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 indecisive (Halting problem), there is no way to successfully give an answer for all code snippets. However, some subsets of the code snipper 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 For the task of matching code elements we are using GumTree API and as for the AST part we decided to use the JDT Core DOM API which unfortunately limits us to only Java code snippets.

Contents

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 definitions [1]. First, let's take a look at the following two 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.

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 equivallence has nothing to do with syntactic equivallence. 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. One might think that in most cases comparing the return values should be enough, however the following example proves otherwise:

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

Side effects like these have an enormous impact on the difficulty of the problem at hand. 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 it's source code. Therefore, side effects (as well as some other constructs) have been excluded from the analysis - we will specify in the following

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

chapters exactly what code constructs have been excluded or what pre-assumptions were made about the given code snippets.

Let's take a look at another example of equivallent code snippets:

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

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

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

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)
{
    if (x >= 0)
        return 1;
    else
        return 0;
}
```

```
int ambFoo2(int y)
{
    return y % 2;
}
```

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.

In the following chapters we will extend the example set with more complicated examples and describe our approach of semantical testing.

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

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.

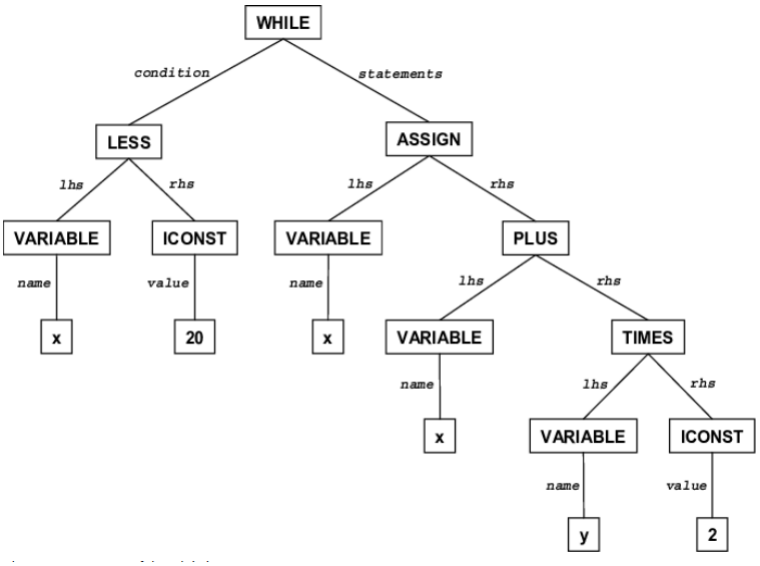

Figure 3.1: AST for the while loop

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.

4 Additional tools used

[1]

5 The analysis algorithm

References

[1] 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.