Introduction to Automatic Software Repair

Martin Monperrus
University of Lille

Wednesday 22nd July, 2015

This document presents an introduction to automatic software repair. It has been first prepared for a course at ECI 2015[1], an Argentine winter school on computer science.

The main URL of the course is `http://www.monperrus.net/martin/eci2015`.

This is a temporary version of the course notes, for printing.

Contents

[1] http://dc.uba.ar/events/eci/2015/

1 Background

1.1 Short bio

I am an associate professor at the University of Lille (France) and a member of INRIA's research group SPIRALS. In 2008-2011, I was a research associate in Mira Mezini's group at the Darmstadt University of Technology (Germany). I received a Ph.D. from the University of Rennes (France) in 2008, for which I was supervised by Jean-Marc Jézéquel, Joël Champeau and Brigitte Hoeltzener. In 2005-2008, I was a research assistant at ENSIETA (Brest, France), thanks to a research scholarship from DGA and CNRS. Previously, I worked as a software engineer for CMB. I spent 6 months working with Yoshua Bengio at the University of Montréal (Canada) in 2004 for my master's thesis. I received a M.Sc. and an engineering degree in computer science from the Compiègne University of Technology (France) in 2004.

2 Concepts

Software engineering is dual. Literally, software engineering is the creation and maintenance of software. But from a research perspective, software engineering is the body of knowledge about the creation and maintenance of software and about the phenomena underlying and emerging from those two activities.

- software engineering: creation and maintenance of actual software
- software engineering research: tools to create software, understanding of the nature of software and its usage.

2.1 In short

Automatic software repair is the process of fixing bugs automatically. There are different kinds of repair

- patch generation
- runtime repair

2.2 Core concepts

The classical terminology about failures [1]

- a **failure** is an observed inacceptable behavior

- a **error** is a propagating incorrect state prior to the failure
- a **fault** is the root cause of the error (in particular incorrect code)
- a bug is both a failure and a fault
- the term "defect" is also used

wikipedia: it's all :-)

My definition: A software bug is a gap between the expected behavior of a program and what it actually does.What I like in this definition is that:

- it does not imply executable specifications
- it does not imply specifications at all

A specification S is the a set of expected correct behaviors and properties.

- a specification may be written in natural language
- a specification may be incomplete
- a specification may be incorrect
- a specification may be inconsistent
- a specification may be implicit (e.g. "the program shall not crash")
- a specification may be executable

An oracle is based on the specification and is intended to determine, for each test (stimuli/input), if the program has violated the specification.

- `assertEquals(6, factorial(3))`
- impacted by encapsulation
- may require white-box observations (mock)

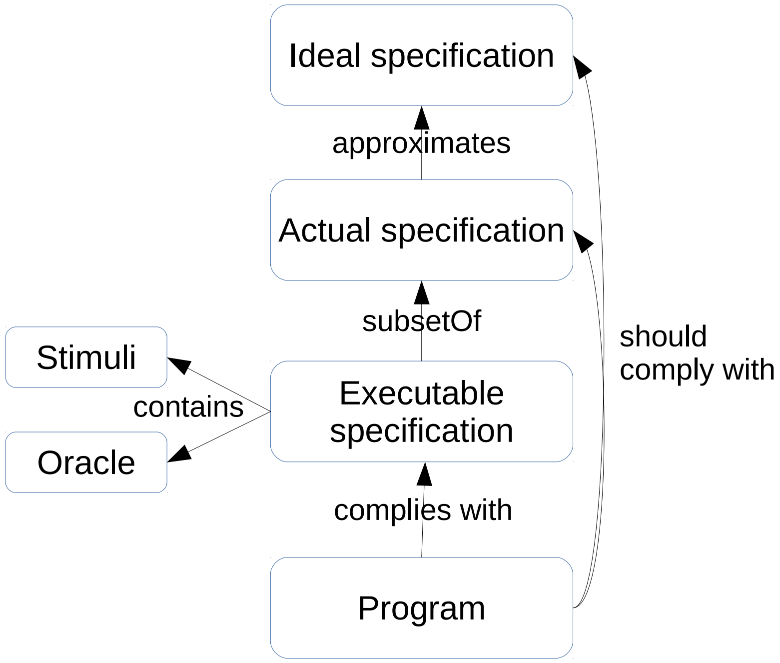

Figure 1: Overview of the concept of "specification"

How to repair bugs automatically?

- Choose a class of bugs
- Identify a good bug oracle
- Set up repair operators
- Add a dose of CPU usage
- Serve it

In repair, there are two kinds of oracles, the bug oracle and the regression oracle. A bug oracle tells you "YES there is a bug", "No, it is fixed"

- Ex: Crashing input: `$ pgm --input foo`
- Ex: Failing test case

- See a good survey [15]

A regression oracle tells you “Ooops, you’ve broken something” or “OK, go ahead”.

- test suite (input-output based)
- pre-conditions, post-conditions, invariants [43]
- logics based specification (LTL, etc) [17]
- if you can reason about the impact of the repair operator, you may avoid a regression oracle

Fault class / defect class:

- Buffer overflow
- Crashes
- Unhandled exceptions
- Infinite loop [25]
- ...

Core repair algorithm:

core repair algorithm

```
While YES there is a bug {
  Try something else
}
```

Because of the loop, a good bug oracle is:

- Automated
- Does not take too long

- Case of not automated: human user [4]

story: human crowd

A **repair operator** (or "repair action") is a modification on the program code or on the program state.
Examples on program code:

-add a precondition
+ if (age>=18)
    serve_adult_content()

Examples of behavioral repair operators:

- add/remove/replace code
- add a precondition [8]
- replace a condition [8]
- replace a method parameter by another
- add a check [22]

talk about Autopag

Examples of state repair operators:

- change a register value [32]
- component restart [38, 37]
- retry [10]
- change object references [9]

Some repair operators may introduce a regression. In this case we need a **regression oracle**:

Listing 1: Basic repair algorithm with regression testing

```
For some time {
  While "YES there is a bug" {
    Try something else
  }
  Is there a regression?
}
```

3 Automatic Patch Generation

In automatic patch generation, the bug oracle can be:

- a failing test case (95% of the cases)
- crashing input
- static analyses [23]

. . . and the regression oracle can be:

- a test suite (95% of the cases)
- none (with a carefully designed repair operator)
- some static analyses

An example of patch generation technique, Genprog [13].

- test suite as oracle of correctness and oracle of bug (at least one failing test case)
- add/ remove/replace statements
- core assumption: redundancy based repair [26].
- evaluation on real bugs

describe redundancy experiment

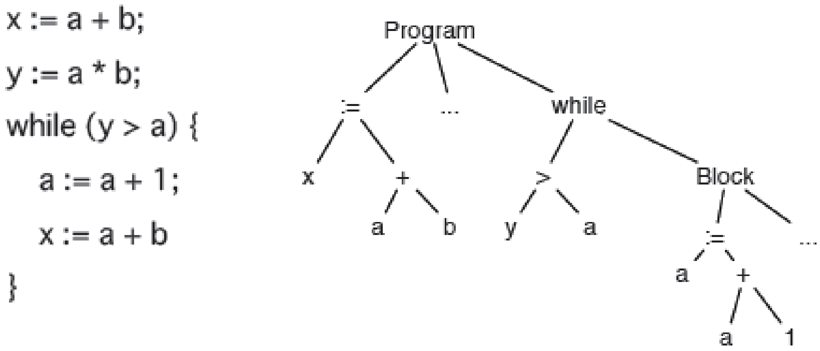

Figure 2: What is an abstract syntax tree (AST)?

- re-implementable in DIY?

Listing 2: Core Genprog Algorithm

```
While some tests fail {
    choose a random modification in add/replace/delete
    perform it
    run tests
}
```

Examples of behavioral repair operators:

- add/remove/replace existing code
- add a precondition [8]
- replace an if-condition [8]
- add a check [22] talk about Autopag

In Semfix [29], Nopol [8]

- Finds a value v such that the failing test case pass (an angelic value)

- Synthesisize an expression e such that $e(context) = v$
- Repair equation: $\forall \text{executions}, e(context) = v$

Semfix [29] and Nopol [8] uses code synthesis to create a patch

- let $I_{x,i}$ be the execution context of expression x at execution i
- let $O_{x,i}$ be the expected value of expression x at execution i
- for a given expression, synthesize exp such that $\forall_i, exp(I_i) = O_i$
- any input/output based synthesizer may be plugged in

There are several ways to find **angelic values**.

- Symbolic/Concolic execution execution [29]
- Value replacement aka **speculative execution** (comes from hardware)
- Model-based diagnosis and trace formulas [18, 30]

Wotawa, early papers

Listing 3: Angelic Value with symbolic/concolic execution

```
void pgm(i) {
  if (x!=1) {
     return 2;
  }
  else {
    return i+2
  }
}

assert pgm(0) = 2
assert pgm(3) = 5

void pgm(i) {
```

```
  if (X) {
      return 2;
  }
  else {
    return i+2
  }
}
```

Constraint: 2 = X ? 0, 0+2
Solution: X = false

3.1 Repair Operators

Now, let's go through different repair operators. We've already seen the ones from Genprog.
Kim et al. [20] has 10 repair templates:

- null pointer checker
- method replacer, parameter adder/remover/replacer, expression adder/remover/replacer
- ...

Kern and Esparza [19] builds a meta-program that is meant to be symbolically executed

Listing 4: Meta-program for repair [19]

```
void pgm(i) {
  if (i != 1) {
      i = 2;
  }
  else {
    i=i+2
  }
  return i
}

// transformed into
void pgm(i) {
```

```
  if (makeSymbolicBooleanVariable() ? i != 1 , i == 1) {
     i = makeSymbolicBooleanVariable() ? 2, 3;
  }
  else {
    i=makeSymbolicBooleanVariable() ? i+2, i-2;
  }
  return i
}
```

// ask JPF: what is valid value of the symbolic variables?

3.2 Sophistication

Fault localization can be used to speed up repair. Most techniques use them.

- try to repair most suspicious statements first
- bugs can be repaired at many different places

The locality of the repair ingredient pool matters [26].

name	commits	redun. global	med .pool size loc	redun. local	med. pool size glo
log4	1687	9.00%	43313	6.00%	57
junit	713	17.00%	8855	16.00%	18
pico	157	3.00%	16911	2.00%	22.5
collections	1019	7.00%	25406	4.00%	35
math	2210	6.00%	69943	4.00%	37
lang	1290	8.00%	22330	6.00%	63

Minimization can be used when the repair process has by-products.

- Use by Genprog
- but no real evolution

4 Runtime Repair

An example of runtime technique, [9].

what about Super Mario? [21]

- uses invariants in data structure
- force restoring those invariants using a constraint solver
- chooses the least costly repair (smallest number of changes)

In runtime repair, the bug oracle is:

- a crash (95% of the cases) (segfault)
- an unhandled exception
- an assertion violation
- a performance problem [40]

Examples of state repair operators:

- change object references [9]
- change a register value [32]
- component restart [38, 37]
- retry [10]
- checkpoint and restart

Reboot/restart is the most common **repair action**.

- can be made recursively [2]
- is related to crash-only property [3]
- much research on this about **rejuvenation** [14]

Failure oblivious computing concentrates on memory errors [35]: validation on Pine

- if the program attempts to read an out of bounds array element, returns the first one
- if the program use an invalid pointer to read a memory location, returns a manufacture value
- if the program attempts to write a value to an out of bounds array element or use an invalid pointer to write a memory location, skip it

Error virtualization consists of transforming an unhandled error case into an handled one [39]:

- returns error code
- transformed into a caught exception [6]
- sophistication: undo changes of the current method, analyzed returned value upper in the stack
- sophistication: trace the artificial value
- error virtualization in the context of exception handling [6]

Clearview is a famous multi-million $ runtime repair system [32].

- learn invariants on register values

- correlate invariants and failures
- repair is reinforcing the invariant

Carzaniga et al. [4] proposed an original runtime repair approach. for web applications.

- the oracle is the end-user, with a button “it doesn’t work”
- the repair action consists of picking up an alternative method from a set of alternatives (e.g. two different method calls for youtube)
- relies on the presence of computational redundancy in software (several ways to do the same thing). Will be discussed later [11].

Some bugs disappear in new versions while others appear. This key insight is behind the repair system proposed by Hosek and Cadar [16].

- the oracle is a Unix signal (SEGFAULT, etc)
- the new version is run in parallel with the old
- the repair action consists of transferring the system state and executing another version

Many bugs are due to the wild space of possible inputs. Generating appropriate filters avoid crashes [24].

- identify structure of inputs
- learn classical values from dataset
- rectify inputs based on standard values

5 Repair research questions

What bugs can be repaired?

- `https://github.com/php/php-src/commit/1e91069`
- Math-280: bug in inverseCumulativeProbability() for Normal Distribution
- ...

What bug kinds can be repaired?

not all bugs in a given fault class

- arithmetic errors [29]
- off-by-one errors [18]
- conditions errors [29, 8]
- infinite loops [25]

How fast can real bugs be repaired?

- "An average repair run took 356.5 seconds" [13]
- Within less than 2 minutes [8]

6 Advanced Discussions

There are other kinds of repairs:

- domain-specific repair [36, 12]
- test repair [7]

Some controversies about Genprog:

- no genetic programming, no evolution [33]

- really bad test suites [34]
- experimental error [34]

Relation between repair and program synthesis. Given a program P and a specification S.

- Classical correctness: P that complies with S, $P \models S$
- Synthesis: find P such that $P \models S$
- Repair: find a change C such that $P + C \models S$

The question of patch overfitting [41]:

- some patches simply hard code the correct answer
- how often does this happen?
- how to mitigate this?

Listing 5: Illustration of patch overfitting

```
assertEquals(3,pgm(6))

void pgm(i) {
  // synthesized patch
  if (i == 3) return 6

 // rest of the program
}
```

The question of **correctness**

- classical correctness is binary
- correctness may be continuous [27]
- correctness may be partial [28]

The question of patch acceptability. Which patch is better?

```
// fix A: code insertion at line 21
+ if (x==2) { foo(x); }

// fix B: code insertion at line 21
+ if (x<=2) { foo(x); }
```

- for impact minimization, #1 is better
- for regular output domain, #2 is better (no spike)

The question of equivalent computational effects.

- there are often several if not dozens of equivalent patches
- some of them are due to the weakness of the test suite
- others are due to some kind of computational equivalence
- fascinating empty research area

The question of the fitness landscape.

- one small change may yield a big difference in output
- to drive a search you need to stack some changes.
- necessity to find smooth repair operators [5]

convex landscape, hill climbing

- and smooth programming languages [31]

One important milestone for automatic repair research:

- fully generated patch accepted by human developers without knowing it has been created by a robot
- or even a flame war
- kind of captcha for repair

Many bugs appear because we have development processes and software stack that are fragile and brittle

Software brittleness, software fragility

- fragile to changes in the environment
- fragile to changes in the code
- fragile with respect to clever and malicious users
- needs for rethinking many points of software engineering [42, 11]

7 Conclusion

Automatic software repair is a young research area. It touches the foundations of software. I think it's fun and I would be glad to help you enter this fascinating field.

Important Concepts

References

[1] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004.

[2] G. Candea and A. Fox. Recursive restartability: Turning the reboot sledgehammer into a scalpel. In *Hot Topics in Operating Systems, 2001. Proceedings of the Eighth Workshop on*, pages 125–130. IEEE, 2001.

[3] G. Candea and A. Fox. Crash-only software. In *Proceedings of the 9th conference on Hot Topics in Operating Systems-Volume 9*, pages 12–12. USENIX Association, 2003.

[4] A. Carzaniga, A. Gorla, N. Perino, and M. Pezzè. Automatic workarounds for web applications. In *FSE'10: Proceedings of the 2010 Foundations of Software Engineering conference*, pages 237–246, New York, NY, USA, 2010. ACM.

[5] S. Chaudhuri, S. Gulwani, and R. Lublinerman. Continuity and robustness of programs. *Communications of the ACM*, 55(8):107–115, 2012.

[6] B. Cornu, L. Seinturier, and M. Monperrus. Exception Handling Analysis and Transformation Using Fault Injection: Study of Resilience Against Unanticipated Exceptions. *Information and Software Technology*, 2014.

[7] B. Daniel, V. Jagannath, D. Dig, and D. Marinov. ReAssert: Suggesting repairs for broken unit tests. In *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering*, pages 433–444. IEEE/ACM, November 2009.

[8] F. DeMarco, J. Xuan, D. L. Berre, and M. Monperrus. Automatic Repair of Buggy If Conditions and Missing Preconditions with SMT. In *Proceedings of the 6th International Workshop on Constraints in Software Testing, Verification, and Analysis (CSTVA 2014)*, 2014.

[9] B. Demsky and M. Rinard. Automatic detection and repair of errors in data structures. *ACM SIGPLAN Notices*, 38(11):78–95, 2003.

[10] G. Friedrich, M. Fugini, E. Mussi, B. Pernici, and G. Tagni. Exception handling for repair in service-based processes. *IEEE Transactions on Software Engineering*, 36(2):198–215, 2010.

[11] R. P. Gabriel and R. Goldman. Conscientious software. In *Acm Sigplan Notices*, volume 41, pages 433–450. ACM, 2006.

[12] D. Gopinath, S. Khurshid, D. Saha, and S. Chandra. Data-guided repair of selection statements. In *Proceedings of the 36th International Conference on Software Engineering*, pages 243–253. ACM, 2014.

[13] C. L. Goues, T. Nguyen, S. Forrest, and W. Weimer. Genprog: A generic method for automatic software repair. *IEEE Transactions on Software Engineering*, 38:54–72, 2012.

[14] M. Grottke and K. S. Trivedi. Fighting bugs: Remove, retry, replicate, and rejuvenate. *Computer*, 2007.

[15] M. Harman, P. McMinn, M. Shahbaz, and S. Yoo. A comprehensive survey of trends in oracles for software testing. Technical Report CS-13-01, 2013.

[16] P. Hosek and C. Cadar. Safe software updates via multi-version execution. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 612–621. IEEE Press, 2013.

[17] B. Jobstmann, A. Griesmayer, and R. Bloem. Program Repair as a Game. *Program*, 507219:226–238, 2005.

[18] M. Jose and R. Majumdar. Cause clue clauses: error localization using maximum satisfiability. *ACM SIGPLAN Notices*, 46(6):437–446, 2011.

[19] C. Kern and J. Esparza. Automatic error correction of java programs. In *Formal Methods for Industrial Critical Systems*, pages 67–81. Springer, 2010.

[20] D. Kim, J. Nam, J. Song, and S. Kim. Automatic patch generation learned from human-written patches. In *Proceedings of ICSE'2013*, 2013.

[21] C. Lewis and J. Whitehead. Runtime repair of software faults using event-driven monitoring. *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - ICSE '10*, 2:275, 2010.

[22] Z. Lin, X. Jiang, D. Xu, B. Mao, and L. Xie. Autopag: towards automated software patch generation with source code root cause identification and repair. In *Proceedings of the 2nd ACM symposium on Information, computer and communications security*, pages 329–340. ACM, 2007.

[23] F. Logozzo and T. Ball. Modular and verified automatic program repair. In *Proceedings of the 27th ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'12)*. ACM Press, New York, NY, 2012.

[24] F. Long, V. Ganesh, M. Carbin, S. Sidiroglou, and M. Rinard. Automatic input rectification. In *Proceedings of ICSE*, 2012.

[25] S. L. Marcote and M. Monperrus. Automatic Repair of Infinite Loops. Technical Report 1504.05078, Arxiv, 2015.

[26] M. Martinez, W. Weimer, and M. Monperrus. Do the Fix Ingredients Already Exist? An Empirical Inquiry into the Redundancy Assumptions of Program Repair Approaches. In *Proceedings of the International Conference on Software Engineering - NIER track*, 2014.

[27] S. Misailovic, S. Sidiroglou, H. Hoffmann, and M. Rinard. Quality of service profiling. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 25–34. ACM, 2010.

[28] M. Monperrus. A Critical Review of "Automatic Patch Generation Learned from Human-Written Patches": Essay on the Problem Statement and the Evaluation of Automatic Software Repair. In *Proceedings of the International Conference on Software Engineering*, pages 234–242, 2014.

[29] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. SemFix: Program Repair via Semantic Analysis. In *Proceedings of the 2013 International Conference on Software Engineering*, 2013.

[30] M. Nica, S. Nica, and F. Wotawa. On the use of mutations and testing for debugging. *Software: Practice and Experience*, 43(9):1121–1142, 2013.

[31] C. Ofria, C. Adami, and T. C. Collier. Design of evolvable computer languages. *IEEE Transactions on Evolutionary Computation*, 6(4):420–424, 2002.

[32] J. H. Perkins, G. Sullivan, W.-F. Wong, Y. Zibin, M. D. Ernst, M. Rinard, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, and S. Sidiroglou. Automatically patching errors in deployed software. *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles - SOSP '09*, page 87, 2009.

[33] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang. The strength of random search on automated program repair. In *Proceedings of the 36th International Conference on Software Engineering*, pages 254–265. ACM, 2014.

[34] Z. Qi, F. Long, S. Achour, and M. Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proceedings of ISSTA*. ACM, 2015.

[35] M. Rinard, C. Cadar, D. Dumitran, D. Roy, T. Leu, and W. Beebee Jr. Enhancing server availability and security through failure-oblivious computing. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation-Volume 6*, pages 21–21. USENIX Association, 2004.

[36] H. Samimi, M. Schäfer, S. Artzi, T. D. Millstein, F. Tip, and L. J. Hendren. Automated repair of html generation errors in php applications using string constraint solving. In *ICSE*, pages 277–287, 2012.

[37] S. Sicard, F. Boyer, and N. De Palma. Using components for architecture-based management: the self-repair case. In *Proceedings of the 30th international conference on Software engineering*, pages 101–110. ACM, 2008.

[38] S. Sidiroglou, O. Laadan, C. Perez, N. Viennot, J. Nieh, and A. Keromytis. Assure: automatic software self-healing using rescue points. In *ACM Sigplan Notices*, volume 44, pages 37–48. ACM, 2009.

[39] S. Sidiroglou, M. Locasto, S. Boyd, and A. Keromytis. Building a reactive immune system for software services. In *Proceedings of the USENIX Annual Technical Conference*, volume 161. 2oo5,(6), 2005.

[40] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. Rinard. Managing performance vs. accuracy trade-offs with loop perforation. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 124–134, New York, NY, USA, 2011. ACM.

[41] E. K. Smith, E. Barr, C. Le Goues, and Y. Brun. Is the cure worse than the disease? overfitting in automated program repair. In *Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, Bergamo, Italy, September 2015.

[42] G. J. Sussman. Building robust systems an essay, 2007.

[43] Y. Wei, Y. Pei, C. A. Furia, L. S. Silva, S. Buchholz, B. Meyer, and A. Zeller. Automated fixing of programs with contracts. In *Proceedings of the International Symposium on Software Testing and Analysis*. AC, 2010.