Towards Version Controlling in RefactorErl

Jenifer Tabita Ciuciu-Kiss
Eötvös Loránd University
Budapest, Hungary
jenifer.girl.98@gmail.com

Melinda Tóth
Eötvös Loránd University
Budapest, Hungary
tothmelinda@elte.hu

Abstract

RefactorErl is an open-source static source code analyzer and transformer tool for Erlang. The analyzed software sources are stored in a structure called Semantic Program Graph. In order to make the source code analysis result available for further use, we need to save the graph. If the source code is changed, the tool updates the graph, so in the new graph only the new information is available. Any kind of backups about different versions are really expensive. This is the reason why we need to save the backups in a much more effective way. An adequate solution to this problem is to save the differences in the same graph. In this way we do not need to make a whole graph for every small change.

1 Introduction

RefactorErl [2, 5] is a static source code analysis and transformation tool for Erlang [1]. During the initial analysis, the tool builds a Semantic Program Graph [4], which contains the lexical, syntactic and semantic information about the source code. The different kinds of data are stored in the nodes of the graph.

The nodes are linked with tagged edges. These edges contain structural information about the nodes. The graph has a specified structure. It starts with a root node, which is followed by a node containing the information about the file (the name, absolute path, etc.). They are linked with an edge tagged with file. The file node is linked to its forms: the function definitions and attributes. The function definition forms are lined to its clauses, and so on. In addition there are module and function semantic nodes to represent the semantic information as well. After properly investigating the structure of the graph, we need to create an algorithm, which defines the differences between the versionsand using this the version control can be specified.

2 Graph structure

We know exactly how the graph is structured, so the differences can be easily detected between the versions.

The algorithm on which we are working on as a first step finds the differences in the graph between the previous version and the new one. There is the question, whether we should continue the search more deeply on the graph after finding the first difference or should we stop? It is problematic to answer this question because at this point we do not know how big is the subgraph. Probably we should look at the actual node, because for example if that is a number we do not need to check the subgraph, but if it is a function name, it is possible, just the name changed and everything else is the same. It takes a lot of work experience and experiments to decide it.

When this question is answered, we can take the next step. How can we save all of the versions in the same graph? The basic idea is that there is only one of each node and edge, so if we start to use the same edges as before. The situation cannot occur in any other way, only if we do version control, so it is possible to teach the tool to handle that.

There are three different case for needing version control: modification, deletion and insertion. We can handle all three with the same algorithm because there are a few specific for each case.

Thanks to RefactorErl, we can easily and efficiently insert new nodes and edges. In the following, we will demonstrate the algorithm for each case.

In Figure 1 we can see a simple Erlang module and in Figure 3 we can observe its graph. This graph contains only the semantic information, and it is big already, nonetheless the analyses are fast.

```
-module(example).
fact(0) -> 1.
fact(X) -> X * fact(X-1).
```

Figure 1. Example Erlang module

2.1 Modification

A modification is when we have had something in a version, and we changed something in it, without inserting a new part or deleting a whole part of the source code.

We can say in general that, node A and node B are different so we need to save them in the same graph and we need to make a difference between the versions. E is the edge to node A and node B in the new graph. We can insert the difference node, in this case, node B right after node A with the same edge, which leads to node A, in this case, edge E. So now we have two similar edges, which we can get only if we do version control. And after node B we can insert everything under, it does not make a difference. We do not need to change anything else, the other nodes under node A can stay because they have different edges. When we need to get a previous version, we can just use all the previous node with the same edge on the graph [3].