Proceedings of the 11th International Conference on Applied Informatics
Eger, Hungary, January 29–31, 2020, published at http://ceur-ws.org

Integration of Incremental Build Systems Into Software Comprehension Tools

Máté Cserép, Anett Fekete

Eötvös Loránd University, Faculty of Informatics, Hungary
{mcserep,afekete}@inf.elte.hu

Abstract

Standalone code comprehension tools and similar features of integrated development environments (IDE) both aim to support the development and the maintenance of large (legacy) software. When applied to actively developed projects, it is essential to process the most recent revision of the source code in real time. Since a complete analysis of the codebase might take up significant time (even hours), the inclusion of incremental parsing is indispensable. However, the utilized build system of a software project is tightly coupled with the source code: over the time not only the content of the source files can be amended, but translation units can be added or removed and the parameters of the existing build instructions might also change.

This paper intends to describe how the incremental update of the build system of a software facilitates the maintenance of the software workspace database in a code comprehension tool by completing the workflow of incremental parsing. We describe why including the build system in incremental parsing is relevant as well as the actual method of parsing build commands. We show that updating the build system is more cost-effective to a ratio than disposing of the existing build command database. The paper also compares the incremental parsing of build systems to that of actual source code.

Keywords: code comprehension, build system, incremental parsing, software maintenance, static analysis

MSC: 68N20

1. Introduction

When programmers change jobs, they usually join a long-running project with a large codebase instead of starting a new project. This might cause a hard time for the programmer since there can be millions of lines of code (LOC) that they have to understand in order to be able to execute their programming tasks and interactively join the project. A significant part of the software maintenance process is *program comprehension*, which is an activity to understand software architectural and programming decisions, choices and logic along with the source code itself.

Although code comprehension is already a hard task, the lack of documentation and other external representation of the code can make it even more difficult. Several software tools exist in order to facilitate software understanding. Numerous code editor features such as find all references or go to definition / declaration can be used by the programmer for code comprehension tasks, but these functionalities cannot replace standalone comprehension and software visualization tools like CodeSurveyor [5], OpenGrok [10], CityVR [8], or CodeCompass [11].

Standalone tools provide various textual and visual data about the source code after processing them utilizing static and/or dynamic analysis. In order to be able to make information available for the programmer, the comprehension software needs to process – *parse* – the source code unit by unit where a unit represents a source file or one line in a file. An actively developed project changes every day, which means the information provided by a software comprehension tool on a day might be incorrect the next day. This is why comprehension tools require frequent reparsing in order to provide correct information. However, frequent reparsing of a whole software project is quite costly, both in computational time and complexity. Hence, incremental parsing, which means parsing only those parts of the project that have been actually changed, is inevitable. Single code lines or entire files can be parsed incrementally. Our method deals with the latter approach.

By applying incremental parsing, severe computational cost can be spared. In case of an actively developed project, the daily usage of the parsed code can be significantly facilitated by incrementally parsing nightly builds, but real-time incremental parsing can be achieved as well by integrating it with code editors [9].

As a case-study we realized incremental parsing of build systems through CodeCompass, a standalone code comprehension tool developed by Ericsson Hungary and Eötvös Loránd University which already encompassed the incremental parsing of source code. In order to test our method, we used the open-source LLVM project, which is under continuous development and has a frequently changing build system.

The rest of the paper is strcutured as follows. Section 2 describes how modern build systems address the issue of incremental building. Section 3 introduces the CodeCompass code comprehension tool, then Section 4 presents our methodology of incremental parsing of build systems and its integration into CodeCompass. Afterwards, Section 5 showcases a performance test on LLVM. Finally, Section 6 discusses the results and concludes the paper.

2. Background

Build systems [6] in general are standalone software whose goal is to provide language-independent solution to define build rules and actions. Incremental building is applied by several build systems such as Ant [7], Maven [12], or GNU Make [4]. However, most build systems do not determine changes in files by detailed, line-by-line static analysis on the file content, nor do they integrate with any version control system's repository.

For example, GNU Make builds the source files based on a file called *Makefile*, which contains rules by which the compilation is executed. We assume that the project has already been built. In this case, we already have binaries that were compiled from the source code using the Makefile. The build system examines the *last modified* timestamp of every source file and the according target. Based on the comparison, the source files are classified either as *unbuilt* (recently added), *obsolete* (recently updated) or *up-to-date*. Other build rules include the examination of dependencies between source files such as include relations. After this, target binaries are built thus old binaries are updated. Figure 1 illustrates the incremental build workflow executed by GNU Make.

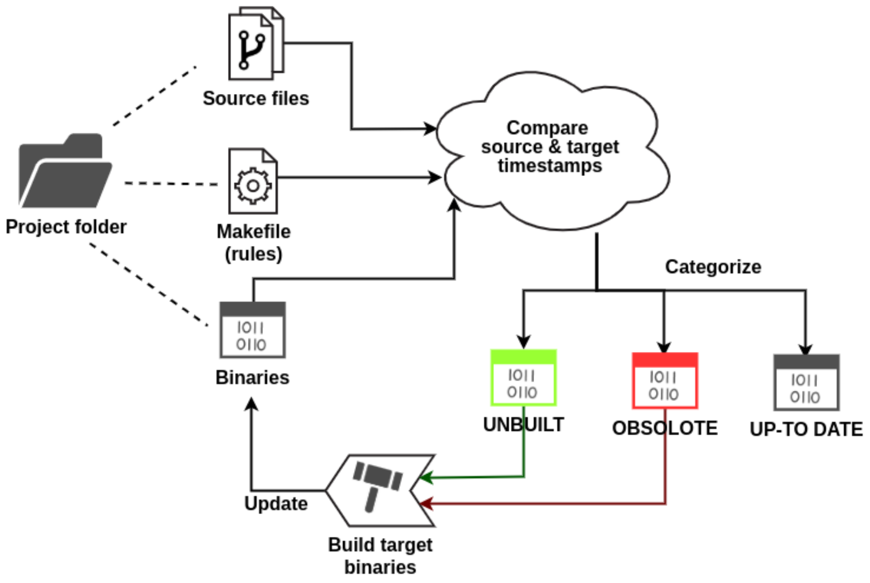


Figure 1: Incremental build workflow of *GNU Make*

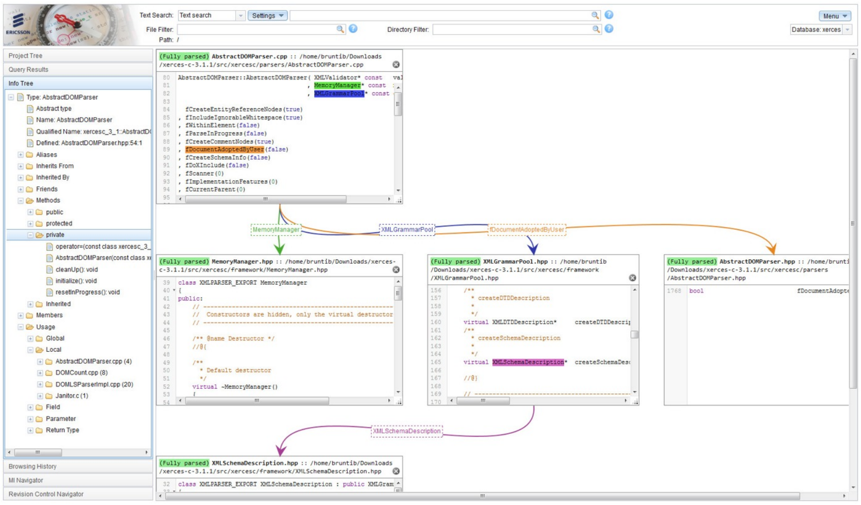
Another example for a build system with such an incremental approach is CMake[1], which is practically a generator for other build systems like GNU Make. Its main purpose is to define the build rules and list them in the Makefile (or other similar file). However, CMake is not only capable of timestamp comparison like GNU Make but also examines compilation parameters. A source file is not only recompiled when its content has been directly modified but when its build rules (e.g. dependencies) have been changed as well.

[1]CMake: Build with CMake: `https://cmake.org/`

3. CodeCompass

CodeCompass [11] 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 [1, 2]. The web user interface of CodeCompass is presented in Figure 2, showcasing a special, interactive *CodeBites* diagram, which supports the understanding of large call chains and type hierarchies. The code comprehension capabilities of CodeCompass is not restricted to the existing code base, but important architectural information is also gained from the build system by processing the compilation database of the project [13] and the version control history of the project when available.

The C/C++ static analyzer component is based on the LLVM[2] compiler infrastructure and the Clang[3] tooling infrastructure for C/C++. The parser 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).



(Fully parsed) **AbstractDOMParser.cpp** :: /home/bruntib/Downloads/xerces-c-3.1.1/src/xercesc/parsers/AbstractDOMParser.cpp

```
AbstractDOMParser::AbstractDOMParser( XMLValidator* const   val
                                    , MemoryManager* const
                                    , XMLGrammarPool* const

  fCreateEntityReferenceNodes(true)
, fIncludeIgnorableWhitespace(true)
, fWithinElement(false)
, fParseInProgress(false)
, fCreateCommentNodes(true)
, fDocumentAdoptedByUser(false)
, fCreateSchemaInfo(false)
, fDoXInclude(false)
, fScanner(0)
, fImplementationFeatures(0)
, fCurrentParent(0)
```

(Fully parsed) **MemoryManager.hpp** :: /home/bruntib/Downloads/xerces-c-3.1.1/src/xercesc/framework/MemoryManager.hpp

```
class XMLPARSER_EXPORT MemoryManager
{
public:
    // -----------------------------------------------------
    //  Constructors are hidden, only the virtual destructor
    // -----------------------------------------------------

    /** @name Destructor */
    //@{

    /**
      * Default destructor
      */
    virtual ~MemoryManager()
    {
```

(Fully parsed) **XMLGrammarPool.hpp** :: /home/bruntib/Downloads/xerces-c-3.1.1/src/xercesc/framework/XMLGrammarPool.hpp

```
    /**
      * createDTDDescription
      *
      */
    virtual XMLDTDDescription*     createDTDDescri
    /**
      * createSchemaDescription
      *
      */
    virtual XMLSchemaDescription*  createSchemaDes

    //@}

    // ------------------------------------------
```

(Fully parsed) **AbstractDOMParser.hpp** :: /home/brunti /Downloads/xerces-c-3.1.1/src/xercesc/parsers /AbstractDOMParser.hpp

```
bool                              fDocumentAdopte
```

(Fully parsed) **XMLSchemaDescription.hpp** :: /home/bruntib/Downloads/xerces-c-3.1.1/src/xercesc/framework/XMLSchemaDescription.hpp

```
class XMLPARSER_EXPORT XMLSchemaDescription : public XMLGram
```

Figure 2: The interactive *CodeBites* diagram of CodeCompass

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

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

4. Workflow

In our previous work [3] we have already described how CodeCompass was extended with the concept of incremental parsing. The implemented feature successfully distinguished added, deleted and modified files (including the traversal of indirectly changed files through header inclusions) and carried out maintenance operations for the database of the code comprehension tool in only the required cases. Therefore, the required costs of parsing (both time and computational resources) of the reanalysis was reduced by multiple magnitudes, by omitting unchanged files in the project. We have also presented [9] how incremental parsing is crucial to provide real-time reanalysis of the modified source code inside integrated development environments (IDEs) by embedding the feature set of code comprehension tools into code editors through remote procedure calls defined by the open-source *Language Server Protocol* (LSP).

For software projects under active development, not only the content of the source files may change frequently, but the configuration or rules of the utilized build system are also modified regularly. While the initial tests evaluated on the Xerces-C++ and LLVM go-to test projects of CodeCompass were promising, the challenge of detecting changes in the build system or the compilation commands were yet unhandled.

The methodology of incremental parsing in CodeCompass was inspired by the incremental strategy of build systems presented in Section 2. However, while build systems usually depend on comparing source file and target binary timestamps along the defined ruleset, the static analyzer component in CodeCompass can only use the source files as input, since the goal of the process is not building the analyzed software and the previously built binaries are not required to be available – if built at all in the first place. The incremental parsing workflow of CodeCompass is depicted in Figure 3. Our approach depends on two input sources: *i*) the actual version of the software, labeled as the *project folder*; and *ii*) the last analyzed state of the software stored in a relational database, labeled as the *workspace database*. In order to detect changes in the build system, compilation commands are stored upon the analysis of a software project and compared between the current and the last analyzed revision. Compilation commands can be defined in the JSON export format of CMake. To support other build systems, CodeCompass includes a *logger* component which can embed the building process of the target project, logging all compilation commands in the required format, thus making CodeCompass compatible with all C/C++ projects, regardless of the build system utilized.

Based on their content and the compilation commands, source files are classified into five categories as follows:

1. Files with different content on in the workspace database and the project folder were classified as **content modified**. Files with indirectly modified content through header inclusion also fall into this category.

2. Files with no content modification, but with a changed compilation com-

mand (even a flag) in the workspace database and the project folder were categorized as **build action changed**.

3. Files present in the workspace database, but not in the project folder were grouped as **deleted**.
4. Files present in the project folder, but not in the workspace folder were marked as **new**.
5. Files present both in the workspace database and in the project folder, with an unchanged content and compilation command were classified as **unchanged**.

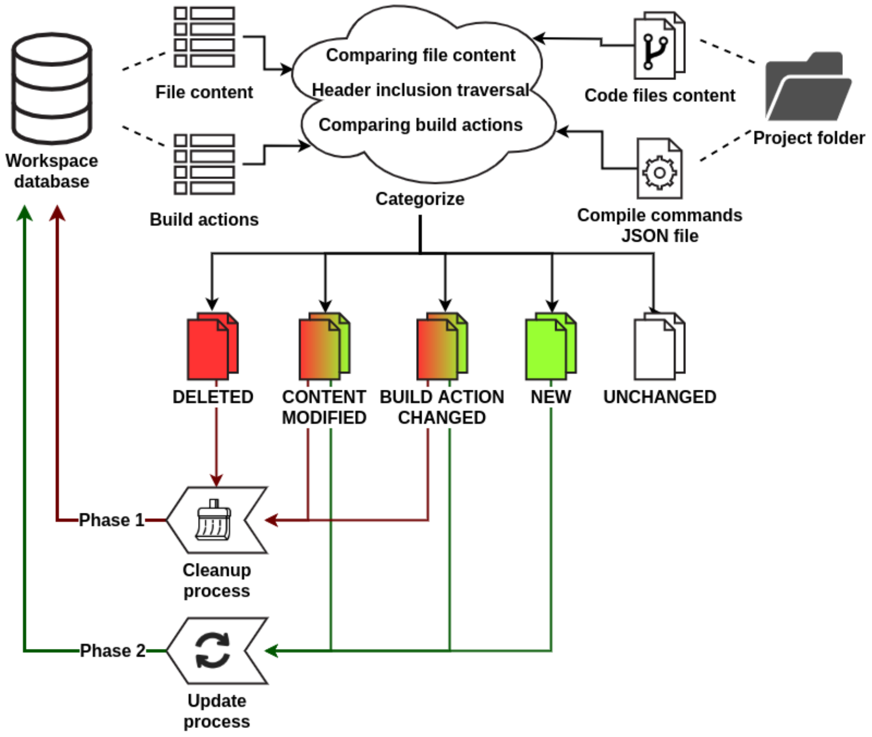


Figure 3: Incremental parse workflow of *CodeCompass*

The maintenance of the workspace database is carried out in two phases. First, a *cleanup process* is executed, removing all related information from the database for the source files categorized as *deleted*, *content modified* or *build action changed*. Afterwards, the new content is parsed during the *update process*, which will include the source files marked as either *new*, *content modified* or *build action changed*. For *unchanged* files, no task has to be executed.

5. Results

To evaluate the performance of the incremental parsing methodology described in Section 4, the prototype implementation in CodeCompass was tested against the open-source LLVM project. LLVM is a large-scale go-to test project for CodeCompass, which contains over 5000 C/C++ source files with more than 1.6 million lines of code; and is actively developed with usually multiple dozens of commits daily affecting hundreds of source files. The testing environment was an average personal computer with 4 CPU cores and 16GB RAM, the incremental parsing was performed as a single thread process.

The goal of the test was to compare the required build and parse time for LLVM in case of *i*) a full clean build and $ii - iv$) on-demand incremental rebuild after code changes were carried out on the original codebase. The $ii - iv$) cases were executed for three changesets: containing 1, 7 and 17 changed C++ source files respectively. Table 1 describes the results in detail. Number of files written in the *Affected files* column include both the directly or indirectly content modified C/C++ files and the files affected by compilation command alteration.

Test case	**Affected files**	**Task**	**Execution time**
Full clean build	all	build	140 min
		parse	387 min
Incremental rebuild #1	1	build	129 sec
		parse	89 sec
Incremental rebuild #2	7	build	112 sec
		parse	110 sec
Incremental rebuild #3	17	build	183 sec
		parse	169 sec

Table 1: Test results of full and incremental parsing on LLVM

6. Discussion and conclusion

In our paper we presented our method on how to determine the changes in a software project's source code or in its compilation commands, refining our previous approach on incremental parsing. Our solution described in Section 4 eliminates the required presence of target binaries, on which build systems with incremental building capability usually depend on, and solely depends on the comparison of the source of the project at two different revisions, categorizing which files have to be reanalyzed. The implementation was carried out as part of the CodeCompass code comprehension tool.

The results presented in Section 5 proved that implementing incremental parsing in a code comprehension tool is capable of processing code changes on the magnitude of a usual version control commit size or local modification in a couple of minutes even on a standard desktop computer. By delivering results in such a significantly reduced time, our approach enables the integration of IDEs and code comprehension tools and utilizing incremental parsing in everyday code development.

Acknowledgements. This work was supported by the construction EFOP-3.6.3-VEKOP-16-2017-00002. The project was supported by the European Union, co-financed by the European Social Fund.

Computer Code Availability. The source code of the CodeCompass Project is available on GitHub at `https://github.com/Ericsson/CodeCompass`. A live demonstration of the parsed *master* branch of the LLVM project is showcased at `https://codecompass.zolix.hu/`.

References

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

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

[3] Fekete, A., and Cserép, M. Incremental Parsing of Large Legacy C/C++ Software. In *21th International Multiconference on Information Society (IS), Collaboration, Software and Services in Information Society (CSS)* (2018), vol. G, pp. 51–54.

[4] Free Software Foundation. GNU Make Manual. Tech. Rep. 0.74, 5 2016.

[5] Hawes, N., Marshall, S., and Anslow, C. Codesurveyor: Mapping large-scale software to aid in code comprehension. In *2015 IEEE 3rd Working Conference on Software Visualization (VISSOFT)* (2015), IEEE, pp. 96–105.

[6] Jameson, K. W. Collection makefile generator, Feb. 21 2006. US Patent 7,003,759.

[7] McIntosh, S., Adams, B., and Hassan, A. E. The evolution of ant build systems. In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)* (2010), IEEE, pp. 42–51.

[8] Merino, L., Ghafari, M., Anslow, C., and Nierstrasz, O. Cityvr: Gameful software visualization. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)* (2017), IEEE, pp. 633–637.

[9] Mészáros, M., Cserép, M., and Fekete, A. Delivering comprehension features into source code editors through lsp. In *2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)* (2019), IEEE, pp. 1581–1586.

[10] Oracle. OpenGrok: A wicked fast source browser. `https://oracle.github.io/opengrok/`.

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

[12] Raemaekers, S., Van Deursen, A., and Visser, J. Semantic versioning versus breaking changes: A study of the maven repository. In *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation* (2014), IEEE, pp. 215–224.

[13] Szalay, R., Porkoláb, Z., and Krupp, D. 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)* (2017), IEEE, pp. 101–110.