Software Evolution by Correctness Enhancement *

Wided Ghardallou
Universityy of Tunis El Manar
Tunis, Tunisia
wided.ghardallou@gmail.com

Nafi Diallo
NJIT
Newark, NJ 07102-1982 USA
ncd8@njit.edu

Ali Mili
NJIT
Newark, NJ 07102-1982 USA
mili@njit.edu

ABSTRACT

Relative correctness is the property of a program to be more-correct than another with respect to a specification; this property enables us to rank candidate programs in a partial ordering structure whose maximal elements are the correct programs. Whereas traditionally we think of program derivation as a process of successive correctness-preserving transformations (using refinement) starting from the specification, we argue that it is possible to derive programs by successive correctness-enhancing transformations (using relative correctness) starting from `abort`. One of the attributes of our approach is that it captures in the same mathematical model, not only the derivation of programs from scratch, but also most (if not all) of the activities that arise in software evolution. Given that most software is developed nowadays by evolving existing products rather than from scratch, any advance in the technology of program transformation by correctness enhancement stands to yield significant practical benefits.

Keywords

Program correctness, Relative correctness, Absolute correctness, software maintenance, software evolution, corrective maintenance, adaptive maintenance, software merger, software upgrade.

1. SOFTWARE EVOLUTION

Relative correctness is the property of a program to be more-correct than another with respect to a given specification. Intuitively, program P' is more-correct than program P with respect to specification R if and only if P' obeys R more often (for a larger set of inputs) than P, and violates R less egregiously (in fewer ways) than P. We came across relative correctness in our attempt to define what is a fault in a program, and what is fault removal [2, 13]. We have since explored the impact of relative correctness in software engineering [3], in software testing [5], and in software design [4]. In this paper, we build upon our results of [4] by showing that relative correctness can be used not only for program development from scratch, but also for various forms of program evolution. We argue, in fact, that virtually all software evolution is nothing but an effort to make some program more-correct with respect to some specification. Given that today most software is developed, not from scratch, but rather by evolving existing software products, we feel that exploration of this avenue may yield substantial returns across software engineering practice. Our purpose, in this paper, is not to offer polished/ validated/ scalable solutions; rather, it is merely to highlight some of the opportunities that are opened by relative correctness in the field of software evolution.

All our definitions and results are formulated in terms of (binary) relations, hence we assume the reader familiar with elementary relational concepts, though we will introduce our own notations as we go. In section 2 we introduce a general framework in which we model relevant concepts such as specifications, program functions, program correctness, refinement, etc; and in section 3 we define relative correctness for deterministic and non-deterministic programs, then we review some relevant properties that we need in the remainder of the paper. In sections 4 to 7, we use relative correctness to model several aspects of software evolution, including: the derivation of a correct program from a specification; the derivation of a (not necessarily correct but sufficiently) reliable program from a specification; the merger of two programs; the upgrade of a program with a new feature; the removal of a fault from a program (corrective maintenance); and the transformation of a program to satisfy a new specification (adaptive maintenance).

2. A PROGRAMMING FRAMEWORK

When a program p manipulates variables x and y, say, of type X and Y, we let S be the cartesian product $X \times Y$, and we refer to S as the *space* of p and to an element of S as a *state* of p. We denote the X-component (resp. Y-component) of state s by $x(s)$ (resp. $y(s)$), and we may write simply x (resp. y) if no ambiguity arises; whatever decoration a state has (e.g. s', or s'') is carried over to variable names (hence we write x' for $x(s')$ and x'' for $x(s'')$). Given a program p on space S, we let the *function* of p be the set of pairs of states (s, s') such that if execution of p starts in state s then it terminates in state s'; we denote this function by upper case P and we may, by abuse of notation, refer to a program p and its function P interchangeably.

Given two relations R and R' on set S, we say that R' *refines* R (denoted by $R' \sqsupseteq R$ or $R \sqsubseteq R'$) if and only if $RL \cap R'L \cap (R \cup R') = R$, where L is the universal relation on S (i.e. $L = S \times S$) and the concatenation of two relations represents their product ($RR' = \{(s, s') | \exists s'' : (s, s'') \in R \wedge (s'', s') \in R'\}$). Note that RL can be writ-

*DOI reference number: 10.18293/SEKE2016-095

ten as $\{(s, s')|s \in dom(R)\}$; it represents the domain of R in relational form. A program p is said to be *correct* with respect to a specification (relation) R on S if and only if $P \sqsupseteq R$. This definition is equivalent to traditional definitions of total correctness [7], [8].

The refinement ordering has lattice-like properties, which we briefly present below:

- Two relations admit a least upper bound if and only if they satisfy the following condition, which we call the *consistency condition*: $RL \cap R'L = (R \cap R')L$.
- Whenever two relations R and R' do satisfy the consistency condition, their least upper bound (*join*) is defined by: $R \sqcup R' = \overline{R'L} \cap R \cup \overline{RL} \cap R' \cup (R \cap R')$.
- Two relations have a least upper bound if and only if they have an upper bound.

3. RELATIVE CORRECTNESS

3.1 Definition

Given a specification R and a program P, we refer to the domain of $(R \cap P)$ as the *competence domain* of program P with respect to specification R. When P is deterministic, the competence domain of P with respect to R is the set of initial states for which P behaves according to R; when P is not deterministic (i.e. it may assign more than one final state for a given initial state), the competence domain of P is the set of initial states for which the set of images by P and the set of images by R overlap (i.e. have a non-empty intersection). We have the following definition, due to [2]:

DEFINITION 1. *Given two programs p and p' on space S and a specification R on S, we say that p' is* more-correct *than p with respect to R (written as: $P' \sqsupseteq_R P$ or $P \sqsubseteq_R P'$) if and only if $(R \cap P)L \subseteq (R \cap P')L \wedge (R \cap P)L \cap \overline{R} \cap P' \subseteq P$. We say that p' is* strictly more-correct *than p if one of the inclusions above is strict ($\subset$ vs. $\subseteq$).*

The first clause provides that p' has a larger competence domain than p; the second clause provides that on the competence domain of p, p' violates R in fewer ways than p (since any pair (s, s') such that s is in the competence domain of p and (s, s') is in P' and is not in R, is necessarily in P). In [2], we find that whenever P' is deterministic, the condition of relative correctness can be simplified as follows.

PROPOSITION 1. *Given two programs p and p' on space S such that p' is deterministic, and given a specification R on S, p' is more-correct than p with respect to R if and only if: $(R \cap P)L \subseteq (R \cap P')L$.*

In the remainder of this paper, we restrict our study to deterministic programs. Figure 1 shows an example of two deterministic programs p and p' and a specification R such that p' is more-correct than p; the competence domains of p and p' are highlighted. Two observations are in order regarding this example: Note that p' is more correct than p with respect to R, but they are both incorrect; also note that even though p' is more-correct than p, it does not duplicate the correct behavior of p, rather it has its own correct behavior.

DEFINITION 2. *A* fault *in program p with respect to a specification R is any feature f (lexeme, expression, statement, set of statements, etc) that admits a substitute that would make the program strictly more-correct. A* fault removal *is a pair of features (f, f') such that f appears in p and program p' obtained by replacing f with f' in p is strictly more-correct than p.*

In the same way that refinement provides a formal model for program derivation, relative correctness, as defined herein, provides a formal model for fault removal; see Figure 3 (a). To give the reader some confidence in the soundness of our definition of relative correctness, we present the following properties, due to [13]:

- Relative correctness is reflexive and transitive, but not antisymmetric.
- Relative correctness culminates in (absolute) correctness, i.e. a correct program is more-correct than (or as correct as) any candidate program.
- Relative correctness logically implies (but is not equivalent to) higher reliability; relative correctness and higher reliability are not equivalent because the former is a logical property, whereas the latter is a stochastic property.
- The fourth property is the subject of the next proposition.

PROPOSITION 2. *Given two programs P and P', P' refines P if and only if P' is more-correct than P with respect to any specification R. We write this as:*

$$P' \sqsupseteq P \Leftrightarrow (\forall R : P' \sqsupseteq_R P).$$

3.2 Relative Correctness and Refinement

Refinement is usually viewed as the touchstone of the derivation of provably correct programs; in such a derivation process, each transformation maps an artifact into a more-refined artifact. But Proposition 2 provides that p' refines p if and only if p' is more-correct than p with respect to *any* specification. If we are trying to derive a program that is correct with respect to a given specification R, then R and only R ought to be the focus of our derivation effort. In [4] we argue that it is possible to derive a correct program from a specification by stepwise correctness enhancing transformations using relative correctness, rather than by stepwise correctness preserving transformations using refinement. In this section, we briefly discuss the difference between a correctness-preserving transformation and a correctness-enhancing transformation.

As an illustrative example, we consider a space S defined by two integer variables x and y and we let R be the following specification on S: $R = \{(s, s')|x' = x + y\}$. We consider the following candidate programs:

`p0: {while (y!=0) {y=y-1; x=x+1;}}`,
$P_0 = \{(s, s')|y \geq 0 \wedge x' = x + y \wedge y' = 0\}$.

`p1: {x=x+y;}`. $P_1 = \{(s, s')|x' = x + y \wedge y' = y\}$.

`p2: {x=x+y; y=0;}`, $P_2 = \{(s, s')|x' = x + y \wedge y' = 0\}$.

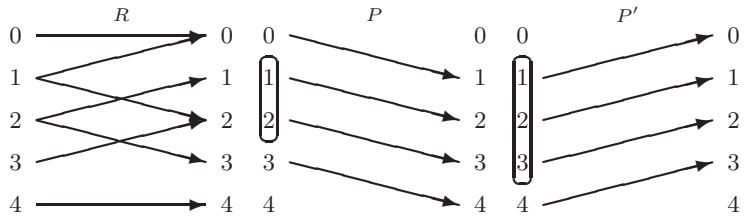


Figure 1: Enhancing correctness without duplicating behavior: $P' \sqsupseteq_R P$

The reader can check (by referring to the definition of refinement given in section 2 and by Proposition 1) that `p1` is more-correct than `p0` but does not refine it, whereas `p2` refines `p0` hence (according to Proposition 2) is more-correct than `p0`. We have a simple explanation for this observation: if we consider the functional attributes of `p0`, we can distinguish between two sources of information:

- Functional attributes that are mandated by the specification, such as the clause $\{x' = x + y\}$ in P_0.
- Functional attributes that are not mandated by the specification, but stem from design decisions, such as the clause $\{y' = 0\}$ in P_0.

In order for a program to refine P_0, it has to preserve all the functional attributes of P_0, regardless of their origin (specification or design); this is the case for `p2`. But in order for a program to be more-correct than P_0, it suffices for it to preserve the functional attributes that are mandated by R; it may, in the process, override / alter functional attributes that stem from design decisions; this is the case for program `p1` which overrides $\{y' = 0\}$ with $\{y' = y\}$. Not surprisingly, `p1` is simpler than `p2`, because it is subject to a less stringent condition.

4. PROGRAM DERIVATION

4.1 Correct Programs

To derive a correct program for a specification R, we have to find an artifact that has two properties: it is a (executable) program; and it is correct with respect to R. To do this in a stepwise manner, we can either preserve correctness until we achieve executability (this is the traditional refinement-based process), or we can maintain executability until we achieve correctness (as we advocate in [4]). These two processes being iterative, we can characterize them by their initial state, their invariant assertion, their variant function, and their exit condition, as shown in Figure 2. See also Figure 3 (b).

4.2 Reliable Programs

In many applications, correctness is unnecessary, and the cost of achieving correctness (as opposed to sufficient reliability) may be unjustified by the stakes attached to (sufficiently infrequent) program failure. In such cases, it may be sufficient to produce a reliable program, rather than a correct program. Thankfully, the derivation model presented herein encompasses the derivation of reliable programs as well as the derivation of correct programs: as we remember from section 3, relative correctness logically implies higher reliability; hence the successive programs generated by this process are increasingly reliable; so that the derivation of a reliable program proceeds in the same way as the derivation of a correct program, except that it terminates as soon as the reliability reaches or exceeds the required threshold. Given that correctness is the culmination of reliability, it is only fitting that the derivation of a correct program be the culmination of the derivation of a reliable program. See Figures 3 (b) and (c).

Attribute	Refinement Based	Based on Relative Correctness
Initialization	$a = R$	$a =$ `abort`
Invariant Assertion	a is correct	a is a program
Variant Function	a increasingly concrete (program-like)	a increasingly correct
Exit test	when a is a program	when a is correct

Figure 2: Iterative Paradigms of Programming

5. PROGRAM MERGER

We consider a specification R and two candidate programs P_1 and P_2 (i.e. programs that are written to satisfy R –they may or may not satisfy it in fact), each of which fulfills the requirements of R to some limited extent, but not necessarily to the full extent. We are interested to merge programs P_1 and P_2 into a program that fulfills the requirements of R to the extent that P_1 fulfills them, and to the extent that P_2 fulfills them. We submit the following definition.

DEFINITION 3. *Given a specification R and two candidate programs P_1 and P_2, a* merger *of P_1 and P_2 with respect to R is any program P' that is more-correct than P_1 and more-correct than P_2 with respect to R.*

We mandate that a merger program be merely more-correct than programs P_1 and P_2, rather than to refine them, for the following reasons:

- *Refinement is Unnecessary.* When we resolve to refine a program, we commit to refine all its functional attributes, those that are mandated by the specification as well as those that stem from design decisions. But we have no reason to preserve design decisions of P_1 and P_2 that do not advance the cause of relative correctness.

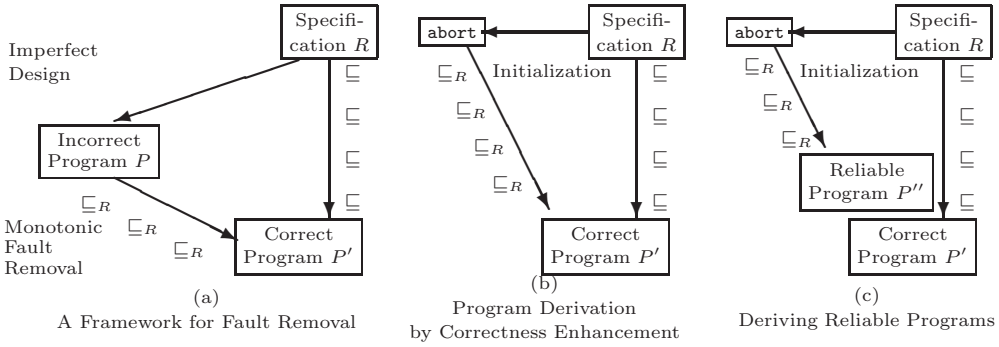


Figure 3: Alternative Program Evolution Paradigms

- *Relative Correctness is Sufficient.* If program P' is more-correct than P_1 and P_2 with respect to specification R, then it delivers all the specification-mandated behavior of P_1 and all the specification-mandated behavior of P_2.

- *Refinement may be Impossible.* Not only is it unnecessary to refine the design-related information of P_1 and P_2, it may actually be impossible: whereas the specification-mandated information of P_1 and P_2 is bounded by R, hence (according to section 2) can be combined by the least upper bound operation, the design-related information of P_1 and P_2 may be incompatible, hence cannot be combined.

We consider the space S defined by three variables x, y and z of type integer, and we let R be the following specification: $R = \{(s, s')|x' = x + y \wedge z' \geq z + 2\}$. Let p_1 and p_2 be the following candidate programs for specification R:

```
p1:  {z=z+2; while (y!=0) {y=y-1; x=x+1;}}
p2:  {z=z+3; while (y!=0) {y=y+1; x=x-1;}}
```

The functions of these programs are, respectively:

$P_1 = \{(s, s')|y \geq 0 \wedge x' = x + y \wedge y' = 0 \wedge z' = z + 2\}$

$P_2 = \{(s, s')|y \leq 0 \wedge x' = x + y \wedge y' = 0 \wedge z' = z + 3\}$.

Indeed, the first program terminates only for initial y greater than or equal to zero, and when it terminates, the final value of x contains $x + y$, the final value of y is zero, and z is incremented by 2. As for the second program, it terminates only for non-positive y, and when it does terminate, the final value of y is zero, z is increased by 3 and x contains $x+y$. So that each program does some of what R asks, but neither is correct. A merger of these two programs is any program P' that is more-correct than P_1 and more-correct than P_2 with respect to R. The systematic derivation of the merger of two programs is beyond the scope of this paper; we content ourselves with presenting a candidate program then showing that it satisfies the definition of a merger. We propose:

```
p':  {z=z+4;
      if (y>0) {while (y!=0) {y=y-1; x=x+1;}}
         else {while (y!=0) {y=y+1; x=x-1;}}}
```

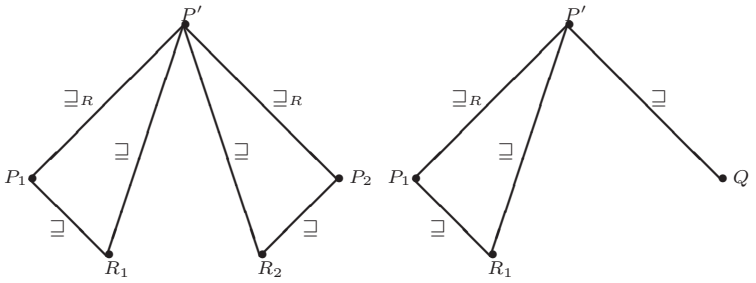
As far as x and y are concerned, this program imitates the behavior of P_1 for non-negative values of y, and the behavior of P_2 for non-positive values of y; as far as z is concerned, this program overrides the behavior of both P_1 and P_2 and increments z by 4. We argue that this program is more-correct than P_1 and more-correct than P_2 with respect to R. The function of this program is:

$P' = \{(s, s')|x' = x + y \wedge y' = 0 \wedge z' = z + 4\}$.

Space restrictions preclude us from showing details, but it is easy to verify that the competence domain of P' ($(R \cap P)L$) is equal to L, hence P' is more-correct than P_1 and P_2. Note that while we found a program that is more-correct than P_1 and P_2, we could not find a program that refines P_1 and P_2. Indeed we can easily check that P_1 and P_2 do not sarisfy the consistency condition, hence they admit no joint refinement. Indeed, no program can simultaneously increase z by 2 (to refine P_1) and by 3 (to refine P_2). This discrepancy between what P_1 does and what P_2 does precludes P_1 and P_2 from having a joint refinement, but does not preclude them from having a program P' that is more-correct than them. The reason is: the statements `{z=z+2}` (in P_1) and `{z=z+3}` (in P_2) are not mandated by the specification (which only requires $\{z' \geq z + 2\}$) but stem instead from arbitrary design decisions; hence both can be overridden by the merger program P'. The difference between refinement and relative correctness is that the former attempts to refine all the behavior of a program, regardless of its source, whereas the latter only refines the behavior that is mandated by the specification. As we see in this simple example, refining all the behavior of P_1 and all the behavior of P_2 is not only unnecessary, it is actually impossible. See Figure 4 (a), where R_1 and R_2 represent the specification-mandated behavior of P_1 and P_2 (we have explicit formulae for these, but their study is beyond the scope of this paper).

6. PROGRAM UPGRADE

We are given a specification R and a candidate program P, and we are interested to augment program P with a new feature that is specified by some relation Q. Typically, P may be a large, complex, comprehensive application that delivers a wide range of services, and Q is a punctual additional function or service that we want to incorporate into P (for example, P is a sprawling corporate data processing application, and Q specifies an additional report to be delivered, or an additional output screen, or an additional statistic on corporate transactions, etc). In transforming P into P', we have every expectation that P' refines Q, because Q is a fairly simple requirement and because it is the main goal of the operation. But we have no expectation that P' refine P, because the implementation of Q may require that some of the behavior of P be altered. Nor do we expect that P' refines R, because in fact we are not even sure P refines R (P is typically incorrect, i.e. it fails to correctly deliver all the required services in all circumstances).



(a) Merger of P_1 and P_2

(b) Upgrading P_1 with Feature Q

Figure 4: Merger and Upgrade

While we do not expect P' to refine P nor R, we most certainly expect P' to be more-correct than P with respect to R; in other words, we do not want that in the process of adding feature Q to P, we degrade the correctness of P with respect to R.

DEFINITION 4. *Given a specification R and a candidate program P, and given a feature Q that we want to add to P, an* upgrade *P' of P with feature Q is any program that refines Q and is more-correct than P with respect to R.*

Given a specification R on space S defined by integer variables x and y, $R = \{(s, s')|x' = x + y\}$ and given the following candidate program,

`p1: {x=x+10; while (y!=10) {y=y-1; x=x+1;}}`

we consider the problem of upgrading program P_1 with feature Q defined by: $Q = \{(s, s')|y > 0 \wedge y' = 0\}$. The function of program `p1` is:

$P_1 = \{(s, s')|y \geq 10 \wedge x' = x + y \wedge y' = 10\}$.

Note that P_1 and Q do not satisfy the consistency condition, since P_1 sets y to 10 while Q mandates that we set it to 0 (for positive values of y). Therefore it is impossible to fulfill requirement Q without altering the behavior of P_1. Fortunately, the feature of P_1 that precludes us from refining Q, namely the clause $y' = 10$, is not a specification-mandated requirement, but stems instead from the specific design of P_1. Hence while it is impossible for the upgrade program P' to refine P, it is not impossible for P' to be more-correct than P with respect to R. We consider the following program:

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

The function of this program is:

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

While space limitation preclude us from showing detailed calculations, it is easy to check that P' does refine Q. On the other hand, we can easily check that the competence domain of P_1 is $\{(s, s')|y \geq 10\}$ whereas the competence domain of P' is $\{(s, s')|y \geq 0\}$. Hence P' is more-correct than P_1 with respect to R. While it is not possible to satisfy Q while preserving all the behavior of P_1, it is possible, and sufficient, to satisfy Q while enhancing the correctness of P_1; this is what P' does. See Figure 4 (b).

7. SOFTWARE MAINTENANCE

7.1 Corrective Maintenance

We argue in this section that corrective maintenance is nothing but an instance of program transformation by relative correctness: in fact it is merely a step in the process we have outlined for program derivation by correctness enhancement; it starts at the current program (rather than `abort`) and it ends a step later (rather than necessarily at a correct program). See Figure 5. As an illustration, we consider the following program on the space S defined by its variable declarations (due to [6], with some modifications):

```
p:  #include <iostream>  ...  ...  ...          // line 1
main {(char q[]); int let, dig, other, i, l; char c;//   2
   i=0; let=0; dig=0; other=0; l=strlen(q);        //   3
   while (i<l) {                                   //   4
      c = q[i];                                    //   5
      if ('A'<=c && 'Z'>c) let+=2;                 //   6
      else                                         //   7
         if ('a'<=c && 'z'>=c) let+=1;             //   8
         else                                      //   9
            if ('0'<=c && '9'>=c) dig+=1;          //  10
            else                                   //  11
               other+=1;                           //  12
      i++;}                                        //  13
```

We define the following sets: $\alpha_A = \{'A' \ldots 'Z'\}$. $\alpha_a = \{'a' \ldots 'z'\}$. $\nu = \{'0' \ldots '9'\}$. $\sigma = \{'+', '-', '=', \ldots '/'\}$, the set of all the ascii symbols. We let $list\langle T\rangle$ denote the set of lists of elements of type T, and we let $\#_A$, $\#_a$, $\#_\nu$ and $\#_\sigma$ be the functions that to each list l assign (respectively) the number of upper case alphabetic characters, lower case alphabetic characters, numeric digits, and symbols. We consider the following specification on S:

$R = \{(s, s')|q \in list\langle\alpha_A \cup \alpha_a \cup \nu \cup \sigma\rangle \wedge$
$let' = \#_a(q) + \#_A(q) \wedge dig' = \#_\nu(q) \wedge other' = \#_\sigma(q)\}$.

The competence domain of P is:

$(R \cap P)L = \{(s, s')|q \in list\langle\alpha_a \cup \nu \cup \sigma\rangle\}$.

This is different from the domain of R, which is

$RL = \{(s, s')|q \in list\langle\alpha_A \cup \alpha_a \cup \nu \cup \sigma\rangle\}$,

hence P is not correct with respect to R. If we let P' be the program obtained from P by changing `{let=+2}` into `{let=+1}`, we find:

$(R \cap P')L = \{(s, s')|q \in list\langle(\alpha_A \setminus \{'Z'\}) \cup \alpha_a \cup \nu \cup \sigma\rangle\}$.

Clearly, $(R \cap P')L \supset (R \cap P)L$. Hence statement `{let+=2}` is a fault in P with respect to specification R and the substitution of `{let+=2}` by `{let+=1}` is a fault removal in P with respect to R.

7.2 Adaptive Maintenance

Adaptive maintenance consists in taking a program P which was originally developed to satisfy some specification

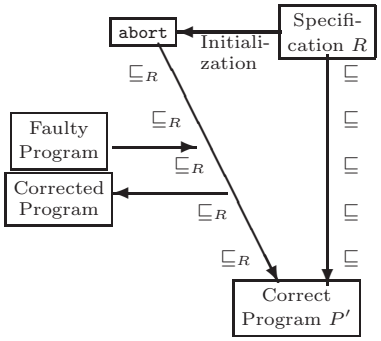


Figure 5: Corrective Maintenance

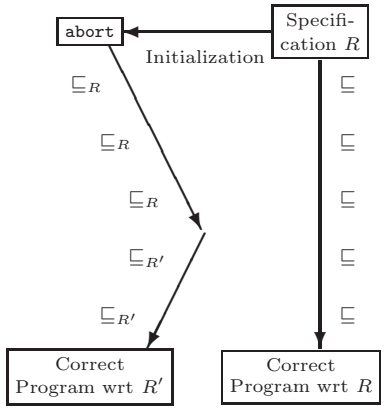


Figure 6: Adaptive Maintenance

R and changing it to make it satisfy some new specification R'. We view this as simply trying to make P more-correct with respect to R' than it is in its current form. Clearly, one does this if one believes that P is close enough to satisfy R' that it is more economical to evolve P than to start from `abort`. Be that as it may, we argue that adaptive maintenance is again a process of making a program more-correct with respect to a given specification. See Figure 6.

8. CONCLUSION

In this paper, we consider the concept of relative correctness (due to [13]), and show that it pervades software evolution, and is potentially more flexible, without being less effective, than refinement-based program transformations. In particular, we find that this concept can provide a formal model for a wide range of software evolution activities, including software design, corrective maintenance, adaptive maintenance, software upgrade, program merger, etc. As a consequence, we argue that by evolving a technology of program transformation with relative correctness, we stand to enhance a wide range of software engineering activities.

Other authors have introduced similar-sounding but distinct notions of relative correctness [1, 9–12, 14]. Their work differs from ours in terms of its specification format (executable assertions vs. input/output relations), its program semantics (execution traces vs. program functions), its definition of correctness (successful assertions vs. refinement), its definition of relative correctness (more successful/ less unsuccessful assertions, vs. larger competence domains), and its goals (fault diagnosis/ program repair vs. software evolution).

9. REFERENCES

[1] Borzoo Bonakdarpour, Ali Ebnenasir, and Sandeep S. Kulkarni. Complexity results in revising unity programs. *ACM Transactions on Autonomous and Adaptive Systems*, 4(1), January 2009.

[2] Jules Desharnais, Nafi Diallo, Wided Ghardallou, Marcelo Frias, Ali Jaoua, and Ali Mili. Mathematics for relative correctness. In *Relational and Algebraic Methods in Computer Science, 2015*, pages 191–208, Lisbon, Portugal, September 2015.

[3] Nafi Diallo, Wided Ghardallou, and Ali Mili. Correctness and relative correctness. In *Proceedings, 37th International Conference on Software Engineering*, Firenze, Italy, May 20–22 2015.

[4] Nafi Diallo, Wided Ghardallou, and Ali Mili. Program derivation by correctness enhancements. In *Refinement 2015*, Oslo, Norway, June 2015.

[5] Wided Ghardallou, Nafi Diallo, Ali Mili, and Marcelo Frias. Debugging without testing. In *Proceedings, International Conference on Software Testing*, Chicago, IL, April 2016.

[6] A. Gonzalez-Sanchez, R. Abreu, H-G. Gross, and A.J.C. van Gemund. Prioritizing tests for fault localization through ambiguity group reduction. In *proceedings, Automated Software Engineering*, Lawrence, KS, 2011.

[7] D. Gries. *The Science of programming*. Springer Verlag, 1981.

[8] E.C.R. Hehner. *A Practical Theory of Programming*. Prentice Hall, 1992.

[9] Barbara Jobstmann, Andreas Griesmayer, and Roderick Bloem. Program repair as a game. In *Computer Aided Verification*, number 3576 in LNCS, pages 226–238, 2005.

[10] Shuvendu K. Lahiri, Kenneth L. McMillan, Rahul Sharma, and Chris Hawblitzel. Differential assertion checking. In *Proceedings, ESEC/ SIGSOFT FSE*, pages 345–455, 2013.

[11] Francesco Logozzo and Thomas Ball. Modular and verified automatic program repair. In *Proceedings, OOPSLA*, pages 133–146, 2012.

[12] Francesco Logozzo, Shuvendu Lahiri, Manual Faehndrich, and Sam Blackshear. Verification modulo versions: Towards usable verification. In *Proceedings, PLDI*, page 32, 2014.

[13] Ali Mili, Marcelo Frias, and Ali Jaoua. On faults and faulty programs. In Peter Hoefner, Peter Jipsen, Wolfram Kahl, and Martin Eric Mueller, editors, *Proceedings, RAMICS: 14th International Conference on Relational and Algebraic Methods in Computer Science*, volume 8428 of *Lecture Notes in Computer Science*, pages 191–207, Marienstatt, Germany, April 28–May 1st 2014. Springer.

[14] Hoang Duong Thien Nguyen, DaWei Qi, Abhik Roychoudhury, and Satish Chandra. Semfix: Program repair via semantic analysis. In *Proceedings, ICSE*, pages 772–781, 2013.