Incremental Parsing of Large Legacy C/C++ Software

Anett Fekete, Máté Cserép
Eötvös Loránd University
Faculty of Informatics
Budapest, Hungary
{hutche, mcserep}@inf.elte.hu

ABSTRACT

CodeCompass is an open source project intended to support code comprehension by providing textual information, source code metrics, version control information and visualization views of the file and directory level relations for the analyzed project. Regarding the typical software development methodologies (especially the agile ones), only a smaller portion of the code base is affected by any change during a shorter amount of time (e.g. between nightly builds), therefore parsing the entire project each time is unnecessary and expensive. A newly introduced feature, incremental parsing is intended to solve this problem by only processing files that have been recently changed and leaving the rest alone. This is achieved by the maintenance of the project workspace database followed by the partial parsing of the project. The feature has been tested both on medium and large scale projects and proved to be an effective tool in CodeCompass.

Categories and Subject Descriptors

D.2.3 [**Software Engineering**]: Coding Tools and Techniques; D.3.4 [**Programming Languages**]: Processors

General Terms

Management, Languages

Keywords

code comprehension, software maintenance, static analysis, incremental parsing, C/C++ programming language

1. INTRODUCTION

One of the main tasks of a code comprehension software tool is to provide exact textual information and visualization views regarding the analyzed codebase to support the (newcomer) developers in understanding the source code. For an enterprise software under development this requires the frequent static reanalysis of the program, which could take several hours for a large legacy software.

Performing a complete static analysis each time is a significant waste of computational resources, since in most cases (e.g. between nightly builds) only a few percent of the file set has been affected by any change. In order to boost the parsing and compilation process and to provide richer user experience in integrated development environments (IDEs) [8], the concept of incremental parsing and compilation has been researched since decades. More recently further approaches, like the involvment of version control systems into incremental parsing [14] and the lazy analysis [10] have been studied. A great overview of pratical algorithms and the exsiting methodology is given by Tim A. Wagner in [13]. C/C++ language-specific compilation tools [12, 4] and programming environments [7] supporting incremental parsing have also emerged as an advancement.

CodeCompass [9] is an open source, scalable code comprehension tool developed by Ericsson Ltd. and the Eötvös Loránd University, Budapest to help understanding large legacy software systems. Its web user interface provides rich textual search and navigation functionalities and also a wide range of rule-based visualization features [5, 6]. The code comprehension capabilities of CodeCompass is not restricted to the existing code base, but important architectural information are also gained from the build system by processing the compilation database of the project [11]. The C/C++ static analyzer component is based on the LLVM/Clang parser [1] and stores the position and type information of specific AST nodes in the project workspace database together with further information collected during the parsing process (e.g. the relations between files). By introducing the concept of *incremental parsing* into CodeCompass we can detect the added, deleted or modified files in the program and carry out maintenance operations for the database of the code comprehension tool in only the required cases. Thus the required time of the reanalysis can be reduced by multiple magnitudes.

In this paper first we present our research in Section 2 on how we extended the static analytical capabilities of the CodeCompass code comprehension tool with incremental parsing. Then Section 3 demonstrates the usability of the concept by showcasing incremental parsing and measuring its performance on a medium and a large size C/C++ software. Finally, Section 4 concludes the results and discusses further research opportunities.

2. METHODOLOGY

A major consideration of the introduced incremental parsing feature was to integrate it seamlessly into the existing parsing process by not differentiating in how an initial or a follow-up incremental parse should be initiated. This was achieved by utilizing the *partial parsing* feature of CodeCompass, which means that the tool is capable of continuing a previously aborted analysis, by omitting the already parsed files which are present in workspace database.

Therefore the main concept of the introduced incremental parsing feature consists of two steps: *i*) perform a *database maintenance* operation, where the project workspace is restored into a state that *ii*) the existing *partial parsing* can finish the procedure.

2.1 Determining file states

When a new parse is being done in incremental mode, the state of each file is determined first. Let F_{DB} be the file set stored in the workspace database and F_{DISK} be the file set stored on the disk. An $f \in F_{DB} \cup F_{DISK}$ file may take one of the three states listed as follows.

Added files f is added to the project since the latest parse if $f \in F_{DISK}$ but $f \notin F_{DB}$.

Deleted files f is deleted from the project if $f \in F_{DB}$ but $f \notin F_{DISK}$.

Modified files f is modified when $f \in F_{DB} \cap F_{DISK}$ at the time of the new parse but its content has changed since the latest. This can be determined by comparing the contents that are stored in the database and on the disk, or by their respective hashes for performance optimization.

2.2 Header inclusion traversal

Specifically when parsing a C or C++ language project, changes in header inclusions provide one more challenge to tackle. Upon the modification of a header file all further files in the inclusion chain depending on it should be considered as modified, even without containing any direct changes themselves. Therefore when determining the *modified* state of a file as defined in Section 2.1, the set of files defined by the header inclusion relationships transitively should be checked for changes. There are two approaches for this, as described below and shown in Figure 1.

Definition 1. For files a, b and c, given that a is included by b and b is included by c, we say that file a is in an upward connection with b and accordingly file c is in a downward connection with b.

Upward traversal model The upward traversal model depends on the upward connection between files. When resolving the state of file a, its included headers have to be checked for modifications transitively.

Downward traversal model Similarly, the downward traversal model uses the downward connections that can be found between files. If a file a is resolved as modified, all files that include a can be marked as modified transitively. Note that with this method, the state of any marked files can be considered final and can be omitted from further inspections.

THEOREM 1. *The downward traversal model has better computational complexity over the upward traversal model, and therefore is preferred to be used through the incremental parsing.*

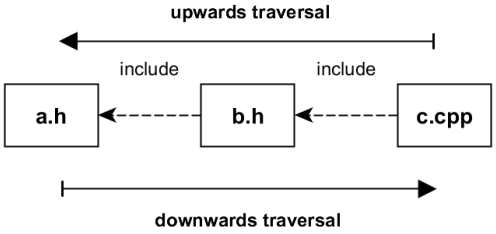

Figure 1: Traversal directions

PROOF. Let $G = (V, W, E)$ be the directed acyclic graph (DAG) of header inclusions with V containing the file set as vertices and E being the set of upward connections, $n := |V|$, $e := |E|$. Let $W \subseteq V$ denote the set of directly changed files, $k := |W|$.

Let $N_G(v)$ be the neighborhood file set of vertex v in G, so $w \in N_G(v) \Leftrightarrow (v, w) \in E$. Therefore for a file v we can define the directly included file set as $N_G(v)$ and the includer files of v as $N_{G^T}(v)$, where G^T is the transpose graph of G.

We define $up(G, v)$ and $down(G, v)$ as the file set result of the upward and downward traversal for $v \in V$ in G by the corresponding traversal model, as formally described below:

$$up(G, v) = \{v\} \cup \forall_{w \in N_G(v)} : up(G, w) \tag{1}$$

$$down(G, v) = \{v\} \cup \forall_{w \in N_{G^T}(v)} : down(G, w) \tag{2}$$

As a simplification in our model lets assume a uniform distribution of header inclusions among the files. Since $\sum_{v \in V} deg^+(v) = \sum_{v \in V} deg^-(v) = e$, the average in-degree and out-degree for a file v is $deg^+(v) = deg^-(v) = \frac{e}{n}$, which will be denoted with d henceforth. As a consequence the length of the longest path in G is $log_d n$, which is the length of the longest header inclusion chain in the project, since G was defined as a DAG.

Therefore the asymptotic tight bound both for $up(G, v)$ and $down(G, v)$ can be calculated as:

$$\Theta(up(G, v)) = \Theta(down(G, v)) = d^{log_d n} = n \tag{3}$$

We define $up(G)$ and $down(G)$ as the upward and downward traversal algorithms which determines indirectly changed files in V through header inclusions from W by the corresponding traversal model. We define the computational complexity of the algorithms as the number of files checked for changes in their content (or by their hash). Based on Equation 3, the asymptotic tight bound both for $up(G)$ and $down(G)$ can be calculated as:

$$\Theta(up(G)) = \sum_{v \in V} \Theta(up(G, v)) = n^2 \tag{4}$$

$$\Theta(down(G)) = \sum_{w \in W} \Theta(down(G, w)) = k * n \tag{5}$$

Since $k \leq n$ and in a typical use case for incremental parsing $k \ll n$: $\Theta(down(G)) < \Theta(up(G))$. □

An example for the downward traversal model is showcased in Figure 2. On the left side of the figure the example file set is shown with header inclusion dependencies denoted as arrows between them. Directly modified files are marked with a dark background, while files requiring expansion through traversal to find indirectly changed files are marked with an italic font. Note, that these two categories are equivalent in the initial stage. On the right side of the figure the effects of downward traversing `a.h` is demonstrated: files `c.h`, `d.h`, `f.cpp` and `g.cpp` are also detected as indirectly changed files. While `c.h` was also a directly modified file, observe that it no longer requires downward traversal.

2.3 Database maintenance

As mentioned above, incremental parsing includes some maintenance of the existing database depending on the state of changed files.

1. *Added files* are perceived as new files to the project and therefore are registered into the database.
2. *Deleted files* need to be purged from the database as they have been removed from the project.
3. *Modified files* are handled as if they were a combination of deleted and added files. First, they are completely wiped out from the database – meaning that all their AST related information and file level relations are erased –, thus considering them deleted, then re-registered like newly added files. Directory level relations are not sufficiently maintainable, but these relations can be effectively computed runtime, on-demand from the file level relations.

3. EXPERIMENTAL RESULTS

The go-to projects on which CodeCompass is usually tested are the Xerces-C++ [3] and LLVM [2] projects. Both are open source projects that have been under development for several years and therefore are considered legacy projects. Incremental parsing was also tested on these two as Xerces-C++ is a medium size and LLVM is a large-scale project and contain enough files (respectively 347 and 2845) to produce a significant difference in runtime between even small portions of changes in the number of files.

Incremental parsing is aimed to reduce the parsing time of builds, especially nightly builds, therefore it was tested on 1, 5 and 10 percent change of the file set, since no bigger difference between two builds is presumable. The changeset was generated automatically by random selection of files.[1] Table 1 shows the results for Xerces-C++, while Table 2 and Table 3 depict the results for LLVM. All measurements were carried out on a standard notebook computer, parsing on 2 processor cores.

In order to keep database consistency in case of a graceful abort or unexpected termination of the parser module, the basic concept is that the maintenance operation of incremental parsing must be performed in a transactional mode, in one of the following ways:

[1]Only leaf nodes from graph G introduced in Section 2.2 were included in the changeset, so header inclusions did not affect the number of changed files.

Table 1: Time measures for incremental parsing the Xerces-C++ project

Parse type	Changed files	Time
Full parse	–	2 min 49 sec
1% change	3	10 sec
5% change	17	21 sec
10% change	35	49 sec

Table 2: Time measures for incremental parsing the LLVM project by one atomic transaction

Parse type	Changed files	Time
Full parse	–	5 h 46 min
1% change	28	7 min 30 sec
5% change	142	1 h 58 min
10% change	284	2 h 45 min

- Carry out all deletions from the database in one *single transaction*, so the maintenance is either completely executed, otherwise no changes are performed.
- Generate multiple *file level transactions*, so information regarding a file is either cleaned from the database or the file is untouched, therefore a consistent state of the database is always kept.

Table 2 and Table 3 compare the differences when the database maintenance is executed through a single and by file level transactions. It is clear that the extensive size of the database rollback log containing all the deletion operations for a larger quantity of files can significantly hinder the effectiveness of incremental parsing, providing significant difference in the timespan of incremental parsing for large size projects like LLVM. Hence while a single transaction may provide stronger guarantees, file level transaction proved to be a more adequate solution, where the required time is more or less linear with the quantity of parsed files, depending on the length and content of the files in question.

Table 3: Time measures of incremental parsing the LLVM project by file level transactions

Parse type	Changed files	Time
1% change	28	9 min 30 sec
5% change	142	49 min
10% change	284	1 h 21 min

4. CONCLUSIONS

Incremental parsing was introduced into CodeCompass to reduce the costs of parsing, both time and computational resources, by omitting unchanged files in the project. The feature distinguishes added, deleted and modified files and handles them accordingly. The early tests of incremental parsing were run on the Xerces-C++ and LLVM projects and showed that it works according to its original purpose, especially in decreasing the timespan of parsing. While the results are promising, further challenges include the improved reduction of the timespan required by incremental parsing through parallelizing the process.

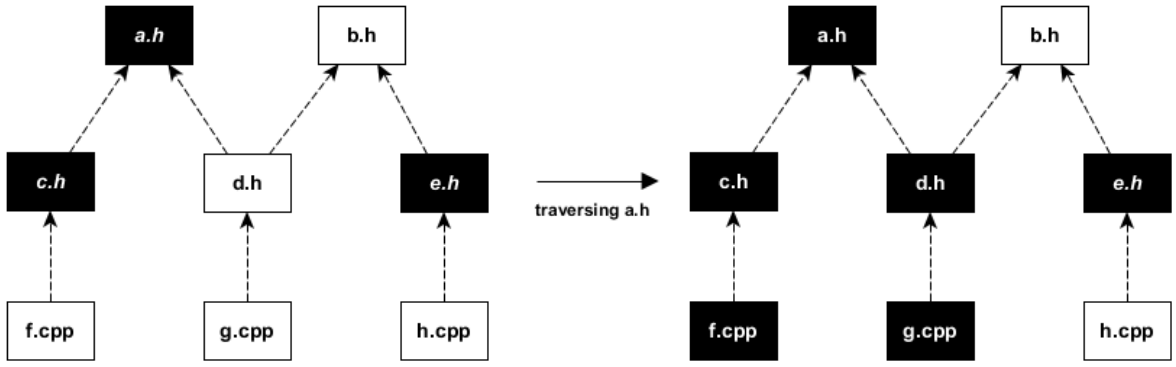

Figure 2: Downward traversing of *a.h* demonstrated on a showcase file set.

5. ACKNOWLEDGMENTS

This work is supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.3-VEKOP-16-2017-00002).

6. REFERENCES

[1] Clang: a C language family frontend for LLVM. `https://clang.llvm.org/`.

[2] The LLVM Compiler Infrastructure. `https://llvm.org/`.

[3] Xerces-C++ XML Parser. `https://xerces.apache.org/xerces-c/`.

[4] Zapcc – A (Much) Faster C++ Compiler. `https://www.zapcc.com/`.

[5] T. Brunner and M. Cserép. Rule based graph visualization for software systems. In *Proceedings of the 9th International Conference on Applied Informatics*, pages 121–130, 2014.

[6] M. Cserép and D. Krupp. Visualization Techniques of Components for Large Legacy C/C++ software. *Studia Universitatis Babes-Bolyai, Informatica*, 59:59–74, 2014.

[7] M. Karasick. The Architecture of Montana: An Open and Extensible Programming Environment with an Incremental C++ Compiler. *SIGSOFT Softw. Eng. Notes*, 23(6):131–142, Nov. 1998.

[8] R. Medina-Mora and P. H. Feiler. An incremental programming environment. *IEEE Transactions on Software Engineering*, (5):472–482, 1981.

[9] Z. Porkoláb, T. Brunner, D. Krupp, and M. Csordás. Codecompass: An open software comprehension framework for industrial usage. In *Proceedings of the 26th Conference on Program Comprehension*, ICPC '18, pages 361–369, New York, NY, USA, 2018. ACM.

[10] V. Savitskii and D. Sidorov. Fast analysis of source code in C and C++. *Programming and Computer Software*, 39(1):49–55, 2013.

[11] R. Szalay, Z. Porkoláb, and D. Krupp. Towards better symbol resolution for C/C++ programs: A cluster-based solution. In *IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 101–110. IEEE, 2017.

[12] T. Tromey. Incremental compilation for GCC. In *Proceedings of the GCC Developers' Summit*. Citeseer, 2008.

[13] T. A. Wagner. *Practical algorithms for incremental software development environments*. PhD thesis, Citeseer, 1997.

[14] T. A. Wagner and S. L. Graham. Efficient and flexible incremental parsing. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 20(5):980–1013, 1998.