

Bugfixing using valid examples

Software Verification course team project
Faculty of Mathematics, Belgrade

Ivan Ristović, Milana Kovačević, Strahinja Stanojević

September 2018.

Abstract

We have been tasked with creating a program which will compare two given code snippets - one of which will be used as a specification whereas the other one will semantically differ from the specification - and synthesize a new code snippet which will use the “invalid” snippet as the base, but edited to fit the specification. Since it has already been proven that the semantical code comparison problem is undecidable (Halting problem), there is no way to successfully give an answer for all code snippets. However, some subsets of the code snippet space can be tested and even though in some cases we cannot say for certain if what we have found is a bug or not we can provide a false-positive warning. The algorithm we use to compare the two requires matching and traversing the abstract syntax trees of both codes. For the task of matching code elements we are using GumTree API and as for the traversal we decided to use the JDT Core DOM API which unfortunately limits us to only Java code snippets.

Contents

1 Problem formulation	2
2 Assumptions and limitations	3
3 Abstract Syntax Trees	4
4 Additional tools used	5
5 The analysis algorithm	5
References	5

1 Problem formulation

Instead of writing a long formulation of the problem at hand we will rather provide a series of examples which will serve as an illustration and hopefully give the reader enough to grasp the matter without formal definitions¹.

Example 1.1. *Let's take a look at the following code snippets:*

```
int foo1()                int foo2()
{                          {
    int x = 1, a = 2;      int x = 0, a = 2;
    return x + a;          return x + a;
}
```

These two snippets are clearly semantically different, because the `foo2` function has different value of `x` variable than function `foo1`. If we wish to synthesize a fix for the given example, we would just replace the initializer for variable `x` in function `foo2` from 0 to 1. \square

We have seen a very simple example which has a simple solution. The problem, however, gets much more complex as new code structures emerge, such as branching and loops. Also, reader might have noticed that semantical equivalence has nothing to do with syntactic equivalence.

Example 1.2. *Suppose we are given two code snippets, as before:*

```
int fooEq1()              int fooEq2()
{                          {
    int x = 1;              return 4;
    int a = 2;              }
    return x + a + 1;
}
```

In this example the syntactic difference is enormous, seeing as that the second function has no variables declared and has a single statement, whereas the first one has variables defined and has three statements. Both functions, however, evaluate to the same result: 4. \square

One might think that in most cases comparing the return values should be enough. While that might be true in most cases, the following example proves that in general this statement does not hold.

Example 1.3. *Side effects like these have an enormous impact on the difficulty of the problem at hand:*

```
int fooSideEff()          int foo()
{                          {
    System.out.println     return 1;
        ("Hello!");
    return 1;              }
}
```

¹Some of the definitions would include semantical equivalence of the two code snippets, which would most probably be hard to formulate and even then would probably be ambiguous.

There is no way to determine for any external function whether it has a side effect or not since we do not have access to its source code. Therefore, side effects (as well as some other constructs) have been excluded from the analysis - we will specify in the following chapters exactly what code constructs have been excluded or what pre-assumptions were made about the given code snippets. \square

Example 1.4. Examples like these though, can be checked and verified to be equivalent even though there are side effects present in the function `anotherFooEq2`.

```
int anotherFooEq1()      int anotherFooEq2()
{
    int x = 0;           {
    x += 3;               int a = 0;
    return x;             a++;
                          a++;
                          ++a;
                          return a;
    }
}
```

The behavior of `a++` and `++a` is exactly the same as long as the value of that expression is not used in that same statement. \square

Example 1.5. We should note that, in general, there is no way of telling whether the execution will reach certain branches of the code due to the lack of determinism if we are to include the user input into the problem:

```
int ambFoo1(int x)      int ambFoo2(int y)
{
    if (x >= 0)           {
        return 1;         return y % 2;
    }                     }
    else
        return 0;
}
```

These functions are definitely semantically different, but the fact that they might produce the same result for different inputs creates an additional problem. Therefore, we have assumed complete determinism - in other words all variables have to be initialized before their use. \square

In the following chapters we will extend the example set with more complicated examples and describe our approach of semantical testing. We will also be using AST representation (see chapter 3) instead of plain code analysis.

2 Assumptions and limitations

In the previous chapter we have shown how certain code structures (loops for example) can be hard to analyze. Therefore we have set a certain number of assumptions on the input code snippets:

- All variables have to be initialized before their use
- All functions are assumed not to have side effects
- Branching condition value needs to be known at compile-time

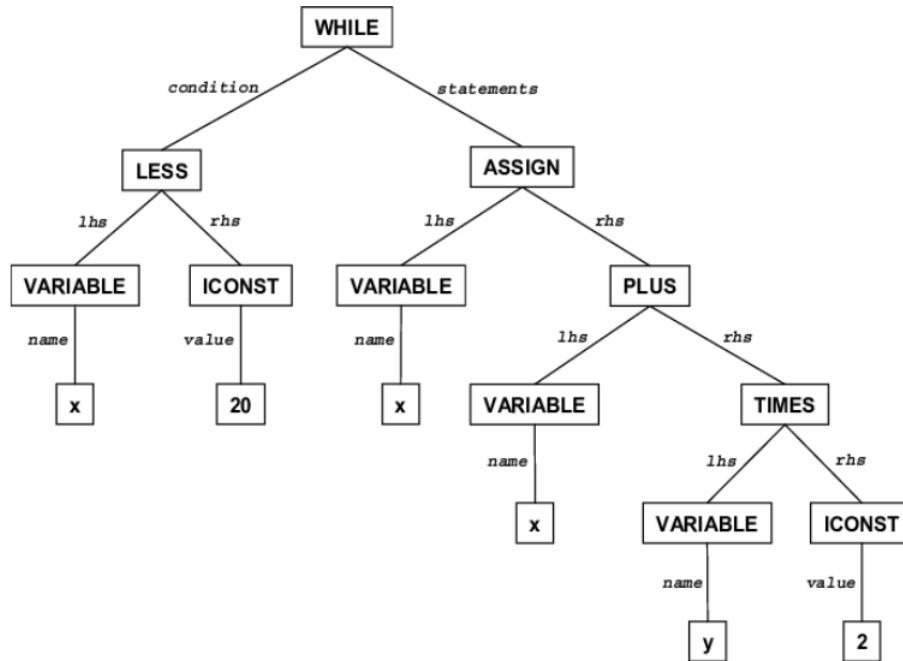


Figure 3.1: AST for the while loop

Apart from these assumptions, we have ignored the following in order to make the problem easier:

- All data types except int
- All code constructs except declarations, assignments, branching statements and arithmetic operators

3 Abstract Syntax Trees

Instead of analyzing the code on the higher level or using some internal representation between assembly and high-level code, we have decided to use the *Abstract syntax trees* (AST from now on).

Definition 3.1. An *abstract syntax tree* is a tree representation of the abstract syntactic structure of source code written in a programming language. Each node of the tree denotes a construct occurring in the source code.

Example 3.2. Consider the following code snippet:

```

while (x < 20) {
    x = x + y * 2
}
  
```

The appropriate AST can be seen on figure 3.1.

Using the AST instead of the pure code has many advantages. Namely, different code snippets might have the same AST representation which itself does the job of semantical analysis. This is a rare case though, however it is still easier to compare the trees instead of code samples

because the tree structure does not contain redundancies like whitespace for example. Also, there exist libraries for AST creation which we use and describe in the following chapter.

4 Additional tools used

We are using two external APIs in order to create/manipulate the code AST:

- GumTree² API [1]
- Eclipse JDT Core API [3]

We use GumTree to create and analyze the ASTs. Via GumTree API it is possible to find mappings between the two trees. Using those mappings we can detect similarities in the trees such as variable renames or whole matching subtrees.

Although GumTree API gives us enormous possibilities, it does not give us a lot of information regarding the specific nodes in the trees. In other words, we may know that the certain part of the code from the first tree has been removed in the second tree, but we can not know whether it means that the trees are equivalent or not (as shown in example 1.2).

For more in-depth traversal of the AST we use the Eclipse JDT Core API (from now on referred to as *JDT API*). Using the JDT API we can define actions which are executed during the visit of a certain node in the AST. More details on the traversal algorithm can be found in the following chapter.

5 The analysis algorithm

References

- [1] Gومتree. on-line at <https://github.com/GumTreeDiff/gومتree>.
- [2] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. Fine-grained and accurate source code differencing. In *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*, pages 313–324, 2014.
- [3] The Eclipse Foundation. Eclipse jdt api. on-line at <https://help.eclipse.org/neon/index.jsp?topic=%2Forg.eclipse.jdt.doc.isv%2Freference%2Fapi%2Forg%2Feclipse%2Fjdt%2Fcore%2Fdom%2FASTVisitor.html>.

²Interested reader can take a look at an excellent article [2] about code differencing written by GumTree authors