctness for non-deterministic programs, and by investigating the use of relative correctness in the patch validation phase of program repair.

2 Background

2.1 Relational Mathematics

In this paper we use relations to represent specifications and programs, hence we devote this section to discussing some operations and properties of relations. We assume the reader familiar with elementary relational algebra [3], hence this section is not a tutorial on relations as much as it is an introduction to some relevant definitions and notations. We represent sets in a program-like notation by writing variable names and associated data types (sets of values); if we write $S$ as:

$x : X; y : Y$

where $X$ and $Y$ are data types (sets of values), then we mean to let $S$ be the cartesian product $S = X \times Y$; elements of $S$ are usually denoted by $s$ and the $X$- (resp. $Y$-) component of $s$ is denoted by $x(s)$ (resp. $y(s)$). When no ambiguity arises, we may write $x$ for $x(s)$, and $x'$ for $x(s')$, etc. A relation $R$ on set $S$ is a subset of $S \times S$. Special relations on $S$ include the universal relation $L = S \times S$, the identity relation $I = \{(s, s) | s \in S\}$ and the empty relation $\phi = \{\}$. Operations on relations include the set theoretic operations of union ($\cup$), intersection ($\cap$) and complement ($\overline{R}$); they also include the converse of a relation $R$ defined by $\overline{R} = \{(s', s') | (s', s) \in R\}$, the domain of a relation defined by $\text{dom}(R) = \{s | \exists s' : (s, s') \in R\}$, the range of a relation defined by $\text{rng}(R) = \text{dom}(\overline{R})$, and the product of two relations $R$ and $R'$ defined by: $R \circ R' = \{(s, s'') | \exists s' : (s, s') \in R \land (s'', s') \in R'\}$; when no ambiguity arises, we may write $RR'$ for $R \circ R'$.

A relation $R$ is said to be reflexive if and only if $I \subseteq R$, symmetric if and only if $R = \overline{R}$, antisymmetric if $R \cap \overline{R} \subseteq I$, asymmetric if and only if $R \cap \overline{R} = \phi$ and transitive if and only if $RR \subseteq R$. A relation $R$ is said to be total if and only if $I \subseteq RR$ and deterministic if and only if $RR \subseteq I$. A relation $R$ is said to be a vector if and only if $RL = R$; vectors have the form $R = A \times S$; we use them as relational representations of sets. In particular, note that $RL$ can be written as $\text{dom}(R) \times S$; we use it as a relational representation of the domain of $R$.

2.2 Program Semantics

Given a program $p$ on space $S$ written in a C-like notation, we define the function of $p$ (denoted by $P$) as the function that $p$ defines on $S$, i.e. the set of pairs $(s, s')$ such that if program $p$ starts execution in state $s$ it terminates in state $s'$; we may, when no ambiguity arises, refer to a program and its function by the same name, $P$. Because our discussion of correctness and relative correctness refers to a notion of refinement, we give here a definition of this property.
Definition 2.1: Given two relations \( R \) and \( R' \), we say that \( R' \) refines \( R \) (abbrev: \( R' \sqsubseteq R \) or \( R \sqsubseteq R' \)) if and only if \( RL \cap R'L \cap (R \cup R') = R \).

Intuitively, this definition means that \( R' \) has a larger domain than \( R \) and that on the domain of \( R \), \( R' \) assigns fewer images to each argument that does \( R \). See Figure 1. We use refinement to define correctness.

Definition 2.2: A program \( p \) on space \( S \) is said to be correct with respect to specification \( R \) on \( S \) if and only if its function \( P \) refines \( R \).

This definition is identical (modulo differences of notation) to traditional definitions of total correctness [14, 15, 27, 32].

3 Relative Correctness

3.1 Relative Correctness: Deterministic Programs

Proposition 1: Due to [30]. Given a specification \( R \) and a program \( P \), program \( P \) is correct with respect to \( R \) if and only if \( (R \cap P)L = RL \).

Interpretation: \( RL \), the domain of \( R \), is the set of states on which execution of candidate programs must produce correct outputs according to \( R \); on the other hand, \( (R \cap P)L \) is the set of states on which execution of candidate program \( P \) does produce correct outputs according to \( R \). The program \( P \) is correct if and only if these two sets are identical.

Definition 3.1: Due to [29]. Given a specification \( R \) and two deterministic programs \( P \) and \( P' \), we say that \( P' \) is more-correct (resp. strictly more-correct) than \( P \) with respect to \( R \) if and only if \( (R \cap P')L \supseteq (R \cap P)L \) (resp. \( (R \cap P')L \supset (R \cap P)L \)).

To contrast relative correctness with the definition of correctness given in Definition 2.2, we may refer to the latter as absolute correctness. We refer to the domain of \( (R \cap P) \) as the competence domain of \( P \) with respect to \( R \); for deterministic programs, to be more-correct simply means to have a larger competence domain. See Figure 2; note that \( P' \) is more-correct than \( P \) but does not duplicate the correct behavior of \( P \); instead, it has a different correct behavior, which is possible due to the (potential) non-determinacy of specifications.

How do we know that our definition of relative correctness is any good? We present below four properties, which we would want a concept of relative correctness to satisfy; we find in [29] that our definition satisfies them all, at least for deterministic programs.
Relative Correctness culminates in absolute correctness. A (absolutely) correct program is more-correct than (or as correct as) any candidate program. Let \( R \) be an arbitrary program on space \( S \) that provides in effect that whatever \( P \) can do as well or better; if such is the case then one would think that for any given specification \( R \), \( P' \) would be more-correct than \( P \) with respect to \( R \). Indeed, we find that program

\[
\begin{align*}
(P' \sqsupseteq_R P) \Rightarrow (\forall P : P' \sqsupseteq_R P).
\end{align*}
\]

Relative Correctness and Refinement. Refinement is a bipartite relation between two programs \( P' \) and \( P \) that provides in effect that whatever \( P \) can do, \( P' \) can do as well or better; if such is the case then one would think that for any given specification \( R \), \( P' \) would be more-correct than \( P \) with respect to \( R \). Indeed, we find that program

\[
\begin{align*}
P' \sqsupseteq_R P \Rightarrow \int_{CD'} \theta(s)ds \geq \int_{CD} \theta(s)ds.
\end{align*}
\]

If \( P' \) is more-correct than \( P \) with respect to \( R \), then it is more reliable than \( P \); but more-reliable is not equivalent to more-correct, as the latter is a logical property whereas the former is a stochastic property. In particular, \( P' \) can be more reliable than \( P \), not because it has a larger competence domain, but because its competence domain covers inputs with higher probability density.
This example illustrates a number of properties:

\[ P' \supseteq P \iff (\forall R : P' \supseteq_R P). \]

To illustrate the definition of relative correctness, we consider the space \( S \) defined by two integer variables \( x \) and \( y \), and we let \( R \) be the following specification on \( S \):

\[ R = \{(s, s') | x^2 \leq x'y' \leq 2x^2 \}. \]

We consider the following candidate programs, denoted \( p_0 \) through \( p_7 \). Next to each program \( p_i \), we represent its competence domain \((CD_i)\). Figure 3 shows how these candidate programs are ranked by relative correctness with respect to \( R \); this graph merely reflects the inclusion relationships between the competence domains.

\( p_0 \): \{ \( x=1; \ y=-1; \) \}. We find: \( P_0 = \{(s, s') | x' = 1 \land y' = -1 \}. \) Whence,

\( (P_0 \land_R) L = \{(s, s') | \exists s' : x' = 1 \land y' = -1 \land x^2 \leq -1 \leq 2x^2 \} = \{\}. \)

\( p_1 \): \{ \( x=2*x; \ y=0; \) \}. We find: \( P_1 = \{(s, s') | x' = 2x \land y' = 0 \}. \) Whence,

\( (P_1 \land_R) L = \{(s, s') | \exists s' : x' = 2x \land y' = 0 \land x^2 \leq 0 \leq 2x^2 \} = \{(s, s') | x = 0 \}. \)

\( p_2 \): \{ \( x=x*x; \ y=0; \) \}. We find: \( P_2 = \{(s, s') | x' = x^2 \land y' = 0 \}. \) Whence,

\( (P_2 \land_R) L = \{(s, s') | \exists s' : x' = x^2 \land y' = 0 \land x^2 \leq 0 \leq 2x^2 \} = \{(s, s') | x = 0 \}. \)

\( p_3 \): \{ \( x=2*x; \ y=1; \) \}. We find: \( P_3 = \{(s, s') | x' = 2x \land y' = 1 \}. \) Whence,

\( (P_3 \land_R) L = \{(s, s') | \exists s' : x' = 2x \land y' = 1 \land x^2 \leq 2x \leq 2x^2 \} = \{(s, s') | 0 \leq x \leq 2 \}. \)

\( p_4 \): \{ \( x=2*x; \ y=2; \) \}. We find: \( P_4 = \{(s, s') | x' = 2x \land y' = 2 \}. \) Whence,

\( (P_4 \land_R) L = \{(s, s') | \exists s' : x' = 2x \land y' = 2 \land x^2 \leq 4x \leq 2x^2 \} = \{(s, s') | x = 0 \lor 2 \leq x \leq 4 \}. \)

\( p_5 \): \{ \( x=2*x; \ y=x/2; \) \}. We find: \( P_5 = \{(s, s') | x' = 2x \land y' = x \}. \) Whence,

\( (P_5 \land_R) L = \{(s, s') | \exists s' : x' = 2x \land y' = x \land x^2 \leq 2x^2 \leq 2x^2 \} = L. \)

\( p_6 \): \{ \( y=x/2; \ x=2*x; \) \}. We find: \( P_6 = \{(s, s') | x' = 2x \land y' = x/2 \}. \) Whence,

\( (P_6 \land_R) L = \{(s, s') | \exists s' : x' = 2x \land y' = x/2 \land x^2 \leq 2x^2 \leq 2x^2 \} = L. \)

\( p_7 \): \{ \( x=x*x; \ y=2; \) \}. We find: \( P_7 = \{(s, s') | x' = x^2 \land y' = 2 \}. \) Whence,

\( (P_7 \land_R) L = \{(s, s') | \exists s' : x' = x^2 \land y' = 2 \land x^2 \leq 2x^2 \leq 2x^2 \} = L. \)

This example illustrates a number of properties:

- Note that this relation is not antisymmetric; so that two programs may be mutually related and still be distinct (such is the case for \( P_1 \) and \( P_2 \), for example).

- The top of the graph represents the programs that are (absolutely) correct with respect to specification \( R \); \( P_3 \), \( P_6 \) and \( P_7 \).

- A program may be more-correct than another without imitating its correct behavior. For example, \( p_3 \) is more-correct than \( p_1 \) and yet it does not behave as \( p_1 \) on the competence domain of \( p_1 \).
In this section we generalize the definition of relative correctness to non-deterministic programs, as provided by [7]. One may want to ask why we want to discuss relative correctness of non-deterministic programs, when the programming language we use is deterministic. There are several reasons why it may be interesting to investigate relative correctness between non-deterministic programs.

- We may want to compare programs by relative correctness without having to compute their function in all its minute detail; for example, if program $P$ manipulates variables $x$, $y$ and $z$ and program $P'$ is more correct than $P$ with respect to $R$ by virtue of dealing better than $P$ with variables $x$ and $y$ (more in keeping with $R$), then we want to reach that conclusion by focusing exclusively on the behavior of $P$ and $P'$ on variables $x$ and $y$. But doing so means that we do not need to determine how $P$ and $P'$ affect variable $z$; hence we will deal with non-deterministic representations of $P$ and $P'$.

- We may want to compare, not only programs, but also partially defined designs or partially specified algorithms.

- Some common programming languages may feature some degree of non-determinacy either by virtue of compile-time decision-making variability or by virtue of run-time-dependent circumstances.

We have the following definition, due to [7].
Definition 3.2: Given a specification $R$ and two programs $P$ and $P'$. We say that $P'$ is more-correct than $P$ with respect to $R$ if and only if:

$$(R \cap P')_L \supseteq (R \cap P)_L \land (R \cap P)_L \cap R \cap P' \subseteq P.$$ 

Also, we say that $P'$ is strictly more-correct than $P$ with respect to $R$ if and only if at least one of the inequalities in this definition is strict.

Interpretation: The first clause provides that $P'$ has a larger competence domain than $P$. To understand the second clause, consider that the left-hand side of this clause represents the set of $(s, s')$ pairs such that $s$ is in the competence domain of $P$ (re: $(R \cap P)_L$) and $s'$ is an image of $s$ by $P'$ that violates specification $R$ (re: $R \cap P'$). What this clause is providing is that any such $(s, s')$ pair is in $P$; so that on the competence domain of $P$, $P'$ does not violate specification $R$ unless $P$ does (but $P$ may violate $R$ in ways that $P'$ does not). In other words: $P'$ is more-correct than $P$ with respect to $R$ if and only if $P'$ has a larger competence domain (first clause), and violates $R$ in fewer ways (second clause). See Figure 4: The competence domain of $P$ is $\{1, 2\}$ and that of $P'$ is $\{1, 2, 3\}$. Program $P'$ violates $R$ by assigning 1 to 1 and 2 to 2; but $P$ is guilty of the same misdeed, in addition to also assigning 0 to 2 and 3 to 1, in violation of specification $R$. Hence $P'$ is more-correct than $P$ with respect to $R$.

3.3 Illustration: Static Analysis

In order to illustrate how non-deterministic programs come about, and how we can reason about the relative correctness of such programs, we consider a sample example. We let space $S$ be defined by the following variable declarations for some natural number $N \geq 1$:

```
int a[N+1]; int x; int f;
```

Also, we let $R$ be the following specification on $S$:

$$R = \{(s, s')|a[f'] = x \land 1 \leq f' \leq N \land (\forall h : f' < h \leq N : a[h] \neq x)\}.$$ 

This specification describes a search of some element $x$ in an array $a$; note that by virtue of the clause $a[f'] = x$, this specification assumes that $x$ is in $a$; note also that although the array is declared of size $N + 1$, we are only using $N$ cells, namely $a[1]$ through $a[N]$. This specification provides that the final value of $f$ must point to the greatest index where $x$ is located in $a$.

We consider the following program $P$ on space $S$:

```
// non deterministic search, assumes that x is in a
#include <iostream>
```
include <cstdlib>
using namespace std;

const int N=30;
void loaddata();
int a[N+1]; int x; int f; // space S
int main ()
{loaddata(); f=0; // loads a and x
 int k; int z; z=1; k=1;
 while (k<=N)
 {if ((a[k]==x) && (z>0))
  {f=k; z=mysteryfunction(z);}
 k=k+1;}
 cout << f << endl;}

In order for us to analyze what this program does, we need to know how
mysteryfunction() is defined; but suppose that this function is too complex, and we
want to reason about the program without having to figure out what it does. Then the most
we can say is that this program returns in $f'$ an index where $x$ occurs in $a$, but we are not
sure which index. Hence we write $P$ as follows:

$$P = \{(s,s')|a[f'] = x \land 1 \leq f' \leq N \land a' = a \land x' = x\}.$$ 

This is clearly non-deterministic because it does not fully specify $f'$, the final value of $f$.
As modeled, this program is not correct with respect to $R$, since it does not necessarily
return the largest index where $x$ is located in $a$; we are interested to generate a more-correct
program.

Now we consider the following program $P'$:

// non deterministic search
include <iostream>
#include <cstdlib>
using namespace std;

const int N=30;
void loaddata();

int a[N+1]; int x; int f;
int main ()
{loaddata(); f=0; // loads a and x
 int k; int z; z=1; k=1;
 while (a[k]!=x) {k=k+1;}
 while (k<=N)
 {if ((a[k]==x) && (z>0))
  {f=k; z=mysteryfunction(z);}
 k=k+1;}
 cout << f << endl;}

The first loop of this program assigns $f$ the first index where $x$ occurs in $a$; if there are
no other occurrences of $x$ in $a$, then the second loop does not change $f$ and the program
returns in \( f \) the unique index of \( x \) in \( a \), hence satisfies the specification. If the array contains two instances of \( x \) in \( a \), then the first loop assigns to \( f \) the index of the first location and the second loop assigns to \( f \) the index of the second location, hence in this case the program necessarily satisfies the specification. We now consider the case where \( x \) occurs more than twice in \( a \), and we let \( f_1, f_2, ..., f_h \) the indices of \( x \) in \( a \) (listed in increasing order); according to specification \( R \), \( f_h \) is the only correct output. Yet program \( P \) may return \( f_1, f_2, ..., f_{h-1} \) (which violate \( R \)) in \( f' \), whereas program \( P' \) may return \( f_2, ..., f_{h-1} \) (which violate \( R \)) in \( f' \). Because the set of incorrect outputs returned by \( P' \) is a subset of that returned by \( P \), we conclude that \( P' \) is more-correct than \( P \) with respect to \( R \)—even though both are incorrect. We are able to make this claim without having to analyze what function \texttt{mysteryfunction()} does.

Let us now consider the following program \( P'' \) on the same space \( S \):

```c
// non deterministic search
#include <iostream>
#include <cstdlib>
using namespace std;

const int N=30;
void loaddata();
int a[N+1]; int x; int f;
int main ()
{loaddata(); f=0; // loads a and x
 int k; int z; z=1; k=1;
 while (k<=N)
 {if ((a[k]==x) && (z+5>0))
 {f=k; z=mysteryfunction(z);}
 k=k+1;}
 cout << f << endl;}
```

Note that if it weren't for the condition \((z > 0)\), program \( P \) would be correct with respect to \( R \) since it would return in \( f \) the largest index where \( x \) occurs in \( a \). By replacing the condition \((z > 0)\) by the condition \((z + 5 > 0)\), we make it more likely to return in \( f' \) the largest index of \( x \) in \( a \), but we do not change the set of possible values that may be returned in \( f' \). Hence while \( P'' \) is more reliable than \( P \) with respect to \( R \), it is not more-correct than \( P \) with respect to \( R \); rather, \( P'' \) and \( P \) are equally correct with respect to \( R \). In fact, the (non-deterministic) function of \( P'' \) is the same as that of \( P \), i.e.

\[
P = \{(s, s')|a[s'] = x \land 1 \leq f' \leq N \land a' = a \land x' = x\}.
\]

Whereas relative reliability deals in terms of plausibility, relative correctness deals exclusively in terms of possibility.

### 3.4 Illustration: Empirical Observations

To corroborate our static analysis of the previous section, we implement a non-deterministic version of \texttt{mysteryfunction()} , as shown below.

```c
int mysteryfunction(int z)
 {return -N + (int) (frand() * (9*N+1));}
```
where \( \text{frand()} \) is defined as follows (to return a random real value between 0 inclusive and 1 non-inclusive):

\[
\text{float frand()}
\begin{align*}
\quad & \{ \text{return ((float) rand()) / RAND_MAX;} \\
\quad & \}
\end{align*}
\]

Also, we load array \( a \) and variable \( x \) as shown below:

\[
\text{void loaddata()}
\begin{align*}
\quad & \{ \text{srand(time(0));} \\
\quad & \quad \text{for (int i=0; i<=N-5; i=i+5)} \\
\quad & \quad \quad \{ a[i+1]=5; a[i+2]=8; a[i+3]=3; a[i+4]=9; a[i+5]=12; \} \\
\quad & \quad x=9; \}
\end{align*}
\]

With this implementation, we make the following observations:

- When we compile \( P \) and run its executable (\( a.out \)) repeatedly, we obtain different results, ranging, for \( N = 30 \), over the set \( \{4, 9, 14, 19, 24, 29\} \).

- When we compile \( P' \) and run its executable (\( a.out \)) repeatedly, we obtain different results, ranging, for \( N = 30 \), over the set \( \{9, 14, 19, 24, 29\} \); this confirms our analysis, to the effect that \( P' \) is more-correct than \( P \), since the set of outcomes of \( P' \) that violate specification \( R \) (i.e. \( \{9, 14, 19, 24\} \)) is a subset of the set of outcomes of \( P \) that violate specification \( R \) (i.e. \( \{4, 9, 14, 19, 24\} \)).

- When we compile \( P'' \) and run its executable (\( a.out \)) repeatedly, we obtain different results, ranging, for \( N = 30 \), over the set \( \{4, 9, 14, 19, 24, 29\} \). A wholly casual/unscientific survey seems to suggest that \( P'' \) returns \( f' = 29 \) (the correct output with respect to \( R \)) more often than \( P \) does, suggesting that \( P'' \) is more reliable than \( P \).

These observations are compatible with our static analysis of the programs of interest, and with our understanding of relative correctness of non-deterministic programs.

## 4 Debugging Without Testing

### 4.1 What is a Fault?

Our study of relative correctness came about when we attempted to answer the question: what is a fault in a program? and when can we say that we have removed a fault from a program? The first matter we have to consider is that any definition of a fault must refer to a level of granularity at which we want to isolate faults. The coarsest level of granularity would be to consider the whole program as a possible fault, but that is clearly unhelpful as far as diagnosing and removing faults; more typical levels of granularity include a line of code, a lexeme, an expression, an assignment statement, elementary programming statements, etc. We use the term feature to refer to any part of the source code at an appropriate level of granularity, including non-contiguous parts.

**Definition 4.1:** Due to [29]. Given a specification \( R \) and a program \( P \), a fault in program \( P \) is any feature \( f \) that admits a substitute \( f' \) such that the program \( P' \) obtained from \( P \) by replacing \( f \) with \( f' \) is strictly more-correct than \( P \). A fault removal in \( P \) is a pair of features \((f, f')\) such that \( f \) is a fault in \( P \) and \( P' \) obtained from \( P \) by replacing \( f \) with \( f' \) is strictly more-correct than \( P \).
Using this definition, we can in principle remove a fault from a program without testing it; all we need to do is prove that the new program obtained by replacing $f$ with $f'$ is strictly more-correct than $P$. But in practice it is very difficult to use this definition, because it is difficult to compute the functions of $P$ and $P'$, then the competence domains of $P$ and $P'$, then compare them. One of the main obstacles to the practical application of this definition to fault removal is that faults are local attributes whereas relative correctness is a global program-wide attribute; hence we need means to bridge the gap between them.

4.2 Proving Correctness and Incorrectness of Loops

The analysis of while loops by means of invariant relations provides a way to infer the relative correctness of iterative programs from partial semantic information. In [25] we discuss how to use invariant relations to prove the correctness or incorrectness of an iterative program of the form $w$: while (t) {b;}. If we let $B$ be the function of the loop body ((b;)) and we let $T$ be the vector that represents the loop condition (i.e. $T = \{(s, s') | t(s)\}$) then an invariant relation of the while loop $w$ is a reflexive transitive superset of $(T \cap B)$.

By definition, an invariant relation is a superset of (hence an approximation of) the reflexive transitive closure of $(T \cap B)$, which represents the function of the loop; hence by looking at a sufficient number of (sufficiently small) invariant relations of a loop, we can learn much about the function of the loop, even when we do not compute it explicitly in all its detail.

We are evolving a tool that can generate invariant relations for common combinations of code patterns; this tool matches an internal representation of the loop against pre-stored code patterns (called the recognizers) for which it has patterns of invariant relations; actual invariant relations are generated by instantiating the patterns of invariant relations with actual program variables. The following table shows, for the sake of illustration, two sample recognizers dealing with numeric variables; more information on this tool can be found at https://selab.njit.edu/tools/fxloop.php.

<table>
<thead>
<tr>
<th>ID</th>
<th>Data</th>
<th>Statements</th>
<th>Invariant Relation Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>1R1</td>
<td>int x; const int a&gt;0; x=x+a</td>
<td>${(s, s')</td>
<td>x \mod a = x' \mod a}$</td>
</tr>
<tr>
<td>2R1</td>
<td>float x, y; const float a, b; x=x+a, y=y+b</td>
<td>$(s, s')</td>
<td>ay - bx = ay' - bx'</td>
</tr>
</tbody>
</table>

Invariant relations are important for our purposes because they enable us to determine whether a loop is correct or not with respect to a specification, as shown in the following two propositions (which are due to [25]). Even though correctness is defined in terms of the program function, invariant relations enable us to rule on correctness or incorrectness long before we have collected all the necessary information to compute the loop function.

**Proposition 2:** Necessary condition of correctness. Let $w$ be a while loop of the form $w$: while (t) {b;} that terminates for all states in $S$, let $R$ be an invariant relation for $w$, and let $C$ be a specification on $S$. If $w$ is correct with respect to $C$ then $(C \cap R)^T = CL$.

This proposition provides, in effect, that any while loop that admits an invariant relation $R$ that fails to meet this condition could not possibly be correct with respect to $C$. We say about such an invariant relation $R$ that it is incompatible with the specification $C$. Any invariant relation that is not incompatible with the specification is said to be compatible.
**Proposition 3:** Sufficient condition of correctness. Given a while loop \( w \) of the form

\[
\text{while (t) \{b\}}
\]

that terminates for all states in its space \( S \), and given a specification \( C \) on \( S \), if an invariant relation \( R \) of \( w \) satisfies the condition

\[
R \cap T \cap CL \cap (C \cup R \cap \hat{T}) = C
\]

then \( w \) is correct with respect to \( C \).

This proposition provides, in effect, that if an invariant relation \( R \) meets this condition, then it contains sufficient information to subsume the specification, and to prove the correctness of the loop with respect to \( C \).

### 4.3 Proving Relative Correctness of Loops

While in the previous section we show how invariant relations can be used to prove the (absolute) correctness or incorrectness of a loop with respect to a specification, in this section we explore how they can be used to prove relative correctness of a loop over another with respect to a given specification. A proposition given in [11] provides an intuitive result to the effect that if we alter a while loop in such a way as to migrate an invariant relation from the incompatible column to the compatible column, while preserving all the compatible invariant relations, we obtain a while loop that is strictly more-correct; in other words, whatever alteration we make to the while loop as described above can be considered as a monotonic fault removal. This proposition can be used to diagnose and remove faults in a while loop:

- **Evidence that a fault exists.** The first step in fault removal is to have evidence that a fault does indeed exist. In our case, the existence of an incompatible invariant relation proves that the program has a fault.

- **Locating the fault.** Among all the incompatible invariant relations, we select that which involves the smallest number of program variables; this enables us to focus our attention on those statements of the program that involve the variables in question.

- **Removing the fault.** We must modify the selected variable(s), while ensuring that all the compatible invariant relations are preserved. The constraint of preserving the compatible invariant relations is used for guidance in deciding how to change the selected variables.

- **Verifying that the fault has been removed.** For each modification that is generated in the previous step, we deploy the invariant relations generator and check whether the incompatible invariant relation identified in the second step has migrated from the incompatible column to the compatible column of invariant relations. Then we are sure that the new loop is strictly more-correct than the original loop (and the fault has been removed).

As an illustration of this approach, we consider the following example, which we borrow from [12]. The space of the specification is defined by the following variable declarations:

- \( \text{char q[]; int let, dig, other, i, l; char c;} \)

The specification \( R \) that we use for relative correctness is:

\[
R = \{(s, s')|q \in \text{list(}A \cup \alpha \cup \varnothing \cup \sigma \text{)} \land let' = let + I_a(q) \land dig' = dig + I_q(q) \land other' = other + I_q(q)\}
\]

where \( \text{list}(T) \) denotes the set of lists of elements of type \( T \), \( I_A, I_a, I_\varnothing \) and \( I_\sigma \) the
functions that to each list \( q \) assign (respectively) the number of upper case alphabetic characters, lower case alphabetic characters, numeric digits and symbols. The (faulty) program that we consider is \( w \):

\[
(i=0; \text{let}=0; \text{dig}=0; \text{other}=0; \text{l}=\text{strlen}(q);
\]

while (i<l) {c = q[i]; i++;
if (A<=c && Z>=c) \text{let}=\text{let}+1;
else if (a<=c && z>=c) \text{let}=\text{let}+1;
else if (0>c && 9>=c) \text{dig}=\text{dig}+1;
else \text{other}=\text{other}+1;})

We find the following invariant relations of this while loop, where \( \sigma_1 \) and \( \sigma_2 \) designate the set of characters whose Ascii codes are less than, and (respectively) greater than or equal to, the code of '0':

- \( V_0 = \{(s, s') | q = q'\} \)
- \( V_1 = \{(s, s') | i \leq i'\} \)
- \( V_2 = \{(s, s') | \text{dig} \leq \text{dig}'\} \)
- \( V_3 = \{(s, s') | \text{other} \leq \text{other}'\} \)
- \( V_4 = \{(s, s') | \text{let} \geq \text{let}'\} \)
- \( V_5 = \{(s, s') | \text{let} - \#_{\sigma_1 \cup A}(q[i..l-1]) = \text{let}' - \#_{\sigma_1 \cup A}(q'[i'..l-1])\} \)
- \( V_6 = \{(s, s') | \text{dig} + \#_{\sigma_1}(q[i..l-1]) = \text{dig}' + \#_{\sigma_1}(q'[i'..l-1])\} \)
- \( V_7 = \{(s, s') | \text{other} + \#_{\sigma_2 \cup \vartheta}(q[i..l-1]) = \text{other}' + \#_{\sigma_2 \cup \vartheta}(q'[i'..l-1])\} \)

The following table shows which of these invariant relations are compatible, and which are incompatible.

<table>
<thead>
<tr>
<th>Compatible Invariant Relations</th>
<th>Incompatible Invariant Relations</th>
</tr>
</thead>
<tbody>
<tr>
<td>( V_0, V_1, V_2, V_3 )</td>
<td>( V_4, V_5, V_6, V_7 )</td>
</tr>
</tbody>
</table>

Because the \( \text{incompatible} \) column is non-empty, we conclude that the program is incorrect with respect to \( R \), hence we must enhance its correctness. To this effect, we select the incompatible invariant relation \( V_4 \) for remediation, which leads us to focus on variable \( \text{let} \) for fault removal. Preservation of the compatible invariant relations mandates that \( \text{let} \) be modified under the following condition: \( \text{let} \leq \text{let}' \). We propose: \( \text{let}=\text{let}+1; \). The generation of invariant relations of the new loop yields the following table:

<table>
<thead>
<tr>
<th>Compatible Invariant Relations</th>
<th>Incompatible Invariant Relations</th>
</tr>
</thead>
<tbody>
<tr>
<td>( V_0, V_1, V_2, V_3, V_4', V_5' )</td>
<td>( V_6, V_7 )</td>
</tr>
</tbody>
</table>

Application of the same process one more time yields the following program:

\[
(i=0; \text{let}=0; \text{dig}=0; \text{other}=0; \text{l}=\text{strlen}(q);
\]

while (i<l) {c = q[i]; i++;
if (A<=c && Z>=c) \text{let}=\text{let}+1;
else if (a<=c && z>=c) \text{let}=\text{let}+1;
else if (0>c && 9>=c) \text{dig}=\text{dig}+1;
else \text{other}=\text{other}+1;})
Analysis of this program produces 8 invariant relations, which are all compatible.

<table>
<thead>
<tr>
<th>Compatible Invariant Relations</th>
<th>Incompatible Invariant Relations</th>
</tr>
</thead>
<tbody>
<tr>
<td>$V_0$, $V_1$, $V_2$, $V_3$, $V_4$, $V_5$, $V_6$, $V_7$</td>
<td></td>
</tr>
</tbody>
</table>

This does not prove that the program is now correct, all it proves is that we have no evidence (in the forms of an incompatible invariant relation) that it is incorrect. To establish correctness, we must ensure that the intersection of all the compatible invariant relations satisfies the sufficient condition provided by Proposition 3, which it does. All the faults have been removed; we now have a correct program.

5 Testing for Relative Correctness

5.1 A Testing Lifecycle

The usual process of software debugging proceeds as follows: We observe a failure of the program; we analyze the failure and formulate a hypothesis on its cause; we modify the source code on the basis of our hypothesis; and finally we test the new program to ensure that it is now correct. But there is a serious flaw in this process: when we remove a fault from an incorrect program, we have no reason to expect the new program to be correct, unless we know (how do we ever?) that the fault we have just removed is the last fault of the program; hence when a fault is removed from a program, the new program ought to be tested for relative correctness over the original program, rather than for absolute correctness. Of course, regression testing is supposed to ensure monotonicity of fault removal, but regression testing is essentially a test data selection matter, whereas the difference between testing for relative correctness and testing for absolute correctness involves other aspects. We argue that testing a program for relative correctness has an impact on three aspects of testing, namely test data selection, test oracle design, and test coverage assessment.

- **Test data selection.** The problem of test data selection can be summarized as follows: We are given a large or infinite test space $S$, and we must select a small subset thereof $T$ such that the behavior of candidate programs on $T$ is a faithful predictor of their behavior on $S$. The difference between absolute correctness and relative correctness is that for absolute correctness with respect to specification $R$, the test space $S$ is $\text{dom}(R)$ whereas for relative correctness over $P$ with respect to $R$ the test space $S$ is $\text{dom}(R \cap P)$.

- **Test oracle design.** Let $\Omega(s, s')$ be the test oracle for absolute correctness derived from specification $R$. Because relative correctness over program $P$ tests a candidate program $P'$ for $\Omega$ only for those states on which $P$ is successful, the oracle for relative correctness $\omega(s, s')$ can be written as:

$$\omega(s, s') \equiv (\Omega(s, P(s)) \Rightarrow \Omega(s, s')).$$

- **Test Coverage Assessment.** It is not sufficient to know that some program $P'$ has executed successfully on a test data set of size $N$ using oracle $\omega(s, s')$; it is also necessary to know what percentage of the test data set satisfy the precondition $\Omega(s, P(s))$ (i.e. is $P'$ more-correct than $P$ because $P'$ is very good or because $P$ is very bad?).
5.2 Program Repair

Research in automated program repair is experiencing a great deal of vitality nowadays, with new or improved tools appearing on a regular basis [28, 31, 4, 1, 21, 33, 5, 36, 6, 22]. Following Mechtaev et. al [28], we distinguish between two broad categories of program repair approaches, which differ primarily by the way they generate candidate patches: whereas search-based approaches, such as GenProg [22, 13], generate patches by scanning a predefined search space, semantic approaches, such as SemFix [33] and Angelix [28], generate patches by solving constraints resulting from a summary symbolic execution of the program. Whereas these methods differ significantly in terms of their patch generation policies, they tend to apply similar methods for patch validation, namely:

- Patch validation is based on the availability of two sets of data: positive test data, which represents correct behavior of the original program, which we want repair candidates to preserve; and negative test data, which represents test data that the original program is not satisfying, and we want repair candidates to satisfy.

- Generated patches are selected as valid repairs if they preserve the positive test data while delivering correct behavior on the negative test data.

- Patch validation can be refined by means of a fitness function, which assigns different weights to different test data, according to their likelihood of occurrence in some usage pattern of the program under repair.

It turns out that both approaches to patch validation approximate relative correctness but do not quite match it.

- When we require that repair candidates preserve positive test data, we are imposing a sufficient condition of relative correct, but an unnecessary condition at that. Indeed, we see from Definition 3.1 and Figure 2 that a program \( P' \) may be more-correct than a program \( P \) without necessarily preserving the correct behavior of \( P \). The condition on \( P' \) for preserving the correct behavior of \( P \) with respect to \( R \) can be written as \( (P' \cap R) \supseteq (P \cap R) \) whereas the condition of relative correctness is \( (P' \cap R)_{L} \supseteq (P \cap R)_{L} \). Clearly, the former is a sufficient, but not a necessary condition of the latter. Using a sufficient but unnecessary condition yields a loss of recall in retrieving valid patches, i.e. we may very well generate patches that are more-correct than the original, but fail to recognize them as such.

- The fitness function that is used by some program repair tools to select repair candidates is an approximation of program reliability, since it assigns weights to successful executions according to their preponderance in some usage pattern. But we find in [9] that relative correctness logically implies enhanced reliability, but is not equivalent to it. In other words, a having a higher fitness function is a necessary condition of relative correctness, but not a sufficient condition. Using a necessary and insufficient condition of relative correctness causes a loss of precision, i.e. we may retrieve repair candidates that have a higher fitness function value than the original but are not in fact more-correct than the original.

Broadly speaking, relative correctness ought to be an integral part of the study of program repair, because it plays for program repair the role that absolute correctness plays for program construction: in the same way that absolute correctness is used to judge the process of...
deriving a program $P$ from a specification $R$, relative correctness ought to be used to judge the process of repairing a program $P$ to obtain a program $P'$ that is more-correct with respect to specification $R$.

5.3 Illustration

In this section, we run a small-scale experiment in which we try to illustrate the use of relative correctness to perform patch validation in the context of program repair. To this effect, we consider the same program as section 4, and we resolve to remove its faults not by static analysis, as we did there, but by testing for relative correctness after each fault removal. To this effect, we proceed iteratively as follows, starting from the original program:

1. Using muJava [26], we generate mutants of the program, and submit each mutant to three tests:
   - A test for absolute correctness, using oracle $\Omega(s, s')$ derived from specification $R$.
   - A test for relative correctness, using oracle $\omega(s, s')$ derived from $\Omega(s, s')$.
   - A test for strict relative correctness, which in addition to relative correctness also ensures that there is at least one state on which the mutant satisfies $\Omega$ whereas the base program fails it.

2. We select those mutants which prove to be strictly more-correct than the base program, make each one of them a base program on which we apply recursively the same procedure, starting from step 1 above.

We invoke muJava with the option of mutating statements and conditions and we test every mutant for relative correctness, strict relative correctness and absolute correctness using randomly generated test data of size 1000. Every invocation of muJava generates exactly 64 mutants, which we label by indices 1 through 64; hence for example $m4.53.8$ is mutant 8 of mutant 53 of mutant 4 of the original program. The outcome of this experiment is shown in Figure 5. The arcs represent relative correctness relationships; at the bottom of this graph is the original program, and at the top is the corrected version of the program. Note that the test for absolute correctness kept coming empty-handed every time except whenever muJava produced the correct program $P'$, i.e. six times. The test for relative correctness returned true for every arc in Figure 5 i.e. 25 times; it enabled us to remove faults one at a time, until we reach a correct program. Note also that many mutations prove to be perfectly commutative; such is the case for 4, 8 and 53. Note further that, if we assume for the sake of argument that our test is exhaustive, then the number of arcs emerging from each program represents the number of faults in that program. For example, program $P$ has four faults even though it is three fault removals away from being correct; we say that $P$ has a fault density of 4 and a fault depth of 3.

6 Programming Without Refinement

For all its interest, program verification is really a dubious bargain: the idea that we write programs in an informal/approximate manner then try to prove their correctness is not very sound. What is more rational is to seek means to write programs that are certified to be correct by construction [14, 32, 15, 34]. Given a specification, we can do so in one of two ways:
Preserve correctness, achieve program quality. Starting from the specification, we transform it progressively to make it more and more program-like, while preserving correctness through refinement. This is the traditional process of program construction by successive refinements.

Preserve program quality, achieve correctness. Starting from the trivial program $abort$, we transform it progressively to make more and more-correct with respect to $R$ until it is correct. In [10], we present this model as an alternative program derivation paradigm, and discuss some of its advantages; we refer to this approach as programming without refinement.

Both processes are iterative, hence can be characterized the way we characterize an iterative program: by their initialization, their invariant assertion, their variant function, and their exit condition, as shown in the table below:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Refinement Based</th>
<th>Based on Relative Correctness</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialization</td>
<td>$a = R$</td>
<td>$a = abort$</td>
</tr>
<tr>
<td>Invariant Assertion</td>
<td>$a$ is correct</td>
<td>$a$ is a program</td>
</tr>
<tr>
<td>Variant Function</td>
<td>$a$ increasingly concrete</td>
<td>$a$ increasingly correct</td>
</tr>
<tr>
<td>Exit test</td>
<td>when $a$ is a program</td>
<td>when $a$ is correct</td>
</tr>
</tbody>
</table>

See Figure 6 (a). One of the main advantages of the proposed paradigm is that it encompasses, not only program derivation from scratch, but also most software evolution activities: many activities of software evolution aim to enhance the relative correctness of a software product with respect to a specification; Figures 6 (b) and (c) show two instances of software evolution, namely (resp.) corrective maintenance and adaptive maintenance. Indeed, corrective maintenance is nothing but an instance of correctness enhancement, as
provided by Definition 4.1. On the other hand, adaptive maintenance simply means that we take a program that was originally written to satisfy some specification $R$, and we adapt it to specification $R'$ by making it more-correct with respect to $R'$. Note that we do not start with a correct program with respect to $R$, since we may start adapting the program to $R'$ before it is known to be correct with respect to $R$. Given that most software is developed nowadays by evolution of existing products rather than by derivation from scratch, any advance in the technology of correctness enhancement stands to yield significant returns in practice.

In [10], we illustrate the process of program derivation by correctness enhancements on a program to compute the Fermat decomposition of a natural number. We illustrate this process on a program that computes the number of alphabetic characters, numeric characters, and special symbols in an input string. We consider the same space and relation as those given in Section 4, and we present a sequence of programs $P_0 \ldots P_4$ that are increasingly more-correct, though not increasingly more-refined. We do not have a methodology for how to do this in general, hence we merely present the programs and their competence domains and briefly comment on why they form a sequence of increasingly more-correct programs.

$P_0 \{\text{abort}\}. CD_0 = \emptyset$. This program fails for any initial state.

$P_1 \{i=0; \text{let}=0; \text{dig}=0; \text{other}=0; l=\text{strlen}(q)\;\text{while (i<l) }\{c = q[i]; i++; \text{if } (\text{'A}'<=c \&\& \text{'}Z'>=c) \text{let}+=1;\}\}$. This program runs correctly for any initial state in which $q$ is made up exclusively of upper case alphabetic characters.

$P_2 \{i=0; \text{let}=0; \text{dig}=0; \text{other}=0; l=\text{strlen}(q)\;\text{while (i<l) }\{c = q[i]; i++; \text{if } (\text{'A'}<=c \&\& \text{'}Z'>=c) \text{let}+=1; \text{else if } (\text{'}a'<=c \&\& \text{'}z'>=c) \text{let}+=1;\}\}$. This program runs correctly for any initial state in which $q$ is made up exclusively of (upper case and lower case) alphabetic characters.

$P_3 \{i=0; \text{let}=0; \text{dig}=0; \text{other}=0; l=\text{strlen}(q)\;\text{while (i<l) }\{c = q[i]; i++; \text{if } (\text{'}A'<=c \&\& \text{'}Z'>=c) \text{let}+=1; \text{else if } (\text{'}a'<=c \&\& \text{'}z'>=c) \text{let}+=1;\}$
Diallo et. al

else if (\'0\'<=c && \'9\'>=c) dig+=1;}

\[ CD_3 = \{ s|q \in \text{list}(\alpha_\text{A} \cup \alpha_\text{a} \cup \nu) \} \]. This program runs correctly for any initial state in which \( q \) is made up of alphanumeric characters.

\[ P_4 \{i=0; \text{let}=0; \text{dig}=0; \text{other}=0; \text{l}=\text{strlen}(q); \]
while (i<l) {c = q[i]; i++;
if (\'A\'<=c && \'Z\'>=c) let+=1;
else if (\'a\'<=c && \'z\'>=c) let+=1;
else if (\'0\'<=c && \'9\'>=c) dig+=1;
else other+=1;}

\[ CD_4 = \{ s|q \in \text{list}(\alpha_\text{A} \cup \alpha_\text{a} \cup \nu \cup \sigma) \} \]. This program runs correctly for any initial state, regardless of what string of characters \( q \) contains.

Clearly, we do have \( P_0 \sqsubseteq_R P_1 \sqsubseteq_R P_2 \sqsubseteq_R P_3 \sqsubseteq_R P_4 \); also, we find \( CD_4 = \text{dom}(R) \), hence (by Proposition 1) \( P_4 \sqsubseteq R \), i.e. \( P_4 \) is correct with respect to \( R \).

This process bears a striking resemblance to test driven derivation (TDD) [16, 35], in the following sense: if we let \( Q_1, Q_2, Q_3, ... Q_n \) be the successive test data samples (in the form of (input, output) pairs) used in successive iterations of TDD, and if we let \( R_i \) be defined as \( R_i = \bigcup_{k=1}^i Q_i \), then we argue that TDD is nothing but an instance of programming without refinement, where the specification is \( R = R_n \) and the successive competence domains are \( CD_i = \text{dom}(R_i) \). Clearly, the competence domains are increasingly large, by construction of \( R_i \), hence we do have \( P_1 \sqsubseteq_R P_2 \sqsubseteq_R P_3 \sqsubseteq_R ... \sqsubseteq_R P_{n-1} \sqsubseteq_R P_n \), where \( P_i \) is the program obtained after considering the \( i^{th} \) test data sample \( Q_i \).

7 Concluding Remarks

In [19, 18, 20, 2] Laprie et al. discuss various aspects of dependability, and argue that faults are at the center of the study of dependability; in particular, most dependability methods can be characterized by the way they deal with faults, including fault avoidance (program proving, program construction), fault removal (program testing) and fault tolerance, and yet, they provide a rather vague definition of a fault: A fault is the adjudged or hypothesized cause of an error [2]. In [29, 9, 7] we attempt to give a formal definition of a fault, and find that a way (the only way?) to do so is to introduce the concept of relative correctness. Beyond defining faults, this concept has many implications, of which we discuss three in this paper, as they pertain to program proving and program testing.

- **Impact on Proving.** Whereas traditionally proving methods are deployed exclusively on correct programs to prove their correctness, we argue that relative correctness enables us to apply proving methods to an incorrect program to prove that despite being incorrect, it is still more correct than another. Given that there are orders of magnitude more incorrect programs than correct programs, this points to a potential expansion of the scope of static analysis methods.

- **Impact on Testing.** We argue that when we remove a fault from a program we ought to test it for relative correctness rather than absolute correctness; of course regression testing is an attempt to test for relative correctness through the selection of targeted test data, but we argue that testing for relative correctness has an impact not only on test data selection, but also on oracle design and test coverage assessment.
• **Impact on Program Derivation.** Whereas the traditional paradigm of program derivation is to proceed by successive correctness-preserving transformations on the basis of refinement, we argue that it is also possible to proceed by successive correctness-enhancing transformations on the basis of relative correctness. One of the main advantages of our paradigm is that it models not only the derivation of programs from scratch, but also many aspects of software evolution. Given that more software is produced by evolution than from scratch, this approach carries significant potential in practice.

Other authors have introduced similar-sounding but distinct concepts of relative correctness [23, 17, 24] in the context of software testing and program repair. Their work differs from ours in terms of its specification format (executable assertions, vs relations), its program semantics (execution traces, vs. program functions), its definition of correctness (all assertions are true, vs refinement), its definition of relative correctness (more valid traces, fewer invalid traces vs larger competence domain and fewer violations), and its goals (fault removal, vs proving, testing, derivation and evolution).

Whereas most other authors approximate while loops by unrolling them a number of times, we approximate them by means of invariant relations. We view the contrast between unrolling a loop and capturing its behavior by invariant relations as a choice between capturing all the functional details of a few iterations, and capturing some functional detail of all the iterations. We argue that capturing all the functional details is typically unnecessary (not all the loop’s functional properties are worthy / relevant), and modeling a limited number of iterations is typically insufficient (the behavior of the loop for a limited number of iterations may not indicate its behavior for an arbitrary number thereof).

**Acknowledgement**

The authors are very grateful to the anonymous reviewers for their thoughtful, valuable feedback.

**References**


