Proceedings of the 9th International Conference on Applied Informatics
Eger, Hungary, January 29–February 1, 2014. Vol. 1. pp. 121–130
doi: 10.14794/ICAI.9.2014.1.121

Rule based graph visualization for software systems

Tibor Brunner[a], **Máté Cserép**[b]

[a]Ericsson Hungary Ltd.
bruntib@caesar.elte.hu

[b]Eötvös Loránd University
mcserep@caesar.elte.hu

Abstract

Supporting the comprehension of complex, industrial-size software systems became an important task of the software industry. Many of the comprehension tools are based on static source code analysis. As direct text-based analysis of the source code usually cannot be performed efficiently, hence such tools use a higher level representation of the source code, i.e. the abstract syntax tree. Having constructed from the source, the abstract syntax tree serves as the base of presenting the system to the user either in textual or graphical way. Graphical visualizations in different views and projections produce efficient assistance in the process of comprehension. A part of these visualization methods are to represent the structure of the source code with graphs revealing various aspects of the program.

Most static source code analysis tools support only pre-defined programming languages and diagram types, restricting the users from a more general use. Our solution presents an abstract, language-independent, rule-driven method that conduces to the creation of arbitrary, user defined graph visualizations from the abstract syntax tree. Our approach lets the user to describe the desired graph to visualize in a declarative way by defining a set of rules. A graph building algorithm then receives the abstract syntax tree as an input and generates the aforementioned visualization alongside of these rules.

Keywords: code comprehension, static analysis, abstract syntax tree, source code visualization, graph representation, graph building algorithm

MSC: 68R10, 05C85, 68N99

1. Introduction

One way to ease the understanding of legacy software systems is the different kind of visualizations, because humans are better at deducing information from graphical

images than from numerical data [1, 8]. While over the past few years, researchers have proposed many software visualization techniques and various taxonomies have been published [2, 3], in our article we focus on node-link graph diagrams with arbitrary nodes and arcs. Graphs are able to support visualization modes for several different levels of the software development process, from low level source code to high level architecture views, such as control flow diagrams, function call diagrams, different type of dependency diagrams, component diagrams, etc. With node-link diagrams it is also possible to create new visualizations to assist the comprehension of a software system from a special aspect of the user when required.

Several tools exist to produce graphical visualizations through node-link based diagrams and a variety of other type of views, some well-known and used examples to mention are the *Architexa* [6, 9], the *Dr. Garbage Tools* [10] or the *NDepend* [12]. While these programs are limited on the given task or the analyzed programming language, our purpose was to create a general algorithm which receives set of rules and an *abstract syntax tree* (AST) [7] of the source code created by a static analyzer or a parser [5]. Based on these two inputs a generalized algorithm creates the specific output graph. This way the users are able to create custom graph visualizations by describing the rules in a declarative approach, without the need to define a complete graph building algorithm.

The paper is structured as follows. In Section 2 we present the rule types handled by our graph building library and discuss a practical example to demonstrate to usefulness of each of them. Section 3 describes the applied algorithm in the graph generation process, while Section 4 demonstrates the usage of our graph building library through examples in large software projects. Finally in Section 5 we conclude the paper and outline the future development directions.

2. Graph building rules

In this section we present our algorithm which produces the output graph based on the AST and a set of rules given by the user.

$$
\begin{array}{lll}
Rule = \{ & isNode: & (N,S) \rightarrow \{true, false\} \\
& children: & (N,S) \rightarrow (N,S)^* \\
& childSequence: & (N,S) \rightarrow (N,S)^* \\
& childSequences: & (N,S) \rightarrow ((N,S)^*)^* \\
& furtherArcs: & (N,S) \rightarrow ((N,S),(N,S))^* \\
& inNodes: & (N,S) \rightarrow (N,S)^* \\
& outNodes: & (N,S) \rightarrow N^* \\
& nodeAction: & (N,S,GN) \\
& arcAction: & (N,S,N,S,GA) \\
\}
\end{array}
$$

Figure 1: Function attributes of the rule record objects.

Our intent was to create a library, which enables the user to build the desired graph not by expressing the respective rules instead of creating new graph iteration algorithms each time. In the object-oriented concept, these rules are record objects and their attributes are functions. The possible function-attributes contained by the rule objects are presented in Figure 1 with their signatures.

In the mathematical notation N means a node of the AST, S is a state object of arbitrary type, GN is a node of the output graph and GA is an arc of the output graph. Assuming X is a mathematical object, X^* is a sequence of Xs and $(X_1, X_2, ..., X_n)$ is az *n-tuple* (ordered list) of the contained objects.

isNode : A boolean returning predicate to determine for a given AST node N in the given state S whether to contain the counterpart of the node in the output graph. The further functions will get this node as a parameter.

children : A function that determines the child nodes of the previously selected node. The traversal algorithm continues its iteration on these nodes.

childSequence : Unlike the *children* function, it is sometimes required to link the defined nodes in a parent-child relation instead of a sibling relation. For example in a control flow diagram the statements of a loop's body should be linked after each other, thus drawing the sequence of these statements.

childSequences : This function is similar to the previous one, but can place several sequences next to each other in a sibling relation.

furtherArcs : In order to provide a the greatest possible flexibility, this function is able to join two arbitrary nodes independently from the other arcs. These nodes can be selected based on the node and state passed as function parameters.

inNodes : It is possible to redirect incoming arcs to a node. This function determines the nodes in which the incoming arcs are redirected.

outNodes : Similarly to the previous function, this one determines the nodes to which the source of outgoing arcs will be redirected.

nodeAction : A procedure (not a function) which runs each time a new node is created in the output graph. This procedure can be used for example labeling the nodes or modifying the state of the node.

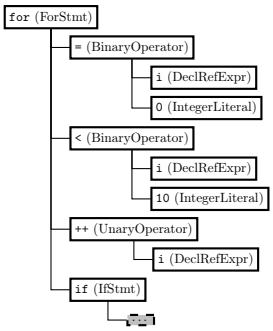
arcAction : Similarly to the *nodeAction* procedure this method runs each time when a new arc is created in the output graph.

2.1. Example

Let's take a short example which may make the comprehension of the method easier. This example is a part of the creation of a control flow diagram, especially the *for* loop. This is quite a complex task, so we can demonstrate all the functions

listed above. The schematic input code and the corresponding AST subtree is presented by Figure 2a and Figure 2b respectively.

```
1 S1;
2 for (init; cond; iter)
3   body;
4 S2;
5
6 for (int i = 0; i < 10; ++i)
7   if ...
```



(a) *for* loop (b) AST of the loop

Figure 2: Input loop of the algorithm.

All we have to do is to create a rule which describes the nodes belonging to a *for* loop in a control flow diagram. If we describe all control structures, then the whole control flow diagram arises. The desired output is depicted by Figure 3.

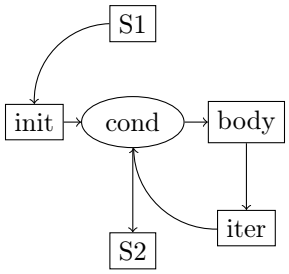


Figure 3: Output control flow diagram of the loop.

First of all we would like to see a node in the output graph if we find an AST node which belongs to a *for* loop, as shown in Figure 4.[1]

```
1 bool isFor(Node n) {
2   return For(n).isValid();
3 }
```

Figure 4: The predicate function for determining whether the output graph should consist a node.

[1]The following code snippets are written in C++, however the concept and process is independent from any specific programming language.

As can be seen the function gets an AST node as a parameter which may belong to any language element. In this example we suppose that we have a helper class called `For` defined (see Figure 5), containing getter functions which return the corresponding nodes of the subtree of the loop node.

```
struct For {
  For(const Node& node);
  bool isValid() const;
  Node getCondition() const;
  Node getInit() const;
  Node getIter() const;
  std::vector<Node> getBodyStmts() const;
};
```

Figure 5: Definition of the *For* helper class.

Since any AST node can be passed to the `For` constructor, it is necessary to have an `isValid()` function which returns true if the given node really belongs to a *for* loop in the AST.

In the control flow diagram there are two arcs pointing out from the conditional node of the *for* loop. These are the first statement of the body if the condition evaluates to true, and the next statement after the loop otherwise. The statements of the body are the children of the conditional node in a control flow diagram, thus the function determining them may look like the code snippet presented by Figure 6.

```
std::vector<Node> forChildren(Node n) {
  For loop(n);
  std::vector result = loop.getBodyStmts();
  result.push_back(loop.getIter());
  return result;
}
```

Figure 6: The rule-function providing the child nodes.

The iteration expression is evaluated at the end of each loop iteration, so this node also has to be appended to the statements of the loop body.

The incoming arc points to the AST node which belongs to the *for* loop but before the first iteration of the loop, the initialization expression is run. This means that we have to redirect the incoming arcs from the loop node to the initialization node, as Figure 7 describes.

There are two further arcs missing: the iteration node has to be connected to the loop node to represent the loop structure and the node of the initialization expression also has to be connected to the loop node, hence the definition of some further arcs are required by this special rule. The related code segment is displayed by Figure 8.

```
std::vector<Node> forIn(Node n) {
  return {For(n).getInit()};
}
```

Figure 7: Redirection of the incoming arcs of the loop node to the initialization node.

```
std::vector<std::pair<Node, Node>> forFurther(Node n) {
  For loop(n);
  return {{loop.getIter(), n}, {loop.getInit(), n}};
}
```

Figure 8: Adding further arcs to the graph by special rules.

Our last task is to create the rule record which describes how to display the control flow graph of a *for* loop, add the rule to the graph builder and finally execute the building process from a specific starting node, as shown by Figure 9.

```
Rule forRule;
forRule.isNode        = isFor;
forRule.childSequence = forChildren;
forRule.inNodes       = forIn;
forRule.furtherArcs   = forFurther;

GraphBuilder builder;
builder.addRule(forRule);
builder.run(startNode);
```

Figure 9: Rule definition and executing the building process.

3. Graph building algorithm

In this section we introduce the algorithm which produces the output graph based on the given rules.

3.1. Basic attributes

In the algorithm we handle a queue data-structure. The method ensures that a node cannot be queued multiple times in the same state, hence enforcing the termination of the progress in a finite amount of time. The process is initialized by inserting the start node in this queue and then beginning a loop which iterates till the queue is not empty. In every iteration of the loop we check whether any of the rules match the current node. A rule matches when its *isNode* attribute returns

true to the current node. If we find several matching rules for a node, then all of them is applied to build the further parts of the output graph.

In the body of the above mentioned loop, the child nodes of the current node are determined by the *children* attribute of the rule. The returned nodes are linked with the current node in the output graph. Every time when a new node is discovered, the new node is placed in the queue so that these will also be processed. Similarly for every new node and arc the *nodeAction* and *arcAction* procedures of the rule have to be run.

Child nodes may also come from the *childSequence* or the *childSequences* functions. In contrast to the *children* function, only the first nodes of these sequences can be linked to the current node, because at this point it is unknown whether there will be any redirections among these nodes. Therefore the only thing that can be done at this stage of the process is to store these nodes in a map data structure to sign that these nodes will be joined later. This *todoMap* allows us to assign a list of nodes to another node for creating these connections in the future.

3.2. Redirection attributes

The development of the algorithm was inspired by producing the control flow diagram of a source code, and that is the reason for the ability of arc redirections and drawing independent arcs became a part of the algorithm. In everyday use cases (see Section 4) these extra features are often unnecessary, as the expressive power of the algorithm is quite strong to create a wide range of graph based diagrams without this extension of the method.

If the current node can be found in the *todoMap*, then it has to be processed as the next step. Let's suppose that according to *todoMap*, the *currentNode* and a *target* node has to be connected. If a rule defines nodes by its *outNodes* attribute (i.e. outgoing arcs have to be redirected), then (*outNode*, *target*) has to be inserted in *todoMap*. The reason of this is that the node where redirection is established, may also have redirection rules for its outgoing arcs, therefore at this point it is unsure whether these nodes will be connected. The motivation of the redirection is to be able to draw the control flow diagram of an `if` statement. In case of nested branching control structures the sequentially connected statements have to be forked. However at the process of the current `if` statement we cannot determine the last nodes of its body, since if the last statement is also a new branch, then the redirection of outgoing arcs has to be delegated to these. So the *outNodes* function has to return the nodes where the redirection will be done in the future progress, when the last statement of the control structures' body is reached. Otherwise if there are no *outNodes* belonging to the current node then an arc has to be drawn between *currentNode* and *target*.

We should not forget the redirections of incoming arcs of a node. So every time when an arc is drawn between two nodes, it has to be checked whether the target node has any redirection rule for incoming arcs. This could be more efficient if the nodes for the current node are stored where the redirections are accomplished. This way we just have to look up this storage when ever a new arcs comes in the

output graph.

Finally *further arcs* are missing. These can also be determined in every iteration of the main loop. The only thing we have to pay attention to here is not to apply the redirection rules on the two endpoints of these arcs, because the user's intention was to directly join these nodes.

4. Usage and results

As the title of this paper suggests, the described method is not only applicable for low level code observations, but also for high abstraction level visualizations as well. The algorithmic method defined in Section 3 is not simply considering mere AST nodes as the traversed input, but so called semantic nodes by which we mean that these nodes are endowed with additional information such as the position, container file or translation unit, documentations, mangled name, etc.

These information give us the possibility to create node-link diagrams on a large scale of variety. Some of the notable use-cases are discussed in this section. It is important to note that our method is not restricted on a given programming language but also can be used on any *domain-specific language* (*DSL*) too.

Function call diagrams In code comprehension it is a constantly actual question to find the caller and called functions of another one. The solution is not trivial when virtual methods, overloaded and overrided functions or calling via function pointers are included. It can also be interesting to visualize all the call paths between two functions. Creating a function call diagram with the described algorithm is a fairly simple task. The desired result can be achieved by defining a single rule of which the *isNode* attribute returns *true* for function nodes and the *children* function returns the called functions of the currently observed one. With the *nodeAction* attribute it is also easy to set distinct visual properties (e.g. color) for the different kind of relations (like virtual functions and its overridden versions) in the output graph.

File level diagrams On a higher level of modularity, the relation between files can be analyzed, to assist the understanding of the software by depicting the different type of dependencies. Some source files might provide an interface written in a given header file, while others are just using its services. Expanding the examination further, an appropriate diagram can visualize which files were compiled into an object file or a shared library, and which libraries constitute an executable binary file. Deducing this knowledge is essential in understanding a legacy system and keeping the software architecture clear. To provide such diagrams it is not enough to regard solely the source code, but further compilation and linking time information must be attached on the AST nodes, thus creating semantic nodes.

Module based diagrams While getting familiar with a legacy code, the first task would often be to determine the main modules (i.e. libraries or packages) and

the higher level inner dependencies of the project. In order to achieve this, the file level relationships should be generalized to more abstract levels.

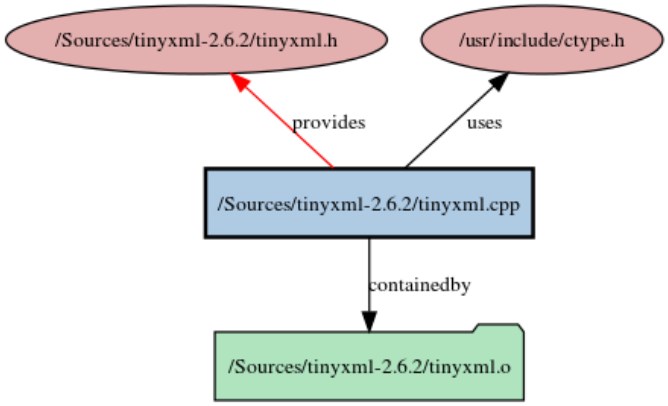


Figure 10: A file level relationship diagram visualized.

With the utilization of the *Graphviz* graph drawing library [4, 11], a diagram visualizing tool was created as part of a larger code comprehension assisting project – named *CodeCompass* – developed in cooperation at Eötvös Loránd University and Ericcson Hungary. In Figure 10 a file level relationship is displayed, assisting the understanding of the well-known *TinyXML* parser project [13], which was written in *C++*. The diagram is fouced on the different kind of relationships of the centered `tinyxml.cpp` file.

5. Summary and further development

Graphical visualization is a great help in understanding a legacy code base, so creating such visualizations should be facilitated by tool support. In the software market several tools can be found, but these are too restrictive considering the programming language or the diagram type. So we have managed to create a method which helps to produce new graph based visualizations in a language independent manner. By this method we could create dependency diagrams, function call diagram, class diagram, and other diagrams representing the higher architecture of a complex software system. An implementation of this algorithm has been realized in C++, as part of a larger static code analizer tool. We successfully managed to include linking time information in our visualization, demonstrating the flexbility of our solution.

The aim of establishment of the library was to facilitate the creation of new visualization methods of an arbitrary software system. Our intention as future work is to integrate our library in a static code analyzer and grocker tool. We are

planning to build a query language by which it becomes possible not only to provide prefabricated graphs for the users but to give the ability to real-time creation of new graph based visualizations.

References

[1] Biederman, I., Recognition-by-components: a theory of human image understanding, *Psychological review*, Vol. 94 (1987), 115–147.

[2] Caserta, P., Zendra, O., Visualization of the static aspects of software: a survey, *IEEE Transactions on Visualization and Computer Graphics*, Vol. 17 (2011), 913–933.

[3] Diehl, S., Software Visualization: Visualizing the Structure, Behaviour, and Evolution of Software, *Springer-Verlag*, (2007).

[4] Ellson, J., Gansner, E., Koutsofios, L., North, S., Woodhull, G., Graphviz – open source graph drawing tools, *Lecture Notes in Computer Science*, Vol. 2265 (2001), 483–484.

[5] Muchnick S., Advanced Compiler Design and Implementation *Morgan Kaufmann Publishers Inc.*, (1997).

[6] Murnane, E., Sinha, V., Interactive exploration of compacted visualizations for understanding behavior in complex software, *Companion to the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, (2008), 763–764.

[7] Salomaa, A., Formal Languages, *Academic Press Professional, Inc.*, (1987).

[8] Spence, I., Visual psychophysics of simple graphical elements, *Journal of Experimental Psychology: Human Perception and Performance*, Vol. 16 (1990), 683–692.

[9] Architexa, `http://www.architexa.com/`

[10] Dr. Garbage Tools, `http://www.drgarbage.com/`

[11] Graphviz – Graph Visualization Software, `http://www.graphviz.org/`

[12] NDepend, `http://www.ndepend.com/`

[13] TinyXML parser, `http://www.grinninglizard.com/tinyxml/`