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 the entire code snippet space. However, some subsets of aforementioned 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 code snippets. 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 **4**

3 Abstract Syntax Trees **4**

4 Additional tools used **4**

5 The analysis algorithm **5**

6 Conclusion and future work **8**

References **9**

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 information to grasp the matter without formal definitions [1].

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

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

```
int foo2()
{
    int x = 0, a = 2;
    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 pair, 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 similarity has nothing to do with syntactic similarity.

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

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

```
int fooEq2()
{
    return 4;
}
```

In this example the syntactic difference is enormous, seeing as 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 sufficient. 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()
{
    println("Hello!");
    return 1;
}
```

```
int foo()
{
    return 1;
}
```

[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.

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

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

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

```
int anotherFooEq2()
{
    int a = 0;
    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. In this case, it is equivallent to* `a += 1`. □

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)
{
    return x >= 0
        ? 1
        : 0;
}
```

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

These functions are definitely semantically different, but if the caller function always invokes functions with argument 1*, their return values will be the same:*

```
void wrapper1()
{
    int x;
    x = ambFoo1(1);

    if (x == 1)
        // ...
}
```

```
void wrapper2()
{
    int x;
    x = ambFoo2(1);

    if (x == 1)
        // ...
}
```

□

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

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 while loop snippet:*

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

The appropriate AST can be seen in figure 5.1.

□

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:

- GumTree [2] API [1]
- Eclipse JDT Core API [3]

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

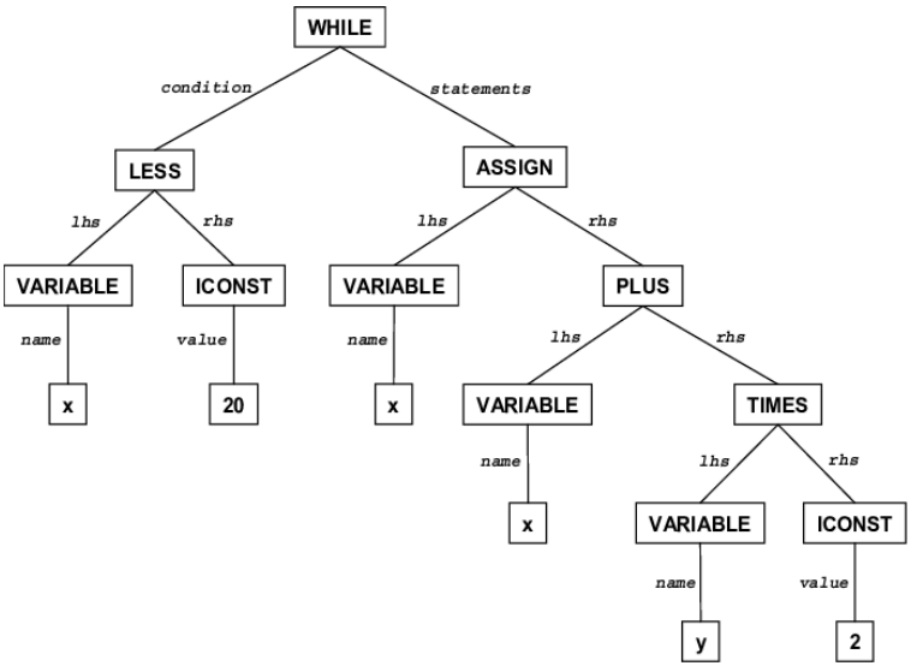

Figure 3.1: AST for the while loop

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 alot 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 equivallent 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

We start off by creating two ASTs, one for each code snipet. The first task is to use matchers to find matching tree nodes. Using the GumTree API, we compute the actions required to create the second tree from the first. If those actions contain only variable name updates, then it is safe to say that the trees are equivallent and that the snippets are semantically equivalelnt. If there are insert or delete actions present, one might be tempted to report them as a difference and automatically assume that the actions needed to be done are reverse of those found in order to "fix" the second code. As shown in example 1.4, this is the wrong way of handling the problem at hand.

The second approach is to phrase the semantical equivallence in a different way. This brings us to per-block variable analysis. We assume the following is true for any given code snippet:

Assumption 5.1. Semantically equivallent code snippets have the same variable values at the end of each block.

To illustrate this, we give an example and explain how we collect and compare variable values for each block.

Example 5.2. *Consider the following code snippets:*

```
int x = 1;
int foo()
{
    int a = 5;

    if (a > 3) {
        x = 2;
    }

    int c = 1;

    if (a > 3) {
        c = 2;
    }
}
```

```
int y = 1;
int foo2()
{
    int a = 2;

    if (a < 3) {
        y = 2;
    }

    int c = 1;

    if (a < 3) {
        c = 2;
    }
}
```

Let's analyze the first snippet "per-block":

```
// Block depth 0, ordinal 1
int x = 1;

int foo()
{
    // Block depth 1, ordinal 1
    // Vars passed: x = 1

    int a = 5;

    if (a > 3) {
        // Block depth 2, ordinal 1
        x = 2;
        // End of block: Update x in parent blocks
    }

    // x = 1 here because of the update

    int c = 1;
    if (a > 3) {
        // Block depth 2, ordinal 2
```

```
        c = 2;
        // End of block: Update c in parent blocks
    }

    // Vars: x = 2, a = 5, c = 2
}

// End of root block: x = 2
```

Using the above information, we create the variable map:

Block depth	Block ordinal	Variables
0	1	x
1	1	x, a, c
2	1	x, a
2	2	x, a, c

The variable values are stored per each block and each update in the child block will be propagated to the parent block when the end of the child block is reached. Using this variable map as a model, we do the same for the second code snippet. Also, note that renamed variables are not a problem - the matcher will give the update pairs before analysis, in this case (x, y) will be an update pair.

```
// Block depth 0, ordinal 1
int y = 1;  // Matches to x

int foo()
{
    // Block depth 1, ordinal 1
    // Vars passed: y = 1

    int a = 2;

    if (a < 3) {
        // Block depth 2, ordinal 1
        y = 2;
        // End of block: Update y in parent blocks
    }

    // x = 1 here because of the update

    int c = 1;
    if (a < 3) {
        // Block depth 2, ordinal 2
        c = 2;
        // End of block: Update c in parent blocks
    }

    // Vars: y = 2, a = 2, c = 2
```

```
        // Conflict found: a = 1 in source, but found a
            = 2
    }

    // End of root block: y = 2
```

The above algorithm does not take into account what actions are done in a block, as long as the end result of the block is the same - i.e. the variables have the same end-value. □

The pseudo code of the algorithm can be seen in figure 5.1. Returned conflicts can be used to either list or repare the code. For now we are just listing the conflicts.

```
Analyze(tree1: AST, tree2: AST)
begin
    if (CreateMatcher(tree1, tree2).OnlyUpdateActionsFound()
        ):
        /* We have found only rename actions */
        print("Given snippets are equivallent")
        return

    /* In general case, traverse the first tree */
    vmap = tree1.TraverseAndRecordVars()

    /* Traverse the second tree and compare vars per block
        */
    conflicts = tree2.TraverseAndCompareVars(vmap)

    foreach (conflict in conflicts):
        print(conflict.Details)
end
```

Figure 5.1: Our analysis algorithm in pseudocode

6 Conclusion and future work

We have created a tool which for a small subset of code snippets returns a list of actions nececary to make them equivallent. These, are however, the first steps in the process of building a broader comparer. We have set some assumptions and limitations, however some of them are not hard to remove. Specifically:

- The usage of the `int` type can be easily extended to primitive types. If we wish to extend this to reference types, there has to exist a way of comparing the objects (which in user-implementation cases does not nececarily mean to compare their references). The `equalsTo()` method satisfies the requirements. Primitive types, therefore, have to be boxed to their container types (`int` to `Integer` etc.) in order to create a unambiguous way to compare the two objects.
- Variable determinism can be retained through the use of the symbolic variables instead of the raw values. Reference types may still cause some problems, though.

- Loops with a compile-time known iteration count can be processed as an array of sequential blocks, without changing the algorithm. Another approach can be to unwind loops to a certain limit and then compare them. However, there are cases where loops have different amount of iterations yet still produce the same result. Ofcourse, this is not the only problem that remains unsolved when it comes to loops.
- Unsupported code constucts can be analyzed simply by implementing the action which will be executed on the visit of that particular node in the AST.

References

[1] Gumtree. on-line at https://github.com/GumTreeDiff/gumtree.

[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.