Correctness and Relative Correctness

Nafi Diallo*, Wided Ghardallou†, and Ali Mili*
*New Jersey Institute of Technology, Newark NJ 07102, USA, ncd8@njit.edu, mili@njit.edu
†Faculty of Sciences of Tunis, University of Tunis El Manar, Tunisia, wided.ghardallou@gmail.com

***Abstract*—In the process of trying to define what is a software fault, we have found that to formally define software faults we need to introduce the concept of relative correctness, i.e. the property of a program to be more-correct than another with respect to a given specification. A feature of a program is a fault (for a given specification) only because there exists an alternative to it that would make the program more-correct with respect to the specification. In this paper, we explore applications of the concept of relative correctness in program testing, program repair, and program design. Specifically, we argue that in many situations of software testing, fault removal and program repair, testing for relative correctness rather than absolute correctness leads to clearer conclusions and better outcomes. Also, we find that designing programs by stepwise correctness-enhancing transformations rather than by stepwise correctness-preserving refinements leads to simpler programs and is more tolerant of designer mistakes.**

I. What is a Software Fault?

In [1], Laprie et al. go to great lengths to define concepts and terminology pertaining to system dependability. In particular, they define the standard hierarchy of *fault*, *error*, and *failure*: a fault is the adjudged or hypothesized cause of an error; an error is the state of the system that may lead to its subsequent failure; a failure is the event whereby the system fails to meet its specification. In the context of software, it is fairly straightforward to characterize a failure: it is the event when the program produces an output that violates the specification. It is much more difficult to characterize errors, because to do so assumes that we have a clear characterization of what state the program must be in at any stage in its execution; it is even more difficult to characterize faults, because to do so assumes that we can trace every error to a well-defined feature of the program. In fact, the same program failure can be remedied in more than one way, involving more than one location in the program, and possibly involving more than one type of remedy. Hence in practice, neither the number, nor the location, nor the nature of faults can be uniquely defined.

In [9] we introduce the concept of *relative correctness*, an ordering relation that ranks candidate programs by how close they are to being (totally) correct with respect to a given specification, and we use this ordering to define program faults: A fault in a given program with respect to a given specification is a program part (which can be an expression, a statement, a block of statements, or a set of non continguous statements) which admits a substitute that would make the program more-correct with respect to the specification. Other researchers have felt it necessary, as we did, to introduce a concept of relative correctness [8], and to test or prove programs for relative correctness [7], [5], [11]; our approach differs from these by referring to specifications rather than executable assertions, by capturing the semantics of programs with functions rather than execution traces, and by studying relative correctness as an intrinsic property of interest with broad applications, rather than simply a tool for program repair.

II. Correctness and Relative Correctness

We define the space of a program as the set of values that its variables may take, and define a state of a program as an element of its space. We define a specification on space S as a binary relation on S, and we represent the semantics of a program p on space S by the function denoted in upper case P that the program defines from its initial states to its final states. Given a relation R on S, we denote the domain of R by *dom*(R).

Definition 1: Given a specification R on space S, and a program p on space S whose function is denoted by P, we say that p is *partially correct* with respect to R if and only if: $dom(R \cap P) = dom(R) \cap dom(P)$. Also, we say that p is *correct* with respect to R if and only if: $dom(R \cap P) = dom(R)$.

Whenever we want to contrast correctness with partial correctness, we may refer to it as *total correctness*; whenever we want to contrast correctness with *relative correctness* (defined below), we may refer to it as *absolute correctness*.

Definition 2: Given a specification R on space S and two candidate programs p and p' on space S, we say that p' is *more-correct* than p with respect to R if and only if: *dom*$(R \cap P) \subseteq$ *dom*$(R \cap P')$. Also, we say that program p' is *strictly more-correct* than p with respect to R if and only if: *dom*$(R \cap P) \subset$ *dom*$(R \cap P')$.

The expression *dom*$(R \cap P)$ represents the set of initial states on which program p behaves as specification R dictates; we call this the *competence domain* of program p; to be more-correct simply means to have a larger competence domain, but it does not mean that program p' is identical to program p on the competence domain of the latter; see Figure 1. In this example, we find: *dom*$(R \cap P') = \{1,2,3,4,5\}$ and *dom*$(R \cap P) = \{1,2,3,4\}$. Therefore p' is more-correct than p with respect to R; yet, p' does not copy the correct behavior of p on its competence domain.

How do we know that our definition of relative correctness is sound? We have reviewed a number of properties that we would want a definition of relative correctness to have, and found that our definition satisfies them all:

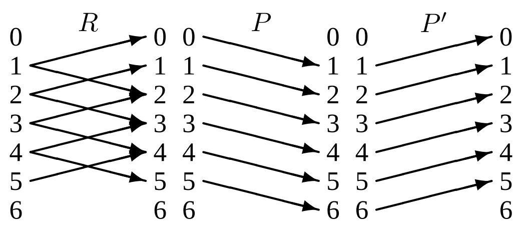

Fig. 1. Enhancing Correctness Without Duplicating Behavior

- *Relative Correctness Culminates in Absolute Correctness.* Indeed, it is very easy to see, from the definitions of correctness and relative correctness that if a program is (absolutely) correct with respect to R, then it is more-correct than any candidate program with respect to R.
- *Relative Correctness Implies Higher Reliability.* The probability of successful execution of the program on a randomly chosen initial state is equal to the integral of the probability distribution over the competence domain of the program; hence the larger the competence domain, the higher the reliability.
- *Relative Correctness as Pointwise Refinement.* We have found in [9] that if and only if a program p' refines a program p, then p' is more-correct than p with respect to any specification R.

III. Applications of Relative Correctness

A. A Model for Monotonic Fault Removal

As programmers, we are all too familiar with situations where we attempt to remove faults from a program, only to find that whenever we correct the behavior of the program for some inputs, it fails for others. So that rather than being a monotonic process of stepwise improvement of the program, the fault removal process is a frustrating exercise of fault recycling, where we remove an obvious fault only to replace it with a more subtle fault. We argue that a program transformation should not be considered a fault removal unless we can establish that it has made the program strictly more-correct. In the same way that program design can be modeled as a sequence of correctness-preserving stepwise refinements from the specification to the program, *program fault removal can be modeled as a sequence of correctness-enhancing transformations from an incorrect program to a correct program.*

B. Mutation Based Program Repair

Automated software repair has achieved great strides in developing tools and techniques to identify and remove faults in software products [6], [10], [12], [3], [13], [2]. We argue that program repair can be improved and made more efficient if instead of testing candidate mutants for correctness, we tested them for relative correctness. Specifically:

- A mutant may run successfully on the test data and still not be more-correct than the original. See Figure 2, where CD and CD' represent (respectively) the competence domain of the original program, and that of the mutant, and T represents the test data set.

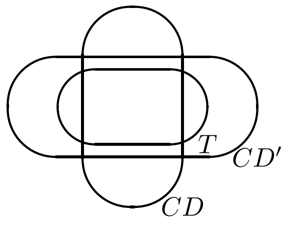

Fig. 2. A Mutant May Succeed on T, yet not be more-correct

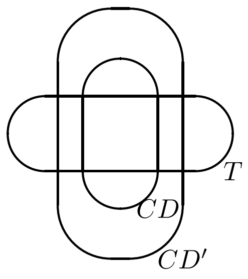

Fig. 3. A Mutant May Fail on T, yet be more-correct

- A mutant may fail on the test data and yet be more-correct than the original. See Figure 3.

The main reason why these techniques select the wrong mutants and reject the wrong mutants (i.e. mutants that should have been selected) is that they are testing mutants for absolute correctness rather than relative correctness.

C. Robust Software Design

Software design by successive refinements generates a program π from a specification R by successive transformations starting with R and ending with π, such that at each step, an intermediate representation p is transformed into a more refined representation p'. We have seen in Section II that p' refines p if and only if p' is more-correct than p with respect to any specification. This raises the question: why would we want to make p' more-correct than p with respect to *all* specifications, when all we care about is specification R? We argue that it is sufficient to mandate that each transformation produces a representation p' that is more-correct than p with respect to R. This requires that we redefine relative correctness to apply to non-deterministic representations, but such a criterion offers some advantages:

- *A Fault Tolerant Process.* In program construction by stepwise refinements, if one step fails to produce an artifact that refines the previous artifact, then the remainder of the design process is doomed. By contrast, in correctness-enhancing transformations, a design fault in one step may be remedied in subsequent steps.
- *Simpler Designs.* We illustrate this premise with an example: Let R be the specification on space S defined by integer variables x and y and relation $R = \{(s, s')|x' =$

$x + y\}$, and let p be an intermediate artifact defined as: {`while (y!=0) {y=y-1; x=x+1;}`}, and let p' and p'' be defined as, respectively: `p': {x=x+y; y=0;}` and `p'': {x=x+y;}`. We find that p'' does not refine p, but it is more-correct than p with respect to R; while p' refines p, hence is more-correct than p with respect to any specification, including R. Imposing refinement rather than relative correctness forces us to generate a more complex artifact (p' rather than p''), as it includes more requirements (relative correctness with respect to all specifications, rather than relative correctness with respect to R alone).

IV. Establishing Relative Correctness

Now that we have defined relative correctness, shown that it satisfies all the desirable properties we associate with this concept, and discussed possible applications thereof, we address the question: How do we prove relative correctness between two programs with respect to a given specification?

A. Testing for Relative Correctness

How does one test a program for relative correctness over another program with respect to a specification, and how is this different from testing a program for absolute correctness? We argue that testing a program for relative correctness has implications for test data generation, as well as for test oracle design:

- *Test Data Generation*. The essence of test data generation is to substitute a large set, say Σ, which is too large to be tested exhaustively, by a smaller representative set, say T, such that if a candidate program runs successfully on set T, we can conclude, with some confidence, that it runs successfully on set Σ. To test program p for absolute correctness with respect to R, set Σ is $dom(R)$, whereas to test program p' for relative correctness over a program p, set Σ is $dom(R \cap P)$.
- *Test Oracle Design*. Let $\omega(s, s')$ be the oracle used to test candidate programs for absolute correctness with respect to R. The oracle that must be used to test a program p' for relative correctness over program p with respect to R is then: $\Omega(s, s') \equiv (\omega(s, P(s)) \Rightarrow \omega(s, s'))$.

B. Proving Relative Correctness

To prove that a program p' is more-correct than a program p with respect to a specification R, we can compute their respective competence domains with respect to R and compare them; of course, computing the competence domain of a program is virtually impossible for all but the simplest programs and specifications, since it requires that we compute the functions of the candidate program in all its minute detail. In the long term, we need to find ways to prove relative correctness without computing the competence domains of the candidate programs (for example, by approximating $dom(R \cap P)$ by an upper bound and $dom(R \cap P')$ by a lower bound).

In the meantime, we have a theorem (due to [4]) that establishes the relative correctness of an iterative program over another using invariant relations. Due to space restrictions, we do not give this theorem here, but discuss how to use it; to this effect, we need to briefly present invariant relations and their use in the analysis of while loops. We consider a while loop w on space S, of the form {`while (t) {b}`}, and we let R be a specification on S. An invariant relation for w is a reflexive transitive superset of $(T \cap B)$, where B is the function of b and T is a representation of t in relational form: $T = \{(s, s') | t(s)\}$. Invariant relations are useful in the analysis of while loops because they enable us to prove claims about the functional properties of a while loop without having to compute its function (W). In particular, we have two invariant relation-based conditions of correctness of a while loop with respect to a relational specification R:

- A sufficient condition of correctness, which holds whenever the given invariant relation subsumes the target specification.
- A necessary condition of correctness, whose negation we use as a sufficient condition of incorrectness: the necessary condition derived from an invariant relation Q is false whenever the invariant relation is incompatible with the specification, i.e. no while loop that admits Q as an invariant relation could possibly be correct with respect to R. When an invariant relation C does satisfy (with R) the necessary condition of correctness, we say that it is *compatible* with R.

The gist of the theorem cited above is that if a while loop w admits an invariant relation Q that is incompatible with a specification R, then the while loop is necessarily incorrect with respect to R (hence in need of repair); then, if the loop is transformed into w' that has the same compatible invariant relations but now has a compatible invariant relation to substitute for Q, then w' is more-correct than w with respect to R.

Using an invariant relations generator, we proceed as follows:

- We generate all the invariant relations we can (depending on how many code patterns we recognize in the loop).
- If the intersection of all the invariant relations satisfies the sufficient condition of correctness, then we conclude that the loop is correct, and no repair is needed.
- If all the generated invariant relations are compatible but their intersection does not subsume the specification, then we conclude that we are unable to determine whether the loop is correct.
- If at least one invariant relation, say Q, is incompatible with R, then we conclude that w is incorrect, hence in need of repair. We consider the variables that are involved in the definition of Q and we resolve that these are the program variables that must be changed in the program. In order to get some guidance on how to change these variables, we use the constraint that these variables must be changed in such a way as to preserve the compatible relations. Even if it does not specify uniquely how we should change the code that alters these variables, this

constraint reduces the range of changes that we can make. For each mutant, we recompute the corresponding invariant relation and check whether it is now compatible; if it is, then the mutant is selected as the proposed repair to the original loop. No testing is necessary.

Note that this procedure does not require that we generate all the compatible invariant relations of the loop, but it does require that we generate all the invariant relations that involve the variables that appear in the definition of Q. Indeed, if an invariant relation C_i does not involve these variables then it is automatically preserved when we alter them, whether we generate it or not. The only practical interest of generating many compatible invariant relations is that they help us define a stronger constraint, hence reduce the number of candidate mutants that we have to consider.

C. Illustration

We consider the following loop, taken from a C++ financial application, where all the variables except `t` are of type `double`, and where a and b are positive constants.

```
w:  while (abs(r-p)>upsilon)
     {t=t+1; n=n+x; m=m-1; l=l*(1+b);
     k=k+1000; y=n+k; w=w+z; z=(1+a)+z;
     v=w+k; r=(v-y)/y; u=(m-n)/n; d=r-u;}
```

We consider the following specification:

$$R = \{(s,s')|b < a < 1 \wedge x' = x \wedge w' = w - z \times \frac{1-(1+a)^{t'-t}}{a}$$

$$\wedge k' = k + 1000 \times (t'-t) \wedge t \leq t' \wedge 0 < l \leq l' \wedge z > 0$$

$$\wedge l \times (1+b)^{-t} = l' \times (1+b)^{-t'}\}.$$

Analysis of this loop by an invariant relations generator derives fourteen invariant relations, five of which are found to be incompatible with the specification. We select the following incompatible invariant relation for remediation:

$$Q = \{(s,s')|l \times (1+b)^{-\frac{z}{1+a}} = l' \times (1+b)^{-\frac{z'}{1+a}}\}.$$

To remediate this incompatibility, we must alter variable z or variable l. We compute the condition on z and l under which a change in these variables does not alter any of the relevant compatible relations, and we find: $z' \geq z \wedge (l = l' \vee l \times (l'-l) > 0)$. We focus our attention on variable z, and apply common mutation operators to the statement {`z = (1+a)+z`} while preserving the condition $z' \geq z$; for each mutant of this statement, we recompute the new invariant relation that substitutes for Q and check whether it is compatible with R. We find that the statement {`z=(1+a)*z`} produces a compatible invariant relation, and conclude that the loop w' obtained when we replace {`z=(1+a)+z`} by {`z=(1+a)*z`} is more-correct than w with respect to R. To illustrate the contrast between absolute correctness and relative correctness, we generate random test data, run the loop on it, and check the oracle for absolute correctness ω derived from specification R; the loop fails at the third test. *But this does not mean our fault removal was wrong: when we run this loop on randomly generated test data using the oracle of relative correctness Ω (section IV-A), it runs for over eight hundred thousand test data without failure*. Indeed, w' is provably more-correct than w, though it may still be incorrect. Running the invariant relations generator on the new loop produces fourteen invariant relations, one of which is incompatible with R (hence w' is indeed incorrect); it seems that by removing the earlier fault we have remedied four invariant relations at once. Applying the same process to w', we find the following loop, which is correct with respect to R:

```
wc: while (abs(r-p)>upsilon)
     {t=t+1; n=n+x; m=m+1; l=l*(1+b);
     k=k+1000; y=n+k; w=w+z; z=(1+a)*z;
     v=w+k; r=(v-y)/y; u=(m-n)/n; d=r-u;}
```

Acknowledgement. The authors are grateful to the anonymous reviewers for their valuable feedback. This publication was made possible by a grant from the Qatar national Research Fund NPRP04-1109-1-174. Its contents are solely the responsibility of the authors and do not necessarily represent the official views of QNRF.

References

[1] Algirdas Avizienis, Jean Claude Laprie, Brian Randell, and Carl E Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004.

[2] Gopinath D., Malik M.Z., and Khurshid S. Specification-based program repair using sat. In *Proceedings, TACAS*, pages 173–188, 2011.

[3] Kim D., Nam J., Song J., and Kim S. Automatic patch generation learned from human-written patches. In *ICSE 2013*, pages 802–811, 2013.

[4] Nafi Diallo, Wided Ghardallou, Marcelo Frias, Ali Jaoua, and Ali Mili. What is a fault? and why does it matter? Technical report, NJIT, Newark, NJ, http://web.njit.edu/~mili/jrn.pdf, 2015.

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

[6] Claire LeGoues, Stephanie Forrest, and Westley Weimer. Current challenges in automatic software repair. *Software Quality Journal*, 21(3):421–443, 2013.

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

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

[9] 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*, Lecture Notes in Computer Science, Marienstatt, Germany, April 28-May 1st 2014. Springer.

[10] Martin Monperrus. A critical review of path generation learned from human written patches: Essay on the problem statement and evaluation of automatic software repair. In *Proceedings, ICSE 2014*, Hyderabad, India, 2014.

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

[12] Debroy V. and Wong W.E. Using mutation to automatically suggest fixes to faulty programs. In *Proceedings, ICST 2010*, pages 65–74, 2010.

[13] Yi Wei, Yu Pei, Carlo A. Furia, Lucas S. Silva, Stefan Buchholz, Bertrand Meyer, and Andreas Zeller. Automated fixing of programs with contracts. In *ISSTA 2010: Proceedings of the 19th international symposium on Software testing and analysis*, pages 61–72, New York, NY, July 2010. ACM.