Collaborative Development Environment: Fusing Human Insights into Automated Program Repair

Chunmiao Li
Shanghai Jiao Tong University
lichunmiao@sjtu.edu.cn

Yijun Yu
The Open University, UK
y.yu@open.ac.uk

Zhenjiang Hu
National Institute of Informatics
hu@nii.ac.jp

1 BACKGROUND

Current integrated development environments (IDE) facilitate programmers to create and debug programs by providing them with a cohesive suite of software development utilities that increase programming productivity with well-designed user interfaces. Such an environment allows a programmer to spot the locations of bugs from instant feedback upon errors. Currently, the task of bug repair is for programmers to discover and patch bugs manually. Although some advices to fix defects are given by IDE as recommendations, they have to understand these suggestions before taking on those countermeasures that might introduce additional bugs [1].

Without human participation, automated bug repair [4] has been proposed to automatically produce correct patches to fix bugs. These techniques use regression tests to force the bugs to reoccur, in order to validate the generated candidate patches by either a formal specification or regression test cases [6]. However, incomplete test cases could lead to incorrect patches, which may eliminate desirable functionalities or may cause security vulnerabilities [7]. Another limitation to automated bug repair is search space explosion [5] due to the assumption that a small modification is enough to fix bugs by trying all the small modifications as potential patches.

2 MOTIVATION

We propose a collaborative development environment (CDE) to support a new paradigm where programmers and bug repair tools can work together to discover and patch bugs. In this scenario, human can focus on front-end programs and tools can operate on back-end structures such as abstract syntax tree (AST). Bug fixing is composed of multi-session collaborative communications between human and machine. In each session, tools can process the ASTs and modify it, then this modification should be reflected to source code so that programmers can recognize them and continue to patch on programs by hand. CDE utilizes the combined work of human and tools to explore the bug repair measures.

Instead of having programmers discover patches manually, CDE aims to find any opportunity of automation that bug repair tools could achieve, whilst leaving the remaining manual work to programmers so that they do not need to understand some elusive hints given by traditional IDE. If the mechanism is made right, the incorrect patches and search space explosion problems could be avoided since programmers can choose which part should be disposed by the tools directly and verify the patches.

3 THE PROBLEM TO SOLVE

To achieve the proposed CDE-supported development paradigm, at least the following two requirements should be satisfied.

- **RQ1.** The operation results produced by tools should be understandable by programmers, i.e., the changes made by tools should be reflected to the source code so that human can recognize those changes directly in programs and decide what to do next;
- **RQ2.** Tools should quickly understand human modifications on the source code in the corresponding ASTs by parsing the modified programs as fast as possible. It would be better that AST only contain those essential parts for the automated tools, e.g., a bug localization tool may rerun only when a suspicious method or a class is changed.

For an example, *Heartbleed* is a security bug found in the OpenSSL C library (CVE-2014-0160)[1], which was caused by a buffer overrun where a pointer accesses an element beyond the allocated array bounds. The bug exists between line 1454 and 1522, as shown in Figure 1. This bug had existed for more than 2 years till it was caught by program analysis tools. With CDE development paradigm, human could have spotted such changes intuitively from the ASTs in the programs produced by static analysis tools.

```
1454        int
1455        dtls1_process_heartbeat(SSL *s)
1456        {
1457        unsigned char *p = &s->s3->rrec.data[0], *pl;
1458        unsigned short hbtype;
1459        unsigned int payload;
1460        unsigned int padding = 16; /* Use minimum padding */
1461
1462        if (s->msg_callback)
1463        s->msg_callback(0, s->version, TLS1_RT_HEARTBEAT,
1464        &s->s3->rrec.data[0], s->s3->rrec.length,
1465        s, s->msg_callback_arg);
    ⋮
1521        return ret;
1522        }
```

Figure 1: A snippet of d1_both.c

4 THE CHALLENGES

To satisfy **RQ1**, there should be a transformation from AST to source code so that the changes made by automated tools can be recognized by humans. A naïve idea is to design this transformation as a printing function independent from the transformation that parses source code into ASTs. However, the source code and related AST should be kept consistent in all changes so that the updated AST can be reproduced by parsing the source code again after those modifications of AST have been reflected onto the source code, i.e., the parsing and printing transformations should satisfy the *round-tripping* laws (i.e., *well-behaveness*) which are as follows, where[2]:

- GET-PUT: put source (get source) = source

[1] https://github.com/openssl/openssl/blob/96db902/ssl/d1_both.c
[2] The functions get and put refer to parsing and printing transformations, respectively

- PUT-GET: get (put source AST′) = AST′

Intuitively, the GET-PUT law states that no extra update should be performed back on the source code when there is no change on the AST, while PUT-GET law hints that put should apply all changes of AST to the source code so that ASTs can be regenerated from the updated source code by get. Two uni-directional transformations cannot promise the *well-behaveness*.

To fulfill **RQ2**, the format and structure of AST should be designed carefully. Currently, ASTs often appear as XML format, e.g., in srcML [2]. The size of this XML ASTs is often bigger than that of source code. XML ASTs preserve all the concrete syntax of the original program, including the layout, which is not actually needed for the task of *bug localization*.

5 BASIC IDEAS

We use *bidirectional transformations* (BXs hereafter) to resolve **RQ1** and propose a binary format ASTs to meet **RQ2**.

5.1 Bidirectional Transformations between Source Code and AST

The theory of BXs was proposed to synchronize two parts of related information with different data representations (i.e., source and view). A BX consists of a pair of forward and backward transformations. A forward transformation (i.e., get) extracts elements from the source to build an abstract view, and the backward transformation (i.e., put[3]) embeds information of the view back into the source and produces an updated source. This pair of transformations should satisfy the *round-tripping* laws. BX allows that view can only contain partial elements of the source. In our case, it benefits us to define partial AST which only contain the essential parts for individual analysis tools. To support the **RQ1**, we aim to implement the BX between a program and AST.

5.2 Binary AST

We apply protocol buffers (PB hereafter) [3] to build a basic schema fitting into the structure of source code. Since PB files can be efficiently encoded into binary bytes streams, the AST represented by PB is called binary AST (bAST hereafter). The textual bAST relating to the code in Figure 1 can be seen from Figure 2. bAST can gain better performance than XML AST especially in parsing from binary bytes stream. Note that one can always encode the textual bAST into a binary bytes sequence by using the API provided by PB.

5.3 Overall architecture

Combining the solutions relating **RQ1** and **RQ2**, we designed a series of BXs as depicted in Fig 3. It describes the BX between source code and bAST. Here srcML can perform the transformation between source code and XML AST without loss of any data, and PB's built-in methods deal with encoding and decoding between textual and binary forms. A common AST format is added as an intermediate between XML AST and textual bAST, which connects the BXs with these two parts.

Specifically, the BXs in Figure 3 are illustrated in detail as follows.

[3]Note here put is not a simple inverse of get. Instead, it accepts the view and the original source as inputs and produces an updated source as output.

```
1          child {
2                  kind: FUNCTION
3                  child {
4                          kind: TYPE
5                          child {
6                                  kind: NAME
7                                  text: "int"
8                          }
9                  }
10                 child {
11                         kind: NAME
12                         text: "dtls1_process_heartbeat"
13                 }
59         ...
60         }
```

Figure 2: A textual dump of the bAST for the function signature `int dtls1_process_heartbeat(SSL *s)`.

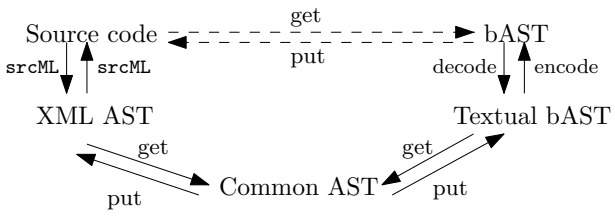

Figure 3: Bidirectional Transformations between source code and bAST

- For the forward *get* transformation, we first use srcML to parse source code to XML form, then run the BX transformation on the XML ASTs to obtain the common AST. After that, another BX transformation between common AST and textual bAST is executed to put the information stored on the former back to the latter. Lastly, we encode the textual bAST into the bAST in binary bytes stream so that the operations can be concluded in this sequence: srcML → get → put → encode.
- For the backward *put* transformation, the actions can be seen as the inverse of *get*. The operations are denoted in this sequence: decode → get → put → srcML.

Since srcML, BX programs, encoding and decoding all satisfy *round-tripping* laws, one is confident that the consistency between source code and bAST in all changes can be firmly promised.

6 OUR CONTRIBUTIONS

We propose a collaborative development environment (CDE) which divides development processes into multiple (potentially interactive) edit sessions, shared among programmers and automated analysis tools. To support the paradigm, we present binary AST, represented by protocol buffers, for tools to manipulate and we design *bidirectional transformations* (BX) to let programmers recognize the modifications made by the tools on familiar source code form. Through BX, the binary ASTs only contain partial elements of the source code needed by the task so that its size can be smaller than that of source code. Using bug localization as an example, we illustrate that programmers can fix a notorious bug collaboratively with static analysis tools.

REFERENCES

[1] Titus Barik, Justin Smith, Kevin Lubick, Elisabeth Holmes, Jing Feng, Emerson Murphy-Hill, and Chris Parnin. 2017. Do developers read compiler error messages?. In *Proceedings of the 39th International Conference on Software Engineering*. IEEE Press, 575–585.

[2] Michael L. Collard, Michael John Decker, and Jonathan I. Maletic. 2011. Lightweight Transformation and Fact Extraction with the srcML Toolkit. In *11th IEEE Working Conference on Source Code Analysis and Manipulation, SCAM 2011, Williamsburg, VA, USA, September 25-26, 2011*. IEEE Computer Society, 173–184.

[3] Google. 2017. Protocol Buffers, Google's data interchange format. (2017). https://github.com/google/protobuf.

[4] Mark Harman. 2010. Automated patching techniques: the fix is in: technical perspective. *Commun. ACM* 53, 5 (2010), 108–108.

[5] Fan Long and Martin Rinard. 2016. An analysis of the search spaces for generate and validate patch generation systems. In *Proceedings of the 38th International Conference on Software Engineering*. ACM, 702–713.

[6] Fan Long and Martin Rinard. 2016. Automatic patch generation by learning correct code. In *ACM SIGPLAN Notices*, Vol. 51. ACM, 298–312.

[7] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. 2015. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. ACM, 24–36.