Automatic Software Repair: A Survey

Luca Gazzola, Daniela Micucci and Leonardo Mariani

Diogo Machado de Freitas

UFG

INSTITUTO DE INFORMÁTICA UFG

I4Soft
Intelligence for Software

Summary

1. Paper selection
2. Software repair
3. Localization
4. Fix generation
5. Fix recommenders

Introduction

Introduction

1. Previous approaches provide useful insights about the possible locations of faults, the inputs and states responsible for the failures, as well as the anomalous operations executed during failures.
2. Program repair techniques:
    a. Automatically repair software systems by producing an actual fix that can be accepted or adapted to fit the system.

Paper Selection

Paper selection

1. Databases:
   a. ACM DL;
   b. IEEE Explorer;
   c. Google Scholar.
2. Search terms:
   a. "program repair", "software repair", "automatic patch generation", "automatic fix generation", "generate and validate", and "semantics driven repair";
3. Papers selected at January 2017;
4. Extract the first 200 references sorted by relevance;
5. 2,573 results.

Paper selection

1. All references where manually analyzed by checking the abstract, the introduction, and the conclusion sections, and scanning the rest of the paper, to quickly eliminate papers clearly unrelated to the topic of the survey;
    a. 107 papers;
2. These papers were read carefully to determine their relevance according to the conceptual framework;
    a. 85 papers;
3. Inclusion of relevant papers that were cited by the selected papers (23 papers).
    a. 107 papers;
4.

Paper selection

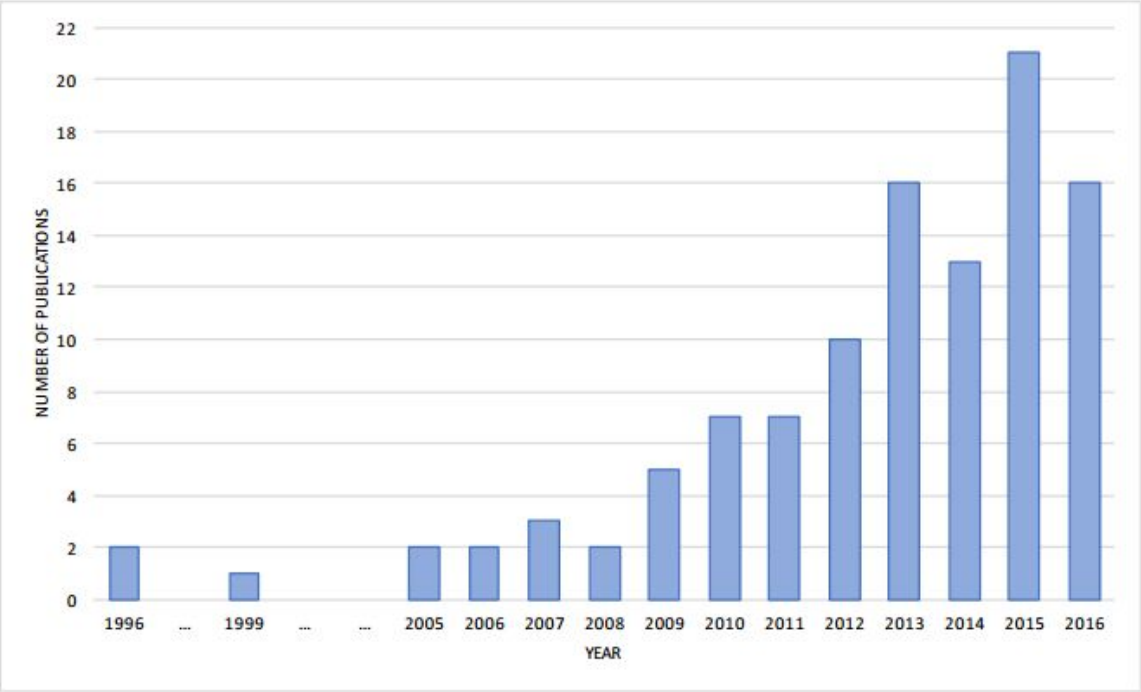

Fig. 1: Publications per year from 1996 to 2016.

Paper selection

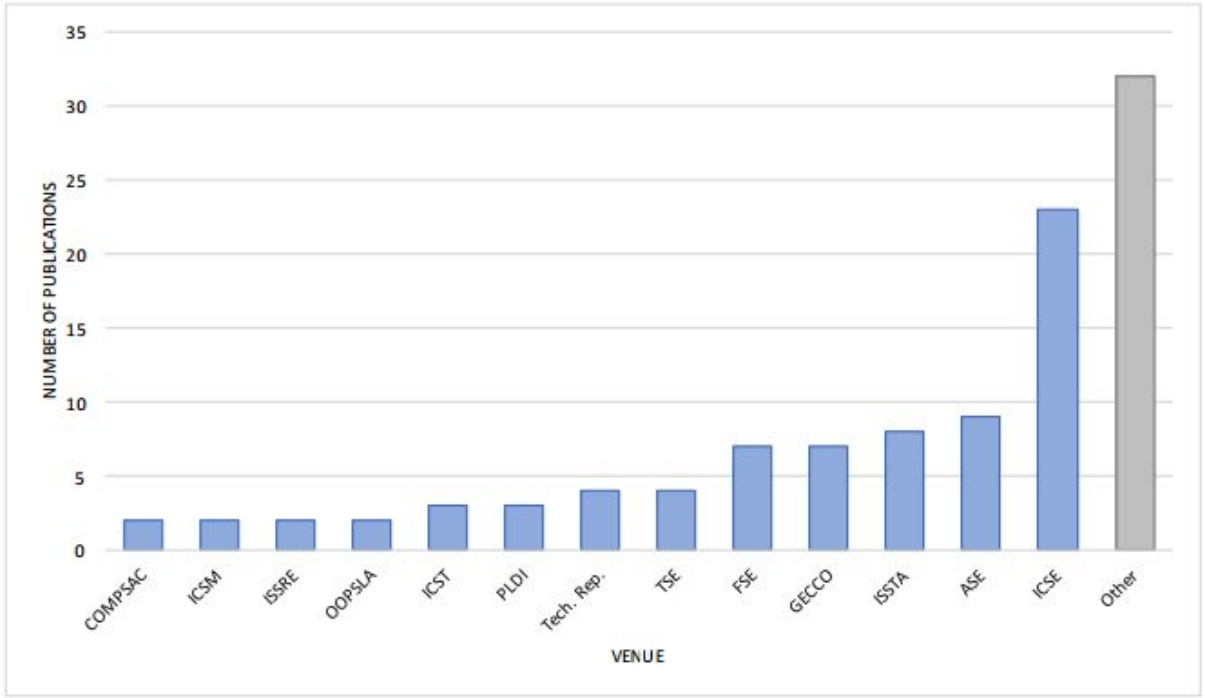

Fig. 2: Publications per venue.

Software repair

Software repair

1. Software healing;
2. Software repair;
3. Healing operations: applied at runtime to turn a failed or a failing execution into a successful execution;
4. Repair operations: performed on the program source code to remove the fault that caused a failure;
5. Workarounds;
6. Fixes.

Software repair

1. ISO/IEC/IEEE 29119-1:2013(E)
2. A mistake or error, made by a human being, causes a defect to appear in the product that the person is working on.
3. The defect has no impact on the operation of the software if it is not encountered when the software is used.
4. But if a defect is encountered under the right conditions when the product is put to use, it may cause the product to fail to meet the user's legitimate need.

> ***Software repairing*** *solutions detect software failures, localize where a fix could be applied, and make the necessary adjustments to fix the fault, and thus prevent further failures caused by the same fault.*

Software repair

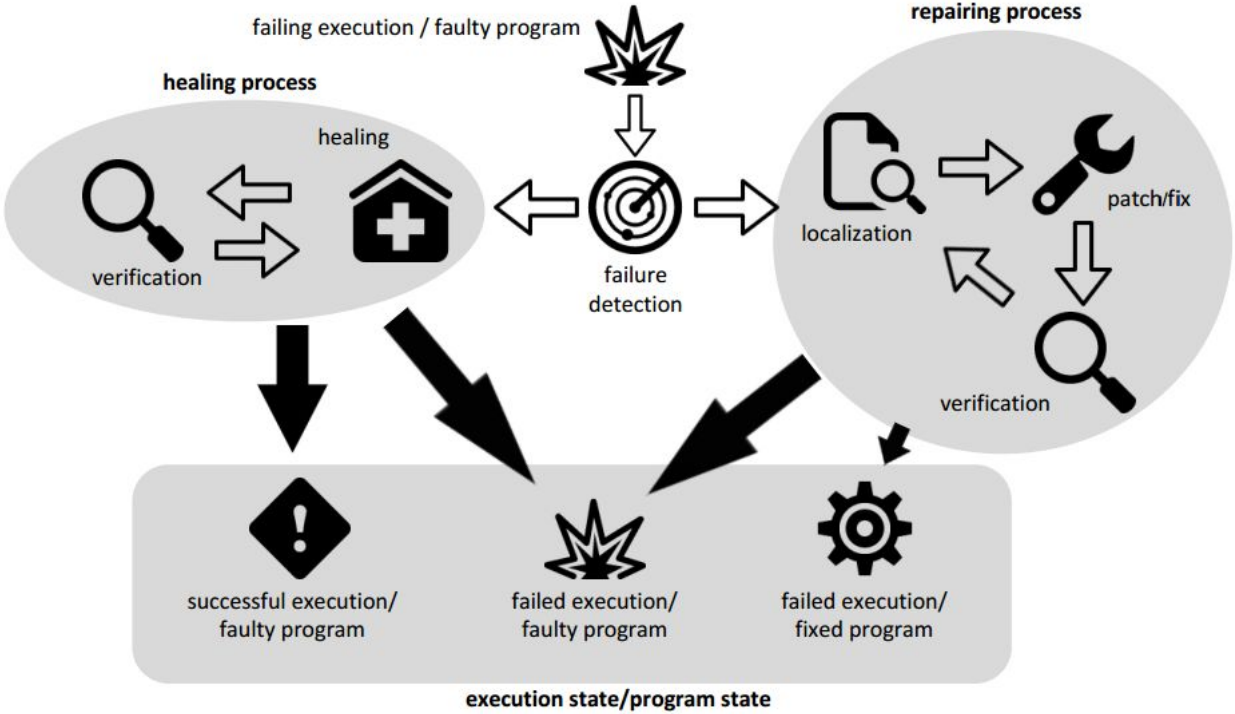

Fig. 3: Healing and repairing processes.

Fault localization techniques

Localization

1. **Fault localization** techniques aim at identifying the faulty statements.
2. **Fix locus localization** techniques aim at detecting the statements where the effect of the fault can be observed and eliminated.
    a. These statements are not necessarily the statements with the fault, but they are potentially modifiable to let the program behave correctly. (compensate the effect of the fault, rather than remove the fault).

Localization | Fault localization

1. Spectrum-based;
    a. **GenProg** (32% of the papers that use SBFL);
    b. **Tarantula** (18% of the papers that use SBFL);
    c. **Ochiai** (18% of the papers that use SBFL);
    d. **Jaccard** (5% of the papers that use SBFL).

Localization | Fault localization

1. Tarantula:

$$susp T(s) = \frac{\frac{failed(s)}{totalfailed}}{\frac{passed(s)}{totalpassed} + \frac{failed(s)}{totalfailed}} \qquad (1)$$

2. Ochiai:

$$susp O(s) = \frac{failed(s)}{\sqrt{totalfailed \times (failed(s) + passed(s))}} \qquad (2)$$

Localization | Fault localization

1. GenProg:

$$susp G(s) = \begin{cases} 0 & failed(s) = 0 \\ 1.0 & passed(s) = 0 \wedge failed(s) > 0 \\ 0.1 & otherwise \end{cases} \quad (3)$$

2. Jaccard:

$$susp J(s) = \frac{failed(s)}{execute(s) + (totalFailed - failed(s))} \quad (4)$$

Localization | Fault localization

	(0,0)	(0,10)	(10,0)	(10,15)	(15,10)	Tarantula	Ochiai	GenProg	Jaccard
gcdWrongPrint(int a, int b) {									
1: if (a == 0){	X	X	X	X	X	0.5	0.4	0.1	0.2
2: printf("%d", a);	X	X				0.8	0.7	0.1	0.5
3: exit(0);}	X	X				0.8	0.7	0.1	0.5
4: while (b != 0){			X	X	X	0	0	0	0
5: if (a >b){				X	X	0	0	0	0
6: a = a - b;				X	X	0	0	0	0
7: }else{				X	X	0	0	0	0
8: b = b - a;}				X	X	0	0	0	0
9: printf("%d", a);			X	X	X	0	0	0	0
10: exit(0);}			X	X	X	0	0	0	0
test outcome	**P**	**F**	**P**	**P**	**P**				

Fig. 4: SBFL with Tarantula, Ochiai, GenProg, and Jaccard

Localization | Fix locus localization

1. Identify the program locations suitable for the synthesis of fixes, regardless of the location of the faults;
    a. A program location where the effect of a fault could be recognized could also be exploited to compensate its effect.
2. **Model-based fix locus localization**: identify statements that use objects illegally in object-oriented software;
3. **Angelic fix localization**: which can localize the statements relevant to missing or faulty *if* conditions faults.

Localization | Model-based fix locus loc.

1. Context of program repair techniques that can repair faults that cause the incorrect usage of class interfaces;
2. Run the passing test cases to collect runtime data and then mine models that represent how classes are used by the program during correct runs;
3. The mined models are then checked dynamically while the failing test cases are executed;
4. A failed execution that violates a mined model indicates the presence of objects that are incorrectly used by the program;
    a. Code that writes data on a file that has been opened in read-only mode.

Localization | Angelic fix localization

1. Targets faults that can be repaired by modifying the *if* conditions in a program;
2. First localize the suspicious conditions according to their suspiciousness score, and then identify which of these conditions might be wrong by systematically forcing the failing test cases to take alternative branches at decision points, regardless of the outcome of the evaluated condition.
3. **Angelic values**: conditional values that make test cases pass;
4. Points that might be used to turn failing test cases into successful test cases are selected for repairing;

Localization | Angelic fix localization

1. Detect missing if conditions;
2. Checks if failing test cases can be turned into passing test cases by skipping the execution of single (simple or compound) statements;
3. Not necessarily the fault locations: places where the flow of the executions could be opportunistically altered, by modifying or adding conditions, to mask or compensate the effect of a fault.

Fix generation

Fix generation

1. Fix generation problem: P_{repair};
2. ***Generate-and-validate***:
   a. Produce fixes by defining and exploring a space S_{repair} of the potential solutions to P_{repair} (note that S_{repair} may contain both elements that solve and do not solve P_{repair}).
3. ***Semantics-driven***:
   a. (also known as *correct-by-construction* approaches) encode the problem P_{repair} formally, either explicitly or implicitly, and once they find a solution, the solution is guaranteed to solve P_{repair}.
4. Any solution is always reported to the developers who cross-check its quality and correctness;
5. Helps to understand the fault in the code and simplifies the implementation of the actual fix.

Fix generation

1. **General techniques** are not designed to target a specific class of faults and can potentially repair any class of faults in a program.
2. **Fault-specific techniques** are designed to address some classes of faults only, such as wrong conditions in conditional statements and buffer overflows.

Fix generation | Generate-and-validate

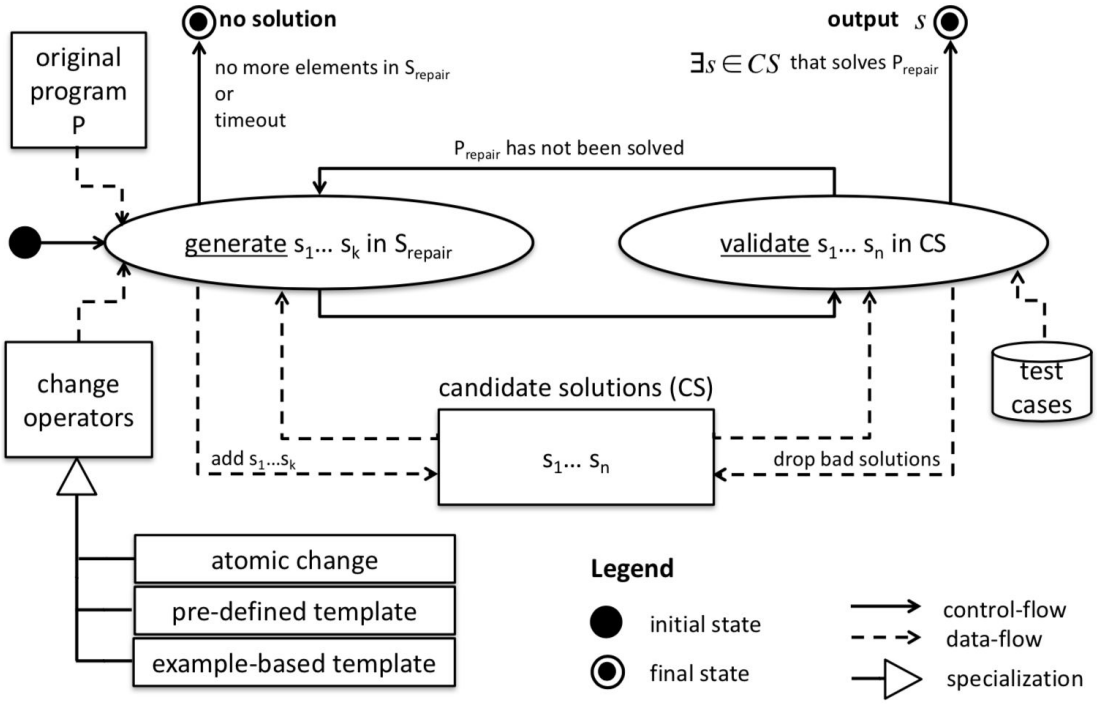

Fig. 5: Generate-and-validate repair process

Fix generation | Generate-and-validate

1. **Atomic change** operators modify P in a single point.
2. **Pre-defined template** operators change P according to potentially complex pre-defined patterns.
3. **Example-based template** operators work the same than pre-defined templates, but the templates are extracted, either manually or automatically, from historical data (e.g., a versioning system).

Fix generation | Generate-and-validate

1. **Search-based** strategies apply change operators randomly or guided by a heuristic or meta-heuristic search algorithm.
2. **Brute-force** strategies systematically produce every possible change that can be obtained within certain bounds with the considered change operators and localization algorithm.
3. In the literature the label “search-based” has been used both to specifically identify the techniques that use search algorithms and to indicate the whole class of generate-and-validate techniques.

Fix generation | Generate-and-validate

TABLE 1: Generate-and-validate techniques - *Atomic change operators*

Strategy	Fault model	Change model	Techniques	Section
search-based	general	AST reuse/insert/delete	GenProg, Marriagent, RSRepair, SCRepair	Sec. 6.1.1.1
		AST modifications	JAFF	
		operator replacement, variable name replacement	pyEDB, MUT-APR, CASC	
brute-force	general	operator replacement, condition negation	Debroy and Wong	Sec. 6.1.1.2
		method call insertion/deletion	PACHIKA	
		functionality deletion	KALI	
		AST reuse/insert/delete	AE	

Fix generation | Generate-and-validate

1. A set of atomic change operators defined at the level of the AST of the program;
2. **GenProg**, **Marriagent**, **RSRepair** and **SCRepair** use the same three atomic change operators: delete, insert and modify.
3. **JAFF** extends this set of atomic changes with operators that can manipulate subtrees of the AST and operators that can randomly generate new statements in the program.
4. Plastic surgery hypothesis;
5. **pyEDB**, **MUT-APR**, and **CASC** use a small set of focused change operators (change operators that can modify the operators and the variables used in the target program).

Fix generation | Generate-and-validate

1. Explore the search space systematically;
2. Four types of change operators: operator replacement and condition negation, method call insertion or deletion, functionality deletion, and AST manipulations.

Fix generation | Generate-and-validate

TABLE 2: Generate-and-validate techniques - *Template-based change operators*

Template type	Strategy	Fault model	Change model	Techniques	Section
pre-defined	search-based	concurrency faults	synchronized region manipulation	ARC	Sec. 6.1.2.1
	brute-force	general	code transformation templates	AutoFix-E, AutoFix-E2	Sec. 6.1.2.2
			condition change, variable value change	SPR, Prophet	
		buffer overflow	buffer manipulation, function replacement	PASAN, AutoPAG	Sec. 6.1.2.3
example-based	search-based	general	code transformation templates, reuse of statements in the same application	History-driven repair	Sec. 6.1.3.1
			code transformation templates	PAR, Relifix	
	brute-force	general	code transformation templates	R2Fix	Sec. 6.1.3.2
		buffer overflow	insertion of code fragments from donor programs in the code under repair	CodePhage	Sec. 6.1.3.3

Fix generation | Generate-and-validate

1. **Pre-defined templates**: pre-defined templates modify programs according to a set of change operators that can affect one or more statements of the program;
2. Complex change patterns can coherently affect the program under repair in multiple locations (hard to obtain with a randomized combination of atomic changes);
3. Ex.: Changes that expand synchronization blocks, perform non-trivial manipulations on program conditions, and add code implementing pre-defined access control policies.
4. Search-based techniques address specific classes of faults, while bruteforce techniques address both general and specific classes of faults.

Fix generation | Generate-and-validate

1. **Example-based templates**: modify programs, either evolving them (search-based techniques) or systematically performing changes (brute-force techniques), according to a set of change operators that are extracted from a sample set of fixes that have been already used to fix programs.
2. Might implement quite complex schema that coherently affect the program under repair in multiple locations;

Fix generation | Generate-and-validate

1. Manual extraction: templates are defined once for all and then used to fix programs.
   a. History-driven repair.
2. Automated extraction: extraction process can be repeated every time from a different set of programs, increasing the generality of the technique.
   a. CodePhage, can automatically extract fixes for buffer overflow problems from a set of correct programs.
3. The effectiveness of examplebased techniques is clearly dependent on the set of cases used to extract the templates.

Fix generation | Semantics-driven

1. Semantics-driven techniques encode the problem P_{repair} formally, either **explicitly**, for instance as a formula whose solutions correspond to the possible fixes of the program under repair, or **implicitly**, as an analytical procedure whose outcome is a fix.
2. A solution s_{repair}, when it can be found, is guaranteed to solve the problem P_{repair} and thus does not need to be validated against P_{repair}.

Fix generation | Semantics-driven

1. Note that although a solution s_{repair} is guaranteed to solve P_{repair}, s_{repair} is not guaranteed to be fully satisfactory for the developers.
2. P_{repair} is an approximated representation of the real repair problem to be solved, and a solution might be problematic according to aspects not represented in P_{repair}.
3. The automatically generated solutions still need to be validated manually or automatically to decide if they can be finally accepted.

Fix generation | Semantics-driven

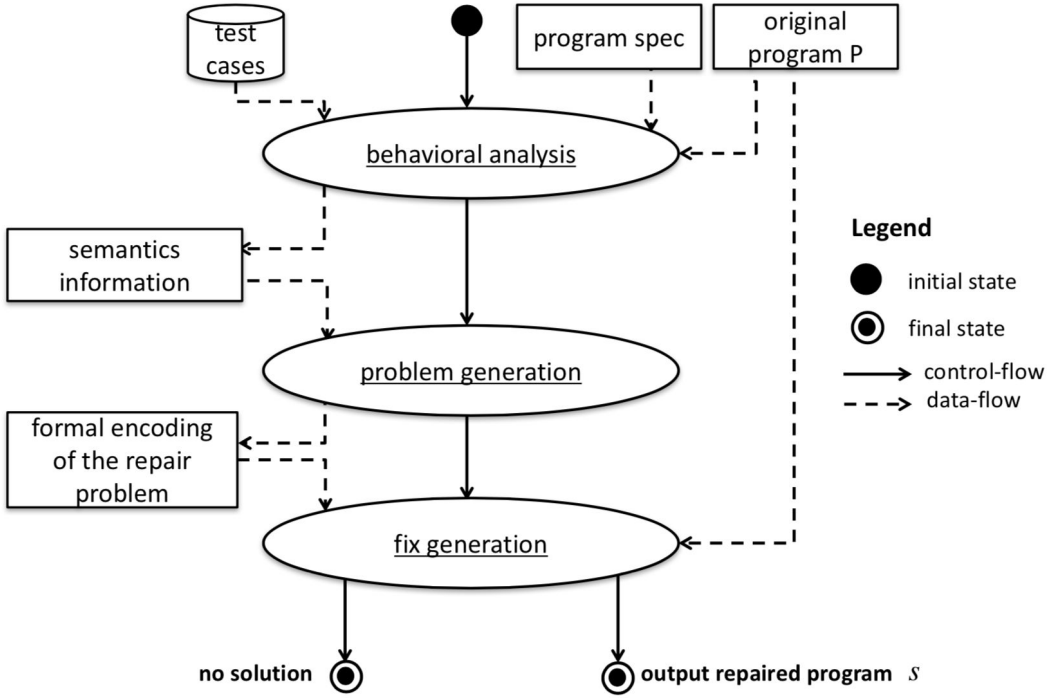

Fig. 7: Semantics-driven repair process

Fix generation | Semantics-driven

1. The **behavioral analysis** activity analyzes the program under repair to extract semantics information about the correct and faulty behaviors of the program;
2. The **problem generation** activity exploits the information collected with behavioral analysis to generate either explicitly or implicitly a formal representation of the repair problem whose solutions are code changes that represent actual fixes.
3. The **fix generation** activity tries to solve the problem generated in the previous step, either identifying the code change that may fix the program, or producing no solution in the case the solution to the repair problem does not exist or cannot be found in a reasonable amount of time.

Fix generation | Semantics-driven

1. In some cases, the problem generation and fix generation activities might be iterated considering different program locations as a target for the fix.
2. The fix generation process may also involve implicitly defining and traversing a fix space driven by the specific formulation of the repair problem.

Fix generation | Semantics-driven

1. Semantics-driven techniques often address specific classes of faults rather than being general.
2. This is because it is easier to find a formal representation of the problem P_{repair} when a specific characteristic of a program under repair is considered (e.g., the locking discipline), than trying to produce a fully comprehensive formalization of the repair problem.

Fix generation | Semantics-driven

TABLE 3: Semantics-driven techniques

Fault model	Change model	Tecqhniques	Section
general	synthesis of new expressions	SemFix, DirectFix, Angelix, SearchRepair	Sec. 6.2.2
wrong conditions and missing preconditions	condition change and if condition insertion	NOPOL, Infinitel, DynaMoth	Sec. 6.2.3
concurrency faults	critical region manipulation, parallelization keyword move	AFix, CFix, HFix, Surendran et al. Axis, Grail, Lin et al., DFixer	Sec. 6.2.4
HTML generation faults	string modification	PHPQuickFix and PHPRepair	Sec. 6.2.5
string sanitization	insertion of checks, string modification	SemRep, Yu et al.	Sec. 6.2.6
access control violations	insertion of role checks	FixMeUp	Sec. 6.2.7
memory leaks	insertion of `free()` statements	LeakFix	Sec. 6.2.8

Fix recommenders

Fix recommenders

1. Do not attempt to produce fixes, but suggest a few changes that might be operated on the software to repair the fault;
2. The recommended changes might fully describe the required fix, or some effort might be required in order to produce the final fix;
3. Do not produce an actual repair (a new version of the software that is supposed to be correct), but their output can be quickly turned into a fix in the best cases.

Fix recommenders

1. **General** techniques aim to be effective with a range of programs, not limiting their scope to a specific class of faults;
2. Other techniques:
    a. **Security faults**, which make programs vulnerable to attacks;
    b. **Data type faults**, which may cause failures due to the misuse of data structures and other types of data;
    c. **Concurrency faults**, which may cause deadlocks and other concurrency issues;
    d. **Performance faults**, which may cause unacceptable execution time for some functionalities.

Fix recommenders

TABLE 4: Fix recommender techniques

Fault model	Techniques	Section
general	BugFix, MintHint, Logozzo et al., QACrashFix	Sec. 7.1
security faults	BovInspector, Abadi et al., CDRep	Sec. 7.2
data type misuses	Coker et al., Malik et al.	Sec. 7.3
concurrency faults	ConcBugAssist	Sec. 7.4
performance faults	Selakovic et al., CARAMEL	Sec. 7.5

Obrigado!

diogom42@gmail.com