A Generic Algorithm for Program Repair

Besma Khaireddine (Tunis, Tunisia)
Aleksandr Zakharchenko (Newark, NJ)
Ali Mili (Newark, NJ)

FormaliSE 2017

Buenos Aires, AR May 27, 2017

YING WU COLLEGE OF COMPUTING

NJIT New Jersey Institute of Technology

Ten Years of Program Repair

Engineering getting ahead of theoretical foundations.

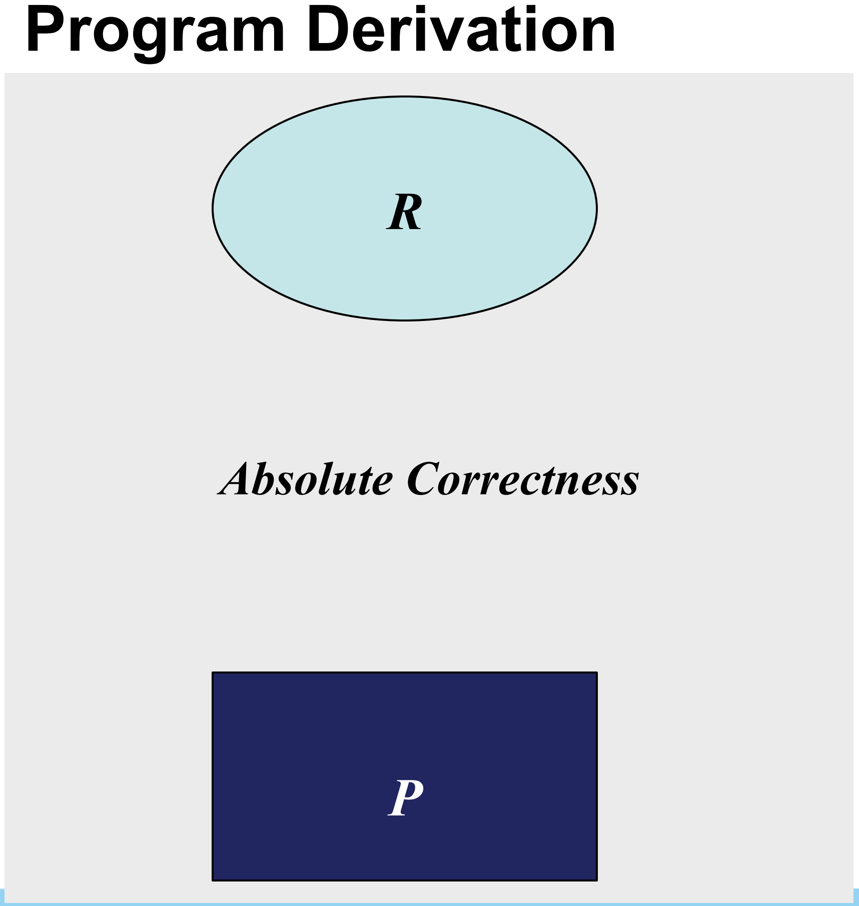
- What foundations are needed for program repair?
  - What does it mean to repair a program?
  - What is a fault? What does it mean to remove a fault?
  - Removing a fault or remedying a failure?
  - How can we tell that the new program is better than the original?
  - How to recognize genuine repairs with optimal precision and recall?

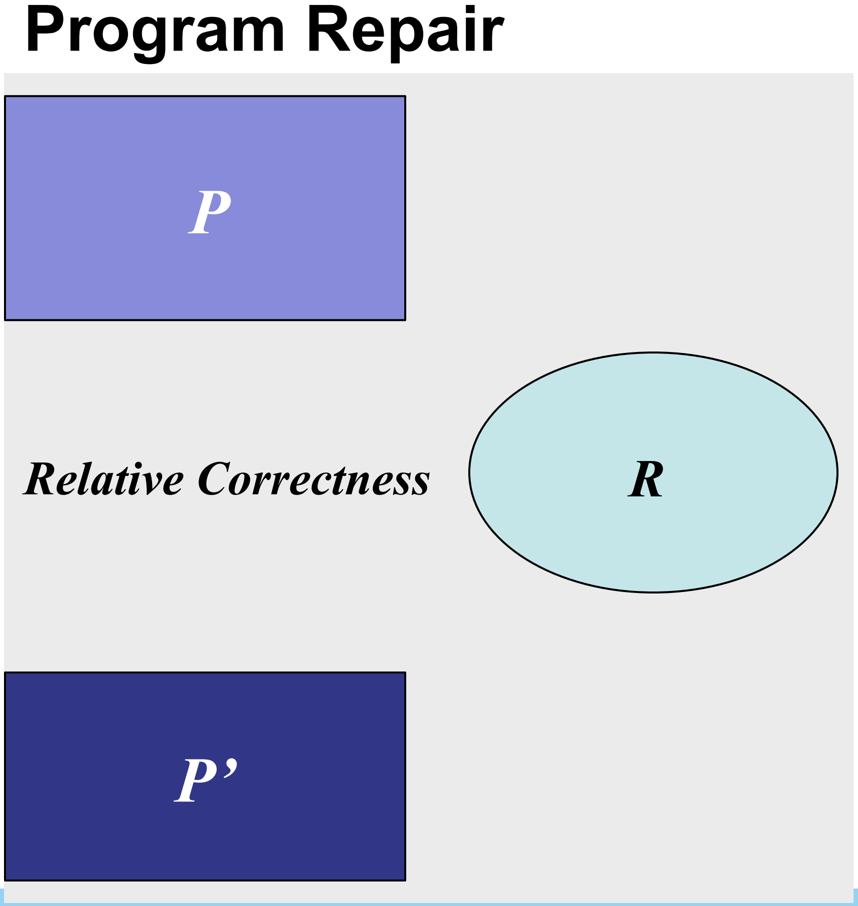
Foundations for Program Repair

Relative correctness: property of a program *P'* to be more-correct than a program *P* with respect to a specification *R*.

- Ought to be an integral part of any discipline of program repair.
  - ***Absolute correctness***: criterion by which we can judge the process of deriving a program *P* from a specification *R*.
  - ***Relative correctness***: criterion by which we can judge the process of deriving a program *P'* as a repair of program *P* with respect to specification *R*.

Foundations for Program Repair





Agenda

Motivation

Absolute and Relative Correctness

Faults and Fault Removals

A Generic Algorithm

Illustration

Conclusion

Motivation

Gains from a theory of program repair, based on relative correctness:

1. Characterizing certifiable fault removal.
   - Strict relative correctness. Re: actually climbing.
2. Distinction: single multi-site vs multiple single-site faults.
   - Importance: counting faults; management of fault removal.
3. Defining Unitary Increment of Correctness Enhancement.
   - Small enough, large enough. Analogy: Mount Everest camps.
4. Insights into Oracle Design.
   - Basis for generic algorithm.

Motivation

5. Distinction: Removing a fault vs Remedying a failure.
   - No one-to-one correspondence between faults and failures.
6. Letting Programs dictate the fault removal schedule.
   - Programs do not expose their faults at once.
   - Remove faults as they appear, failure will be remedied.
7. Distinction: Debugging vs Testing.
   - Debugging without testing; static analysis.
8. Distinction: Fault density vs Fault Depth.
   - Difference between
     - *Program P has N faults*, and
     - *Program P requires N fault removals*.

Motivation

Overall, in the absence of a formal definition of faults, we tend to reason about faults by analogy with bad apples in a bushel of otherwise good apples: When we say that a program has N faults, we assume that

- All the faults are visible/ accessible.
- We can remove them in an arbitrary order.
- We need N fault removals.
- There is only one way to remove each fault.
- Whenever we remove a fault, we have one fewer fault, and one fewer fault removal.

All true for apples, not for faults.

Agenda

Motivation

Absolute and Relative Correctness

Faults and Fault Removals

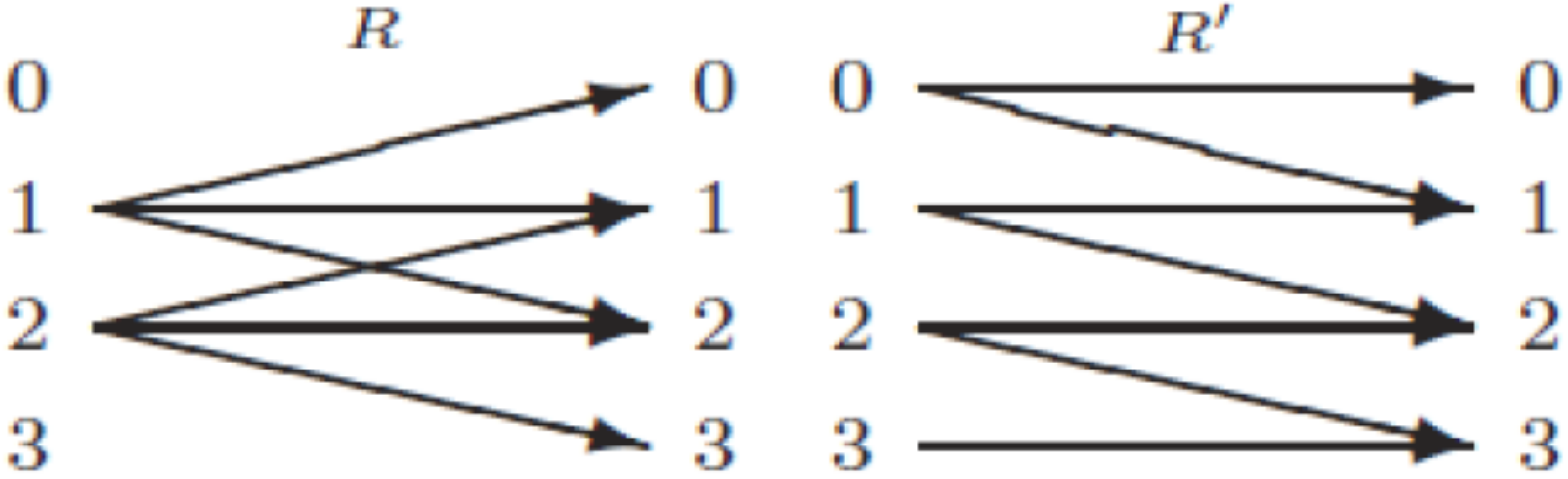
A Generic Algorithm

Illustration

Conclusion

Absolute and Relative Correctness

Refinement: $(R' \sqsupseteq R) \Leftrightarrow (RL \cap R'L \cap (R \cup R')).$



Absolute and Relative Correctness

Absolute correctness, Deterministic Programs:

Specification *R*, Program *P*.

- **Definition**. *P* is said to be correct with respect to *R* if and only if *P* refines *R*.
- **Proposition**. *P* is correct with respect to *R* if and only if $dom(R \cap P) = dom(R)$.

$dom(R \cap P)$: set of initial states for which program *P* satisfies specification *R*; the *competence domain* of *P* with respect to *R*.

Absolute and Relative Correctness

Absolute correctness, Deterministic Programs:

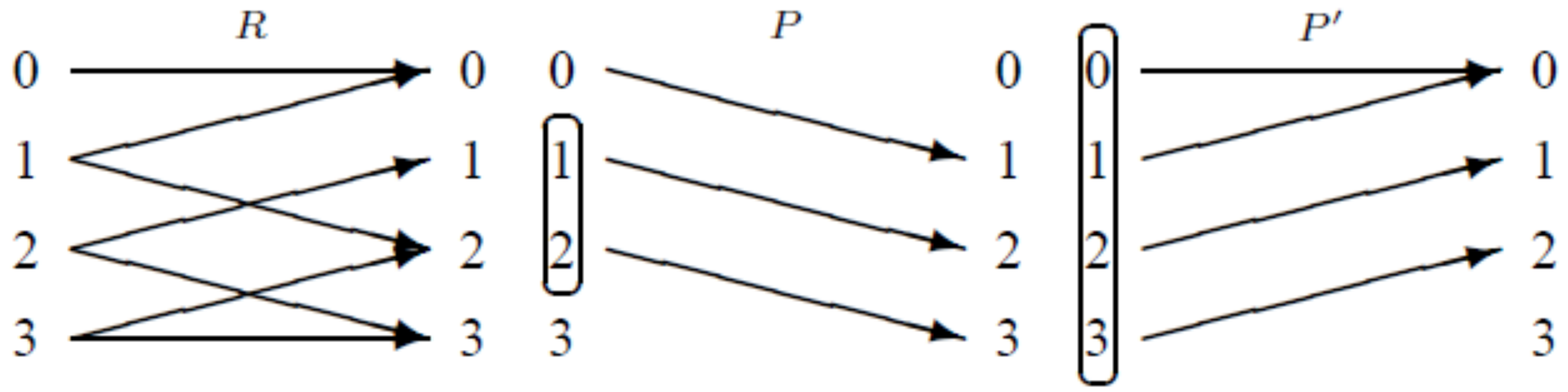


Fig. 1. P is incorrect, P' is correct with respect to R

Absolute and Relative Correctness

Relative Correctness, Deterministic Programs:

- Program *P'* is said to be (strictly) more-correct than program *P* with respect to *R* if and only if the competence domain of *P'* with respect to *R* is a (proper) superset of that of *P*.
  - Whereas absolute correctness distinguishes between two classes of candidate programs: correct and incorrect.
  - Relative correctness ranks candidate programs over a partial ordering whose maximal elements are absolutely correct.

Absolute and Relative Correctness

Relative Correctness, Deterministic Programs:

- *P’* is more-correct than *P*, but does not duplicate correct behavior of *P*.

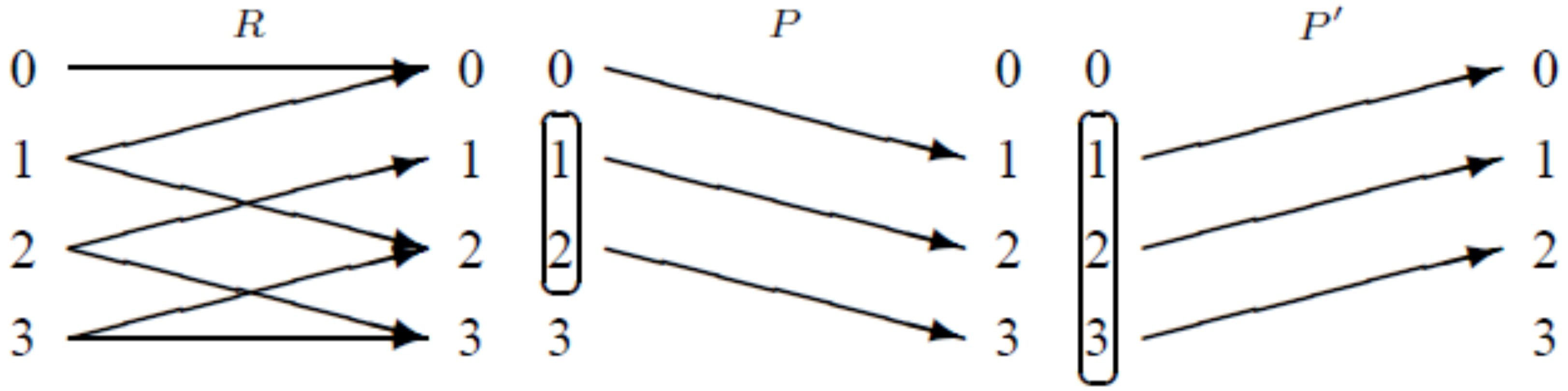


Fig. 2. $P' \sqsupseteq_R P$, Deterministic Programs

Absolute and Relative Correctness

Illustration: $S = \{a, b, c, d, e\}$:
$R = \{(a,a), (a,b), (a,c), (b,b), (b,c), (b,d), (c,c), (c,d), (c,e)\}$

- $P_0 = \{(a,d), (b,a)\}$. $CD_0 = \{\}$.
- $P_1 = \{(a,b), (b,e)\}$. $CD_1 = \{a\}$.
- $P_2 = \{(a,d), (b,c)\}$. $CD_2 = \{b\}$.
- $P_3 = \{(b,e), (c,d)\}$. $CD_3 = \{c\}$.
- $P_4 = \{(a,b), (b,c), (c,a)\}$. $CD_4 = \{a, b\}$.
- $P_5 = \{(a,d), (b,c), (c,d)\}$. $CD_5 = \{b, c\}$.
- $P_6 = \{(a,c), (b,e), (c,d)\}$. $CD_6 = \{a, c\}$.
- $P_7 = \{(a,a), (b,b), (c,c), (d,d)\}$. $CD_7 = \{a, b, c\}$.
- $P_8 = \{(a,b), (b,c), (c,d), (d,e)\}$. $CD_8 = \{a, b, c\}$.
- $P_9 = \{(a,c), (b,d), (c,e), (d,a)\}$. $CD_9 = \{a, b, c\}$.

Absolute and Relative Correctness

Illustration:

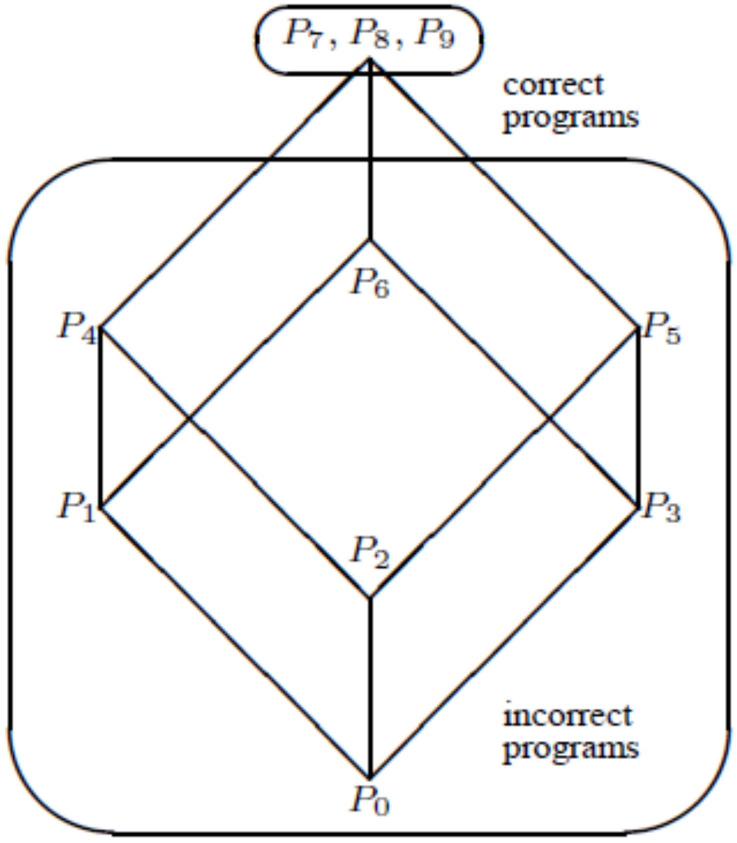


Fig. 3. Ordering Candidate Programs by Relative Correctness

Absolute and Relative Correctness

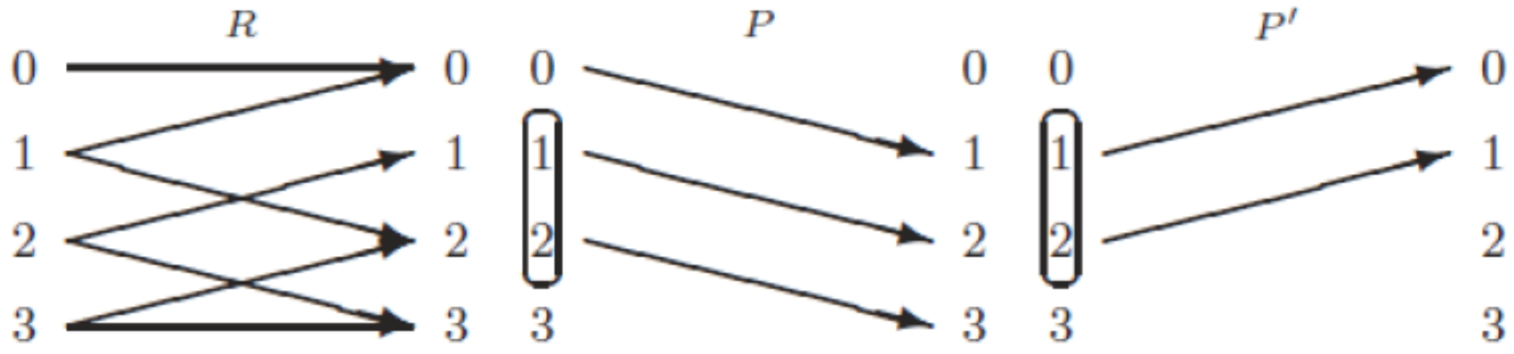
Is Our Definition any Good?

- Reflexive and Transitive, but not antisymmetric.
- Culminates in absolute correctness.
- Logically implies enhanced reliability.
- Pointwise refinement.

Absolute and Relative Correctness

Reflexive and Transitive, but not antisymmetric.

- Equally correct but distinct.



Absolute and Relative Correctness

Culminates in Absolute Correctness:

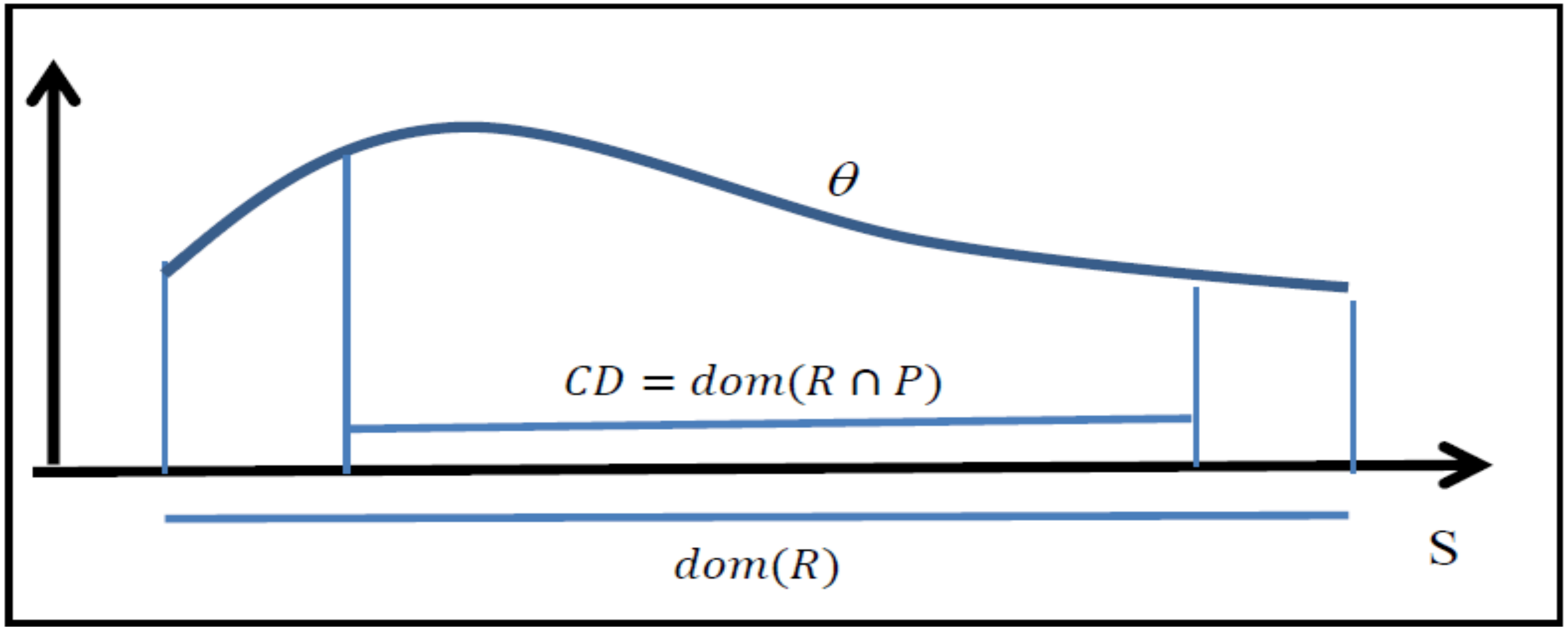
- *P* is correct with respect to *R* if and only if $dom(R \cap P) = dom(R)$.
- By monotonicity of intersection and domain, for any candidate program Q we have

  $$dom(R \cap Q) \boxed{?} dom(R).$$
- Hence *P* is more-correct than *Q*.

Absolute and Relative Correctness

Relative Correctness and Reliability: the reliability of a program is defined in terms of two parameters,

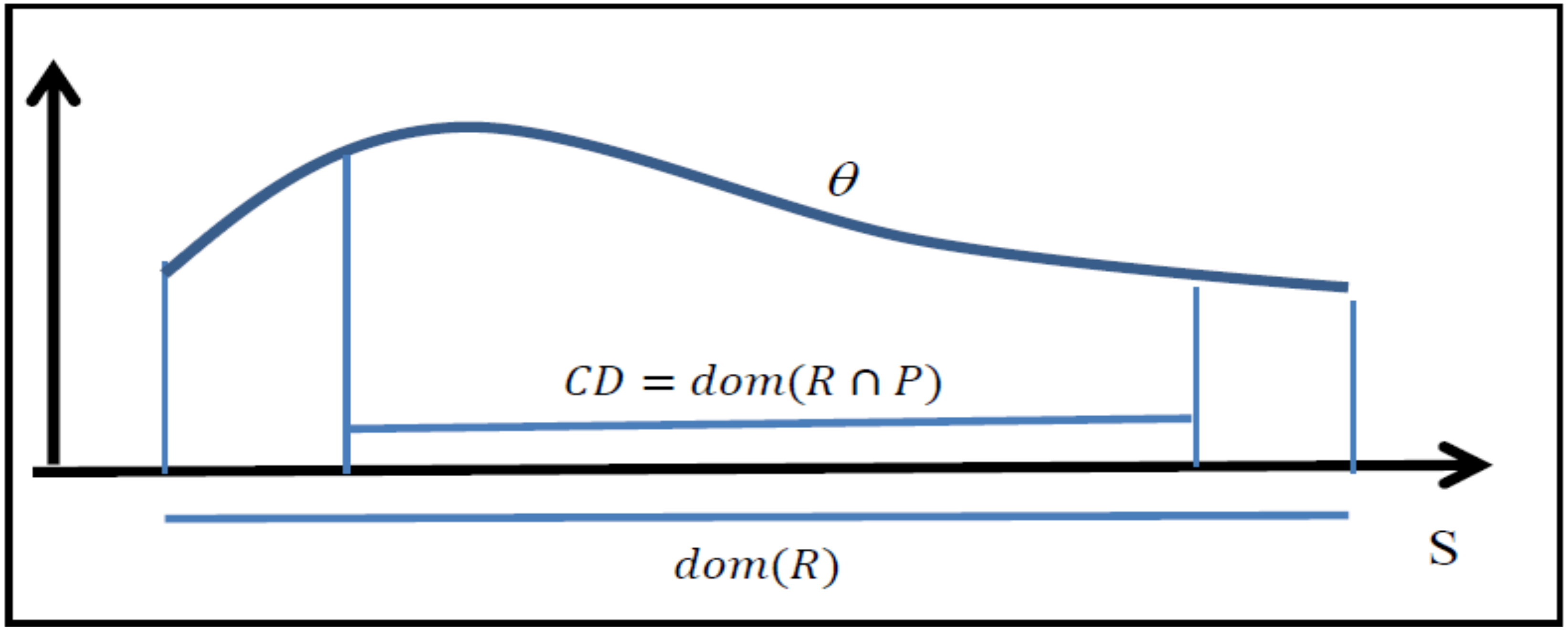
- Specification *R*,
- Probability distribution θ over the domain of *R*.



Absolute and Relative Correctness

Relative Correctness and Reliability:

$$\boxed{(P' \sqsupseteq_R P) \Leftrightarrow (\forall \theta : \rho_R^\theta(P') \geq \rho_R^\theta(P)).}$$



Absolute and Relative Correctness

Relative Correctness and Refinement:

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

- *P'* refines *P*: Whatever *P* does, *P'* can do as well or better.
  - P' more-correct than P with respect to any specification.

Absolute and Relative Correctness

Reliability, Relative Correctness and Refinement:

$$\begin{tikzcd}
 & P' \sqsupseteq P & \\
\sum_{s \in dom(P \cap P')} \theta(s) \geq \sum_{s \in dom(P)} \theta(s) \arrow[ur, "\forall \theta"] & & P' \sqsupseteq_R P \arrow[ul, "\forall R"'] \\
 & \rho_R^\theta(P') \geq \rho_R^\theta(P) \arrow[ul, "\forall R"] \arrow[ur, "\forall \theta"'] &
\end{tikzcd}$$

Fig. 2. Reliability, Relative Correctness, Refinement

Absolute and Relative Correctness

Now that we have vetted our definition of relative correctness,

- We can use it to comment on program repair practice:
  - Using regression for patch validation: Sufficient but not necessary. Leads to loss of recall.
  - Using fitness functions for patch validation: Necessary but not sufficient, as fitness functions are approximations of reliability. Leads to loss of Precision.
- We argue: patch validation by means of strict relative correctness.

Absolute and Relative Correctness

Relative Correctness for Non-Deterministic Programs

- Why: To analyze programs for relative correctness without having to compute their function in detail.

Formula:

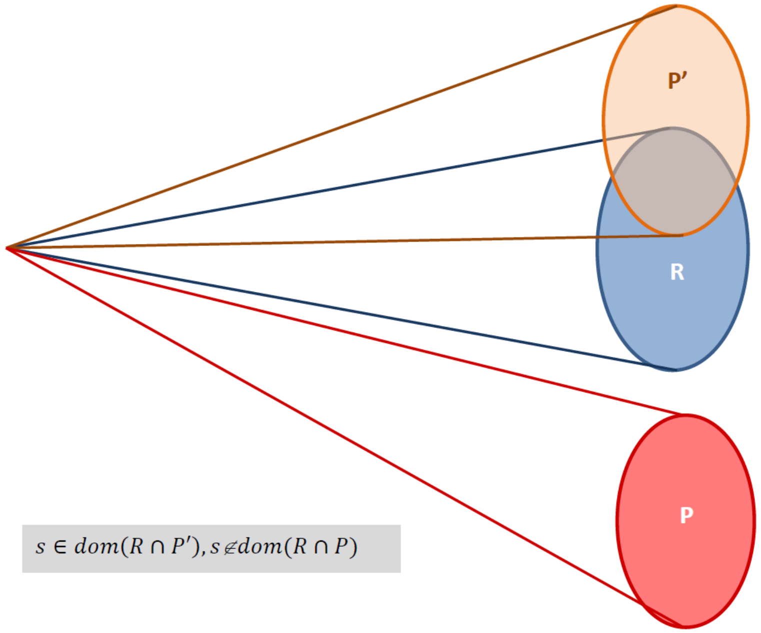
$$\boxed{(P' \sqsupseteq_R P) \Leftrightarrow ((R \cap P)L \subseteq (R \cap P')L \wedge (R \cap P)L \cap \overline{R} \cap P' \subseteq P).}$$

$$\boxed{(P' \sqsupseteq_R P) \Leftrightarrow ((R \cap P')L \cap (R \cup P') \sqsupseteq (R \cap P)L \cap (R \cup P)).}$$

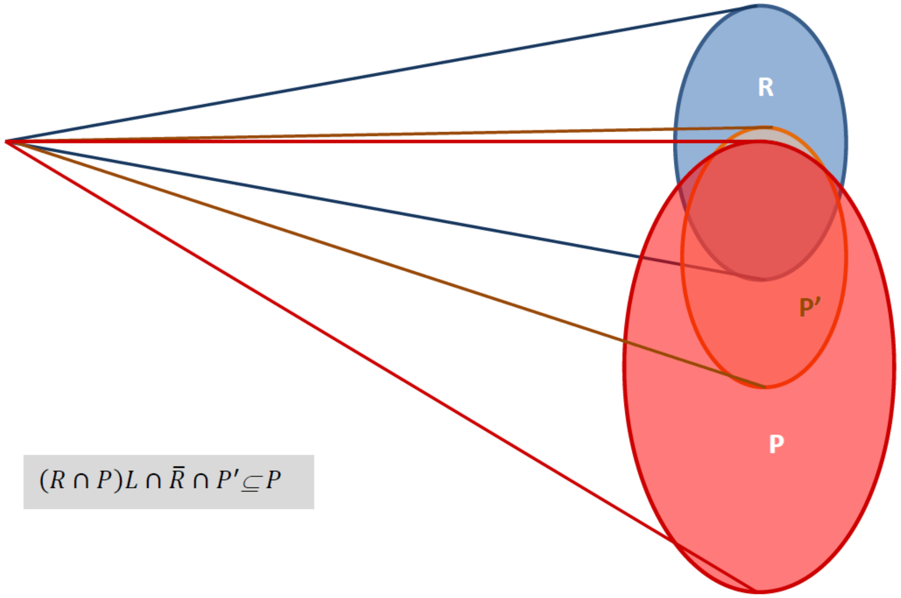
Absolute and Relative Correctness

Interpretation:

- Larger competence domain.
- Fewer outputs that violate *R*.



$s \in dom(R \cap P'), s \notin dom(R \cap P)$



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

Absolute and Relative Correctness

Illustration:

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

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

Absolute and Relative Correctness

Illustration:

```
P:  int main ()
   {f=0; 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;}}
P': int main ()
   {f=0; int k; int z; z=1; k=1;
    while (a[k]!=x) {k=k+1;} f=k; k=k+1;
    while (k<=N)
       {if ((a[k]==x) && (z>0))
           {f=k; z=mysteryfunction(z);}
        k=k+1;}}
P'': int main ()
   {f=0;  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;}}
```

Absolute and Relative Correctness

Illustration:

- *P'*: more-correct than *P*.
- *P''*: more reliable than *P*, not more-correct.

```
P:   int main ()
    {f=0; 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;}}
P': int main ()
    {f=0; int k; int z; z=1; k=1;
     while (a[k]!=x) {k=k+1;} f=k; k=k+1;
     while (k<=N)
         {if ((a[k]==x) && (z>0))
              {f=k; z=mysteryfunction(z);}
          k=k+1;}}
P'': int main ()
    {f=0;  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;}}
```

Agenda

Motivation

Absolute and Relative Correctness

Faults and fault Removals

A Generic Algorithm

Illustration

Conclusion

Faults and Fault Removals

Any definition of a fault must implicitly refer to a level of granularity at which faults are isolated:

- Statement, expression, lexeme.
- Not necessarily contiguous.

Feature:

- Program part at the appropriate level of granularity.

Faults and Fault Removals

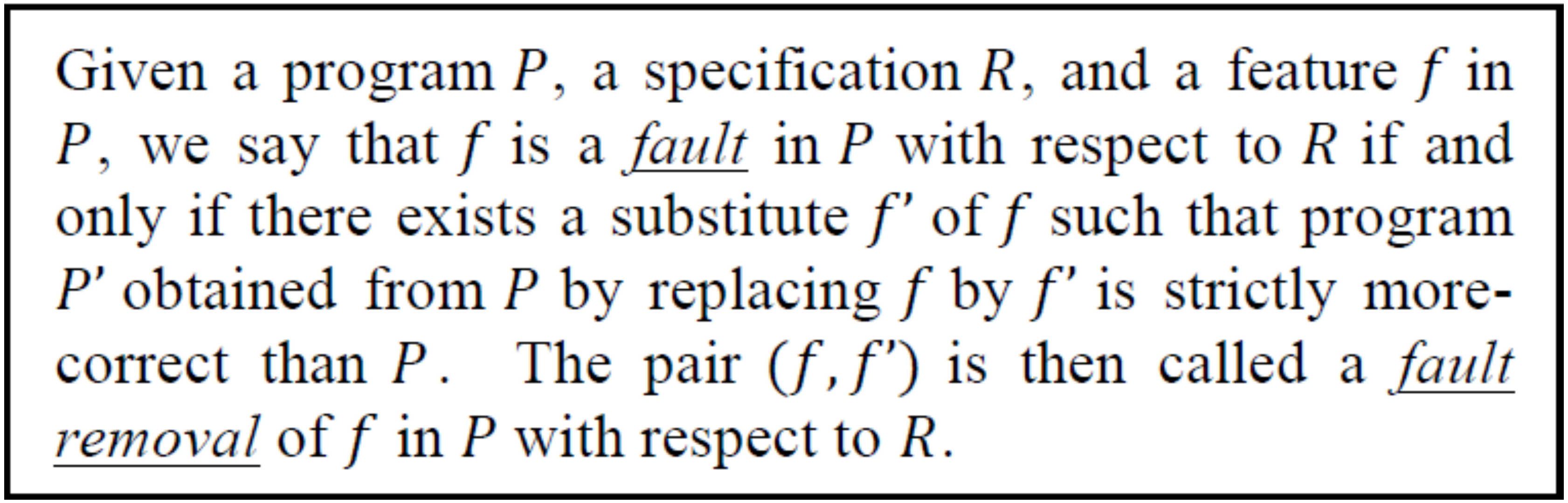
Definition of a fault:

- the (faulty) feature,
- the program,
- the specification.

Faults and Fault Removals

Definition of a fault:

- the (faulty) feature,
- the program,
- the specification.



Given a program P, a specification R, and a feature f in P, we say that f is a <u>*fault*</u> in P with respect to R if and only if there exists a substitute f' of f such that program P' obtained from P by replacing f by f' is strictly more-correct than P. The pair (f, f') is then called a <u>*fault removal*</u> of f in P with respect to R.

Faults and Fault Removals

We consider the following specification/ program:

$$R = \{(s, s') \mid x' = \sum_{i=1}^{N} a[i]\},$$

P: {x=0; k=0; while (k!=N) {x=x+a[k]; k=k+1}}.

We need to change two statements: (k=0) and (k!=N).

- Do we have one two-site fault or two one-site faults?

Faults and Fault Removals

Of course, answer depends on whether one change produces a more-correct program:

- P: {x=0;k=0;while(k!=N){x=x+a[k];k=k+1;}}
- P0: {x=0;**k=1**;while(k!=N){x=x+a[k];k=k+1;}}
- P1: {x=0;k=0;while(k!=**N+1**){x=x+a[k];k=k+1;}}
- P’: {x=0;**k=1**;while(k!=**N+1**){x=x+a[k];k=k+1;}}

Competence Domains:

- $CD=\{s|a[0]=a[N]\}$
- $CD0=\{s|a[0]=0\}$
- $CD1=\{s|a[N]=0\}$
- $CD' =S$.

One two-site fault.

Faults and Fault Removals

$$\begin{tikzcd}
 & \boxed{P'} & \\
\boxed{P0} \arrow[ur, "\sigma 2"] & \boxed{P} \arrow[u, "(\sigma 1, \sigma 2)"] \arrow[l, "\sigma 1"'] \arrow[r, "\sigma 2"] & \boxed{P1} \arrow[ul, "\sigma 1"']
\end{tikzcd}$$

Figure 6: Pattern of a Single Two-Site Fault

Faults and Fault Removals

Same question for:

$$R = \{(s, s') | \ a[0] = a'[0] \wedge \ a'[1..N] = 0\},$$

P: { k=0; while (k!=N) {a[k]=0; k=k+1}}.

We need to change two statements: (k=0) and (k!=N).

- Do we have one two-site fault or two one-site faults?

Faults and Fault Removals

Initialization Example:

- P: {k=0; while(k!=N){a[k]=0;k=k+1;}}
- P0: {**k=1**; while(k!=N){a[k]=0;k=k+1;}}
- P1: {k=0; while(k!=**N+1**){a[k]=0;k=k+1;}}
- P’: {**k=1**; while(k!=**N+1**){a[k]=0;k=k+1;}}

Competence domains

- $CD=\{s|a[0]=0 \boxed{?} a[N]=0\}$
- $CD0=\{s|a[0]=0\}$
- $CD1=\{s|a[N]=0\}$
- $CD1' =S$.

Two one-site faults.

Faults and Fault Removals

$$\begin{tikzcd}
 & \boldsymbol{P'} & \\
\boldsymbol{P0} \arrow[ur, "\sigma 2"] & & \boldsymbol{P1} \arrow[ul, "\sigma 1"'] \\
 & \boldsymbol{P} \arrow[ul, "\sigma 1"] \arrow[ur, "\sigma 2"'] \arrow[uu, "(\sigma 1, \sigma 2)" description] &
\end{tikzcd}$$

Figure 7: Pattern of Two Single-Site Faults

Faults and Fault Removals

Elementary fault (for a given level of granularity):

- A fault such that no part of it is a fault.
  - (k=0,k!=N) is an elementary fault in the sum program, not in the initialization program.
- All single-site faults are elementary faults.

Faults and Fault Removals

Fault Density: Number of elementary faults in a program.

Fault Depth: Minimal number of elementary fault removals that separate program from correctness.

- Faults may hide each other, fault removal affects subsequent fault configuration,
  - Hence depth is a more reliable measure of faultiness.
- Density does not decrease by 1 with each fault removal, $density(P') \boxed{?} density(P) - 1.$
- Depth kind-of-does: $depth(P') \boxed{?} depth(P) - 1.$
  - Equality if *P'* is on the minimal path from *P* to a correct program.
- For a given fault depth,
  - Greater fault density is better. Fault density a quality attribute?

Agenda

Motivation

Absolute and Relative Correctness

Faults and fault Removals

A Generic Algorithm

Illustration

Conclusion

Generic Algorithm

Input	**Program P** **Test Data set T** **Predicate R(s,s')** **Predicate domR(s)**
Output	**Program *Pprime*, more-correct than *P* wrt *T\R*.** **Maybe (if patch generation is good): abs. cor. wrt *T\R*.**

```
Pprime=P;
while (! Abscor(Pprime))
    {Pprime = StrictRelCorrect(Pprime);}
```

Producing an Absolutely Correct Program?

- Yes, with respect to *T\R*: pre-restriction of *R* to *T*.
- Under some conditions: *Pprime* more-correct than *P* with respect to *R*.

Generic Algorithm

- Patch Generation: Immaterial for our purposes.
- Patch Validation:
  - Absolute Correctness:

$$\Omega(s, s') \equiv (s \in dom(R) \Rightarrow (s, s') \in R).$$

  - Relative Correctness:

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

  - Strict Relative Correctness:

$$\sigma_T(P') \equiv (\omega_T(P') \wedge (\exists s \in T : \Omega(s, P'(s)) \wedge \neg\Omega(s, P(s))))$$

Generic Algorithm

- Patch Generation: Immaterial for our purposes.
- Patch Validation:
  - Absolute Correctness:
    - *P'* passes this oracle for all s in *T*: absolutely correct wrt *T\R*.
  - Relative Correctness:
    - *P'* passes this oracle for all s in *T*: more-correct than *P* wrt *T\R*.
  - Strict Relative Correctness:
    - *P'* passes this oracle: strictly more-correct than *P* wrt *T\R*.

Agenda

Motivation

Absolute and Relative Correctness

Faults and fault Removals

A Generic Algorithm

Illustration

Conclusion

Illustration

To illustrate our discussions

- We take the *replace* component of the Siemens Benchmark (563 LOC).
- We enter six modifications to it (provided in the benchmark).
- We take the test data set provided by the benchmark (5542).
- R(): the original program. domR(): true.
  - Non-deterministic specifications: in progress.
- Patch generation: mutant generator,
  - Parameterized to the same nature, scale as modifications.
  - Generates 90 mutants per call.
- Patch validation: oracle infrastructure.
- Experiment: compute *all* the correctness enhancement paths.

Illustration

Experimental Algorithm

- Input: *P, T, R, domR*.
- Output: Graph showing all the paths from *P* to correct programs.

Process:

1. Initial graph = {P}.
2. If all the maximal nodes of the graph are absolutely correct, DONE.
3. Else, for each maximal node that is not absolutely correct,
    1. Generate mutants
    2. Select those that are strictly more-correct, add them to the graph. Goto 2.
4. If all maximal nodes are not abs cor and admit no mutants that are strictly more-correct, then increase multiplicity, Goto 3.2

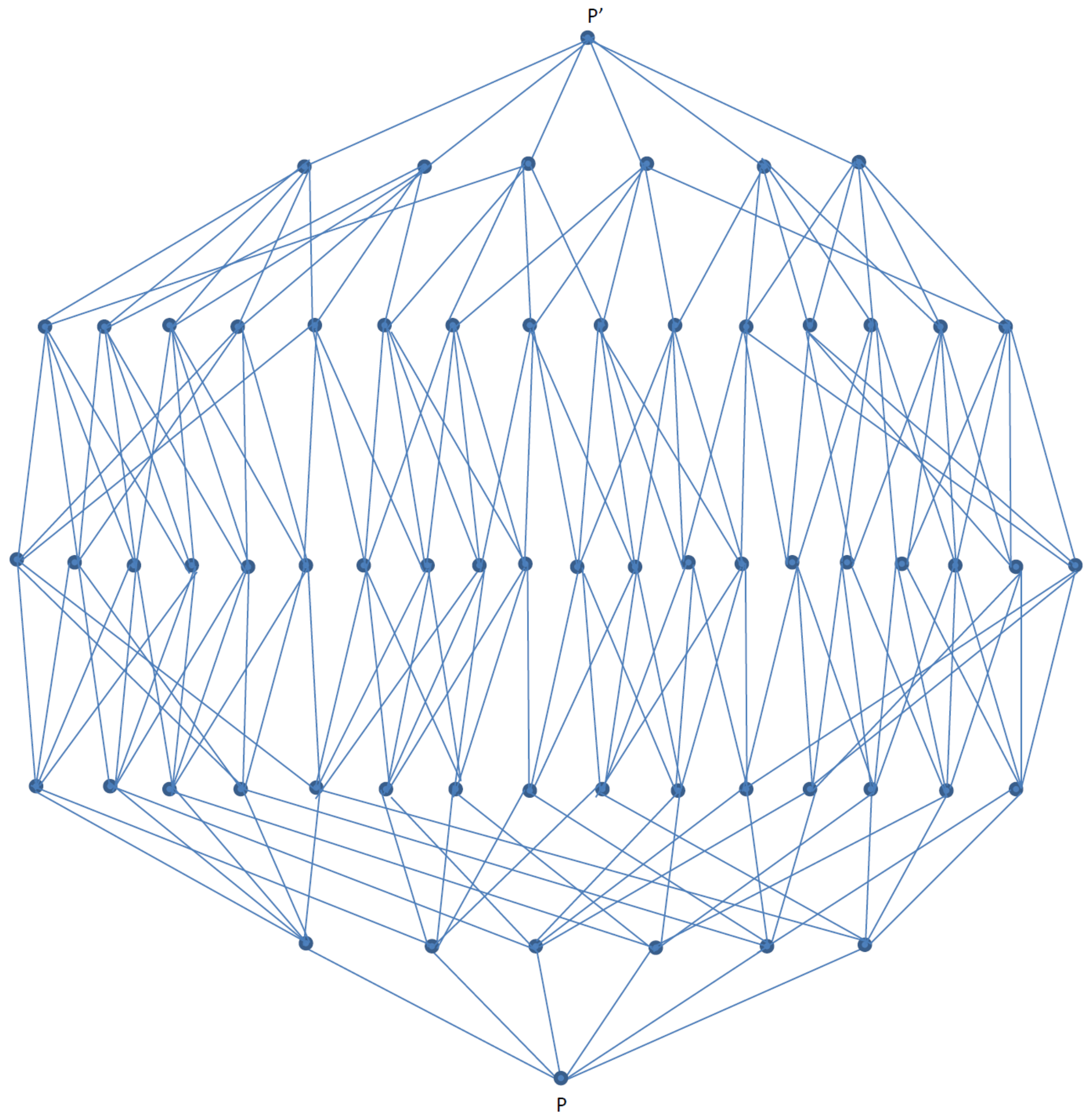
Illustration

If the *bad apple* analogy held,

- P has 6 faults.
- Next layer 5, then 4, then 3, etc..
- density = depth.
- Both decrease by 1 at each layer.

Ready to see reality?

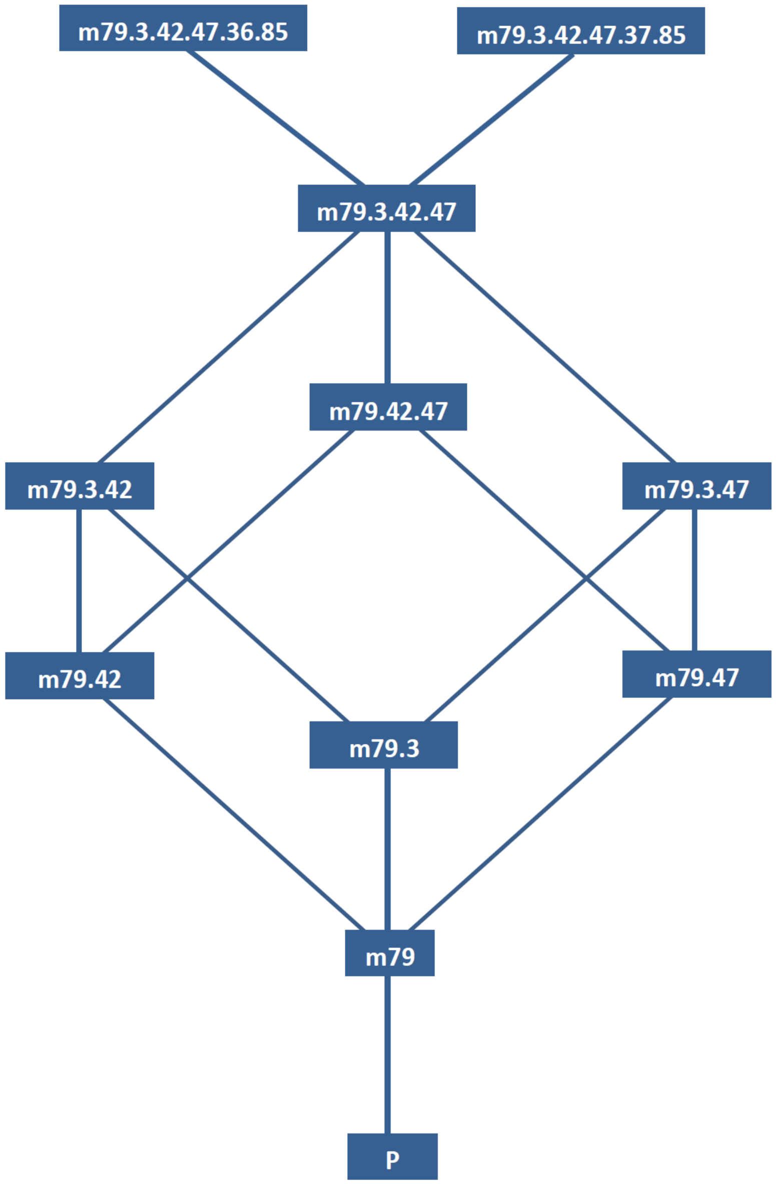
– *drum roll*................



Illustration

Observations:

- 6 modifications in P, 1 fault.
- <u>*depth*</u> decreases by 1 for each fault removal.
- <u>*density*</u>: all over the map.
- m79.3.42.47 not absolutely correct, admits no relcor mutant.
  - Double mutation yields two programs, both absol. correct.
  - One of them original <u>*replace*</u>.
- The cost of failure-based repair:
  - Fault-based: $depth \times O(N) = O(N)$.
  - Failure-based: $O(N \uparrow depth)$.



Agenda

Motivation
Absolute and Relative Correctness
Faults and fault Removals
A Generic Algorithm
Illustration
Conclusion

Conclusion

- Relative correctness ought to be an integral part of the study of program repair.
  - The same way that absolute correctness is part of the study of program construction.
- Removing faults without a definition of fault and fault removal is inherently flawed:
  - Confusion between multi-site faults and multiple single-site faults.
  - Confusion between density and depth.
  - Confusing between remedying a failure and removing a fault.
  - Unnecessary conditions cause loss of recall.
  - Insufficient conditions cause loss of precision.
- The *bad apple* analogy is a bad *apple analogy*.

Conclusion

- Short term Prospects
  - Combine existing patch generation with our oracle-based patch validation.
- Longer term Prospects
  - Turn the mathematics of relative correctness from means to validate repair candidates to means to generate them.
    - Generating more-correct-by-construction repair candidates.
  - In the same way that many researchers in the 80's and 90's turned mathematics of program correctness into means to generate correct-by-construction programs.
    - Dijkstra, Gries, Hehner, Hoare, Morgan, etc.
  - Correctness Enhancement pervades Soft. Engineering.

Agenda

Motivation

Absolute and Relative Correctness

Faults and fault Removals

A Generic Algorithm

Illustration

Conclusion