Automatic, Efficient, and General Repair of Software Defects using Lightweight Program Analyses

Dissertation Proposal

Claire Le Goues

September 22, 2010

Software Errors Are Expensive

"Everyday, almost 300 bugs appear [...] far too many for only the Mozilla programmers to handle."
– Mozilla Developer, 2005[1]

- Even security-critical errors take 28 days to fix.[2]
- Software errors in the US cost $59.5 billion annually (0.6% of GDP)[3].

1. J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In *International Conference on Software Engineering*, pages 361–370, 2006.
2. P. Hooimeijer and W. Weimer. Modeling bug report quality. In *Automated software engineering*, pages 34–43, 2007.
3. NIST. The economic impacts of inadequate infrastructure for software testing. *Technical Report NIST Planning Report* 02-3, NIST, May 2002.

Proposed Solution

Automatic Error Repair

Previous Work

- Runtime monitors + repair strategies [Rinard, Demsky, Smirnov, Keromytis].
  - Increases code size, or run time, or both.
  - Predefined set of error and repair types.
- Genetic programming [Arcuri].
  - Proof-of-concept, limited to small, hand-coded examples.
- Lack of ***scalability*** and ***generality***.

Insights

1. Existing program code and behavior contains the seeds of many repairs.
2. Test cases scalably provide access to information about existing program behavior.

Proposal

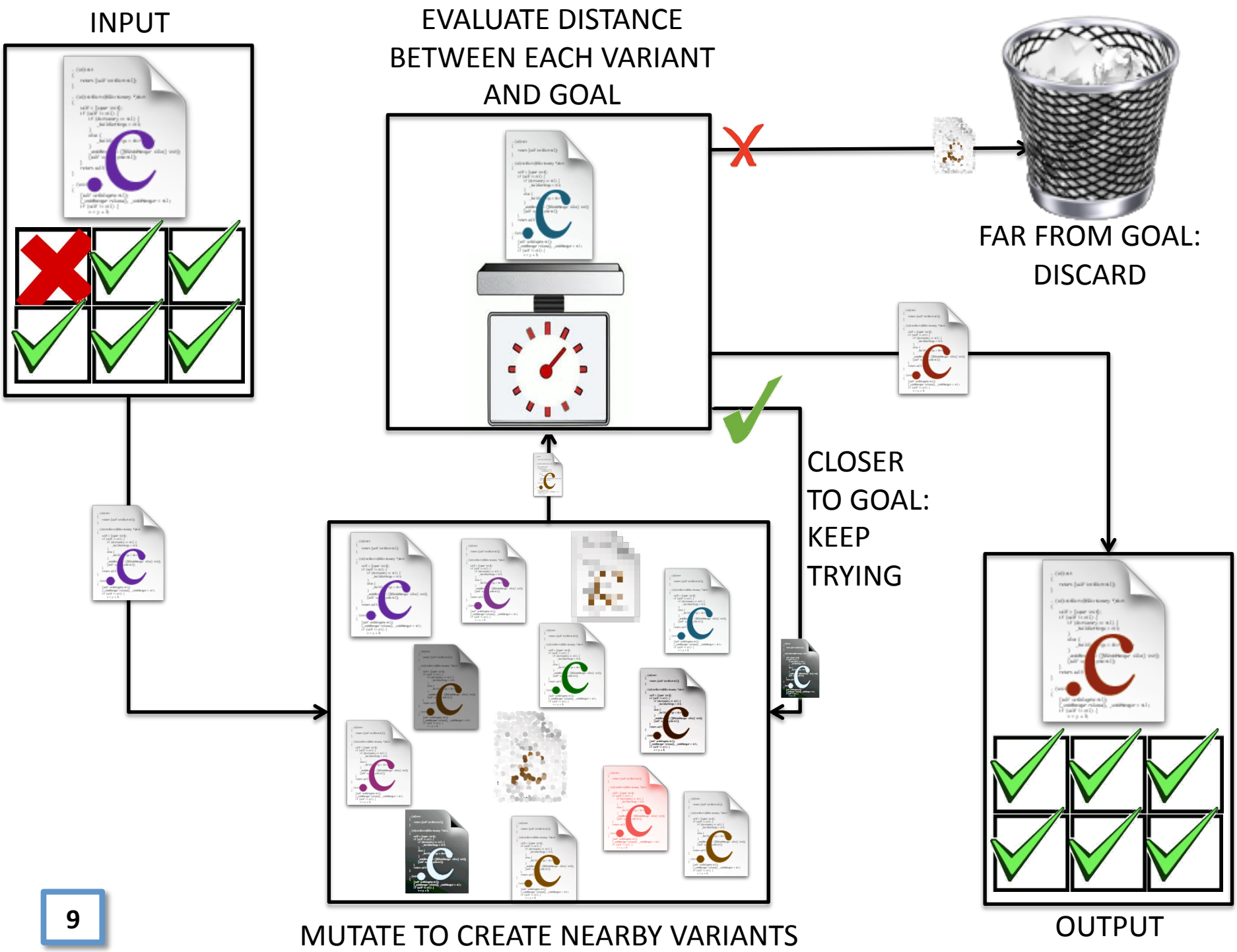
Use **search** strategies, **test cases**, and lightweight **program analyses** to quickly find a version of a program that doesn't contain a particular error, but still implements required functionality.

Outline

- Repair technique metrics
- System overview
- Four research contributions, including preliminary results
- Schedule
- Conclusions

Overall Metrics

- **Scalability**
  - Lines of code. Success: hundreds of thousands of lines.
  - Time. Success: minutes.
- **Generality**
  - Varied benchmark set.
  - As much as possible, real programs (open source) with real vulnerabilities (public vulnerability reports).
- **Correctness**
  - Large, held-out test suites.
  - Performance on workloads.

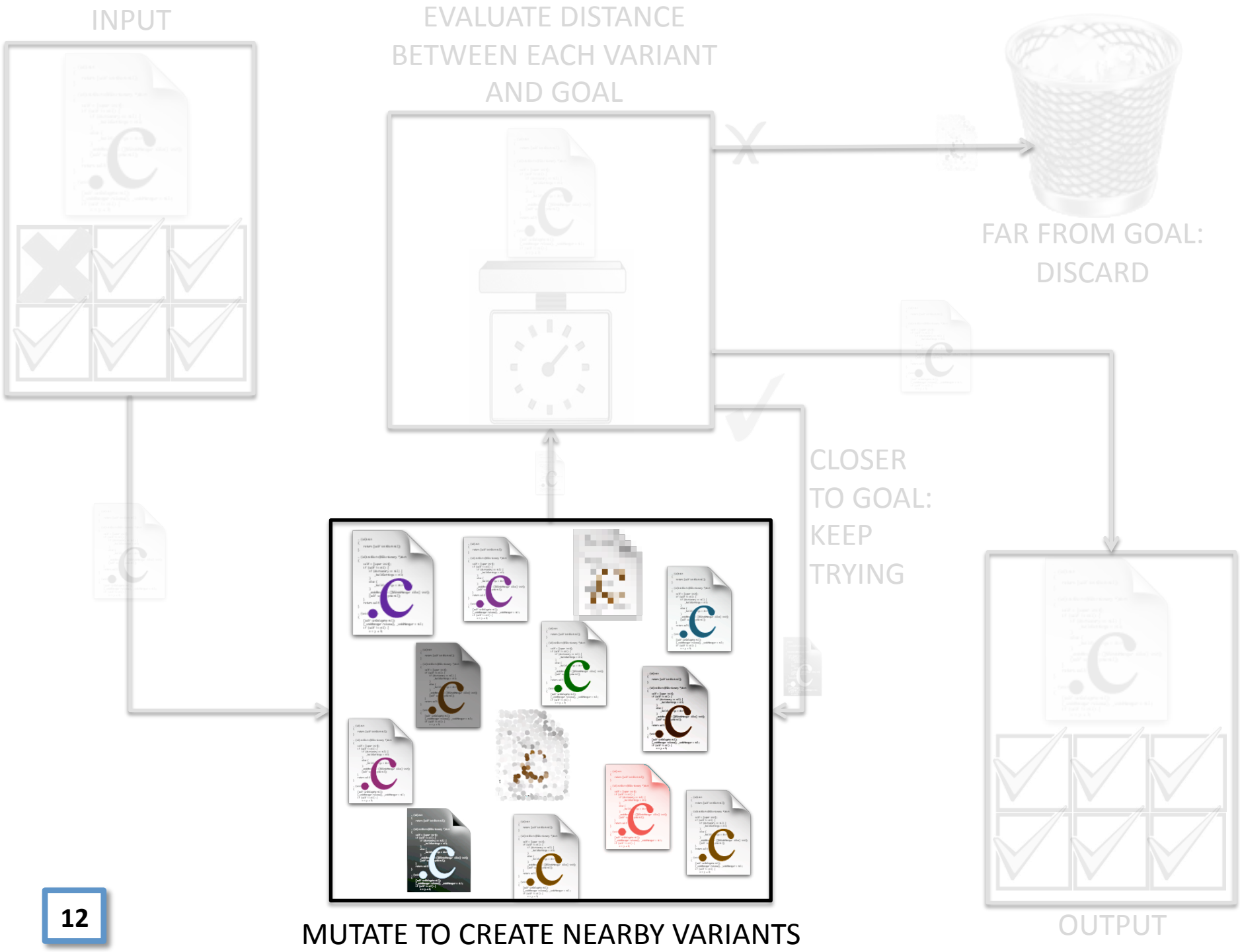

INPUT
EVALUATE DISTANCE BETWEEN EACH VARIANT AND GOAL
FAR FROM GOAL: DISCARD
CLOSER TO GOAL: KEEP TRYING
MUTATE TO CREATE NEARBY VARIANTS
OUTPUT

Four Proposed Contributions

1. **Initial prototype**, with baseline representation, localization, and variant evaluation choices.
2. **Fault and fix localization**: Identify code implicated in the error (that might profitably be changed), and code to use to make changes.
3. **Repair templates:** Generalize previous work by mining and using repair templates, or pieces of code with "holes" for local variables.
4. **Precise objective function**: Develop a precise way to estimate the distance between a variant and a program that passes all test cases.

Preliminary Results

Program	Description	Size (loc)	Fault	Time (s)
gcd	example	22	Infinite loop	149 s
zune	example	28	Infinite loop	42 s
uniq	Text processing	1146	Segmentation fault	32 s
look-ultrix	Dictionary lookup	1169	Segmentation fault	42 s
look-svr4	Dictionary lookup	1363	Infinite loop	51 s
units	Metric conversion	1504	Segmentation fault	107 s
deroff	Document processing	2236	Segmentation fault	129 s
nullhttpd	webserver	5575	Remote heap overflow	502 s
indent	Code processing	9906	Infinite loop	533 s
flex	Lexer generator	18775	Segmentation fault	233 s
atris	Graphical tetris game	21553	Local stack overflow	69 s
Total/Avg		**63K**		**171.7 s**


INPUT
EVALUATE DISTANCE BETWEEN EACH VARIANT AND GOAL
FAR FROM GOAL: DISCARD
CLOSER TO GOAL: KEEP TRYING
OUTPUT
MUTATE TO CREATE NEARBY VARIANTS

Four Proposed Contributions

1. **Initial prototype**, with baseline representation, localization, and variant evaluation choices.
2. **Fault and fix localization**: Identify code implicated in the error (that might profitably be changed), and code to use to make changes.
3. **Repair templates:** Generalize previous work by mining and using repair templates, or pieces of code with "holes" for local variables.
4. **Precise objective function**: Develop a precise way to estimate the distance between a variant and a program that passes all test cases.

Mutating a Program

- Given program A1:
  - With some *probability*, choose code at a location.
  - Insert code before it, or replace it entirely, by copying code from elsewhere in the same program, chosen with some *probability*.
- Result: program A2

Fault localization defines probability that code at a location is modified.

- Goal: Code likely to affect bad behavior without affecting good behavior = high change probability

Fix localization defines probability that code is selected for insertion.

- Goal: code likely to affect repair = high probability of selection.

Search space size is approximated by combining these probabilities over the entire program (how much we can change * how many ways we can change it).

Fault and Fix Localization: Idea

- Plan: use **machine learning** to relate lightweight features to fault/fix probability.
  - Statistics relating statements and dynamic data values to important events, like failure.
  - Static features shown by previous work to correlate with quality.
- Identify code that might affect variables implicated in failure, or code that is similar, but not identical, to likely-faulty code (the same, but includes a null-check, for example).

Fault and Fix Localization: Evaluation

- Effect on search space size (**scalability**):
  - ***Score*** metric: proportion of code eliminated from consideration (higher is better).
  - Measure space size by summing returned probability over the entire program (lower is better)
- Find/create benchmarks with difficult-to-localize errors, like SQL injection attacks (**generality**).

Four Proposed Contributions

1. **Initial prototype**, with baseline representation, localization, and variant evaluation choices.
2. **Fault and fix localization**: Identify code implicated in the error (that might profitably be changed), and code to use to make changes.
3. **Repair templates:** Generalize previous work by mining and using repair templates, or pieces of code with "holes" for local variables.
4. **Precise objective function**: Develop a precise way to estimate the distance between a variant and a program that passes all test cases.

Moving Code: Baseline

```
1.void gcd(int a, int b) {
2.   if (a == 0)
3.     printf("%d", b);
4.   while (b > 0) {
5.     if (a > b)
6.       a = a – b;
7.     else
8.       b = b – a;
9.   }
10.  printf("%d", a);
11.  return;
12.}
```

```
1.void gcd(int a, int b) {
2.   if (a == 0)
3.     printf("%d", b);
4.     return;
5.   while (b > 0) {
6.     if (a > b)
7.       a = a – b;
8.     else
9.       b = b – a;
10.  }
11.  printf("%d", a);
12.  return;
13.}
```

Repair Templates: Idea

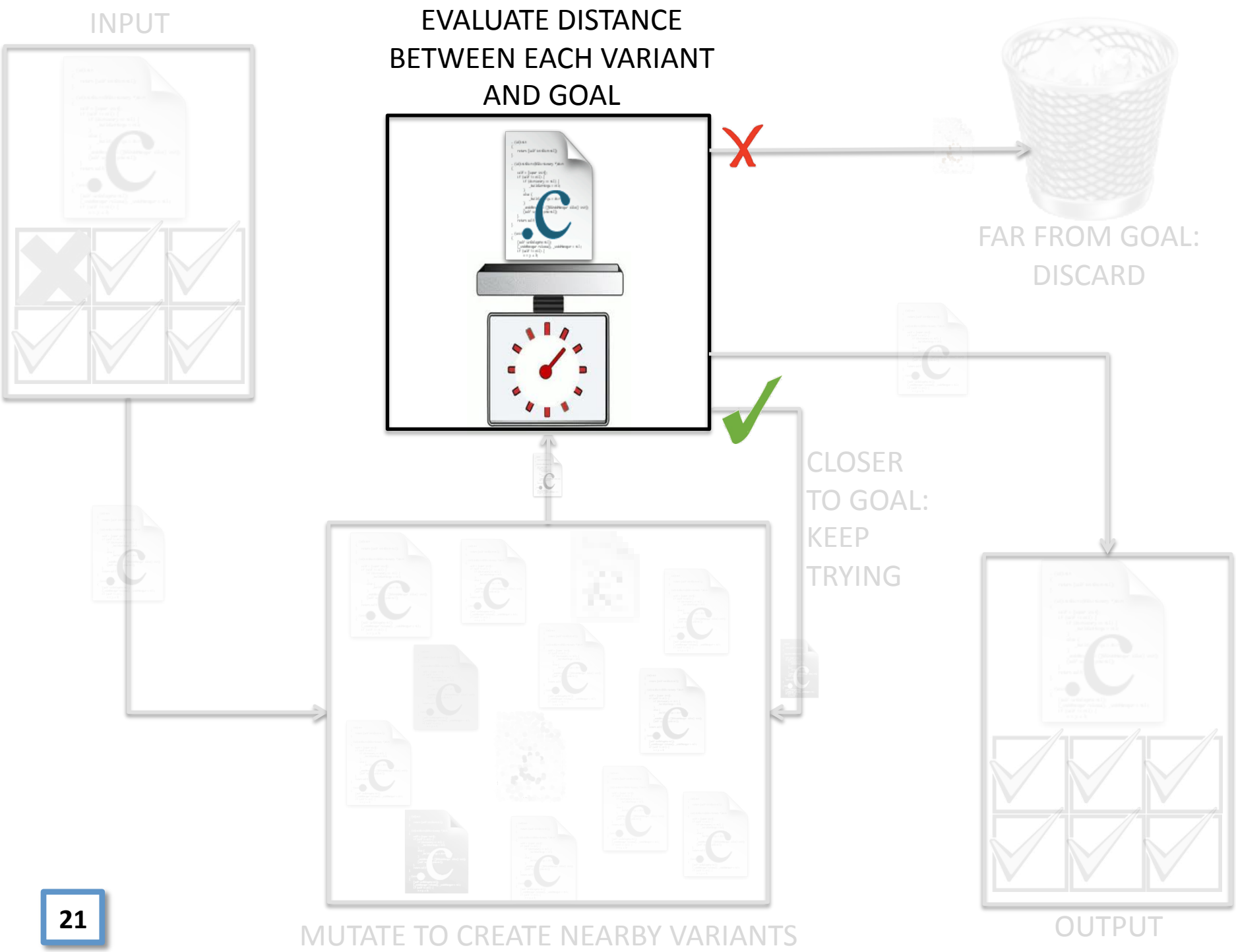
```
1. int gcd2(int a, int b) {
2.   if (a == 0)
3.     printf("%d", b);
4.   while (b > 0) {
5.     if (a > b)
6.       a = a – b;
7.     else
8.       b = b – a;
9.   }
10.  printf("%d", a);
11.  return a;
12.}
```

?

1. **Mine** promising template candidates from existing source code or the source control repository.
2. **Synthesize** templates from candidates, generating code with annotated "holes."
3. **Use** a template to do mutation, as in previous work in error repair or dynamic compilation techniques.

Repair Templates: Evaluation

- Measure proportion of intermediate variants that compile (more is better).
- Formalize: small-step contextual semantics (optional).
- Find/create benchmarks with errors amenable to templated repairs (i.e.: errors handled in previous error repair work or repaired in the source code history).


INPUT
EVALUATE DISTANCE BETWEEN EACH VARIANT AND GOAL
FAR FROM GOAL: DISCARD
CLOSER TO GOAL: KEEP TRYING
MUTATE TO CREATE NEARBY VARIANTS
OUTPUT

Four Proposed Contributions

1. **Initial prototype**, with baseline representation, localization, and variant evaluation choices.
2. **Fault and fix localization**: Identify code implicated in the error (that might profitably be changed), and code to use to make changes.
3. **Repair templates:** Generalize previous work by mining and using repair templates, or pieces of code with "holes" for local variables.
4. **Precise objective function**: Develop a precise way to estimate the distance between a variant and a program that passes all test cases.

Evaluating Intermediate Variants

- The **objective function** estimates the distance between an intermediate variant and the goal (i.e., to pass all test cases); variants closer to goal are used in the next mutation round.
- Natural **baseline**: how many test cases does a variant pass?

A Buffer Underflow Vulnerability

```
1.void broken(int sock) {
2.   char* line, buff=NULL;
3.   int len;
4.   sgets(line,socket);
5.   len = atoi(line);
6.   // no bounds check
7.   buff=calloc(len * 2);
8.   // vulnerable recv
9.   recv(sock,buff,len);
10. return buff;
11.}
```

```
1.void fixed(int sock) {
2.   char* line, ff=NULL;
3.   int len;
4.   sgets(line,socket);
5.   len = atoi(line);
6.   if(len>0 && len<MAX){
7.     buff=calloc(len * 2);
8.     recv(sock,buff,len);
9.   }
10.  return buff;
11.}
```

Objective Function: Idea

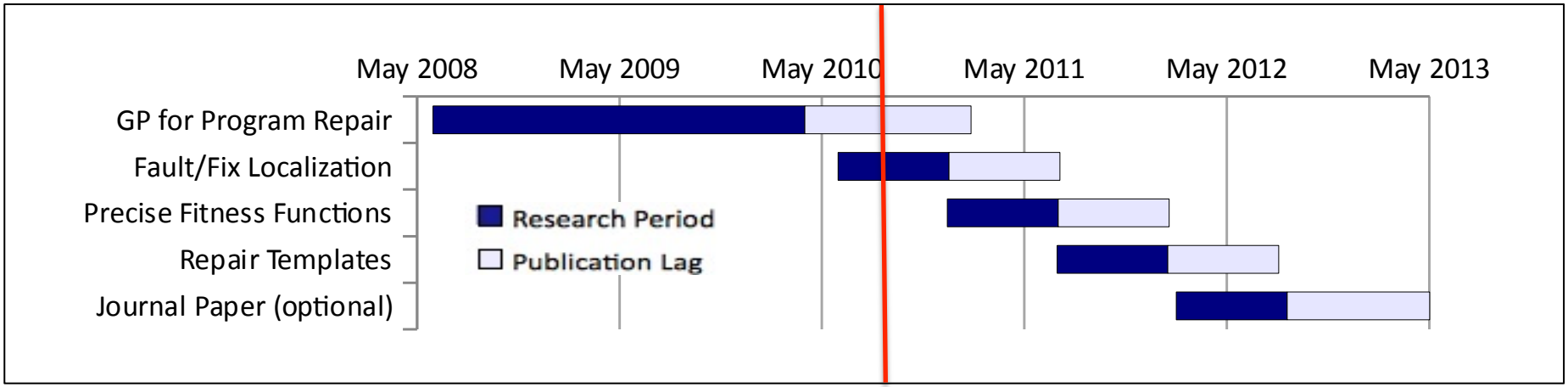
```
1.void almost(int sock) {
2.   char* line, ff=NULL;
3.   int len;
4.   sgets(line,socket);
5.   len = atoi(line);
6.   if(len>0 && len<MAX){
7.     buff=calloc(len * 2);
8.     recv(sock,buff,len);
9.   }
10. len = 5 / 0;
11.  return buff;
12.}
```

- Function should be **precise,** correlating well with actual distance; counting test cases is **imprecise** because it throws away intermediate information.
- Plan: use **machine learning** to relate differences in dynamic behavior between broken program and intermediate program to distance.

Objective Function: Evaluation

- Starting points for "actual" distance: tree-structured differencing, profiles of dynamic behavior.
- Estimate the function's **fitness distance correlation**, or the correlation between it and the "ground truth".
- Find/create benchmarks that require more than one change to repair.

Schedule



- Graduate May 2013 (3 more years).
- Journal article on contribution 1 under revision.
- Slack in schedule: another internship, collaborative project on safety-critical medical equipment software, new ideas that arise from proposed research.

Conclusions

- Goal: scalable, general, correct automatic error repair.
- Approach: search closely-related programs for a version that passes all of the test cases.
- Questions to be answered:
  - What representation choices are necessary to make this possible? ***(Initial Prototype)***
  - How should intermediate variants be created from nearby programs? ***(localization, templates)***
  - How should intermediate variants be evaluated, to effectively guide the search? ***(Precise objective functions)***

Journal

1. C. Le Goues and W. Weimer. **Measuring code quality to improve specification mining**. *IEEE Trans. Software Engineering* (to appear), 2010.
2. W. Weimer, S. Forrest, C. Le Goues, and T. Nguyen. **Automatic Repair with Evolutionary Computation.** *Communications of the ACM.* Vol 53 No. 5, May, 2010, pp. 109-116.

Conference

3. E. Fast, C. Le Goues, S. Forrest and W. Weimer. **Designing Better Fitness Functions for Automated Program Repair.** *Genetic and Evolutionary Computation Conference (GECCO)* 2010: 965-972.
4. S. Forrest, W. Weimer, T. Nguyen and C. Le Goues. **A Genetic Programming Approach to Automatic Program Repair**. *Genetic and Evolutionary Computation Conference (GECCO)* 2009: 947-954.
5. W. Weimer, T. Nguyen, C. Le Goues and S. Forrest. **Automatically Finding Patches Using Genetic Programming.** *International Conference on Software Engineering (ICSE)* 2009:364-374.
6. C. Le Goues and W. Weimer. **Specification Mining With Few False Positives.** *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)* 2009: 292-306

Workshop

7. C. Le Goues, S. Forrest and W. Weimer. **The Case for Software Evolution.** *FSE/SDB Workshop on the Future of Software Engineering Research* (to appear), 2010.
8. T. Nguyen, W. Weimer, C. Le Goues and S. Forrest, **Extended Abstract: Using Execution Paths to Evolve Software Patches.** *Search-Based Software Testing (SBST)* 2009.

Please ask difficult questions.