Use of graph databases for static code analysis

Q. Dauprat[1,2] ✉  ***, P. Dorbec***[1]***, G. Richard***[1]***, JP. Rosen***[2]
1: Université de Caen Normandie, Campus 2, Boulevard Maréchal Juin, 14000 Caen, France;
2: Adalog, 2 rue du Docteur Lombard, 92130 Issy les Moulineaux, France; email: 1:
{quentin.dauprat,paul.dorbec,gaetan.richard}@unicaen.fr; 2: {dauprat,rosen}@adalog.fr

Abstract

This paper deals with static code analysis. With analysis needs becoming more and more complex, and code volumes getting bigger and bigger, scalability of code analysis tools is becoming one of the current challenges. We explore the use of recent technologies, like graph databases, to represent the source code and pattern matching to find information into a graph. We hope that this will reduce the time to analysis a source code and improve the effectiveness of the analysis. When trying to answer the same query compared to AdaControl, we manage to find results that were missed by the programatic approach. We expect further improvement on future benchmarking.

Keywords: Ada, Static analysis, ASIS, graph databases, Neo4J, AST

1 Introduction

In some activity sectors like industry (railway, avionics, space), the life cycle of the programs can extend over several decades. Over time, programs become huge and complex. With many engineers working on the code, the needs of coding guidance, and more generally, code analysis, is at the heart of the concerns.

It is in this objective that code analysis tools were created. This kind of software allows to check the quality of the code, to validate the implementation, to find performance issues, etc. Since their emergence in the 70s, static code analysis tools kept working on the same structure to analyze the source code, namely an Abstract Syntax Tree (AST). The problem with AST is that we have often needs to re-explore previously visited nodes (sub-tree) to find related information, a variable to this declaration, a type to his definition, etc. We cannot store the information when we firstly explore it, because we do not know in advance what we really need, and this can take a lot of memory. Therefore, AST is not perfectly suited to the needs of static code analysis.

Furthermore, more and more needs for advanced analysis have emerged [1,2,3], and, consequently, the complexity of analysis induces a long analysis time. The scalability of static analysis tools become one of the current challenges [4, 5].

Ada programming language is an interesting subject of study, since all its complexity has been delegated to the compiler. As a result, it is very difficult to compile, and the resulting AST is heavy and complex, making it complicated to understand and to query.

In this work-in-progress paper, we try to take advantage of recent technologies (graph databases) to represent the source code, with Code Property Graph (CPG) [6], and pattern matching to find information into a graph. Our goal is to decrease the time of analysis of a source code and improve the effectiveness of the analysis. Lastly, Ada is a good starting point for our study, and we hope that will lead us to provide a general approach for static analysis of other programming languages.

2 Background

Code analysis is the process of analyzing a program (its source code or its execution). Here, we focus on static analysis that only analyzes the source code. It is mainly used for the computation of code metrics (cyclomatic complexity, etc.) and checking coding rules.

Nearly all static code analysis software works with a structure which is globally the same: an Abstract Syntax Tree (AST). This structure is generally provided by a compiler, or hand built by this software. The resulting AST can be complex and hard to query, especially in the case of Ada. Unfortunately, we cannot provide a simple but complex (in terms of analysis) example of such structures.

3 Approach

The main goal of this research is to take advantage of a new way to represent a source code for static code analysis, with the main objective of reducing the time of analysis, notably for scalability.

3.1 Structures and definitions

Our approach is to use a directed graph to represent the code. We call it a Code Property Graph (CPG) [6]. A property graph is a directed graph where nodes and edges can have properties, whereas in an AST only nodes have them. The following figure shows an example of a property graph:

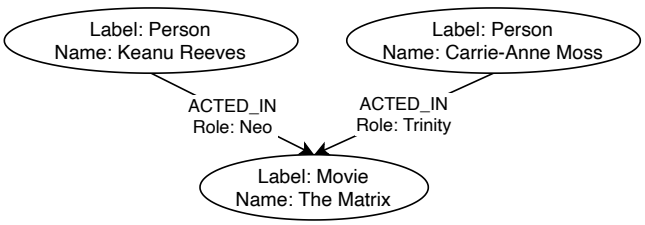

Figure 1: Example of a Property Graph

We can see the CPG as a structure that encompasses AST, the call graph (calls of subprograms), the dependency graph, and many more, simply by adding relations between nodes of the previous AST. The basic intuition is that we use additional edges (with properties) to represent semantic and non-local information. One simplest example is that we can add an (usage) edge from any reference of a variable to the definition of the variable. Enriching the graph with semantic information allows thus easier checking properties. However, the main challenge is the trade-off between the gain induced by using the new edges and the cost resulting from the increase in size of the graph. For the former, we identify below some edges concepts that seems to be the most promising. For the latter, the recent development of graph databases (see next section) gives high hopes of keeping efficient queries. In fact, our experiment search to give a realistic proof of concept to validate the benefits of using CFG.

Furthermore, where in an AST we have a fixed starting node (root of the node), that can be possibly split by compilation units, with a graph, there is no fixed starting node. So we have to choose from which node of the graph we want to start our queries. This can be very convenient to target some kind of nodes by using indexes.

By taking advantage of our experience at Adalog, we have introduced new relationships that we found relevant from a query point of view. These relationships are based on common usage we have to often deal with in static analysis, and from which require deep search to find the corresponding information. The added relationships are:

- **CORRESPONDING_NAME_DEFINITION**: Given an expression of type *Identifier*, *Operator Symbol*, *Character Literal* or *Enumeration Literal*, this refers to where its name has been defined.

- **IS_OF_TYPE**: Given a *component, constant (dereferenced), discriminant specification, formal object, parameter (loop) specification, constant/variable return specification, simple tasks/variable protected object, element iterator specification, object renaming declaration*, this refers to the type associated with this declaration.

- **CORRESPONDING_INSTANCIATION**: this indicates whether an element is part of a generic instantiation.

- **CORRESPONDING_PARAMETER_-SPECIFICATION**: Given *A_PARAMETER_-ASSOCIATION* this points out the corresponding *A_PARAMETER_SPECIFICATION*. This is useful to get all information about a parameter, notably since the order of parameters can differ between the user and the specification.

Nevertheless, we have to keep a number of kinds of relationships as few as possible in order to not increase the size and the time of creation of the database in a disastrous way.

3.2 Graph databases

In addition to this emerging structure, we can take advantage of recent technologies for the scalability. With the rise of NoSQL databases, one type of database appears to be particularly suited to our needs : Graph Databases. A graph Database (GDB) is a kind of non-relational database where data is stored in a graph structure. This kind of database rely on pattern matching to find information. Unlike relational databases, there is no costly joins or foreign keys in a GDB. Here, we can see relations as a pointer. Furthermore, where an AST is stored in files or RAM, a GDB stores the graph in files and RAM but in a way that makes it efficient for querying, especially thanks to indexes.

We believe that the use of Graph Databases could solve current problems of static code analysis, notably for the scalability, and thus, its ability to quickly analyze a large volume of code. Regarding the scalability, some studies [7, 8] demonstrate that GDB is easily scalable for representing the source code.

Nevertheless, GDB lack a standardized query language. Indeed, there is no universal query language like SQL to query any Graph Database Management System (GDBMS). However, a movement of standardization has been launched with GQL (Graph Query Language) [9, 10] and an ISO normalization is on the way.

In this study, we decided to use the Neo4j database. **Neo4j** is a GDBMS written in Java. It is the world leader in directed graph databases. It has its own query language, Cypher, allowing expressing queries in ASCII art (visual) form, like `(node)-[RELATED_TO]->(anotherNodes)`, where parentheses are used for nodes and braces for relationships. We do not believe that the use of one graph database rather than another has a significant impact on this research.

3.3 Thinking about the use of GDB versus the AST approach

We think that the graph approach is interesting because, on the one hand, we have to query the whole AST to get some information, on the other hand, only a specific portion of the graph is queried, thanks to indexes, which only contains the nodes we are interested. To illustrate that the use of GDB comes across as to be a promising approach, we introduce the following example: Given a program, with multiple compilations units, we want to list every declaration of a function that returns a specific type *A*. With an AST, we have to traverse the whole AST to find every function declaration, and apply a filter to find every function that returns the *A* type. With a GDB, we can directly target any function declaration nodes, and apply a filter to find every function that returns the *A* type.

Now, a user may want to make another query, to control that none of the function of a program returns a specific type *B*. With an AST, we can make sure that the controls follow each other, and thus, only traverse the AST once. Using GDB, we can take advantage of another useful feature, the cache. Since we have to start with the function declaration (like the previous query), this result is stored in the cache and can be reused with the new query.

4 Experiments

4.1 Example of Ada code into Neo4j

As an illustration of our research, one example can be given by the code below that is stored in Neo4j as depicted in the figure:

```
package Pack is
   A1, A2 : Integer;
   A3 : Integer range 1..10 := 1;
end Pack;

-- Later in the program

Pack.A1 := Pack.A3;
```

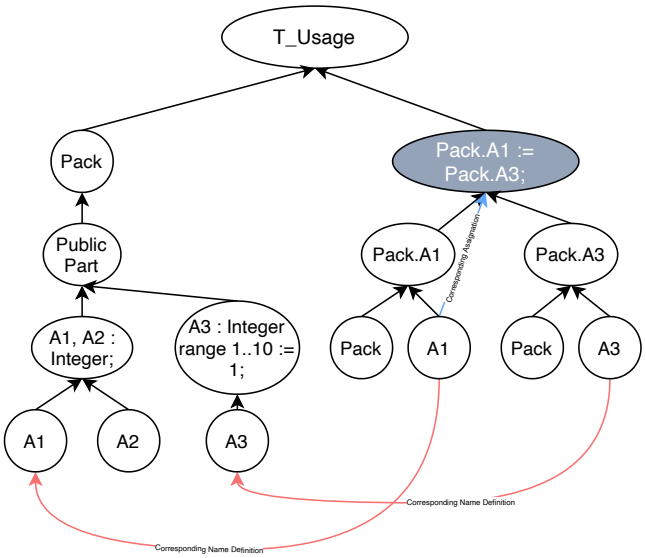

Figure 2: Example of program stored into Neo4J

Note that in the figure, only some nodes are represented. Black edges represent the "IS_ENCLOSED_IN" relation, which is the relation from one node to its parent. This is the only relation which is present in the AST, all other (colored) edges are relations we have created. The gray node is the starting node, where we start the query.

With this tiny example, we could formulate the following request: Given an assignment statement, I want to access to the declaration of the assigned variable. Using a GDB, we start by retrieving all assignment statement (thanks to indexes), and we follow the added relation **CORRESPONDING_ASSIGNATION**, then **CORRESPONDING_NAME_DEFINITION** to finally go to the parent node (using the dark arrow) to obtain the result. Using an AST, we would have been forced to only use black arrows. Furthermore, we have to start from the root node, and exploring child in prefix order. When we find an assignment statement, we have to explore the previously visited sub-tree to retrieve the corresponding declaration. As a reminder, not all the nodes of the graph are represented on the previous figure, so the process of retrieving the node would be costly.

The next figure present the result of the following Cypher query, which obtains the declaration of the variable from the assignment of a variable:

```
MATCH (a:AN_ASSIGNMENT_STATEMENT)
<-[:CORRESPONDING_ASSIGNATION]-
(corrAssi:AN_IDENTIFIER)
-[:CORRESPONDING_NAME_DEFINITION]->
(Id:A_DEFINING_IDENTIFIER)
-[:IS_ENCLOSED_IN]->
(varDecl:A_VARIABLE_DECLARATION) RETURN *
```

Figure 3: Result of the Neo4J query

Where we would have been forced to explore the whole AST to find the usage of all variables, with a graph we can start from a specific kind of node (the variable declaration for example, or the variable assignment in the previous example) to retrieve all information about the variables, without exploring the whole graph. This is possible thanks to indexes, and because there is no notion of "root" for starting point in a graph, and thanks to the extra edges in the graph.

4.2 Our experiment

To make our experiments measurable, we are trying to implement some AdaControl's rules, from simple rule, to more complex, in terms of analysis. Currently, we try to answer the following query: "For each variable of a program, is the variable is **READ** and/or **WRITE**?"

The first step will be to populate the database with an Ada program. We started from the source code of AdaControl where we have changed the analysis of rules by populating the database. The goal of starting with the source code of AdaControl is to reduce the time on prototyping, since AdaControl uses the Ada Semantic Interface Specification (ASIS) in the core to query the source code. Thereafter, we wrote the query using Cypher (the query language of Neo4j) and compare the results with what AdaControl obtains.

The first experiment has revealed that our approach found more results than AdaControl (AdaCtl 1.22r15). We note that using Cypher and thus "pattern matching" to query the code, we found some cases that were forgotten in the traditional (programmatic approach). This is due to a missing case in AdaControl. The current query can find usage of variables in any context (normal, in generic packages and in instantiation), but it does not currently support usage within a *renames*, since this case is tricky because of pointers, arrays, etc. that can be hidden inside the *renames*.

4.3 Current limitations of our approach

Even if the use of the graph database is convenient for the scalability and so, the response time, there are some drawbacks.

The first one is the extra time required for populating the database compared to simply building the AST. Therefore, we accumulate the time to create the AST by ASIS, plus the time to traverse the whole AST again by adding relations and sending it to Neo4j. Though we made no precise timing comparison, our database initialization was visibly slower than the computation of the AST. So, on the one hand, we can have a quick answer to the query, but, on the other hand, the time to populate the database is slow. We are currently focusing on reducing code analysis time, and we do not take into account

the time required to populating the database. An improvement will be to populate the database during the creation of the AST, but this will require to create our own parser or to use a different library than ASIS, like libadalang to process the source code. This is currently out of our research.

Secondly, though Cypher allows to express complex queries, it is quite a verbose query language. Indeed, the query made to answer our "simple" question about usage of variable took more than 200 lines, without considering the special case of **renames**, that would probably double the number of lines. An improvement could be to split the query into several elementary queries, or to generate the query using an intermediate (programming) language.

Moreover, graph databases use pattern matching approach to query the graph, compared to the tree traversal of traditional approach. Event if this approach allows finding some missed case of traditional approach, we are not immune to false negatives.

Finally, we have to keep the number of relationship types and indexes as small as possible in order to reduce as much as possible the impact in size of the database, to be more manageable, and not to sky rocket the time required to populating the database.

5 Related work

During the last decade, more and more research focused on the representation of the code in order to perform code analysis, but mainly for the search of vulnerabilities [6, 11]. Even if [6] focuses on vulnerabilities, it introduces Code Property Graph, for which it is referenced in numerous research (more than 220). This research ([6]) led to the creation of a company specialized in the detection of vulnerabilities in code. There tool is fully based on the CPG introduced into their research.

In his thesis [5], Fan introduces three hard-to challenges: hard-to-employ, hard-to-scale, and hard-to-be-recognized. Our study is focused on the hard-to-scale challenge. He explored different ways to improve the scalability, but the use of graph databases has not been discussed. In [4], authors take advantage of their experience and their industrial vision at Facebook to provide an overview of current challenges and opportunities of static and dynamic code analysis.

Ramler et al. [8] have studied the use of graph databases for various kinds of code analysis, for Java, C, C++ and C#. However, the tools used in the study sound proprietary or non-disclosed.

Furthermore, some studies are focused on the query language. Alves et al. [12] establish a comparison of different query languages, but it looks like that the literature on this subject is quite poor, and it is not our main problematic here.

6 Further Development

In this paper we demonstrate that the use of Graph DBMS could be interesting for code analysis. We have formulated a query to demonstrate the efficiency of this approach, but we have not yet benchmarked this approach on a large volume of code. Currently, some construct into the standard library (GNAT version), cause some trouble when creating the graph. We have to fix these problems before making a benchmark on large volume of code.

The future work will be to provide a benchmark on large code to validate the efficiency of our approach. We suppose that this can be beneficial for large volumes of the code, but irrelevant (slower) on small volumes compared to current approaches. The first step will be to manage some construct that we can see into the Ada standard library implementation provided by AdaCore. Next, we will be able to perform a benchmark, by comparing the execution time between a query made using a graph versus the same rule with AdaControl.

We have to select a subset of AdaControl's rules to perform our benchmark, in order to have enough use cases to have a relevant benchmark. We have to define a "frontier" between the programatic querying versus the GDBMS query language. We have some insights regarding this problem. We can cut the query into several "elementary" queries (like ASIS), or, we can generate the request through another language [13]. We could also add a way to make incremental updates, to only update compilation units that have been modified. Another development will be to switch from ASIS to libadalang for the parsing. This will provide a compiler independent Ada code analysis tool.

References

[1] Y. Xie and A. Aiken, "Saturn: A scalable framework for error detection using Boolean satisfiability," *ACM Transactions on Programming Languages and Systems*, vol. 29, p. 16, may 2007.

[2] T. Ball and S. K. Rajamani, "The SLAM project: Debugging system software via static analysis," in *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '02*, ACM Press, 2002.

[3] "Coverity Scan." `https://scan.coverity.com/projects/`, 2018.

[4] M. Harman and P. O'Hearn, "From Start-ups to Scale-ups: Opportunities and Open Problems for Static and Dynamic Program Analysis," in *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, IEEE, sep 2018.

[5] G. Fan, *Practical static code analysis : challenges, methods, and solutions*. PhD thesis.

[6] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," may 2014.

[7] R.-G. Urma and A. Mycroft, "Source-code queries with graph databases—with application to programming language usage and evolution," *Science of Computer Programming*, vol. 97, pp. 127–134, jan 2015.

[8] R. Ramler, G. Buchgeher, C. Klammer, M. Pfeiffer, C. Salomon, H. Thaller, and L. Linsbauer, "Benefits and drawbacks of representing and analyzing source code and software engineering artifacts with graph databases," pp. 125–148, dec 2018.

[9] R. Angles, A. Bonifati, S. Dumbrava, G. Fletcher, K. W. Hare, J. Hidders, V. E. Lee, B. Li, L. Libkin, W. Martens, F. Murlak, J. Perryman, O. Savković, M. Schmidt, J. Sequeda, S. Staworko, and D. Tomaszuk, "PG-keys: Keys for property graphs," in *Proceedings of the 2021 International Conference on Management of Data*, ACM, jun 2021.

[10] A. Deutsch, N. Francis, A. Green, K. Hare, B. Li, L. Libkin, T. Lindaaker, V. Marsault, W. Martens, J. Michels, F. Murlak, S. Plantikow, P. Selmer, H. Voigt, O. van Rest, D. Vrgoč, M. Wu, and F. Zemke, "Graph pattern matching in gql and sql/pgq," Dec. 2021.

[11] A. Ponomarev, H. S. Nalamwar, and R. Jaiswal, "Source code analysis: Current and future trends challenges," vol. 685, pp. 877–880, feb 2016.

[12] T. L. Alves, J. Hage, and P. Rademaker, "A comparative study of code query technologies," sep 2011.

[13] T. Zhang, M. Pan, J. Zhao, Y. Yu, and X. Li, "An open framework for semantic code queries on heterogeneous repositories," in *2015 International Symposium on Theoretical Aspects of Software Engineering*, IEEE, sep 2015.