МАТЕМАТИКА И МАТЕМАТИЧЕСКО ОБРАЗОВАНИЕ, 2013
MATHEMATICS AND EDUCATION IN MATHEMATICS, 2013
Proceedings of the Forty Second Spring Conference
of the Union of Bulgarian Mathematicians
Borovetz, April 2–6, 2013

CREATING CALL AND REFERENCE GRAPHS USING TERM REWRITING [*]

Todor Cholakov, Dimitar Birov

In this article we describe the idea of using term rewriting in an algorithm for extracting call and reference graph information out of source code. Three different representations of the call graphs are described depending on the potential usage of the generated graph for refactoring and reengineering purposes. We also described several ideas about a tool that makes use of the created call graphs and analyzed some of the currently available similar tools.

1. Introduction. Call and reference graphs allow analyzing the system's behavior and software dependencies and are widely used by developers in the processes of refactoring, reengineering and discovering and fixing bugs. Although they have different usages, they have similar structure and both help the developers to analyze the dependencies between different software pieces. That is why we consider them together. In the next paragraphs, we will give a short description of both call and reference graphs and what each of them is useful for.

The purpose of this paper is to implement an algorithm for extracting call and reference graphs out of java sources using term rewriting. The output of this algorithm may be used by tools that further analyze the resulting graphs either for debugging, refactoring or documentation purposes. This implies the need to define several different output formats for the resulting graph that would be useful in different cases depending on the purpose of the user.

We focus on term rewriting, because it allows us to transform and filter the program text, ignoring non-significant parts of the code. This approach will further allow us to use the same algorithm on programs written in different programming languages with little or no modifications.

In the next paragraphs we give some basics about call graphs, reference graphs, term rewriting and the platform that we use for the implementation.

Call Graphs [1] are graphs representing method, function, procedure or constructor invocations within the analyzed program or piece of software. The vertices of such graphs are methods (In the rest of the paper, we will use the term "method" to designate the three of them). And there is an edge (oriented arrow) from ***method1*** to ***method2*** if

[*]**2000 Mathematics Subject Classification:** 68N19.

Key words: call graphs,reference graphs, reengineering, term rewriting.

The author gratefully acknowledges financial support by the Bulgarian National Science Fund within project DO 02-102/23.04.2009

method1 calls directly ***method2***. The call graph may have weights for the edges, which depend on how many times ***method2*** is called by ***method1***.

Call graphs are useful for analyzing program behavior and structure. They allow the programmer to follow easily the program execution path and eventually to find and fix bugs that would be hardly to reproduce otherwise. Although the edges of the graph are oriented from callers to callees, in real life tools we need to follow the graph in both directions (see the examples below).

Call graphs may be combined with domain specific language (DSL) for making queries on the graph. Examples of such queries would be:

– Which methods call **A**?

– Which methods call **A** n levels deep?

– Which methods are called by **A** n levels deep?

– Which are the methods in class **C** that are never called from within the class itself?

– Which are the most often called methods from within class **C** or from within a group of classes?

All these questions and many others can be answered by analyzing the call graph. This will lead to better and easier understanding of the piece of software by the developer working on it, and to faster correcting of the errors.

For example if we have a field that has an invalid value during program execution at method **m1** and we consider that it may not be initialized, we may ask which methods call this method and also ask which methods initialize the problematic field and construct their call graph. Then we make the difference between the first and the second set and see the potential places where **m1** is called but the field is not initialized.

Except for using call graphs for demonstration and querying, they are also extensively used in refactoring tools. For example in order to implement "*rename method*" refactoring, one must be able to trace all references to that method and a call graph would be very useful for this purpose.

Reference Graphs represent dependencies and references between classes. Class ***A*** is considered to refer class ***B*** if class ***A*** is subclass of ***B***, ***A*** calls a method of ***B*** or ***A*** has variable of type ***B*** or ***A*** refers method or field of ***B***. The edges of the graph may be weighted depending on the count of the references from class ***A*** to class ***B***.

Reference graphs are useful for determining high level dependencies. Especially they are useful to determine which classes should be in the same component or package. For example, if we have several classes that have a high degree of references each other, but they refer rarely classes outside the group, we may consider them belonging to one and the same component or package. This is helpful for refactoring tools that want to reengineer an existing system and recreate or rearrange its components.

We chose to use languages based on term rewriting instead of procedural or object oriented languages, because it gives a unified way to work with all kinds of program elements. This greatly reduces the amount of code that needs to be written. Additionally we chose a platform (Stratego/XT) that has an already written implementation of a java parser and type checker (the Dryad library).

ATerms [2] (annotated terms) provide an easy and human readable way to represent hierarchical information. As long as any class or piece of code may be represented by its parse tree, the annotated terms may be used. They have the additional advantage that they support annotations – any arbitrary data (such as type, current value and so on) may be associated with a term without interfering in the hierarchy.

A simple ATerm defining a boolean literal used in method invocation looks like this:
`Lit(Bool(False)){Type(Boolean)}`

Term rewriting is a concept that defines the rules for transforming terms. Depending on the implementation a term rewriting system may give a powerful set of commands that work on terms. Such systems give a very easy way for manipulating terms and therefore software code. Usually such systems are extended to support term annotations (ATerms) to achieve additional language expressivity.

Stratego/XT [3] is a language/system based on term rewriting. It has a large set of commands that allow easy manipulation of the term hierarchy. Additionally it has libraries that allow parsing of Java and C source code and converting it into a set of ATerms.

The Dryad library is very useful in our case, because it gives instruments to parse any java class or interface and transform it in a set of annotated terms. The type checking abilities implemented in this library annotate each term with its type. The output that this library produces is the raw data for our algorithm.

The easy navigation through the terms, the type checking abilities of Dryad and the easy parsing of Java code are the main reasons to choose Stratego as implementation base for our analysis. In Java for example implementing call graphs would be much more complicated task.

Canonical form of class or method is a form that uniquely defines the class or method and ignores any information that is not important. For classes this is the form
`<package name>.<class name>`

For methods this is
`<defining class canonical name>.<method name>(<parameter1 type in canonical form>, <prameter2 type in canonical form>)`

We don't need the names of the variables or the returning type because this information will not change which method is being referred.

2. Extracting the reference graph. In order to extract the reference graph we used the following algorithm:

Step 1. We call a routine of the Dryad library that parses the Java sources to a set of "CompilationUnit" ATerms and each call or variable usage is annotated with its corresponding type. That is the raw data needed for the next steps.

Step 2. Out of each CompilationUnit we get the following info for all the classes in it:

The exact class name in canonical form (for example `"com.analyzed.Test1"`)

The parent class name also in canonical form. Generally we need all the class names to be in canonical form in order to be able to connect them later

Step 3. We infer the types of all typed expressions. This includes variable, parameter and field declarations and type castings. It also gets the return types of all methods.

Step 4. We infer the types of all other expressions and method calls. For example `"System.out.println()"` would cause that the class `java.lang.System` is counted as referred by our code, but also `java.io.PrintStream` is referred because the type of `System.out` is `java.io.PrintStream`.

We divide the types in three categories – user defined, defined in the standard libraries and primitive types.

We implemented an option to ignore all references to library classes and primitive types. This is because the standard java libraries are so widely used that in most cases they don't give us any useful information. However there are cases when such information is useful. For example consider a tool that want to separate UI classes, business logic classes and IO oriented classes. In this case we can leave the library references and state that classes that mostly refer to `"java.awt"`, `"javax.swing"` or `"org.eclipse.swt"` classes are part of the UI and classes that refer to `"java.io"` are IO classes and classes that refer more to classes from within the analyzed software than to library classes are considered business logic.

Step 5. We transform the result to a list containing all the class names, and a list of triples containing the edges of the graph (`<start>`, `<end>`, `<weight>`). A simple example of this representation is:

```
    (["org.proj.Main","org.proj.FileManipulation","org.proj.Dialog"],
[(1,2,3),(1,3,3),(3,2,6)])
```

In the above example the Dialog class refers the FileManipulation class six times. The elements in the triples are numbers which refer the corresponding element in the string list. In this way we may manipulate the graph by any graph manipulation routine and then get back to the original semantics of class names.

Although the result in the previous step is enough for the purpose of manipulating the graph, it is not human readable, so we take one next step and print the graph in GraphWiz's `".dot"` format [4] file which is readable from the ZEST [5] visualization framework.

We chose the ZEST visualization framework, because the description of the graphs to be visualized is in plain text, which makes the resulting graph easier to manipulate and debug, and also there is a visualization plugin for Eclipse, which allows to easily view and analyze the results of our program.

3. Implementing the call graph. The algorithm for extracting the call graphs is very similar to the on for reference graphs. However, there are some differences between the algorithms:

In step 3 and 4 we have to go to method level and extract the canonical form of each defined method. Apart of method definitions we need to analyze only method calls – we don't count variable declarations as references. Constructing new objects may or may not be considered a call to the appropriate constructor. We chose not to count constructor invocations as method calls in our implementation because too many questions arise about the implicit parameterless constructors that are always considered present if no other constructor is defined.

In step 5 we have three possible representations of the call graph that must be supported. This is because we may need to preserve relative positioning between calls or we may choose to aggregate them depending on the usage of the result. For example consider the following method:

```
package myPkg;
class A{
      public int m1(){
```

```
                    B bVar;
                    C cVar;
                    bVar.doSomething();
                    cVar.createReport(bVar);
                    bVar.doSomething();
        }
}
```

The resulting graph may be in one of the following forms:

1. Natural – `myPkg.A.m1()->myPkg.B.doSomething(), myPkg.C.createReport(myPkg.B),myPkg.B.doSomething() )`

In this form we have for each edge all its heirs in the same order that the calls were executed. This form is useful when we want to analyze the sequence of calls, coding patterns or when we use the result in the refactoring process. If createReport() changes bVar, it does matter that it is called before doSomething(). But this form is not useful for creating metrics based on the call graph. Also it makes the graph larger and more difficult for processing.

2. Metrics – `myPkg.A.m1()-> ((myPkg.B.doSomething(),2), (myPkg.C.createReport(myPkg.B), 1))`

This form of presentation is basically the same as the first but it ignores duplicate calls, and instead adds weights for each call, stating how many times it was executed from the specified method. This presentation of the graph, gives much simpler graphs, which are more suitable for applying metrics or searching for trends.

3. Machine processable – `(["myPkg.A.m1()","myPkg.B.doSomething()", "myPkg.C.createReport(myPkg.B)"],[(1,2,2),(1,3,1)])`

This form of presentation separates the semantics (method signatures) from the graph representation, thus allowing a graph processing routine to work on it without knowing the exact semantics.

All these representations have their usages and our program is able to produce as output any of them.

The last part of this program is to print the result in human readable form. Again the graph is printed in GraphWiz's dot format, as all the methods of a class are in a single sub graph.

4. Experiments. We implemented the proposed algorithms as Stratego scripts and we managed to generate a call and reference graphs in the machine processable form. The software that we analyzed had about 1382 classes (an IDE) and the resulting call graph consisted of more than 8684 nodes (methods). There were 28939 connections (method calls) in the call graph and 7758 references in the reference graph.

Although the algorithm does its job, there is much to be said about the abilities of Stratego and Dryad library to process large amounts of code – we had to pass the classes one by one to Stratego in order to produce the ATerms at step 1. After all the ATerms were generated we managed to generate the graph without any further problems.

We also generated call and reference graphs for a small example project consisting of three classes and several methods. In this case there were no issues with the Stratego platform and the generation ran flawlessly.

In both experiments we generated also a file for GraphWiz. The visualization for the small graphs was quite good, but for the large ones, the generated view was unusable, because there were too many nodes and edges to be able to understand anything useful.

5. Similar works. Currently there are several tools that work with call or reference graphs. We show them here with their main features to demonstrate the usages of the concept of call graphs and to show what is still not implemented or needs improvement.

• **Call graph viewer for Eclipse** – uses Zest for graph visualization (which is a possible solution for our tool, too). However we were unable to run the tool with none of the recent versions of Eclipse (3.7.2, 4.2) to test its functionality more thoroughly. According to the documentation the user must be able to add caller and callees to the graph. Obviously it does not analyze the whole program, but instead analyzes only the class that the use works on and its callers and callees.

• **The call hierarchy feature in Eclipse** [6] – it generates a partial call graph about from where a method is called. However it does not generate a full call graph. The call hierarchy is created incrementally when the user clicks on a method and tries to expand it. Apparently the eclipse plugin uses some kind of indexing, because even for relatively large projects getting the callees is quite fast, but still takes several seconds. The plugin allows traversing through the call hierarchy and clicking on an element opens it in the code editor.

• **Most of the profilers like JProfiler, gprof and OptimizeIt** generate call graphs based on dynamic execution and measurements to help developers to discover bottlenecks in their software. These call graphs are usually partial, because they do not use analyzing the complete software, but only the calls that were really executed in the current run. This is quite useful approach, because it ignores potential executions that never happen, and focuses on those that happen. However this is not suitable for refactoring purposes where the whole call graph is needed in order to determine dependencies and software components. The static approach that we use have the advantage of taking in account all possible flows, but the results are more difficult to use when a single flow is needed.

• **Reacher** [7, 8] – a tool for visualizing and analyzing call graphs. It has some very useful features concerning querying for different reachability questions. It allows the developers to ask questions about which methods are called in the downstream of a given method or to search methods that are in the upstream of the callers of a method.

6. Future research. Creating reference and call graphs is only a step to implementing a more complicated refactoring and source analysis framework. Next steps for developing this idea include:

• To develop a clustering algorithm that will be able to determine whether two code pieces (classes or methods) belong to a common unit (a module or a class)

• To create and implement a query language that will allow the developers to trace references to a method or dependencies between classes and methods.

7. Conclusion. Call graphs and reference graphs have many usages in the process of software reengineering. They can help the developers to form components and classes, especially when the code comes from procedural language. They are also useful for discovering bugs in the software and for analyzing software performance and quality.

In this paper we have introduced a simple algorithm for generating call and reference graphs using the Stratego language and the Dryad library, and shown different possibilities for representing these graphs depending on their potential usages.

REFERENCES

[1] B. G. Ryder. Constructing the Call Graph of a Program. *IEEE Transactions on Software Engineering*, **SE-5** (1979), No 3, 216–226.

[2] D. J. Howe. A type annotation scheme for Nuprl, Theorem Proving in Higher Order Logics. *Lecture Notes in Computer Science* **1479** (1998), 207–224

[3] M. Bravenboer, K. T. Kalleberg, R. Vermaas, E. Visser. Stratego/XT 0.17. A language and toolset for program transformation. *Science of Computer Programming*, **72** (2008), Issues 1–2, 52–70, ISSN 0167-6423.

[4] The DOT Language, `http://www.graphviz.org/content/dot-language`

[5] Zest visualization framework, `http://www.eclipse.org/gef/zest/`

[6] Eclipse Indigo (3.7) Documentation, `http://help.eclipse.org/indigo/index.jsp`

[7] T. D. Latoza, B. A. Myers. Visualizing Call Graphs. *Visual Languages and Human-Centric Computing* (2011).

[8] T. D. Latoza, A. Kittur. Answering reachability questions, 2011.

[9] D. Grove, G. DeFouw, J. Dean, C. Chambers. 1997. Call graph construction in object-oriented languages. *SIGPLAN Not.* **32** (1997), No 10, 108–124.

Todor Plamenov Cholakov, Dimitar Yordanov Birov
Faculty of Mathematics and Informatics
University of Sofia
5, James Bourchier Blvd
1164 Sofia, Bulgaria
e-mail: todortk@abv.bg, birov@fmi.uni-sofia.bg

СЪЗДАВАНЕ НА ГРАФИ НА ИЗВИКВАНИЯТА И ИЗПОЛЗАНИЯТА ЧРЕЗ ИЗПОЛЗВАНЕ НА ПРОЕОБРАЗУВАНИЯ ВЪРХУ ТЕРМОВЕ

Тодор П. Чолаков, Димитър Й. Биров

Тази статия описва идеята за използване на преобразуване на термове в алгоритъм за извличането на информацията за извикванията и референциите от програмния код. Описани са три възможни представяния на графите на извикванията в зависимост от потенциалното използване на генерирания граф за целите на анализ, рефакторинг или реенжинеринг. Описваме няколко идеи за инструмент, който използва генерираните графи и анализираме някои съществуващи подобни инструменти.