




Bugfix: a standard language, database schema and repository for research on bugs and automatic program repair

Victoria Kananchuk 
Constructor Institute of Technology
Schaffhausen, Switzerland
Viktoryia.Kananchuk@constructor.org

Ilgiz Mustafin 
Constructor Institute of Technology
Schaffhausen, Switzerland
Ilgiz.Mustafin@constructor.org

Bertrand Meyer 
Constructor Institute of Technology
Schaffhausen, Switzerland
Bertrand.Meyer@inf.ethz.ch

Abstract—¹ Automatic Program Repair (APR) is a brilliant idea: when detecting a bug, also provide suggestions for correcting the program. Over the past decades, researchers have come up with promising techniques for making APR a mainstay of software development. Progress towards that goal is, however, hindered by the absence of a common frame of reference to support and compare the multiplicity of ideas, methods, tools, programming languages and programming environments for carrying out automatic program repair.

Bugfix, described in this article, is an effort at providing such a framework: a standardized set of notations, tools and interfaces, as well as a database of bugs and fixes, for use by the APR research community to try out its ideas and compare the results on an objective basis.

The most directly visible component of the Bugfix effort is the Bugfix language, a human-readable formalism making it possible to describe elements of the following kinds: a *bug* (described abstractly, for example the permutation of two arguments in a call); a *bug example* (an actual occurrence of a bug, in a specific code written in a specific programming language, and usually recorded in some repository); a *fix* (a particular correction of a bug, obtained for example by reversing the misplaced arguments); an *application* (an entity that demonstrates how a actual code example matches with a fix); a *construct* (the abstract description of a programming mechanism, for example a “while” loop, independently of its realization in a programming language; and a *language* (a description of how a particular programming language includes certain constructs and provides specific concrete syntax for each of them — for example Java includes loop, assignment etc. and has a defined format for each of them).

The language is only one way to access this information. A JSON-based *program interface* (API) provides it in a form accessible to tools, which may record and access such information into databases. Bugfix includes such a database (*repository*), containing a considerable amount of bugs, examples and fixes.

We hope that this foundational work will help the APR research community to advance the field by providing a common yardstick and repository.

Index Terms—testing, software correctness, bugs, bug fixes, software IDEs, program verification, bug database, automatic program repair

¹An early step towards this article was a short contribution [1] to the 2024 Automatic Program Repair at the International Conference on Software Engineering (ICSE 24). The present text reuses a few elements of introduction and motivation but is otherwise thoroughly reworked and extended.

I. INTRODUCTION

Techniques of software verification, from tests to static analysis and proofs, help identify bugs; in recent years the idea has emerged that while spotting a bug is good, devising a possible *correction* for the bug is better. This approach has come to be known as “Automatic Program Repair” or APR.

In practice, Automatic Program Repair techniques usually do not actually perform changes to the program but *suggest* one or more possible corrections to the programmer, who remains in charge of choosing and applying one of them. The basic toolset that programmers use (Interactive Development Environment, or IDE) should include tools that detect bugs and, for example, pop up a message that signals the bug and proposes one or more valid corrections. The present article does not examine these (important) practical matters but concentrates on the underlying techniques making such a scenario possible.

Interest in APR techniques has developed considerably in the past two decades, starting with early work such as [2]–[7]. There is now a considerable literature; good surveys can be found in [8]–[10].

One of the major obstacles to widespread deployment and use of APR tools is the heterogeneity of the field: any solution must deal with the wide diversity of APR approaches, of IDEs, of programming languages, and also of bugs. These factors force every APR project to invest a major part of its effort in recreating a basic bug and fix description framework; such spurious repetition of work considerably impair progress towards making Automatic Program Repair a standard and effective part of the software developer’s daily experience.

The Bugfix project, described in the present article, addresses this obstacle by establishing a joint framework, Bugfix, that all APR efforts can use. The starting point for Bugfix was a short position paper [1] at the APR workshop of ICSE. The present proposal makes a number of different design choices and its description below provides enough detail to allow testing and APR projects to start using it.

It includes mechanisms for specifying, in a standard way, elements of six different kinds. The first four pertain to bugs

and their fixes:

- A **bug** element, describing a bug pattern, such as using a strict comparison ($i < n$) instead of a nonstrict-one ($i \leq n$) (section III-A).
- A **fix** pattern for a bug pattern, such as replacing the nonstrict operator by the strict version (section III-B).
- An **example** of a fix in a particular program which shows the before- and after- states of the code (section III-D).
- An **application** of a fix to an example which shows how exactly the bug pattern matches the before-state of the code and how the fix pattern transforms the code to the after-state (section III-C).

Elements of these kinds are described independently of a particular programming language (although some of them may be meaningful in some programming languages only, as in the case of bug in a “repeat... until...” loop, which can only occur in a language such as Pascal supporting a loop of that kind). Actual bugs occur in actual programming languages; the remaining two kinds of element enable Bugfix to achieve full generality by specifying the constructs in which bugs can occur and fixes can be applied:

- A **construct** element (IV-A) describes a particular programming language construct, such as the “repeat... until...” loop. Although such a description is guided by the availability of a construct in existing programming languages, it is still language-independent: it provides a structural description of the construct (an abstract syntax), by listing its components, such as an exit condition and a loop body in this case, without providing any particular concrete syntax.
- A **language** specification (IV-B) indicates how a programming language uses particular constructs. It specifies the list of constructs in the language (at least those for which bugs have been identified) and, for each of these constructs, what concrete syntax it uses. For example Java includes loops, assignments, conditionals etc. and provides a specific way of expressing each of them, such as “`target = source;`” for an assignment, where `source` and `target` are components of the corresponding (language-independent) **construct** specification.

These six kinds of element provide the core mechanisms of Bugfix. They are available in various ways. In particular:

- The description of the mechanism (in sections III and IV) uses the Bugfix **language**, a clear human-readable notation best suited for learning the concepts and discussing bugs, examples, fixes, constructs and languages.
- These mechanisms must also be usable through programs, or through queries on bug and fix databases. For that purpose, a **program interface (API)** is available, using the JSON conventions (section V). The functionality it provides is the same as the functionality of the language.
- To illustrate the approach and provide a directly available resource, we have collected a large number of bugs, examples and fixes in a publicly available **repository** (section V).

II. THE BUGFIX LANGUAGE

At the core of Bugfix lies a language for describing identified patterns for both bugs and their fixes. Two observations:

- The Bugfix effort focuses on bugs that manifest themselves through code patterns that are wrong and should be replaced (fixed) by adapting the patterns. For example, code that uses $f(a, b)$ when it should use $f(b, a)$. We do not at this point consider deeper or more elaborate bugs, such as design bugs, as analyzed for example in [11].
- To illustrate the Bugfix language, we use a concrete syntax. For clarity the syntax is keyword-oriented. Such concrete syntax details are not essential; as noted, the same functionality is available through an API (section V). Other versions of the syntax are possible.

Section III presents bug-related elements: **bug** (III-A), **fix** (III-B), **application** (III-C) and **example** (III-D). Section IV presents language-related elements: **construct** (IV-A1) and **language** (IV-B).

The Bugfix language description is informal, through examples; a full language reference will be available on the Bugfix website included as part of the Artifacts.

III. BUGS, FIXES, APPLICATIONS AND EXAMPLES

Part of the Bugfix language is dedicated to specifying bug patterns, examples of such patterns, and proposed fixes.

These descriptions are language-independent and based on the idea of a universal Abstract Syntax Tree (AST) which consists of constructs (specified per the rules of IV-A below).

A. Specifying bug patterns

A bug specification presents a certain program pattern, of which some instances have been identified as buggy (although not all instances necessarily are).

Such a specification describes the general scheme for a bug, not a particular occurrence. Defining specific occurrences is the role of *application* specifications (III-C), each of which refers to a given **bug** specification. (As seen in III-E below, it is also possible, for convenience, to specify an example together with the corresponding bug.)

The following is an example of bug specification:

```
bug WRONG_ARGUMENT_IN_CALL_1
-- In a call with an actual argument list
-- starting with 'pre' and
-- ending with 'post',
-- the actual argument between these parts
-- may not be the right one.
-- So in 'f (a, b, m, u, v)'
-- 'm' is not right.
-- Here 'initial' is 'a, b',
-- 'final' is 'u, v',
-- 'wrong_arg' is 'm'.
-- Note that 'pre' and 'post'
-- may each be empty.
where
```

```

(ARGUMENT_LIST
  (,)* @pre
  (,)* @wrong_arg
  (,)* @post)
)
end

```

The specification defines a bug pattern called `WRONG_ARGUMENT_IN_CALL_1` which matches any `ARGUMENT_LIST` node which has at least one argument (zero or more `@pre` arguments, one `@wrong_arg` argument and zero or more `@post` arguments).

The syntax used for the pattern is based on S-expressions (sexps) and is similar to the query syntax of the tree-sitter [12]. Tree-sitter is discussed more in section VI.

Each sexp starts with a name of a construct or a kind or which must match at this position. The wildcard symbol “_” can be used for matching any node.

The following elements of sexp define the children of the node. Each child definition starts with an optional construct feature name like `right:`, followed by a required sexp specifying the pattern, followed by an optional quantifier (such as “*” for 0 or more occurrences, “+” for 1 or more, “?” for 0 or 1), followed by an optional capture name like `@wrong`.

Nodes which have a specified capture name can be used in **fixes**. The top-level node always has an implicit capture name `@bug`.

B. Specifying fixes

A fix specification presents a proposed replacement for a bug pattern defined by a **bug** specification per III-A. Since a fix specification always describes a fix for a particular bug, it will refer explicitly to that bug in the **for** clause. (An alternative is to declare the fix with the bug, as explained in the next subsection.)

The fix specification uses the **parameter** clause to list the required constructs which are needed to fill the blanks in the fix; and the **then** clause to specify the replacement subtree for the whole pattern captured by the **bug**. For example:

```

fix CORRECT_ARGUMENT_IN_CALL_2
  -- In the call,
  -- replace the '@wrong_arg' argument
  -- by '@correct_arg'.
bug_id
  WRONG_ARGUMENT_IN_CALL_2
parameter
  correct_arg: EXPRESSION
then
  (ARGUMENT_LIST
    @initial
    @correct_arg
    @final)
end

```

The **feature** clause specifies that a node of type `EXPRESSION` named `@correct_arg` is used in the **then** clause.

The **then** clause defines the replacement subtree for the subtree captured by the **bug** pattern. The syntax used in the **then** part is also based on sexps: the first element is a name of the construct followed by the child definitions. Child definition starts with an optional feature name like `right:` and is followed by a sexp, a bug capture name or a fix feature name; or a string value for the leaf nodes.

An alternative syntax for might be used for splicing (splating in Ruby or spreading in JavaScript) captured nodes and fix features using the splice symbol “*”. A sexp starting with the splice symbol followed by a name is equivalent to a sexp with the same contents as the node represented by the name. Child nodes can be overridden by simply specifying them in the sexp.

Splice symbol can appear after the construct name as in `(X *@y)` and is equivalent to a sexp with the construct name set as `X` and children as in the node represented by `@y`.

C. Specifying applications

An **application** specification presents how a real bug occurrence in an **example** is captured by a **bug** pattern and how the **fix** pattern produced the fixed version of the **example**.

The **tree** clause shows how the real code is represented by the construct-tree and which nodes are captured with capture names.

The **parameter** clause specifies the values for **fix** features which lead to the fix matching the **example**.

For example:

```

application
  CORRECT_ARGUMENT_IN_CALL_1_CLOSURE_14
example
  CLOSURE_14
fix
  CORRECT_ARGUMENT_IN_CALL_1
tree
  argument_list [766, 28] – [766, 66]
    identifier [766, 29] – [766, 37] @pre
    method_invocation[766, 39] – [766, 52]
      @wrong_arg
      object: identifier [766, 39] – [766, 45]
      method: identifier [766, 46] – [766, 52]
      identifier [766, 54] – [766, 65] @post
parameter
  @correct_arg = (field_access
    object: (identifier "Branch")
    field: (identifier "ON_EX"))
end

```

The specification presents an application of the fix `CORRECT_ARGUMENT_IN_CALL_1` which matches the real code **example** `CLOSURE_14`. The bug pattern matches the `argument_list` which spans from the line 766, column 28 to line 766, column 66. The spans of the named captures (`@pre`, `@wrong_arg`, `@post`) are also given. The feature `@correct_arg` required by the fix is provided as a `field_access` node with the specific children.

D. Specifying examples

An **example** specification represents the actual code change applied to fix a bug in a code base.

The code change is represented by two versions of the code: the buggy “before” version and the fixed “after” version.

If the code base is stored using Git, Apache Subversion or another compatible version control system, then the **example** specification can specify the “before” and “after” versions as commit identifiers.

```
example CLOSURE_14
repository
  https://github.com/google/closure-compiler
before
  b7c2861bf45b358b26ebc5ee1be9b6ce96bec78a
after
  4b15b25f400335b6e2820cb690430324748372f9
language
  JAVA
hunk
diff --git a/src/com/google/javascript/jscomp/
  ControlFlowAnalysis.java b/src/com/google/
  javascript/jscomp/ControlFlowAnalysis.java
index 5c6927f9c08..980deff1df6 100644
--- a/src/com/google/javascript/jscomp/
  ControlFlowAnalysis.java
+++ b/src/com/google/javascript/jscomp/
  ControlFlowAnalysis.java
@@ -764,7 +764,7 @@ private static Node
  computeFollowNode(
    } else if (parent.getLastChild() == node){
      if (cfa != null) {
        for (Node finallyNode : cfa.finallyMap.
          get(parent)) {
- cfa.createEdge(fromNode, Branch.UNCOND,
  finallyNode);
+ cfa.createEdge(fromNode, Branch.ON_EX,
  finallyNode);
    }
  }
  return computeFollowNode(fromNode,
    parent, cfa);
end
```

E. Specifying fixes and applications together with a bug

In the preceding examples the bug, fix and application specifications are separate. There are two reasons for enabling this possibility:

- Several fixes and applications may be available for a single bug.
- For a bug specified in one place, bug fixes may be proposed in many different places, by authors who do not have the ability or permission to modify the original bug specification; the same observation applies to applications.

In some cases, however, it may be convenient when describing a bug to specify one or more fixes, as well as one or more applications, directly with it. The syntax support these possibilities by allowing one or more **fix** clauses, and one or more **application** clause, after the **where** clause of a **bug** specification. Here is a case involving a bug-with-fix specification, equivalent to the separate specifications given above, with an **application** clause also added:

```
bug WRONG_ARGUMENT_IN_CALL_2
parameter
  @correct_arg: EXPRESSION -- This is a feature
  -- of the fix
where
  (ARGUMENT_LIST
    ( )* @pre
    ( ) @wrong_arg
    ( )* @post)
fix CORRECT_ARGUMENT_IN_CALL_2
then
  (ARGUMENT_LIST
    @pre
    @correct_arg
    @post)
application
  @correct_arg = (identifier "a")
end
```

To specify more than one fix, it suffices to add more **fix** clauses, each with its name (such as `CORRECT_ARGUMENT_IN_CALL_2` here). In the same way, any number of **example** clauses may be present. While such a specification always starts with a single **bug** clause, the order of **fix** and **example** clauses, if any, is arbitrary, although the recommended order is: bug, fixes if any, examples if any.

IV. CONSTRUCTS AND LANGUAGES

The bug specifications as illustrated above, and through them the example and fix specifications, refer to language constructs. The second part of Bugfix includes a way to describe the corresponding constructs, at two levels:

- In a language-independent way: for example you may specify the concept of loop, in one of its variants such as the “while loop” available in some programming language. Such a **construct** specification indicates the components of a loop, for example the initialization part *i*, condition *c* and body *b*. Such a description does not prescribe any particular programming language.
- In language-specific way: for example, a while loop in the C language is of the form `for { i ; c ; b }` where the elements in italics correspond to the components of a particular loop.

The following two subsections describe these mechanisms.

A. Specifying constructs

A construct specification simply indicates how a particular programming construct is made up of elements.

Terminology: a “construct” is a language concept, such as `WHILE_LOOP` below. A program in the language is made up of “specimens” (instances) of those constructs. For example, a particular while loop is a specimen of the construct `WHILE_LOOP`.

1) *Describing a construct with required components:* The following example describes “while” loops:

```
construct WHILE_LOOP
kind
  INSTRUCTION
feature
  body: INSTRUCTION
  condition: EXPRESSION
  initial: INSTRUCTION
end
```

The basic idea is simply to express a construct as consisting of a number of named components, describing the abstract syntax. Here a `WHILE_LOOP` includes a body, a condition and an initialization part, of respective types `INSTRUCTION`, `EXPRESSION` and `INSTRUCTION` again. The order of their listing is irrelevant since the specification only addresses abstract syntax; order will only become significant in the concrete syntax specification for a particular language (**language** specifications, section IV-B); that will be the place to specify that, in a particular programming language, a loop has the form **from** initial **while** condition **do** body **end**.

2) *Alternation constructs:* The **kind** clause lists a construct, `INSTRUCTION`. Its semantics is that the construct being defined here, `WHILE_LOOP`, is a particular case of another construct `INSTRUCTION`; the idea is similar to inheritance in object-oriented programming (where a compiler could have a class `WHILE_LOOP` inheriting from an abstract class `INSTRUCTION`).

Such constructs are not permitted to have their own **construct** specifications but are declared contextually by appearing in the **kind** clause of at least one **construct**.

This convention is important for the flexibility of the language description since it enables each language to pick its own variants. It applies to “alternation” constructs such as `INSTRUCTION` which in traditional BNF would be defined through the specification of a set of alternatives, as in `INSTRUCTION = ASSIGNMENT | WHILE_LOOP |`

`CONDITIONAL | ...`

Such an approach would, however, defeat the Bugfix purpose of specifying programming mechanisms in a language-independent way, and then, separately, their combination and rendition in a particular programming language:

- While all imperative programming languages have some kind of instruction construct with a number of variants, they differ in their exact choice of variants. For example, while “for loops” and “while-do” loops appear in both Java and C, Java does not have the C “do-while “while loops”.
- If the Bugfix specify a construct such as `INSTRUCTION` by listing its alternatives, that construction would need

many different variants, such as `C_INSTRUCTION` and `JAVA_INSTRUCTION`. But then it would become impossible to define other constructs such as `WHILE_LOOP` since some of their components such as `body` must refer to `INSTRUCTION`.

- The Bugfix solution is instead to consider such alternation constructs as defined simply by their appearance in at least one **kind** clause for a **construct** specification.
- For a particular programming language, the alternation is defined as a choice between all such constructs included in the **language** specification (section IV-B).

With this convention, an alternation construct such as `INSTRUCTION` becomes a construct without an explicit specification, simply by appearing in **kind** clauses for some other constructs.

3) *Optional and list components:* The following example completes the illustration of how to specify constructs in Bugfix, by introducing two other facilities: optional and list components.

```
construct GENERAL_CONDITIONAL
kind
  -- A conditional instruction
  -- with a "then" part,
  -- 0 or more of "else if"
  -- and an optional "else" part.
  INSTRUCTION
feature
  condition: EXPRESSION
  then_part: CONDITION_INSTRUCTION
  else_if*: CONDITION_INSTRUCTION
  else_part?: INSTRUCTION
end

construct CONDITION_INSTRUCTION
kind
  -- A <condition, instruction> pair.
  INSTRUCTION
feature
  condition: EXPRESSION
  instruction: INSTRUCTION
end
```

The question mark “?” after a component name, as in `else_part?`, indicates that the component is optional: a specimen of the construct may, or not, include a specimen of that component.

An asterisk after a component name denotes a sequence (list): here the `else_if*` component consists of zero or more specimens of `CONDITION_INSTRUCTION`. Since zero specimen is a possibility, the component may be absent. To specify one or more, use “+” instead of “*”.

The first construct definition above defines a “general conditional” as consisting of: an expression, called `condition`, required; an expression, the `then_part`, also required; zero or more “else if” parts, each made of a condition

and an instruction (as specified by the auxiliary construct `CONDITION_INSTRUCTION`); and an optional “else part”.

B. Specifying languages with existing grammars

To define a language in a very formal way, existing Tree-sitter grammars can be used. The approach for defining a language this way is similar to defining bug-fix pair in Bugfix.

Each language entity is defined using 4 different kinds: a *source* (the Tree-sitter query in the concrete language grammar); a *construct* (the pattern for constructing the Bugfix **construct**); a *construct_name*; a *language*.

For example, the following code extract shows how `ARRAY_ACCESS` construct can be extracted from Java and Python programs:

```
language
  JAVA
construct_name
  ARRAY_ACCESS
source
  ASSIGNMENT_INSTRUCTION:
  (ARRAY_ACCESS
   array: ( ) @array
   index: ( ) @index)
construct
  (ARRAY_ACCESS
   array: @array
   index: @index)
end

language
  PYTHON
construct_name
  ARRAY_ACCESS
source
  (SUBSCRIPT
   value: ( ) @array
   subscript: ( ) @index)
construct
  (ARRAY_ACCESS
   array: @array
   index: @index)
end
```

The nodes captured in the query pattern and used in the construction pattern are translated recursively.

V. BUGFIX WEBSITE AND API

The Bugfix specifications are available as a website at <https://icbf813.github.io/> which has both the human-readable pages and the machine-readable JSON API. The website’s data takes advantage of the already cited earlier sketch, [1].

The website is built using the static website generator Jekyll [13] which generates both the HTML code for the webpages and the JSON files for the API. The input for the generator is a collection of Jekyll data files which contain specifications in the Bugfix language format. This allows

the website to be used for exchanging information related to different projects.

Both the API and the webpages provide the same data, so the following description focuses on the API view.

The website has a Jekyll collection for each kind of Bugfix elements. Each collection has the index page which lists all elements of this collection. For example, the `/bugs` page lists all **bug** elements present in the database. This page and all other pages have the JSON API counterpart which is located at the same path but with the `.json` extension appended (in this case it is `/bugs.json`). Same for `/fixes`, `/applications`, `/examples`, `/constructs`, `/kinds` and `/languages` for language elements (the full list of collections is visible on the website).

Each element has an ID. Index JSON routes respond with a JSON object which has element IDs as keys and the main fields of the elements as values.

The extended information of each element is available on the element’s page in the collection. The path to the element’s page is constructed by appending the element’s ID as a path segment. For example, one of the bugs in the database has an ID `null_check_missing_1`, so the page of the bug is located at `/bugs/null_check_missing_1` and the JSON API is located at `/bugs/null_check_missing_1.json`.

The fields of the JSONs reflect the clauses of the Bugfix specifications of the respective element kind. The clauses containing the sexp-based patterns are available both as a string and as JSON objects to simplify the creation of tools using Bugfix. For example, the JSON of the bug `NULL_CHECK_MISSING_1` has the `"where"` field for the parsed **where** pattern and the `"where_raw"` field for the raw string with the S-expressions:

```
{ "id": "wrong_return_statement_1",
  "where": {
    "type": "return_statement",
    "children": [{
      "field": "return_value",
      "node": { "type": "true" },
      "name": "@wrong_value" } ] },
  "where_raw":
    "(return_statement return_value: (true)
     @wrong_value)",
  "fixes": [ "correct_return_statement_1" ] }
```

The webpage versions of the pages include hyperlinks in the pattern fields for the ease of navigation.

VI. RELATED WORK

The most important precursor to the design presented here is the important work pursued by several groups, over the past decades, to provide repositories of bugs and bug fixes; this concerted effort is a prerequisite to any attempt at enabling teams working on Automatic Program Repair to assess and compare their methods. Some of the most significant such repositories are the following [11], [14]–[21].

Another precursor is the work on code analysis tools. Boa [22] is a domain-specific language and infrastructure for analysing software repositories. Boa has a common AST structure for Java, Python and Kotlin programs present in the Boa datasets. To analyze the AST of programs in other languages programs must be translated into Boa AST, this was done in [23] for Python. By default, Boa AST is not extendible. A separate project Boidae [24] allows extensions but they must be done directly in the Java source code of the system.

Research in multilingual code analysis can use several distinct AST structures such as [25] which combines results of several parsers and [26], [27] in which the Semantic [26] does code analysis on different ASTs produced by different tree-sitter parsers. Another approach is to to analysis on a pre-existing common AST as in [28] which uses the Boa AST for Java, Python and Kotlin.

A short paper [1] was to our knowledge the first to propose a possible framework for describing bugs and fixes. It is, however, very short and more of a position paper, whose results are not directly usable; in addition, unlike the present work, it does not provide any directly usable resources.

ASTs are produced by parsers according to specified grammars. Parser generators such as ANTLR [29] and others have been used both in language compilers and in research [30]. Tree-sitter [12] is a parser generator which has language grammars available for many languages. Tree-sitter is used in code editors and code analysis research [25], [27], [31]. Another feature of tree-sitter is its query language. The query language allows specifying subtree patterns which are found in the parsed AST.

VII. ASSESSMENT, LIMITATIONS, CONCLUSIONS AND FUTURE WORK

Departing from the usual practice of describing specific techniques in testing and Automatic Program Repair, the present work positions itself at what could be called a “meta” level: it abstracts from individual research projects to provide instead a set of concepts, tools and resources for everyone working in the field.

The authors’ own evaluation has consisted of considering all the bug and fix repositories which we could get hold off, more specifically all the repositories covering *coding* bugs and fixes. and painstakingly ensuring that the notation and tools could cover all their contents simply and effectively.

By that measure, the effort is successful, since we have Bugfix descriptions for all these examples from the literature and widely used repositories. Full validation can only come from widespread use by researchers and practitioners working in complete independence from the authors.

While it would be desirable to cover more sophisticated kinds of bugs, such as design bugs, the restriction to the coding case seems justified think such further advances will only possible if we have a fully satisfactory solution to this application, covering the bugs that are best understood and on which APR techniques have produced their most significant results so far.

The contributions to the advancement of Automatic Program Repair described in this article include:

- A standard language for describing *bugs and their fixes*.
- Supporting it, a standard language for describing *programming constructs* from virtually any programming language.
- For these languages, use of *widely accepted techniques*, tree-sitter parsing and querying access to the information in a standard way (unlike the approach of [1] which only used new notations).
- Along with all these (human-oriented) notations, *API (programming interface) versions* making the functionality accessible to any production or research tool.
- The associated *database mechanisms*.
- An *actual database*, publicly available, providing a repository of important bugs and fixes collected from a wide variety of sources.
- A working *website* providing access to all this information, both in human-readable HTML and machine-readable JSON.
- For all these applications, an *open, modular approach* making it possible to add new constructs, new languages, new kinds of bugs and fixes and new data sets in a flexible way.

The present work should be viewed as a **community-service contribution**: it is meant to help everyone interested in bugs and their automatic repair, particularly researchers in the field, to achieve new advances by relieving them of common and repetitive tasks, facilitating the assessment of new techniques, and making it possible to compare the effectiveness of competing or complementary approaches by providing a common reference.

Unlike previous attempts, which were just sketches (albeit in the right direction), the proposal presented here, while still incomplete, provides enough information (in particular, the notations and API) and resources (in particular, the repositories) to enable actual use by bug-analysis and Automatic Program Repair projects.

We have made every effort to make these tools and resources attractive and useful to the community, and will be happy to adapt them in response to user feedback. The most significant evaluation will come from such use.

Just as understanding bugs is fundamental to software engineering, progress in Automatic Program Repair is fundamental to the progress of practical software development. We strongly hope that the community will find Bugfix a valuable resource and that it will provide a new impetus to further advances in the field.

REFERENCES

- [1] B. Meyer, V. Kananchuk, and L. Huang, “Bugfix: towards a common language and framework for the automaticprogram repair community,” in *APR Workshop at ICSE 2024 (International Conference on Software Engineering)*, Lisbon, 14-20 April 2024. ACM and IEEE, 2024. [Online]. Available: <https://arxiv.org/abs/2402.14471>

- [2] K. Liu, A. Koyuncu, K. Kim, D. Kim, and T. F. Bissyandé, “Lsrepair: Live search of fix ingredients for automated program repair,” in *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2018, pp. 658–662.
- [3] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, “Semfix: Program repair via semantic analysis,” in *International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 772–781.
- [4] J. Xuan, M. Martinez, F. DeMarco, M. Clément, S. L. Marcote, T. Durieux, D. Le Berre, and M. Monperrus, “Nopol: Automatic repair of conditional statement bugs in java programs,” *Transactions on Software Engineering*, vol. 43, no. 1, pp. 34–55, 2017.
- [5] Y. Pei, C. A. Furia, M. Nordio, Y. Wei, B. Meyer, and A. Zeller, “Automated fixing of programs with contracts,” *Transactions on Software Engineering*, vol. 40, no. 5, pp. 427–449, 2014.
- [6] Y. Pei, C. A. Furia, M. Nordio, and B. Meyer, “Automated program repair in an integrated development environment,” in *International Conference on Software Engineering (ICSE)*, vol. 2. IEEE, 2015, pp. 681–684.
- [7] R. van Tonder and C. L. Goues, “Static automated program repair for heap properties,” in *International Conference on Software Engineering (ICSE)*. ACM, 2018, p. 151–162.
- [8] M. Monperrus, “Automatic software repair: A bibliography,” *ACM Computing Surveys (CSUR)*, vol. 51, no. 1, pp. 1–24, 2018.
- [9] —, “The living review on automated program repair,” Ph.D. dissertation, HAL Archives Ouvertes, 2018.
- [10] L. Gazzola, D. Micucci, and L. Mariani, “Automatic software repair: A survey,” in *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 1219–1219.
- [11] G. Catolino, F. Palomba, A. Zaidman, and F. Ferrucci, “Not all bugs are the same: Understanding, characterizing, and classifying bug types,” *Journal of Systems and Software*, vol. 152, pp. 165–181, 2019.
- [12] M. Brunsfeld, A. Qureshi, A. Hlynskyi, P. Thomson, J. Vera, ObserverOfTime, P. Turnbull, dundargoc, T. Clem, D. Creager, A. Helwer, R. Rix, D. Kavolis, H. van Antwerpen, M. Davis, Ika, T.-A. Nguyễn, A. Yahyaabadi, S. Brunk, M. Massicotte, bfredl, N. Hasabnis, C. Clason, M. Dong, S. Moelius, S. Kalt, W. Lillis, S. Finer, and Kolja, “tree-sitter/tree-sitter: v0.23.0,” Aug. 2024. [Online]. Available: <https://doi.org/10.5281/zenodo.13375512>
- [13] T. Preston-Werner *et al.*, “Jekyll,” Dec. 2023. [Online]. Available: <https://github.com/jekyll/jekyll/tree/v4.3.3>
- [14] E. C. Campos and M. de Almeida Maia, “Common bug-fix patterns: A large-scale observational study,” in *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 2017, pp. 404–413.
- [15] M. Soto, F. Thung, C.-P. Wong, C. Le Goues, and D. Lo, “A deeper look into bug fixes: patterns, replacements, deletions, and additions,” in *Proceedings of the 13th International Conference on Mining Software Repositories*, 2016, pp. 512–515.
- [16] B. Livshits and T. Zimmermann, “Dynamine: finding common error patterns by mining software revision histories,” *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5, pp. 296–305, 2005.
- [17] C. C. Williams and J. K. Hollingsworth, “Automatic mining of source code repositories to improve bug finding techniques,” *IEEE Transactions on Software Engineering*, vol. 31, no. 6, pp. 466–480, 2005.
- [18] X. Sun, T. Zhou, G. Li, J. Hu, H. Yang, and B. Li, “An empirical study on real bugs for machine learning programs,” in *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2017, pp. 348–357.
- [19] J. A. Duraes and H. S. Madeira, “Emulation of software faults: A field data study and a practical approach,” *Ieee transactions on software engineering*, vol. 32, no. 11, pp. 849–867, 2006.
- [20] D. Lin, J. Koppel, A. Chen, and A. Solar-Lezama, “Quixbugs: A multi-lingual program repair benchmark set based on the quixey challenge,” in *Proceedings Companion of the 2017 ACM SIGPLAN international conference on systems, programming, languages, and applications: software for humanity*, 2017, pp. 55–56.
- [21] V. Sobreira, T. Durieux, F. Madeiral, M. Monperrus, and M. de Almeida Maia, “Dissection of a bug dataset: Anatomy of 395 patches from defects4j,” in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2018, pp. 130–140.
- [22] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen, “Boa: Ultra-large-scale software repository and source-code mining,” *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 25, no. 1, pp. 1–34, 2015.
- [23] S. Biswas, M. J. Islam, Y. Huang, and H. Rajan, “Boa meets python: A boa dataset of data science software in python language,” in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 2019, pp. 577–581.
- [24] B. Sigurdson, S. W. Flint, and R. Dyer, “Boidae: Your personal mining platform,” in *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*, 2024, pp. 40–43.
- [25] S. Nielebock, R. Heumüller, and F. Ortmeier, “Programmers do not favor lambda expressions for concurrent object-oriented code,” *Empirical Software Engineering*, vol. 24, pp. 103–138, 2019.
- [26] R. Rix, J. Vera *et al.*, “Semantic,” May 2024. [Online]. Available: <https://github.com/github/semantic/tree/ad281b52e3a399ff88126e3d06063c48eb245135>
- [27] T. Clem and P. Thomson, “Static analysis at github: An experience report,” *Queue*, vol. 19, no. 4, pp. 42–67, 2021.
- [28] A. M. Keshk and R. Dyer, “Method chaining redux: an empirical study of method chaining in java, kotlin, and python,” in *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*. IEEE, 2023, pp. 546–557.
- [29] T. J. Parr and R. W. Quong, “ANTLR: A predicated-LL(k) parser generator,” *Software: Practice and Experience*, vol. 25, no. 7, pp. 789–810, 1995.
- [30] A. Latif, F. Azam, M. W. Anwar, and A. Zafar, “Comparison of leading language parsers-antlr, javacc, sablecc, tree-sitter, yacc, bison,” in *2023 13th International Conference on Software Technology and Engineering (ICSTE)*. IEEE, 2023, pp. 7–13.
- [31] Q. Le Dilavrec, D. E. Khelladi, A. Blouin, and J.-M. Jézéquel, “Hyperast: Enabling efficient analysis of software histories at scale,” in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–12.