TECHNISCHE UNIVERSITEIT EINDHOVEN
Department of Mathematics and Computer Science

MASTER'S THESIS

Fact Extraction, Querying and Visualization of Large C++ Code Bases

Design and Implementation

by
F.J.A. Boerboom
A.A.M.G. Janssen

Supervisor:

Dr. A.C. Telea (TUE)

Eindhoven, August 2006

Abstract

Reverse engineering aims to increase the developer's understanding on software products. The source code of a software product typically contains all the knowledge of the software product. The first task in reverse engineering is the processing of the source code so as to obtain the desired data, and is performed by programs known as fact extractors. There are many variations in the requirements of a fact extractor depending on the task at hand. Facts can range from low-level information on expressions as well to high level information such as class diagrams and metrics. Our original goal was to find a suitable fact extractor for the C++ programming language that is able to handle industry-size source code bases of millions of lines of code, as a basis for a source code visualizer. We have conducted a survey and found no fact extractor to be suitable for our project. Instead of matching our visualization capabilities to a certain fact extractor, we have decided to create a new fact extractor, which we call EFES, based on Elsa, one of the survey participants. Our fact extractor retrieves all information about the source code, the abstract semantic graph (ASG) and more, and stores this in an efficient custom database format. We have created a source code visualizer. The visualizer allows for the visualizing of the source code represented by the ASG and is based on a versatile visualization framework. The visualizer offers a scalable view with multiple columns for the display of up to 50 KLOC in a single image. We have created a query system on top of our visualizer. The query system enables the rapid searching for potentially complex patterns in the ASG.

Acknowledgments

First and foremost, we thank dr. Alex Telea for his excellent support throughout the entire project. His experience in the area of research helped us create a high grade program.

We thank LaQuSo, in particular dr. Alexander Serebrenik, for providing us with a real world scenario. We successfully applied our fact extractor and query system, for reverse engineering of industrial software, in their scenario. Their scenario gave us new insights in the difficulties of reverse engineering.

During our project we have applied a new software visualization due to ir. Danny Holten for which we are grateful to him. He kindly provided a version of his tool in development and supported us by promptly implementing our feature requests.

We thank Scott McPeak, Ph.D. the author of the outstanding Elsa C++ parser, for releasing his project under the BSD license. Only because of his work we were able to create our fact extractor which is a crucial component of all our work.

Contents

List of Figures

Chapter 1

Introduction

1.1 Problem statement

Understanding nowadays software systems is a tremendous task. Middle-sized code bases, containing the source code of a system, can easily have millions of lines of code, spread in thousands of files, developed by tens of programmers over many years. For industry-size applications such as banking systems or telecom frameworks, these figures are orders of magnitude larger. Many activities related to software quality assessment and improvement, such as empirical model construction, program flow analysis, reengineering, and reverse engineering [66], rely on static source code analysis [30] as the first and fundamental step for gathering the necessary input information. Static code analysis encompasses all the activities performed to extract facts about the software directly from the source code. Facts range from low-level information, e.g. complete syntax trees, to refined information such as metrics [16, 22] and code patterns [27, 10]. C++ and its predecessor C are the most widespread programming languages nowadays and have been used to code some of the largest, most complex software systems in use [67]. C++ offers probably the richest, but also dauntingly complex, set of features among the general-purpose, widely used, programming languages nowadays. However, this sophistication comes together with an added difficulty to understand C++ code, especially in large systems.

A first, crucial step in supporting the understanding of large C++ code is the availability of a powerful fact extractor for this language [72, 23]. Fact extractors are tools that directly read, or parse, source code and produce various information used in the static code analysis [8]. Many requirements exist for fact extractors, which have led to the development of a wealth of such tools. However, this has also led to many problems in practice. First, the often subtle assumptions and capabilities of existing extractors are rarely clearly explained and compared. Users have thus difficulties choosing the right extractor that matches some given requirements on the input (C++ source code) and output (quality and type of extracted facts). Every specific analysis task will ultimately come up with its own set of requirements. Yet, in practice, to satisfy a broad spectrum of users, an extractor should satisfy the following typical requirements (see Sec. 2.2.1 for a detailed list): The extractor should be able to handle tens of millions of lines of code (MLOC) in reasonable time. All facts should be correctly extracted, yielding a complete annotated syntax tree of the input. Complex language constructs, such as C++ templates, should be analyzed correctly. The extractor should be able to handle incomplete and/or incorrect source code gracefully, i.e. code that will not compile due to missing definitions, `#include`s or plain errors. However, most popular extractors we are aware of do not fully handle all these requirements. Finally, even the most versatile extractor might need to be modified, as the language evolves and/or users might have custom requirements [72]. Yet, few C++ extractors have open, documented designs that facilitate extension and/or modification. This leads many users to the necessity of constructing their own extractor, e.g. by extending an existing one with the desired extra features. Unfortunately, this process is very complex and its success depends on many minute technical details which are not

often emphasized in the literature.

Once a fact database is produced by a fact extractor, we arrive at the next step of supporting the understanding of large C++ code. A fact database contains too much unstructured information for the user to comprise. There are many different ways to look at this information. A method is needed for selecting the facts of interest and visualizing this information in an insightful way.

First, the source code text can be visualized. This is the most familiar method for visualizing C++ code because it is used by almost all source code editors. The visualization of the source code text can be enhanced by coloring, highlighting or underlining parts of the text. For example, unmatched braces can be colored red, or erroneous code can be underlined.

Most source code editors provide only simple lexical highlighting to give visual cues about the structure of a program. However, this only visualizes the structure at the lowest possible level. Source code also has a higher level structure, namely its syntactical structure. The reason most source code editors do not visualize this information, is because it is hard to extract this information directly from the source text.

Visualization tools based on a fact database have the information about the syntactic structure of the source code directly available and can utilize this information to enhance the visualization. The only condition is that the fact database must contain all information about the source code, because the original source text must be reconstructed. Some fact databases also store semantical information, which further increases the possibilities for source code visualization. For example, the visualization can use the information for highlighting all missing includes or all calls to undeclared functions.

The drawback of visualizing source code lines, is that at most 50 KLOC can be visualized on a single screen [73]. Other visualizations are necessary for giving higher lever overviews of a software system. Examples are call graph, class diagram, and tree map visualizations.

Finally, there should be an appropriate method for navigating through source code. There are relations between elements in the source code, and these can be used for navigation purposes. For example, it is awkward if the user has to scan the source code to find the declaration of a variable used in an expression. This can be done automatically if the information is available in the fact database.

1.2 Solution

Our solution consists of two components, viz. a fact extractor for C++, and a visualization framework. In the following, we shall use the name EFES for the fact extractor itself, and Visual EFES for the complete framework, which integrates the fact extractor, query system, and visualization tools in a single application.

EFES extracts all information from source code and stores it in a fact database. The extractor is fault-tolerant and can be applied on incorrect and erroneous source code. It supports almost all language features of standard C++, as well as several compiler specific language extensions. The extractor has its own preprocessor solution, a modified GCC preprocessor, which stores the begin and end locations of all tokens.

The visualization component, described in chapter 5, gives insight in a fact database by providing several kinds of visualizations of the source code. The visualization component is a very flexible framework with light weight interfaces. For each level in the hierarchy the visualization framework can use a different renderer. One of the renderers we implemented for the visualization framework is a source code renderer, which can reconstruct and render the original source code based on the information stored in a fact database. The basis of every visualization is a selection of nodes. Another substantial part of the visualization component is the integrated query system. The query systems offers the functionality to search for arbitrary structural information in the source code. In chapter 3.7 we detail the design of the query system. We use the query system to make selections of elements in source code. These selections can then be visualized, for example by highlighting them in the source code visualization. The query system is also used for source code navigation.

1.3 Roadmap

Chapter 2 gives an overview of available fact extractors and visualizers for code and call graph visualization. It lists a number of requirements that we formulated for evaluating fact extractors. A number of well-known, widely-used fact extractors are then evaluated against our requirements. Their limitations with respect to the proposed requirements are described. From this analysis, conclusions are drawn as to how to build a new fact extractor that complies with all our requirements.

In chapter 3 the design of our fact extractor, based on the Elsa C++ parser, is presented. A description is given of the design steps we took to construct the proposed extractor. It is explained how we overcame several technical difficulties to comply with the proposed arguments. Chapter 7 discusses the effectiveness and efficiency of our proposed tool by presenting concrete results and benchmarks on industry-sized code bases.

Fact databases contain a tremendous amount of information, and users want a specific look at this information. In chapter 4 our needs for querying and enriching facts are stated. Subsequently, the design of our solution, a generic query system, is presented. The chapter finishes with a number of examples of common queries. For each query an example of its use and a possible implementation is given.

Chapter 5 shows how the information in fact databases and query results can be visualized. Facts and queries are useful, but extremely complicated datasets. Using a visual and interactive presentation they can be understood easier and faster.

In chapter 6 several applications of our tool set are given. The performance on a variety of benchmark projects is elucidated and two reverse engineering scenarios are given. The reverse engineering examples show two real world scenarios of using our tools for carrying out a specific task.

Finally, chapter 7 reflects on the previous chapters and describes in which degree we were able to meet our objectives stated in the introduction. The chapter finishes with the section future work, presenting directions of future work.

1.4 Allocation of work

Since this is a double thesis, written by two people, we will detail the allocation of work in this section.

The following table discriminates which sections have been authored by whom. Each section not mentioned in the table has been jointly written.

section	author
3.2.1	Frans
3.2.2	Arjan
3.3	Arjan
3.4	Arjan
3.5	Frans
3.6	Frans
3.7	Arjan
4	Arjan

section	author
5.2	Arjan
5.3	Frans
6	Arjan
7.3	Arjan
7.4	Frans
8.3	Arjan
8.4	Frans
8.5.2	Arjan
8.5.3	Frans

Chapter 2

Previous work

2.1 Introduction

The first task of our project was to look for and evaluate existing fact extractors. This chapter presents the results of our survey. We discuss these fact extractors further in chapter 7. In the second part of this chapter we introduce existing call graph and source code visualizers that relate to our work.

2.2 Fact extraction

Numerous C++ source code fact extractors exist nowadays. We discuss here how a number of such tools, ranging from limited but simple to complex but powerful comply with our requirements from Section 2.2.1. In contrast to the related discussion of Van den Brand *et al.* in [72] on the pros and cons of various parser technologies, our focus here is on well-known, publicly available extractors, parsing only C++, and from a tool end-user perspective. We shall however also include a few commercial, closed source extractors in the discussion, when such tools are important players in the field. We use markers like (req. 3) to refer to the requirement set from Section 2.2.1. We summarize our findings in Table 8.1 and draw conclusions as to how to architect a C++ fact extractor that overcomes the various limitations of the discussed solutions.

2.2.1 Fact extractor criteria

As outlined in Section 1.1, fact extractors come in many variations and comply with a highly variable set of requirements. We present next the set of most common requirements for high-quality, versatile C++ fact extractors, structured in ten categories:

1. Fault-tolerance

   Fact extraction must be possible on incomplete and/or incorrect source code, i.e. code that does not directly compile. Such code can either suffer from missing include files, causing missing declarations, or syntactic or semantic errors in the code itself. Non fault-tolerant extractors are of limited use. Similar to Knapen *et al.* [42], our experience is that one misses, during analysis, the complete compilation context in which the code at hand was written. For example, when analyzing Linux code running a Windows-based extractor, the include set is not (fully) available. Moreover, localized small-scale syntax errors that halt a compiler should not block the fact extraction.

2. Completeness and correctness

   All non-faulty, i.e. parseable, syntactical constructs should be extracted correctly so that a complete syntax tree is produced. This is particularly important for a general-purpose

extractor, as one does not know which constructs are important or not for the subsequent analysis. Thus, all syntactic information should be made available by the extractor, including complex language constructs such as templates. Moreover, the extractor should guarantee correct extraction. As we shall see, this is not always the case for some popular C++ extractors.

3. Compliance

   The extractor should handle the latest C++ standard, and also understand the specifics of major C++ dialects, e.g. g++, Borland C++, and Visual C++. The extractor should be easily extendable to support future C++ language extensions or dialects (e.g. Symbian C++), as much as possible in a modular way.

4. Cross-references

   Cross-references in the source code should be resolved correctly and completely by the fact extractor. Cross-references exist between variable declarations and their use; function declarations, definitions and calls; class declarations, definitions and use, and more. This requires the extractor to correctly apply all C++ language lookup and disambiguation rules [67]

5. Preprocessing

   The fact extractor should be able to preprocess source code of arbitrary complexity. All information from the preprocessing stage, such as macros or original line numbers, should be preserved and saved into the extractor's output.

6. Full coverage

   The C++ grammar allows code to be written with a context dependent meaning, causing possible local ambiguities. Fact extractors should leave such ambiguities in the output only if the ambiguity cannot be resolved due to lack of sufficient contextual information.

7. Output completeness

   The output of the fact extractor should include all extracted information, e.g. construct source code location, type, and preprocessing information. Combined with requirement 2, this enables users, or other tools, to examine all aspects of the analyzed code. The exact construct location, i.e. line and column position in the source code of every C++ construct, is important if one wants to use the extracted information for visualization purposes, as discussed in [40, 47, 73, 70]

8. Performance and scalability

   The extractor should be stable and efficient enough to parse industry-sized code bases, containing hundreds of millions of lines of code (MLOC) and complex C++ constructs in a time comparable to the compilation time of that code.

9. Portability

   The extractor should run on different platforms, e.g. Linux and Windows.

10. Availability The extractor should be available in one of the many open source variants. This implies it should not be a commercial product, and also that its source code should be available for potential modifications.

Clearly, the above set of requirements is not absolute. For us, the above set of requirements has been simply distilled from daily practice in building source code analysis, reverse engineering, and visualization tools in the last six years. Interestingly, a very similar set of requirements on C++ fact extractors (except the availability one) are presented by Semantic Designs, the company behind the powerful DMS program analyzer tool, in their on-line comparison of C++ front ends [62]. This is not by chance, as both the creators of DMS and ourselves aim at a C++ fact extractor that efficiently extracts all low-level facts from industry-size, possibly incorrect and incomplete C++ code written in various C++ dialects.

2.2.2 GccXFront / BisonXML

BisonXML [58] is a modification of the parser generator Bison [20], used by some versions of the g++ compiler. BisonXML writes XML output whenever it performs a grammar reduction rule. This method works for all parsers built with Bison. Yet, few C++ parsers use Bison, because the ambiguities in C++ make it a very hard language to parse with Bison's LALR(1) parsing method [11]. For example, the g++ 3.4.5 parser is hand-written in C. Overall, BisonXML is quite limited. The resulting syntax tree output in XML reflects only the grammar productions executed, and thus is very different from the complete annotated syntax graph (ASG) we are interested in. Moreover, BisonXML maintains no symbol table information, so cross-references are missing in the output (req. 4). Since BisonXML uses a strict grammar, just as the g++ compiler, it will halt on incomplete or incorrect code after a small number of syntax errors(req. 2). Although BisonXML won't directly support non g++ dialects, e.g. Visual C++, it could be modified relatively easily to do so. Finally, BisonXML does not give location information in the output (req. 7).

2.2.3 Src2srcML

Src2srcML [17] is a fact extractor written from scratch by a research group, able to recognize C++ code in a liberal way (req. 1). The parser of Src2srcML is written using the ANTLR parser generator [52]. It uses the so-called island grammars instead of a complete C++ language grammar [50]. The Src2srcML tool has been used for several software projects, such as XWeaver [49].

Src2srcML uses the srcML file format for output [48]. XML tags are interleaved with the original source code. Src2srcML does have the advantage that incorrect input does not easily halt the parser. Output files are compact, but that is mainly because much information is lost (req. 7). The tool tries to parse the code inside function bodies. However, our tests showed that the output facts were often incorrect (req. 2). Src2srcML does not resolve C++ ambiguities correctly, but merely seems to 'guess' the code structure and semantics (req. 6). No symbol cross-references are output (req. 4), which suggests that Src2srcML does not maintain a symbol table. The tool does not come with a preprocessor (req. 5). Src2srcML reports encountered comments and preprocessor directives, but does not actually perform the textual substitutions. Hence, Src2srcML needs an external preprocessor. Src2srcML is relatively fast, as is it ignores include files and does not do the extra work needed to correctly resolve ambiguities. Overall, Src2srcML does not comply well with our requirements.

An interesting question is whether ANTLR, the parser generator that underlies Src2srcML, could be used to construct a parser for the complete C++ language. ANTLR is a powerful $LL(k)$ parser generator that allows driving the parsing via complex predicates using semantic and syntactic context information, combines lexical and syntactic analysis in one step, and provides mechanisms to write user code to handle parsing errors [52]. ANTLR was used in numerous industrial and academic projects. However, to fulfill our completeness requirement, one would need to provide ANTLR with a complete C++ language grammar *and* the associated predicates *and* symbol table management needed to disambiguate parsing and to store extracted facts. There have been several attempts to build C++ fact extractors with ANTLR by writing a C++ grammar, predicate set, and related symbol table, e.g. [42]. However, we are aware of no such result that implements a full symbol table (i.e. is complete in terms of Sec. 2.2.1), has full language coverage, and also handles incomplete and incorrect code. Hence, we have ruled out ANTLR as a basis for our desired fact extractor.

2.2.4 Doxygen

Doxygen [31] is a widely used documentation system for C++, C, Java, Python, Objective-C, IDL (Corba and Microsoft flavors) and to some extent PHP, C#, and D. Doxygen includes a fuzzy parser [43], meta programmed using the Flex scanner generator [53], and implemented as a scanner with state. Unknown constructs are simply ignored (req. 1). Doxygen treats source code in local

(e.g. function) scopes as ordinary text, which it immediately formats by inserting line numbers and adding markup. Hence, information about the structure of this code is not propagated to other components of the system, and cannot be exported at a later stage. Parsing of interface elements, e.g. global type and function declarations, is taken care of quite extensively and is fault tolerant. However, local scope constructs are not supported. Adding such support in the light of requirements (req. 2) and (req. 4) would require an almost complete rewrite of the tool. The tool has the option to generate XML output. However, the information stored therein focuses only on documentation aspects of the code (req. 7). The DOXYGEN code is stable, well structured, modular, and ported on many platforms (req. 9). The modular structure makes it easy to add new output generators, if this is desired. Overall, however, DOXYGEN is intended as an interface documentation tool rather than a full-fledged complete fact extractor.

2.2.5 Sniff+

SNIFF+ is a commercial product available from Wind River, currently part of the larger Wind River Workbench development environment [40]. SNIFF+ is an efficient and portable C++ programming environment which makes it possible to manage, edit, browse, compile, and debug large software systems via a mixed textual and graphical user interface. SNIFF+ and related tools are available on a wide range of platforms, and are used in the development of many large-scale industrial software projects. Here, we shall focus only on the C++ parser component used by SNIFF+ [13]. This is a fuzzy parser, which can handle incomplete and incorrect C++ code. SNIFF+ provides an open symbol table API for obtaining symbol information, e.g. for communication with the various tools available in the development environment. Although SNIFF+ has an open architecture and documented set of APIs for tool integration, the actual details and internal parser construction are not disclosed. By studying the provided documentation, APIs, as well as the original design published by Bishofberger in [13] and related studies that used SNIFF+ [66, 8], we found out that SNIFF+ supports only a partial parsing of the C++ language, focused on global scope symbols, e.g. global class, function, and variable declarations. Information inside function bodies is, for example, almost entirely skipped from analysis, and thus not present in the provided symbol table. Also, SNIFF+ does not always guarantee the correctness of the extracted information and related cross-references [13]. Moreover, the C++ parser itself is closed source, so it is not possible for third parties to extend this component towards a complete C++ language support. Besides C++, SNIFF+ does support several languages, such as C#, Java, FORTRAN, and Python. Several parsers are provided that extract facts from these languages and store them in the C++ language-based symbol table of SNIFF+, by mapping them to analogous constructs in C++. This is relatively easy to achieve since both the parsers and the symbol table support only a relatively small subset, i.e. global scope symbols, of the full languages considered, as already mentioned. All in all, SNIFF+ does not satisfy our (output) completeness and correctness requirements, and is also a closed source project.

2.2.6 DMS

DMS is a commercial program analysis and transformation system developed by Semantic Designs [62]. DMS has been developed over a period of more than eight years and contains several interrelated tools. At the core of DMS is a so-called *hypergraph*, which contains the basic source representation, stored as a forest of abstract syntax trees (AST's), call and flow graphs, and so on [11]. This data representation is accessible by the various DMS tools via a procedural API. Atop this data representation, DMS offers several tools, such as source-to-source transformations for code optimization or refinement, program analysis, and pretty printing. The scope of the DMS toolset goes far beyond parsing and fact extraction, and covers several other languages besides C and C++, as detailed in several papers [2, 11, 57]. In our present analysis, however, we shall focus strictly on the C++ parsing and fact extraction capabilities of DMS, since this is our specific goal. A DMS parser is constructed following the traditional tool combination of a lexer, preprocessor, and the parser itself. The parser uses Generalized LR (GLR) parsing [61] to produce a forest

of alternative parse trees, and provide, in the words of [11], "full context-free parsing". Name lookups and scoping is done after the parsing phase, using a generic symbol table implementation based on interrelated scopes containing (name,type) pairs. The generic symbol table is used via a domain-specific API which implements language-specific name lookup rules, e.g. for C++. Since DMS is a closed source product, further details on the symbol table and language-specific lookup implementations are, however, not available. DMS was specifically built with the goal of scalability in mind. As described in [11], DMS was successfully used to analyze C code of millions of lines and thousands of files. To provide good performance, DMS is implemented to make use of symmetric multiprocessing support on x86 machines. All in all, DMS is a professional product. Unfortunately, it fails our availability requirement, since it is a closed source, commercial product. Also, the language-independence of DMS makes it relatively slower as compared to language-specific C++ parsers which are hand-tuned to excel at a single job.

2.2.7 Gccxml

GCCXML is a fact extractor originally developed by the Kitware company [28]. It builds atop of g++ to extract interface elements from source code, being essentially a modified compiler without code generation [7]. GCCXML outputs the information in a custom format XML file. The original initiators of the GCCXML project were interested in a utility to extract only interface elements of C++ code. Interest in being able to parse the complete C++ language is mentioned by the authors, no advances were made in this direction. GCCXML does not extract all information in local scopes, so it does not meet the requirements (req. 2) and (req. 4). Being based on a compiler parser, it cannot handle incomplete or incorrect code (req. 1). Our previous experience with GCCXML shows it us useful for extracting interface information, but not much more [47]. Overall, GCCXML is more powerful than DOXYGEN, but still far from a full-fledged C++ fact extractor.

2.2.8 CPPX

CPPX [46] is a stand-alone project built atop of g++. CPPX is designed specifically as a fact extractor. It produces an annotated syntax graph (ASG) according to the Datrix [35, 12] fact model. The graph can be exported to different formats, namely GXL [32], TA [34], or VCG [45]. The exported files retain most information about the parsed code structure (req. 7). The Datrix format is very consistent and clearly structured.

However, CPPX's scalability is severely diminished by the huge size of the generated output files (req. 8). In many of our test cases, output files were more than two orders of magnitude larger than the input source code. The output for a mid-sized project of 100 KLOC requires multiple gigabytes of storage. Moreover, the output is not a complete ASG dump, as e.g. generated by the `fdump-tree` option of g++. Some information, e.g. about template classes, is missing (req. 2),(req. 7). Since CPPX is built on g++, it inherits many of the latter's characteristics. g++ has a very mature C++ parser, being both stable and standard compliant (req. 3) and is integrated with a highly configurable preprocessor (req. 5). However, as any compiler-based extractor, CPPX suffers from a compiler's strictness. If a construct contains more than a certain number of syntax errors, the parsing halts (req. 1). Moreover, CPPX accepts only code in the g++ dialect of C++ and is very strict about deviations from the standard (req. 3). In g++, parsing and type checking are interweaved. g++ uses so-called feedback loops for parsing. This means that source code is only parsed when enough context information is extracted from other parts of the code to fully disambiguate the parsed part (req. 6). Hence, it is impossible to simply ignore part of the type checking and export ambiguities in the ASG.

CPPX uses the rather old g++ 2.97 version, even though several improved versions of g++ exist. Porting CPPX to a more recent g++ version would be needed in order to benefit from the latest enhancements of the g++ parser. However, this would require significant effort, as the internal ASG access interfaces of g++ have changed. We were able to compile CPPX under Linux natively and under Windows using Cygwin (req. 9). Overall, CPPX does not comply sufficiently with our requirements, mainly due to its parsing strictness and huge output size.

2.2.9 Columbus

COLUMBUS is a fact extractor developed by the company FrontEndART [18]. Although not open source, we included it in our analysis since binary versions can be obtained freely for testing purposes. COLUMBUS is a complete, professional-looking system. It comes with a C++ Analyzer CAN, a C++ preprocessor CANPP, a linker CANLINK, and the exporter EXPORTCPP. A graphical user interface allows creating projects and configure and use these tools. COLUMBUS accepts a superset of C++, which means that particular compiler dialects and slight deviations from the standard are parsed as valid constructs (req. 1). Supported compiler extensions are g++, Microsoft Visual C++, and Borland C++. COLUMBUS uses a similar, though not identically implemented, parsing technology as DMS, i.e. separates the parsing proper and type checking phases. COLUMBUS also has error recovery, which means that syntax errors do not stop the parse. However, when we tried parsing erroneous code fragments, the parser sometimes took very long to get 'back on track'. For template instantiation, a two pass parsing method is used. A fuzzy parser is used for finding template definitions. This has a performance cost. If desired, however, the template instantiation pass can be disabled. COLUMBUS outputs various formats, such as CPPML and XMI. We verified that most information about the code structure is preserved in CPPML (req. 7). The format is quite verbose, however. For example, every ASG node stores the filename in which it occurred as a string. We also found out that not all locations are correctly reported in the CPPML output (req. 2). COLUMBUS comes with its own preprocessor (req. 5), which allows specifying a forced include file. This allows defining preprocessor macros e.g. to cope with nonstandard language constructs, such as C++ dialect extensions. However, we noticed this preprocessor is less fault-tolerant than e.g. Visual C++'s preprocessor. Actually, the COLUMBUS support team recommended us to use the latter.

COLUMBUS comes with good documentation, making it easy to understand and use. Several publications describe using COLUMBUS as a fact extractor [23, 24, 74, 10, 25]. However, COLUMBUS has several drawbacks. First, it is closed source software, so it is impossible to modify or extend it. Second, its C++ language support has a few deficiencies (req. 3). Template support is rudimentary, i.e. does not cover member templates, expression templates, and a few other template features. These features are used extensively in many popular software libraries, e.g. the GCC 4 Standard Template Library (STL), and the Boost [38] and Loki [3] libraries. Now that compilers come with very good standard compliance, many industrial projects start to use these features, so a fact extractor should support them. Third, COLUMBUS can only parse C code as a subset of C++. C is not a strict subset of C++ and several real-world projects use esoteric C code that will not parse in C++. Native C support is important since many industry C++ projects still include large legacy C library code. Finally, the performance of the COLUMBUS parser is not competitive with the performance of most existing compilers (req. 8). In most of our tests, COLUMBUS was almost an order of magnitude slower than Visual C++ or g++ (see Section 7). This makes it less attractive when working on very large code bases. Although COLUMBUS supports precompiled headers, which can result in a speed advantage, not all code bases are structured in such a way that they can benefit from this.

2.2.10 Elsa

ELSA is a C++ parser built using a set of very powerful tools, including a GLR parser generator written in C++, ELKHOUND [61], with a mini-LR core, and an ASG generator (ASTGEN) that can automatically generate an ASG structure complete with visitors and XML serialization code. The ELSA parser recognizes most C++ constructs and works on a large set of code bases (req. 3). ELSA is released under the BSD license.

Using a GLR grammar makes the extractor much easier to maintain and extend than traditional LR parsers. The entire C++ grammar is only about 4300 lines, including all comments and reduction action code. ELSA can successfully parse industry-size projects of millions of LOC, such as Mozilla [51], Quake 3 Arena [36], VTK [39], and wxWidgets [64] (req. 8).

ELSA has recently been improved greatly with the addition of template type checking support

(req. 3). Good technical documentation on the parser internals is available, making it relatively easy to understand. Elsa has good standard compliance (req. 3). The most important missing feature is support for template template parameters. Template template parameters is an advanced C++ feature that allows types to be parameterized with parameterized types. Production code does not frequently use this language feature, because it is not easy to use and requires a very up-to-date compiler. However, some advanced template libraries, such as Loki [3] and Boost [38], make use of it to some extent. A smaller limitation of Elsa was that member declarations posed an unhandled ambiguity. For example, declaring a constructor with one unnamed parameter is ambiguous as being also the declaration of a member variable. However, we were able to easily extend Elsa's parser grammar to handle the above problems, and also support the Microsoft extensions for assembler statements, `throw(...)`, and `__finally` clauses for exception handling. This allowed our patched Elsa to parse Windows platform headers as well as several STL implementations. The additions we performed to the grammar did not include support for actual type checking of templates that use template template parameters. However, we see this as a possible and doable extension.

All ASG information is serialized to XML by Elsa (req. 7). The authors state that a typed ASG that is serialized and later de-serialized commutes with the original. We verified that, indeed, all extracted information is retained in the output.

Elsa lacks a preprocessor (req. 5), so it does not know about preprocessor directives. Also, Elsa is not entirely robust. Parsing parts of Boost, i.e. the Wave library, resulted in an internal crash. For all other code bases we tried, parsing completed without severe errors (see also Sec. 7). If the GLR parser of Elsa encounters an ambiguity, all possibilities are stored in the ASG (req. 6). These ambiguities are resolved in a later phase. This makes the parser very elegant, as no low-level tricks, e.g. feedback loops, are necessary. Extracted facts are output in XML, which may pose space and speed problems when parsing very large projects. For a GLR parser, Elsa is extremely fast (req. 8). Its performance is competitive with that of compiling the code e.g. with g++ 3.4.4, even though the latter uses a hand-written, carefully optimized, parser.

2.3 Fact visualization

In our work we look at both source code visualization and call graph visualization. There have been previous efforts in both directions. First we will discuss previous works on source code visualization namely SeeSoft, CSV, and CVSScan/CVSGrab. We will present our own source code visualization in Section 5. We also use several programs for call graph visualization. We present them in this section and we will discuss them in detail in Section 5.2, where we apply the visualizations on data sets generated by our query system (cq. Section 4.

Last but not the least, a strong limitation of many software visualization tools is that they are not designed to work in tight integration with other important parts of the forward or reverse software engineering process, such as IDEs, code editors, or fact extractors. This unfortunately limits their effectiveness in practice and hinders their acceptation by the software developing community. Besides designing an effective and efficient fact extractor, one of the goals of our work is to tightly integrate it in a framework for fact data querying and visualization, in order to provide an as complete as possible solution (given our resources) for C++ source code investigation and analysis

2.3.1 Code visualization

SeeSoft

SeeSoft [77] provides a line base source code visualization. SeeSoft's data input is a set of files and a set of attributes. For each file there is a map of lines and for each line a set of values corresponding to the different attributes. SeeSoft displays the files as a series of columns, each column representing one file. The width of the columns are constant, the heights are proportional to the file sizes. Within each column, each file's text line is displayed as a one pixel high line.

The start and end of the drawn line is proportional to that of the corresponding text line. This aids the visualization as the natural text indentation as written by the programmer is a helpful guide to source code structure. At any time only one attribute can be visualized (or perhaps the cartesian product of two attributes). The visualized attributes has a set of possible values. Each value is mapped to a color and these colors are used for drawing the lines.

The SeeSoft visualization is interactive. Using mouse input the colors can be scaled and lines can be selected for detailed visualization. The detail visualization is a basic local view in plain text. In the column, a rectangle is drawn around the lines that are displayed in detail as feedback. When hovering the mouse cursor over the legend of value-color mappings, the visualization will highlight those lines that have the value of choice.

The SeeSoft visualization gives a clear high level view of the source files. SeeSoft is able to visualize between 20000 and 50000 LOC while remaining readable. It is restricted in the way that only line-bound and higher level attributes can be visualized. The detail view does provide a direct link to the original source code but there is no option to view source code in place of lines which would yield an intermediate detail level.

CSV

Code Structure Visualization, or CSV [47], visualizes software at the source code level. The visualization shows the original source file in full text as one layer and a cushioned view of the source code syntax as another layer. Both layers are drawn on the same area, each with a certain transparency. CSV lets the user choose colors for syntax elements. For example, C++ if statements can be highlighted with a striking color while other syntax elements are not rendered. CSV supports gradual zooming up to the point where a line of text becomes a one pixel high line. Applying the syntax view to the line view of the source code enables one to see source code structure with ease.

The drawing style of the syntax cushions can be highly configured. CSV supports the visualization of multiple files simultaneously making it capable for comparing files on syntactical characteristics. The text layer has no visualization possibilities other than drawing text in a certain font. This excludes options for lexical highlighting.

CVSScan

CVSScan [73] is a tool for visualizing the evolution of line-based software structures, semantics and attributes using space-filling displays. It is aimed at developers involved on maintenance projects. The tool shows versions as columns, and uses the horizontal direction for time. Software metrics and the source code are shown in the tool in separate linked displays.

CVSScan accesses a CVS database using the CVSGrab data extractor and obtains different versions the software system. For each version, author, date, and source code are known. CVSScan can visualize this data the line based evolution view. A limitation of CVSScan is that it has no facility for parsing source code. Therefore, it can show which developer changed which line of code, but it cannot visualize in a column which language construct the line contains.

2.3.2 Graph visualization

VCG

Visualization of Compiler Graphs, or VCG [76] is a graph visualization tool. It reads a textual specification of a graph and creates a visualization of the graph. The tool can layout the graph using several heuristics such as reducing the number of edge crossings. The VCG tool can interactively fold dynamically or statically specified regions of a graph. A call graph may consist of thousands of calls spread over hundreds of modules, so it is very convenient that we can collapse modules cluttering our view. The VCG tool is available for X11 and MS Windows 3.11.

The VCG tool is not integrated with fact extractors. The input is a file containing a graph, generated using another tool. The tool itself has no option to interactively specify for which functions a call graph must be produced. This means we need another tool for this purpose.

aiSee

Another graph visualization tool is aiSee [78], which is based on VCG. aiSee reads graphs specified in GDL (graph description language) [] and automatically calculates a customizable layout. The layout is then visualized and can be explored interactively. aiSoft offers zooming and navigation facilities for this purpose. Like VCG, aiSee can interactively fold dynamically or statically specified regions of a graph. This makes the tool suitable for visualizing very large call graphs. The tool is available for many different platforms.

CGDI

CGDI, or Call Graph Drawing Interface [79] is a tool to extract dynamic call graphs from C or C++ programs. It uses GNU gprof [80] to obtain profile information about an executed program. Gprof produces an output file containing profile information, which is then filtered and visualized using VCG. There is no method to step through the program and interactively specify for which sub systems a call graph must be produced.

CGDI produces a dynamic call graph, a graph containing functions that were actually called during program execution. In contrast, static analysis produces a static call graphs containing all functions potentially called during a run of the program. Hence, a dynamic call graph is a subset of the complete static call graph of a program. Dynamic call graphs are often more interesting than static call graphs. Functions that are not actually called are omitted, and consequently, a more concise call graph is produced. Specific bugs that occur during an execution of a program, for example an infinite recursion, can often be spotted easily in a dynamic call graph.

Unfortunately, is not always possible to create a dynamic call graph in this fashion. Gprof can only profile programs for which a working executable is available. To build such an executable, all source code and exactly the right build environment must be available. Furthermore, the source code must be fully compilable by a compiler that can generate profile information in a format supported by gprof. Clearly, this is rarely the case, and this severely limits the usability of the tool.

Rigi

Rigi is a system for understanding large information spaces such as code bases, documentation and world wide web [84]. It extracts facts from a system and organizes these facts into higher level abstractions. System components and their dependencies are represented by nodes and edges in the graph model.

Rigi can visualize the resulting model in a graph view of the user interface. The graph view visualizes graphs by hierarchically clustering them into subcomponents. The user can edit, analyze and interactively navigate through the graph. Elements of the graph can be selected and collapsed into a single. Rigi can also generate several reports with statistical information to facility maintenance and re-engineering tasks. The user interface of Rigi is fully programmable using the Tcl scripting language. The user can write his or her own scripts, but the tool also offers a library of scripts for common reverse engineering tasks. Graphs are stored in the Rigi Standard Format (RSF). This is a textual graph format consisting of a sequence of triples defining nodes, arcs, and attributes of the graph. The graph can also contain references to external information, such as web pages or source code.

ClassViz

ClassViz is a commercial tool for class hierarchy visualization from the company Toolsfactory GmbH [85]. The tool is applicable to a variety of languages, because it has a fuzzy parser for

C/C++, C++.NET, C#, Delphi, Java, IDL and VB.NET source code. The tool also supports several commonly used project files, such as Microsoft Visual Studio projects. The tool can read a project file and automatically parses all files in the project. The parser produces an inheritance graph, which the tool can automatically lay as a tree structure. The resulting graph can be navigated, zoomed, and edited. However, the tool has no option to export the graph to well-known graph formats such as dot. Moreover, The tool lacks a facility for hierarchically clustering classes, for example based on the namespace or module where they are defined. This decreases the effectiveness of the tool on large code bases. Also, because the tool has no preprocessor facility, it often produces incomplete or erroneous class diagrams for correct code.

Chapter 3

Fact extractor design

3.1 Introduction

We now describe a fact extractor we constructed based on the Elsa parser. We name our extractor EFES, standing for Elsa Fact Extractor System, to acknowledge the use of the original Elsa parser.

The weak points of the original Elsa tool are as follows: It has no preprocessing capabilities. No information about preprocessor directives is included in the output. Location information is only partial, i.e. reported for the beginning of syntax elements only, and only as line (no column) numbers. Error recovery lacks. Elsa can produce parse errors caused by incomplete and incorrect code, and internal parser errors (crashes) caused presumably by some design fault. Elsa covers most, but not all, of the C++ standard, and does not support some dialect-specific, e.g. Visual C++, constructs. Elsa has no project concept, so it must be invoked on a per-file basis. The produced output cannot be filtered to reflect only portions of the input, e.g. filter out standard header symbols, or perform a link step to identify common symbols across translation units.

We first introduce some terminology. A *translation unit* is the contents of a file, i.e. the smallest compilable, or parseable, unit. A *topform* is a syntactic element that appears on the highest scoping level in a translation unit, e.g. a global declaration. A translation unit is thus a sequence of topforms.

Figure 3.1 shows an overview of EFES's architecture, which resembles a typical compiler pipeline. Since Elsa uses a GLR parser, we fortunately can avoid the complex interweaving of a parser with a type checker. The input consists of the C++ files to analyze plus any available system headers that will be included by the preprocessor. Elsa lacked a preprocessor, so we incorporated a customized preprocessor that allows selecting different profiles for the most commonly used compilers. This is needed for analyzing code that uses the preprocessor for conditional compilation, e.g. cross-platform libraries. It is thus very important to be able to simulate the behavior of various vendor-specific preprocessors. Our profile-based solution allows this flexibility: Each profile corresponds to a set of preprocessor defines and include directories. We also customized the preprocessor to simply ignore missing headers, instead of stopping the parsing.

Our fact extractor works in four phases. In the first phase, the parser, preprocessor and preprocessor filter operate in parallel. The preprocessor reads and processes the input files and outputs a token stream. The parser reads this stream as it performs reductions and builds up the ASG. In the second phase, the ASG gets disambiguated and type checked. In the third phase, the ASG is filtered to retain the information the user is interested in. In the fourth and last phase, output is generated and written to file. These phases are detailed next. Section 3.4 details how we added a preprocessor to Elsa so that we comply with requirements (req. 2) and (req. 5). Section 3.2.1 details how we extended the parser of Elsa to allow for incomplete and incorrect input (req. 1). Section 3.2.2 details how we modified Elsa's type checker to handle internal parser errors without stopping the type checking efforts. Section 3.5 describes the filtering strategy,

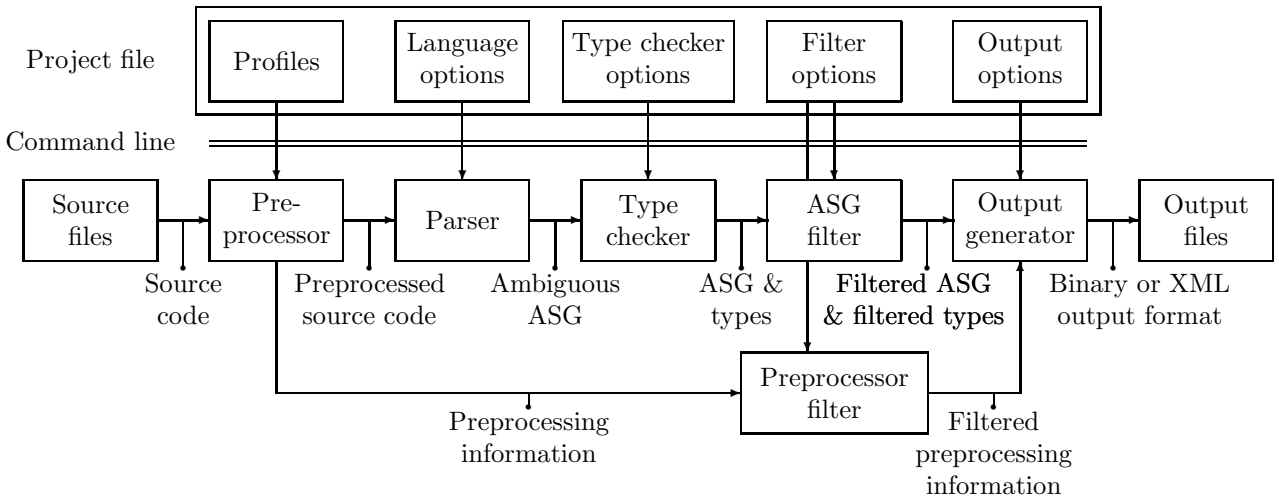


Figure 3.1: Architectural overview of EFES

which lets one specify which extracted elements to be output, using Elsa's original XML output generator or our newly added, performant binary output generator. Finally, Section 3.7 describes a query API we built atop of our extractor to enable users define and execute arbitrarily complex queries on the extracted facts.

3.2 Robustness

We apply two kinds of error recovery to improve the fault-tolerance 1 of EFES. First, there is the syntactic error recovery, which makes sure that an incorrect fragment of C++ code is fit into a grammar rule in such a way that parsing can continue. This essentially discards the incorrect fragment. Next, we discuss recovery of the type checker environment. Invalid or correct but esoteric C++ code may confuse the type checker, and we employ a method for recovering the type checker environment to a consistent state, such that the parser can continue type checking the next high level construct.

3.2.1 Handling incorrect and incomplete code

Elsa requires its input to be syntactically correct and complete, similar to CPPX and GCCXML. EFES does not, in order to address requirement 1. To support parsing incorrect and incomplete code, we modified both Elsa and its GLR parser generator, Elkhound. Our recovery method does not do any efforts of guessing the intended meaning of erroneous code, e.g. by trying to fit the input into the correct grammar rules, as done by other methods [15, 42], since this can easily lead to input misinterpretation. We handle parse errors by special rules in the grammar which start with the special *garbage token*. Lexical errors are handled similarly, i.e. they produce a garbage token which triggers one of the special rules. If no parse error occurs, i.e. input is correct C++, these rules are never used by the GLR algorithm. While parsing, we track in the GLR algorithm so-called *error recovery states*, i.e. states in which the garbage token can be shifted. When a parse error occurs, we set the parser with the most recent error recovery state, at the front of the parse tree. As a result, all unreachable states in the parse tree get cleaned up. Most states that get cleaned up reflect parse positions within the erroneous part. For a few iterations through the parsing after this error recovery, the scanner output will be prefixed by a sequence of tokens. The first token is the garbage token, followed by the right amount of opening or closing braces to match the nesting level in which the parse error occurs, relative to the error recovery state. After the scanner output is prefixed the normal GLR algorithm continues. The scanner output,

with prefix, from occurrence of the parse error onwards, gets matched to an error recovery rule in the grammar, after which the erroneous input was passed and we can continue parsing normally. Essentially, our approach, where error-related grammar rules get activated on demand only, is similar to the hybrid parsing strategy suggested (though not implemented) in [42]. Compared to the ANTLR parser generator [52], our approach lies between ANTLR's simple error recovery, i.e. consuming all tokens until a given one is met, and its more flexible parser exception handling, i.e. consuming tokens until a state-dependent condition is met. However, just as ANTLR's authors underline, the complexity here is not at the grammar level, but efficiently saving and restoring of the parser's quite intricate internal state, including the ASG, in the error recovery process.

The grammatical error recovery rules match any token to the nearest semicolon or match the outermost brace pair. Hence, EFES can recover from errors at the level of topforms. To implement this error recovery, we have added 6 rules to the grammar of Elsa. This is, for practical purposes, an optimal balance between error recovery granularity, and performance and simplicity. We could implement error recovery at finer levels, e.g. expressions and declarations. However, this would require to maintain more information in the error recovery state, the recovery rules would get substantially more complex, and thus the overall parsing slower.

3.2.2 Handling internal parse errors

Recovery from parse errors is important for parsing incomplete or incorrect code. However, there is a second kind of errors that is sporadically caused by correct, albeit esoteric, code fragments. These errors, called *internal errors*, originate from the fact that the original Elsa parser is not perfect. Occurrence of such errors means that requirement 1 is not well satisfied. We discuss here a solution to make fact extraction robust in presence of such errors.

We observed that complex template code can confuse the Elsa type checker. In this case, Elsa generates an error message and no output. Halting on internal errors is undesirable, since a small problematic code fragment can prevent generation of output for other code fragments that could easily be type checked. This is particularly bad if the problematic code fragment resides in a header file that is included by many translation units.

We implement internal error recovery by making backups of the parser state at topform level. We call this a *checkpoint*. If an internal error occurs in a topform, we restore the checkpoint saved before its encounter. The topform in which the error occurred will not be marked as type checked and parsing continues at the next topform. The parser continues type checking as if the problematic topform were not present. We implement recovery at topform level for performance reasons. Backing up the parser state takes significant amounts of time and memory. Doing this at each ASG node would provide finer grained error recovery, but would reduce performance even more. Our tests showed that internal parse errors are rare, mostly occurring in isolated pieces of complicated code. Moreover, such internal errors usually make it impossible to correctly type check the rest of the topform they occur in. We only back up the parser state that could possibly be changed by the type checker. This makes the parser about 20% slower in the average case. In extreme cases, where a translation unit consists of thousands of small topforms, the performance can drop by as much as 50%.

3.3 Project concept

Elsa does not have the notion of a project. The user must run Elsa on every individual source file with the correct command line parameters, e.g. include paths, compiler-specific flags, and preprocessor defines. For code bases of thousands of files and tens of include directories, this manual approach is clearly not scalable (req. 8).

We created a project concept for EFES that allows to efficiently process industry-size projects. Source files are specified in a project by means of so called *batches*. Each batch consists of an input path, a filename filter and an output path. The input path can either point to a single file or an entire directory. If the input path is a directory, the tool searches the directory for all files

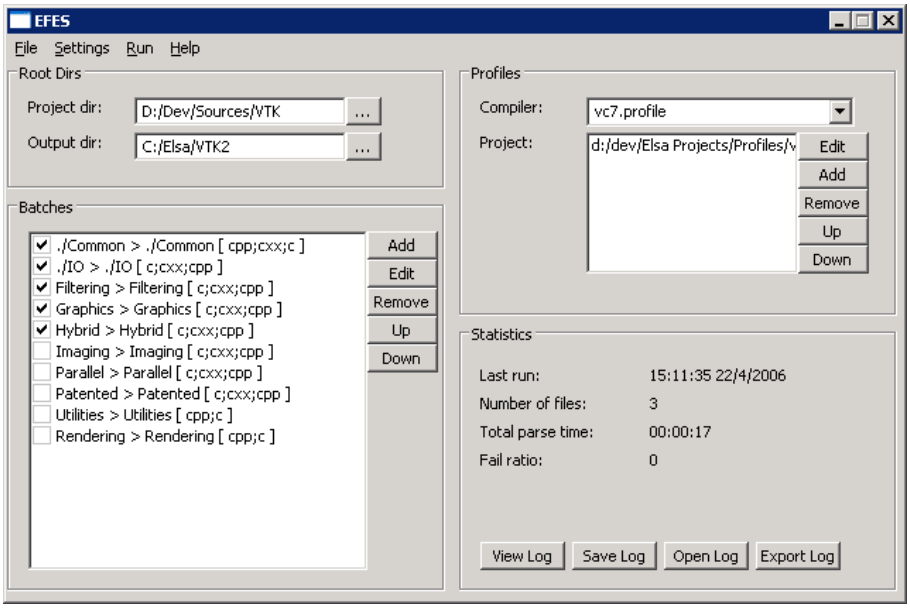


Figure 3.2: User Interface for Project Setup

matching an extension specified in the filter. We allow the user to enable or disable each particular batch, and process them in arbitrary order. Batch ordering has proven to be very useful to specify the order in which results are produced. This can save time, as results can be analyzed before the entire project has been processed. For large projects, which take hours to process, this can be very important (Section. 6.3).

Besides input files, projects also specify a compiler profile and a list of project profiles. Compiler profiles specify which compiler dialect to emulate and which standard library to use, e.g. the various versions of GNU g++, Microsoft Visual C++, or Borland C++. Supporting specific compiler vendors and versions is essential, as many real-world projects can make extensive use of particular features thereof, e.g. template support or vendor-specific language extensions. We support such specific language extensions, e.g. Visual C++'s `throw(...)` directive, `__finally` clause and `__asm {}` statement, by adding extra rules to the grammar. The GNU g++ extensions are already supported by the original Elsa parser. Profiles provide an open way to extend EFES's support of C++ language extensions. To give just a well-known example, we could easily add support for Qt's `signal` and `slot` extensions to C++ [60].

Project profiles specify system and user include directories, preprocessor defines, and forced includes. Often, certain libraries or third-party code are used by multiple projects. With multiple project files, we only need to configure the fact extraction for these libraries once, and can next use the resulting profile in different projects. Finally, we have created a translator that constructs EFES projects automatically from the XML-based Visual Studio project files. In this way, users can analyze complex projects with virtually no setup effort. We designed a simple user interface for project setup. A screenshot of the interface is shown in Figure 3.2. The user interface shows a project file for VTK [39]. The image shows that input and output directories are specified, respectively "D:/Dev/Sources/VTK" and "C:/Elsa/VTK2". Moreover, the "vc7.profile" compiler profile is selected, and the project consists of several batches, some of which are disabled. The interface also shows a summary of the statistics of the previous run. The buttons located at the bottom right are for viewing, loading, and saving, and exporting the statistics. The menu bar of the tool can be used, for example, to change a number of settings, and run EFES on the project once the project setup is complete.

Columbus proposed a different approach for extracting large projects, called *compiler wrapping* [23]. This overloads the standard compiler and linker tools on a platform by the fact extractor

and extractor linker respectively, using the *PATH* variable, and runs the original makefiles to perform the extraction. Compiler wrapping is a very interesting technique that we plan to implement in the near future.

3.4 Preprocessor solution

As explained, Elsa does not come with a built-in preprocessor. We require correct and exact location information of tokens in the original source file 7. This information can only be collected by the preprocessor, as macro expansions change the token locations. As outlined in Sec. 2.2.1, exact location information for the constructs is needed e.g. when displaying the extracted facts in a code visualizer or browser, such as [13, 40, 47, 73, 70]. The preprocessor solution we designed covers both requirement 5 and 7. Since designing this solution and integrating it with the parser exhibits several subtleties, we detail it here further.

The original Elsa parser keeps a line map from which construct locations can be calculated. However, macro expansions in the preprocessor make this line map unreliable. The parser does not know when a location arises from a macro expansion and thus cannot always exactly relate locations to the *original* source file. We solve this in EFES by modifying both the preprocessor and Elsa parser so that each syntactic construct has a location recorded with it. A location consists of a begin and an end *marker*. Attributes to each marker are the *row* and *column* numbers in a certain *file* and if the marker is inside a *macro* expansion. We record the file for begin *and* end markers because this information may differ for a single syntactic construct, due to more complex uses of the #include directive. Indeed, it is possible that part of a syntactic construct has arisen from macro expansion while the other part has not, which is why we need to flag for macro expansion both at begin and at end.

We first considered using the preprocessors coming with compilers such as g++ or Microsoft Visual C++. However, these do not report location changes due to macro expansions. We next evaluated two preprocessor libraries, namely Wave, part of the Boost [38] library, and Libcpp [65], which is part of g++. Libcpp met our requirement (req. 5) closest, and has a clean, easy to adapt design, so we chose it as the preprocessor for EFES.

We connected the preprocessor to the Elsa parser by streaming tokens directly. We modified Libcpp to provide macro expansion and file id information with the locations. These locations are streamed to the Elsa parser alongside of the tokens. The location information generated by the preprocessor flows through the parser without alteration, as depicted in Fig. 3.3. EFES uses two lexical scanners, one in the preprocessor (scanner A) and one in the parser (scanner B). Using a single scanner, as e.g. in ANTLR [52], would result in a slightly faster and cleaner design, but it would mean heavy re-coding of both the preprocessor Libcpp and the Elsa parser. We took here a pragmatic approach, as in Sniff+ [13], i.e. use existing tooling with minimal modification, as long as it matches our requirements. Besides location information, we also stream token length, in order to keep the token stream between the preprocessor and the parser synchronized in the presence of two different scanners.

3.5 Output filtering

Since EFES is to be used on very large code bases, output size can become a problem. For some analysis tasks, e.g. comparing code, mining for implementation patterns, or checking for low-level bugs, we need all facts extracted from the source code. However, for other tasks, e.g. producing a call or class inheritance graph, dumping the entire AST and type information of every translation unit in the output is unnecessary, as only a subset of this information is needed.

The output of EFES consists of the ASG, type information and preprocessing information. We provide a modular way to specify what to include in the output and what not in terms of different *output filters*. The most commonly used filter outputs information originating in a subset of all input files, called the set of *interest*. We output the extracted ASG nodes that originate

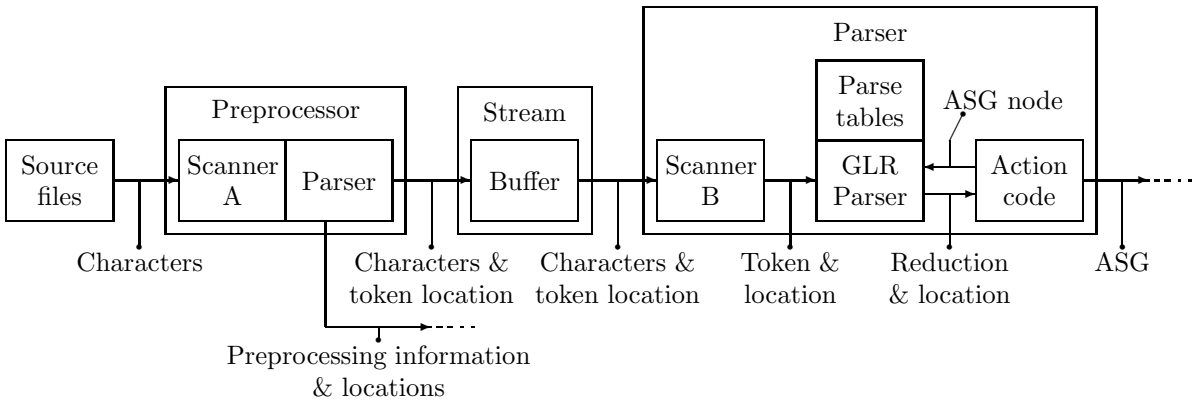


Figure 3.3: Flow of location information in EFES

from the interest set, as follows. ASG nodes usually originate from a single file. However, ASG nodes can originate from more files, e.g. when a syntactic construct starts in a file and ends with a macro call, the macro being defined in another file, or by including files that provide parts of the definition. We output an ASG node if at least one of its originating files is in the set of interest. Since an ASG node needs full context to be useful, we output the topform which contains it. For all output ASG nodes, we only output type information for the referenced (used) types in the set of interest, thereby limiting output size. A second output filtering is to mimic a compiler and output only the ASG nodes that are interesting to a *linker*, i.e. data and function objects only, and of course the types referred in their declarations. The result of this is that declarations, e.g. as present in large system headers, do not bloat the output.

The implementation of the output filter is split in two parts. First, we filter all preprocessing information right away in the preprocessor filter (see Fig. 3.1), using the desired filtering strategy. The preprocessor filter is implemented in LIBCPP's callback functions [65]. Next, we filter the ASG produced by the parser, using the visitor pattern interface [27] provided by ELSA's ASG, and the desired filtering strategy. The binary and XML output modules recognize this flag in the following stage of output production. Finally, the output of filtering is streamed to the actual output generator which creates XML or binary output files. Writing and reading binary, compared to XML, massively saves time and space: The binary output is roughly five to ten times smaller than the XML output, and more than one order of magnitude faster to read and write. Since the binary format is not as easy to work with for potential third-party tools, we keep supporting the original XML output format of ELSA. However, we should note this format is of limited use, as it does not conform to any well-known standard and does not come with a document type definition (DTD).

3.6 Output format

Speed is a highly desired property. This goes for the fact extractor, as well as for programs reading the fact database. We aim at highly interactive querying and visualization of the fact database and it is necessary that these utilities can read the fact database efficiently. We have sacrificed writing speed over reading speed, as we expect the fact database to be read more often than it gets constructed. The fact database for a complete project is a collection of smaller fact databases per extraction unit and an index file listing all those smaller fact databases. For simplicity we will refer to a single file as a fact database. The fact database is a binary file. We have witnessed binary files to be ten times smaller in size compared to xml files containing the same data and binary files need less processor time to be interpreted (we estimate them to be an order of magnitude less processor intensive to read). Another great advantage that the binary format provides is the ability to skip over large quantities of data, useful when only a subset needs to be read.

The fact database houses several kinds of data: The generic statistics, the file table, the token locations, the preprocessor information, the type information and the abstract syntax tree. The generic statistics include statistics on parse time, parse errors, type errors and input size. The file table is a list of the files that make up the extraction unit. The list assigns an index to each file so that other data elements in the fact database can refer to the file by index. The token locations is a list of token locations which, like the file table, implies an indexing on the token locations. The ASG refers to token locations using this index.

The preprocessor information is a list of lists. The main list contains 8 lists which store the relevant information on macro calls, definitions, conditionals et cetera. The generator and parser for this data are hand written but highly maintainable. Our file format is not set up in a way that allows for skipping over the smaller lists individually, because we think this is not a desired feature.

The type information is a tree structure typically very wide and shallow in shape. Stored in this part of the database is type information such as the classes, enumerations, typedefs, function types and the likes. The type information generator and parser are automatically generated by MultiGen. We have achieved this by setting up definitions for each kind of type, orthogonal to the way Elsa has the description of the ASG, and we have extended MultiGen to produce a generator and parser for this kind of data. There is no option to search for, and load a single specific type in the fact database. This choice is based on several factors: One argument that always stands is that it would complicate the parser if we were to support this. Another argument is that types are meaningless without having an ASG. The complete ASG refers to every type in the database directly or indirectly. A single type can be referred to from the ASG in several places and the ASG can, in close context of these places, refer to other types which are likely to be relevant to the end user. Types do not form an easy hierarchy where a type's relevant data is a subtree of the type.

The abstract syntax tree is the last data element to be discussed. The ASG is a tree (or actually it is a DAG but this can be largely ignored) with nodes similar to those present in the fact extractor. MultiGen creates an ASG structure in the shape of a C++ class hierarchy for use by the fact extractor. Certain data elements in that ASG structure are not relevant after fact extraction and as such we do not save them in the fact database. This causes the fact extractor and the fact database parser (and utilities) to use different ASG structures. This is not much of a concern. Like with the type information we have meta-programmed a generator and parser for the fact database by extending MultiGen. Since we can precisely steer the generator and parser to match on a high level, we manage to avoid difficulties introduced by having two different ASG structures.

The ASG parser facilitates online searching. Online searching is faster and more memory efficient, and hence it is preferred when it provides sufficient data. Each ASG node in the file has a header describing the kind of node and the size of the subtree rooted at the node. While reading the ASG one can jump ahead to any following sibling of a previously encountered node. It is also possible to read certain attributes of a node without reading its subtree. We have designed the file format to support online search on the ASG because of the strong structure the ASG has.

The ASG is not necessarily a tree. In certain degenerate cases it is a DAG. The GLR parser can parse ambiguous source code. Some action rules of the parser decide that a subtree of two parses of the same ambiguous input gets shared between two ASG nodes (each representing one of the parses). When the type checker can not resolve such an ambiguity, the ambiguity remains present in the ASG and is saved as such to the fact database. While the fact database is being created, the ASG is visited to write every node in pre order in the output file. While visiting the ASG, when a subtree is encountered that has already been traversed once before, a stub is written instead, pointing at the subtree that already exists in the fact database.

The fact database is divided in separate sections containing different types of information but the information in the fact database is only complete when all the links between the data elements are resolved by the reader. Figure 3.4 depicts the dependencies between the different sections. To be able to record links between data elements, those data elements need to have identifiers. Because we optimize reading speed, the identifiers run from 1 to the number of data elements

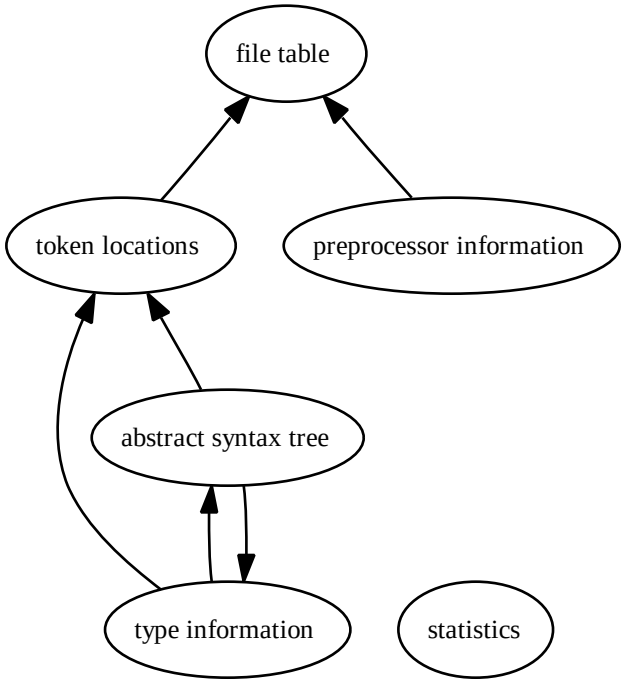


Figure 3.4: Fact database dependencies

present in the file. The reader can then construct a mapping of data elements in a vector that can be directly indexed. By giving each data element a unique identifier we also facilitate selections, which will be introduced in Section 4.3.1. The reader can optionally skip reading of certain data at the cost that not all links can be restored.

3.7 Query interface

The features of EFES described so far allow one to easily create a complete low-level fact database from source code as a set of XML or binary files. However, for this database to be useful, one must have a mechanism to *query* the stored information. Most professional extractors, in particular the heavyweight ones, e.g. COLUMBUS, SNIFF+, DMS and ASF+SDF provide a so-called *query API* to examine the extracted data. Lightweight extractors, e.g. DOXYGEN andGCCXML, provide no API but just save the data in some file format, which the user should be able to parse relatively easily, given the output is not too large and/or complex. Since EFES is a heavyweight extractor, we designed a query API atop of the basic extractor. The query API has two functions. First, it lets users define arbitrarily complex queries that involve the extracted elements. Second, it lets users apply these queries on a given database. We next sketch these functions in brief.

Queries can be defined by constructing a so-called *query tree*. The tree nodes correspond to the C++ grammar elements, e.g. classes, identifiers, functions, declarations, and so on. Each node has several attributes that describe its grammar element, as well as children nodes corresponding to its contained grammar elements. For example, a class node has a 'name' attribute and several 'baseclass' and 'members' children. Query node attributes can be set to specific values to indicate what one searches for, using regular expressions. For example, to look for all classes whose name begins with "Foo" and have a base called "Bar", one should set the class node's name attribute to "Foo*" and the name attribute of the 'parent' child node to "Bar". The query nodes are C++ classes which we generate automatically from the ELSA C++ grammar using a design analogous to the ASTGEN tool [61]. The query API consists of all these classes plus a single `query` function

that applies a given query tree to a given set of 'input' ASG nodes, yielding an 'output' subset of the input nodes which match the query. A carefully optimized implementation of this function allows us to execute complex queries on databases containing millions of ASG nodes in less than one second.

Predefined query trees can be saved in a query library. Users can pick any query from such a library and apply it on any desired input. Queries can be cascaded using set operations (AND, OR, etc). This allows one to easily construct arbitrarily complex queries, ranging from e.g. "select all functions whose name starts with Foo" to e.g. "select all classes which inherit virtually from a class Bar, have at least one virtual method with a template parameter, and have a member of type T or derived from T" or "select all functions which nobody calls (dead code) or which recursively call themselves, directly or not". A rich set of query examples is given further at [21].

Chapter 4

Fact enrichment and querying

4.1 Introduction

Source code contains a wealth of information. The user often wants a specific look at this information. The user may want to see a class diagram showing inheritance relations, a function call graph starting from the main function, an overview of the types defined in the system, or the original source code of some header file. If we want to generate such views, we need a way to filter the information in the database and select only the facts of interest.

We can select facts of interest by searching for patterns in source code. Tools such as grep [81] and AWK [82] can be used to search patterns in source code. Unfortunately, their power for querying source code is very limited. There are many complex relations between elements in source code, which cannot be directly obtained from the source code text. A variable assignment, for example, refers to a variable which may be defined at many different places in the source code. Our fact extractor can extract this information from source code. EFES produces extremely large databases storing entire abstract semantic graphs along with much other information about source code. The ASG stores all elements occurring in the source including their relations. If we can query the ASG, then we can query almost everything about the source code.

We need a query mechanism with a facility where the user can specify exactly what part of the ASG to select. A query system that can only query specific information in the ASG severely limits the possibilities for constructing queries. Suppose, for example, that the user wants to create a visualization of ambiguous code fragments. In this case the user needs a query to select all ambiguous constructs. However, if the query system has no ability to query the ambiguity link of an ASG node, then this query cannot be written. The only options for the user are to extend the query system such that it can query the ambiguity link of an ASG node, or bypass the query system entirely and write his own query functions. In any case it takes a significant amount of time and skill to add functionality to the query system. Clearly, a generic query system is needed that can query all attributes of all ASG nodes in the database.

Furthermore, the query system should offer the ability to build arbitrarily complex queries. For example, we want to be able to write queries such as 'select all recursive functions with more than 3 parameters of type *float*', and even more complex queries such as 'select all functions which recursively call themselves, directly or not'. We prefer to write complex queries by combining and configuring existing queries.

It should be relatively easy to create queries for the query system. A developer with reasonable knowledge of C++ should be able to create a complex new query in a matter of minutes. We do not expect inexperienced programmers able to write complex queries. However, they should at least be able to change the arguments of an existing queries.

Finally, we want to specify queries while Visual EFES is running. This way the query developer does not have to reload the database, which may take a long time, every time he or she changes a query.

4.2 Method

In order to achieve our query desiderata, we have implemented a generic and flexible query system atop of the EFES fact extractor. We call the combination of all ASG nodes (including syntax, type, and preprocessor nodes), data nodes, files, extraction units, and fact databases *selectable nodes*, or simply *selectables*. Figure 4.3 shows the class hierarchy of selectable nodes. We designed a query system that can query all selectable nodes and all their attributes.

A query can be formulated as a function Q accepting a selection S and n query specific arguments $a_1, a_2, \ldots, a_n$ as input parameters and producing *result selection* S' as output. A selection is simply a set of nodes from the ASG produced by the fact extractor. Selections are detailed further in Section 4.3.1.

$$Q(S, a_1, a_2, \ldots, a_n) \rightarrow S'$$

The query system works by searching for patterns in the ASG. These patterns can be specified using a *query tree*, a tree structure built from query nodes. Figure 4.1 shows the class diagram for the fundamental query tree classes. Analogous to how a regular expression engine matches a regular expression in a sequence of text, our query system search patterns in the ASG matching the specification of the query tree.

For every selectable node type σ, the query system defines a corresponding query node type σ'. Selectable nodes of type σ potentially contain references to other selectable nodes. For example, an if-statement node references an expression node, a then statement node, and an else-statement node. For each reference to an object of type τ in a node of type σ the user can provide zero or more query objects of type τ' in query σ'.

The query system works by traversing the query tree in depth first order. During the traversal the query system tries to match, in a bottom-up fashion, selectable nodes to a query node. The matching of a complete subtree of the query tree with ASG nodes, may result in an arbitrary number of *selections*. A *selection* boils down to picking an ASG node and inserting it into the result selection of the query.

A query can be executed using the function `select`. The function takes as input a selection S, and produces a result selection S' as output. Selection S stores the nodes on which the query is executed. Selection S' contains the nodes that were selected by the query.

4.3 Query system design

The GLR grammar that EFES uses defines over 300 ASG node types. It is an unfeasible job to construct a query node type for each ASG node type by hand. Even if this task could be completed, the maintainability of EFES would be severely diminished, because any change in the ASG requires a change in the query system. Extensibility is one of the hallmarks of EFES so we clearly want to avoid that.

Fortunately, the types of nodes used in the ASG of Elsa, which serves as a basis for EFES, are entirely generated using the ASTGen tool. ASTGen is part of the Elsa package and is documented in detail in [56]. Due to lack of space, we shall limit ourselves to a very brief description of the way ASTGen operates. ASTGen takes as input a file that describes the structure of the ASG and produces a C++ ASG and several supporting classes, such as visitors and factories [27], as output. The ASTGen tool works by building a class diagram of ASG nodes. For each ASG class, ASTGen maintains a list of all fields with their corresponding types. ASTGen traverses these lists to generate C++ code for the ASG.

Our idea is to use ASTGen to automatically create the query node types as well as the ASG node types. To this end, we have modified ASTGen in depth and created a new tool, called MultiGen. MultiGen extends the functionality of ASTGen by adding a query system generator. The query system generator generates all node types of the query tree as well as several supporting code such as visitors, factories, and name maps. Figure 4.2 shows the input files accepted by MultiGen, and the output files it produces.

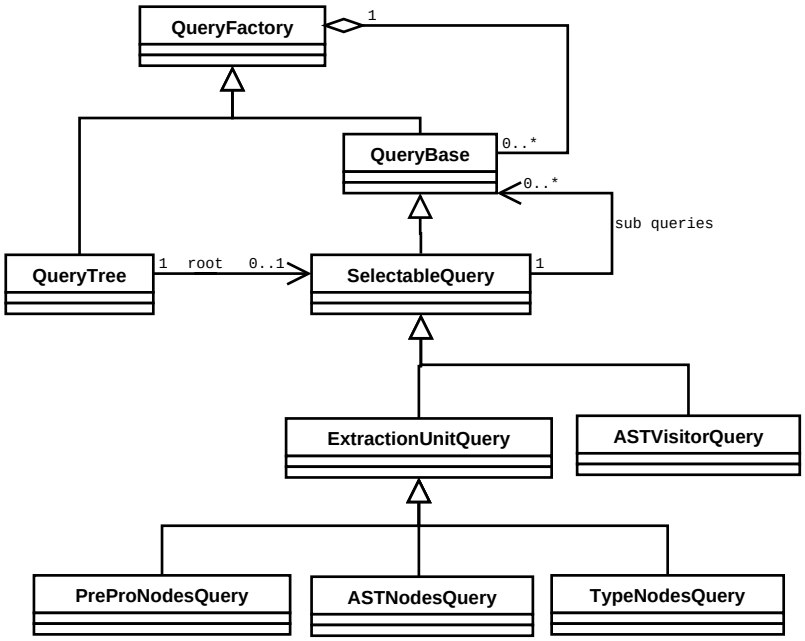


Figure 4.1: Query system class hierarchy

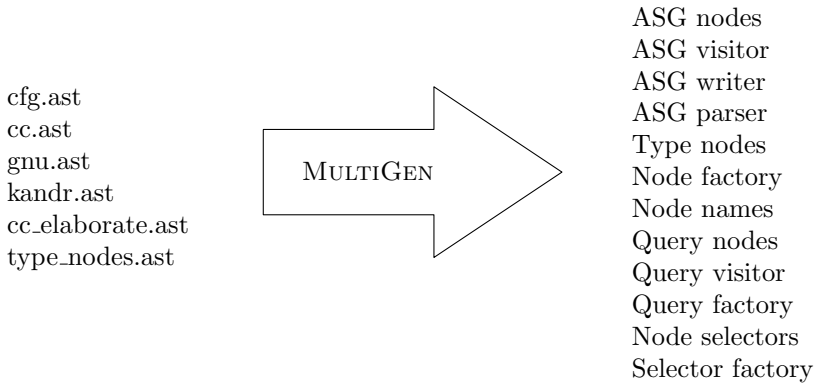


Figure 4.2: Generating source code with MultiGen

Each class description contains a list of attributes. For each attribute the name and type can be requested in MultiGen. MultiGen generates query classes by iterating through the list of ASG classes and generating a query class for each class it finds in the list. Attribute information is used for the generation of the class declaration and function bodies. MultiGen decides, based on attribute information, what kind of sub query to emit. The query generator uses the algorithm described in listing 4.1.

The ASTGen specification in Elsa included only part of the ASG. About 30 nodes, mostly from the type system, were written directly in C++. We decided to add an ASTGen specification for the type system to EFES.

We added a type generator to MultiGen that works in the same fashion as the AST generator. Based on option specified in the input files, MultiGen determines which generator to invoke. We added an option to MultiGen where the name of the query system to generate can be specified.

We were able to write a MultiGen description for the entire type system, although this turned out to be a nontrivial task. MultiGen is designed for class hierarchies purely based on single inheritance, whereas the type system uses multiple inheritance regularly. We avoided this

```
procedure emitQueryClasses
begin
  open file out
  for each top level class x
    emitQueryClass(out, x)
    for each subclass y
      emitQueryClass(out, y)
  close file out
end

procedure emitQueryClass(File out)
begin
  write class head to out
  for each attribute x
    emitQueryAttribute(out, x)
  write class tail to out
end

function emitQueryAttribute(File out, Attribute x)
begin
  if isTreeNode(x.type)
    emitTreeQuery(out);
  else if isTreeNodePtr(x.type)
    writeTreeQuery(out);
  else if isTypeNode(x.type)
    writeTreeQuery(out);
  else if isListType(x.type)
    writeListQuery(out);
  else if isSimpleType(x.type)
     writeSimpleQuery(out);
  else isFlagEnum(x.type)
     writeFlagQuery(out);
end
```

Listing 4.1: Generating query classes using MultiGen

problem by augmenting the generated code with some handwritten code that is copied verbatim by MultiGen. MultiGen also has trouble parsing certain function bodies, so we had to move these to a translation unit with custom implementations.

In contrast to queries concerning the C++ syntactic information (which is described as ASG nodes), queries concerning C++ preprocessing elements are still written by hand. There are about 10 different preprocessor nodes, and we believe that constructing a MultiGen description for the preprocessor nodes is too large a task compared to the benefits it provides on the short term. Preprocessor nodes have a simple structure, and its corresponding query node is easy to write.

4.3.1 Selections

An EFES fact database contains a variety of objects which a user should be able to select. This includes the fact database itself, extraction units, files, ASG nodes, type nodes, data nodes, and preprocessor nodes. Figure 4.3 shows the class hierarchy for selectable nodes.

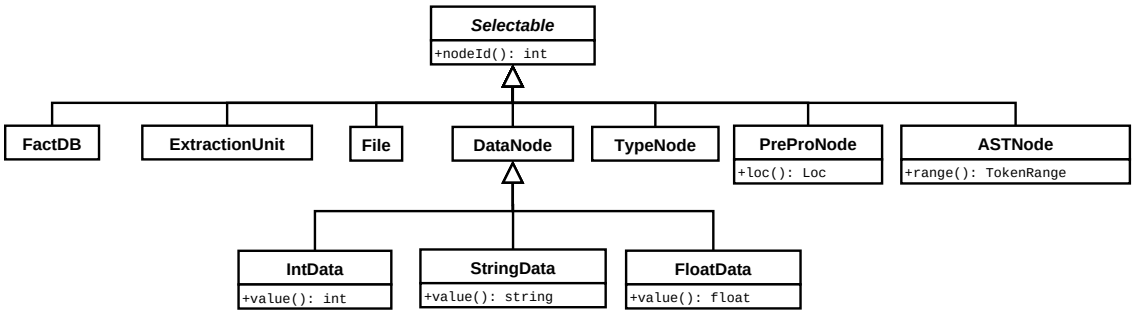


Figure 4.3: Selectable node class hierarchy

All selectable nodes have a *node identifier* that is unique across a single extraction unit. The node identifier of a selectable object can be queried in constant-time. By combining the node identifier with an *extraction unit identifier*, or simply *unit identifier*, it forms a *global identifier*. Global identifiers uniquely identify a node in the fact database for an entire project. We call a set of global identifiers a *selection*.

In selections global identifiers are used instead of pointers, because extraction units may constantly be unloaded and reloaded during the lifetime of a selection object. ASG nodes are placed at different memory locations every time an extraction unit is loaded, which would clearly invalidate a selection. This is unacceptable because often databases are so large that all their extraction units do not even fit into memory. We still want to perform selections on those databases.

Queries accept a selection as input and produce a result selection as output.

The query system uses a *selection object* for storing input and output selections. The presence of a node in a selection object can be queried using the `contains` function, which accepts a global identifier as argument. It is also possible to iterate through all the nodes stored in the object. The `begin` and `end` functions of a selection object return iterators to its sequence of global identifiers.

A selection can be written to a file. This way the result of a query can be stored on disk. The selection can later be recovered by loading the file.

4.3.2 Data types

The ASG of EFES is a collection of typed objects. These objects can be classified into three distinct groups.

Primitive types Primitive types are the elementary data types in the ASG. Examples of primitive types are boolean, integer, string, and enumeration types. Attributes of simple types are not represented as nodes in the ASG, and do not implement the selectable interface.

Instead, attributes of primitive types are always attributes of other nodes. These attributes are selected by wrapping them into selectable objects. This is detailed in Section 4.3.3.

Node types Node types are AST node, type node, and preprocessor node types that make up the ASG. ASG nodes represent syntactic elements of the source language. Examples are *class specifier*, *variable* and *for-statement*. Node types are aggregate types consisting of simple types, container types, and references to other ASG nodes. ASG nodes are hierarchically structured in a class hierarchy. For example an *if-statement* is a special kind of *statement*, which is in turn a special kind of *AST node*. The query system utilize the class hierarchy. For example, an *if-statement* can be queried by a *statement* query.

Container types Containers, for example lists or sets, hold collections of other objects. EFES uses type safe containers holding a specific type of ASG nodes. A *translation unit*, for example, has a list of *topforms*. A *class specifier* has a list of *base specifiers*.

4.3.3 Data nodes

Many attributes of ASG nodes have a primitive type, such as integer or string. Primitive types do not implement the selectable interface, and hence these attributes lack a global identifier. Nevertheless, queries often need to select these attributes. For instance, if we want to cluster a set of functions based on their name, or a set of integer constants based on their value.

We considered changing the ASG and turn primitive attributes into selectable objects. However, we dismissed the option, because we estimated that it would at least double the memory requirements of the ASG. Instead, we utilize the fact that often relatively few data attributes are actually selected. By wrapping an attribute of a primitive type into a *data node* only if it is actually selected. When the selection is destroyed, all its data nodes are released. Data nodes are allocated using factory methods of the *data node manager*. The manager creates a new data node, generates a global identifier for it, and manages the constructed object. When a selection object is destroyed, it informs the data node manager, and the manager destroys all data nodes associated to the selection.

Global identifiers of data nodes can not be written to disk directly, because these identifier refer to temporal objects. Instead, all content of a data nodes must be serialized. During deserialization, the data node can be reconstructed, and receives a new global identifier.

4.3.4 Query nodes

Query nodes are the atomic building blocks of a query tree. Query nodes are always part of exactly one query tree. We implemented the query system such that nodes cannot be shared between multiple query trees, because they have context dependent state.

Each node ν in the query tree defines a *selection predicate* P_ν. The predicate takes an ASG node as argument and returns a boolean value.

The query system evaluates this predicate on queried ASG node n. If $P_\nu(n)$ returns true, then all selector functions $S_0, S_1, \ldots, S_n$ attached to ν are called with n as argument. Each selector function returns a selected node, which the query system inserts into the result selection.

$$P_\nu(n) \to \mathbb{B}$$

Query predicates are evaluated in depth-first order. A query node accumulates its result and the results of all its sub queries using an *accumulator*. The query predicate returns this value.

Once a query tree is constructed, it often necessary to traverse the nodes of the tree. For example to alter the parameters of query nodes. The query system uses the visitor design pattern for this purpose. It offers a *query visitor* class that can serve as the base class for custom visitors. The query visitor is automatically generated using MultiGen.

4.3.5 Accumulators

We noted in the previous Section that each node ν in the query tree defines a predicate P_ν. Here we will explain how such a predicate can be constructed.

Most query predicates are based on predicates of sub queries. For example the selection predicate in query 'select all functions with name "parse" and return type "int" or "float" ', can be based on the predicates 'is the function named parse?', 'is the return type "int"?'. and 'is the return type "float"?'. The results of the sub queries can be combined using the boolean operators "and" and "or".

Queries such as 'select all functions with at least 3 parameters of type "int" ', cannot be constructed with a simple boolean operation. Here we have to count the number of times the predicate 'is this parameter of type "int"?' evaluates to true. The result of the query predicate is true iff the number of times a sub predicate evaluates to true is greater than 3.

In our query system, query predicates can be based on sub predicates. The operation to use for combining the results of several sub queries can be specified using *accumulators* objects. Accumulators are objects responsible for accumulating the results of queries and producing a single result value.

Most query nodes only have a single accumulator. Operations such as "x and (y or z)" can be expressed by nesting sub queries. Each query node can have multiple sub queries that are applied on the same node. In this example there is a query with an "and" accumulator having a sub query with an "or" accumulator".

Accumulator types are specialized for the operation they perform. There are several predefined accumulator types in EFES. First, there are accumulators specialized for the binary operators such as "and", "or", and "xor". The default accumulator for a query is the binary "and" accumulator. This is essentially a "forall" operation. The accumulator returns true if and only if all all accumulated values are true. Adding a new sub query to a query with an "and" accumulator, results in a stricter query, which generally produces smaller selections. The "or" accumulator essentially implements the "exist" operation. The accumulator returns true if and only if at least one of its accumulated values is true.

Secondly, there are accumulators based on a counter. These accumulators count the number of sub predicates evaluating to true, and compare this value with a user specified reference value. The comparison operation can be specified by the user. Available comparison operations are "less than", "at most", "equal", "different", "at least", and "greater than". For example the query for 'Select all functions with at least 6 parameters', can be implemented using an "at least" accumulator with reference value 6 for the parameter list query.

4.3.6 Selectors

We explained how query nodes can be combined to form query trees, and how the query system uses these trees for pattern matching in the ASG. The user is hardly ever interested in the node matching with the root node of a query tree, because this results in at most one match per node in the input selection. Queries such as 'select all function parameters', executed on a selection with an extraction unit, cannot be constructed as such.

Often part of a query merely serves as a path to some ASG node ν for which the user is interested in predicate $P(\nu)$. The results of query predicates executed on nodes the path to ν are in that case irrelevant and ignored.

A selector is an object that can be attached to a query node. It defines a *selection function* S taking a node as argument and producing a node N as result:

$$S_{\nu'}(\nu) \rightarrow \mathrm{N}$$

A query node ν' contains a list of selectors. If P_ν evaluates to true for some node n, then all selectors in L are invoked on n. The selected nodes are inserted into the result selection.

Like query objects, selectors mirror the structure of ASG nodes. For every ASG node σ there is a selector σ''. We have extended MultiGen to generate selectors automatically, similarly to ASG node types, query node types, parsers, and a number of supporting classes.

Selectors can be combined in such a way that they form a *selection path* to an ASG node. Every selector has a number of *ports* to which another selector can be attached. For every ASG node attribute in σ, there is a corresponding selector port in σ''.

The selection algorithm works as follows:

```
function select(Selectable *node) : Selectable*
begin
 if !node->isX()
  return null;
 for each attribute y
 begin
  if selector(y) != null
   return selector(x)->select(x->attribute[y]);
 end
 return node;
end
```

The algorithm first tests if the node type matches with the selector type. If this not the case, null is returned to indicate that the node is not selected. It has the wrong type. Otherwise the algorithm iterates through all selector ports. If a selector is present for some port corresponding to an attribute y in the ASG node, then its selection function is called with y as argument. The returned node is then immediately returned by the algorithm. If no port contains a selector, then the selector is located at the end of a selection path, and the ASG node itself is returned.

A concrete example of using a selection path is the implementation for Query 6 given in Section 4.6.1.

4.3.7 Link map integration

The linker, which is discussed in Section 4.4, produces a link map data structure. The link map links similar occurrences of the same symbol in multiple translation units to a single definition. This essentially generates cross-links between different translation units. Of course, the query system can handle link maps. The link map is a requisite for queries spanning multiple translation units. Examples of such queries are 'select all calls to functions with more than 10 lines of code', or 'select all assignments to global variables defined in file console.cpp'.

Queries nodes for the type system can optionally perform a link map lookup for a type node. Its query predicate is then evaluated on the result of the lookup, instead of the type node that was provided as input to the query.

There is no global option for using the link map, instead it must be specified per query node whether or not it should perform a link map lookup. Link map lookups are relatively expensive and not always necessary. Therefore, they should be used with care.

4.3.8 Special query nodes

Selectable nodes often have more attributes than just references to other selectable objects, such as string values and bit fields. Querying these attributes is particularly interesting, because they store store information such as the name or the value of a particular ASG node. Our query system can query these attributes using *special query nodes*.

We differentiate between *selectable queries* and *leaf queries*. Leaf queries are queries without having a sub query, and are executed on node attributes. Some leaf queries can have a selector. In this case the attributes are selected by wrapping them into a data node which is added to the selection. Often leaf queries function solely as sub queries of a selectable query. We differentiate between several kinds of attribute. For each kind of attribute the query system has a unique leaf query.

Simple query

The most common attributes are those with a simple type, such as integer and enumeration values. These attributes can be queried using a *simple query*. Simple queries allow for the specification of a reference value and a compare functor. The reference value is a value of the same type as the attribute. The compare functor compares the reference value to the attribute and returns a boolean value indicating the result of the comparison. The query system defines several common compare functors, such as "greater than", "equals", and "at most". The query node forwards the result of the compare functor.

Flag query

Some enumeration types are defined in such a way that their constant values can be used as flags. The positions of the set bits in the bitwise representation of the enumeration constants of such types are disjoint. Hence, an individual enumeration flag can be either turned on or off using bitwise operations. An example of such an enumeration type is the "DeclSpec" type, which has values for flags such as "virtual", "member", "register", and "inline". Function declarations have a bit field "flags" that can, for example, be flagged both as "member" and "virtual" using the constants of the enumeration type. The query system offers *flag queries*, for querying such attributes conveniently. Such query nodes can test whether individual flags are either set or cleared. This query exists purely for convenience, since it is essentially a simple query using the "bitwise and" compare function.

Location query

All ASG and preprocessor nodes have a begin and an end location. The begin location indicates the position where the first character of the construct is placed, the end location indicates where the last character of the construct is placed. Locations consist of column index, line number, and file identifier and are stored in *location objects*. Location information is required for queries such as 'Select all function bodies occupying more than 10 lines' and 'select all function declarations in header windows.h'. Locations can be queried using *location queries*. In a location query you can specify ranges where a row, column, or file identifier should lie.

Type query

Each selectable type σ has a corresponding query type. A query object of this type can be used for selecting objects of type σ. The drawback of this method is that type σ is hardwired into the query tree. Queries such as 'select all objects of type "x" ', where "x" is provided by the query user, cannot be constructed in this way, because changing type σ involves changing the query tree. We would like to specify the type as parameter, such that the query user can specify a type for a given query. For this purpose the query system offers a *type query*. A type query uses runtime time information to query the type identifier of a node. This type identifier is then compared to the reference value specified by the query user.

Scope query

Often the user wants to test the scope in which an ASG node is defined. The query 'Select all function calls to functions declared in namespace std' can be constructed easily if such a test is available. It is erroneous to simply query the parent node of the called function, because new scopes can be opened inside the std namespace. If a selectable object x is defined within scope y then scope y is located somewhere along the root path of x. For testing the scope where an ASG node is located, our query system contains a *context query*. It can evaluate a *scope query* of a node x on all ASG nodes on the root path of x.

List query

Some selectable nodes contain lists of references to nodes. For example, a translation unit has a list of topforms, and a function declaration has a list of parameters. These lists can be queried using *list queries*. A list query can have a single *item query*. The element query is executed on the nodes in the list. The user is not always interested in querying all elements in a list. For example, the query 'Select all enumerations where the second enumeration constant has identifier RED' cannot be constructed in this way. Therefore the list query allows for the specification of a range. The range specifies a half-open interval $[x..y)$ of list indices where the item query is evaluated.

Visitor query

Sometimes it is impractical or simply impossible to specify an exact pattern of selectable nodes. For example it would be hard to create the query 'Select all functions having at least 3 goto statements' this way. This is where the *visitor query* comes in. It traverses the entire subtree of an ASG node. On each node it executes one or more *visit queries*. This produces the same results as first performing a "flatten" operation on the tree, thereby producing a temporary selection T of objects, and then executing the visit queries on T. Each visit query has its own accumulator. After all nodes are visited, the results of all queries are accumulated using the visit accumulator. By allowing multiple accumulators and visit queries, we avoid repeated traversals, which are time consuming. For example the query 'select all functions having at least 2 return statements and at least 1 goto statement in the function body' can benefit from this.

Closure query

Some queries, such as *closure queries*, cannot be expressed with a single query tree. We call such queries *compound queries*. Compound queries use the results of another query as the parameters for another query. Since queries produce selections as output, it is convenient if we have a query node that can test if some selectable occurs in a given selection. The query system offers a *selection query* for precisely this purpose. An selection can be linked to a selection query

4.3.9 Query storage

A query tree can be constructed procedurally by using the (C++) API of our query system described so far, but constructing queries this way has several drawbacks. Users often want to run an existing query, observe the results, and then refine the query and run it again. This cycle is typically repeated many times over, so it is important that this is efficient. Ideally, this cycle should be close to realtime. This way the user can try many different approaches, and the system immediately presents query results.

This is only attainable if the user can write a query while Visual EFES is running and the entire database stays in memory. But since Visual EFES is a monolithic program, writing a query in C++ requires a recompilation of the entire system.

Users can specify queries in XML while the Visual EFES is running. This is implemented using object serialization, complete query trees can be either written to or read from XML. We chose XML because it is a simple declarative language that allows for the specification of tree structures in a compact manner. It is also standardized and many good tools are available for processing XML.

The following XML file contains a query to search for all C++ style cast expressions:

```
<QueryTree>
  <Root Type="NodeQuery">
    <NodeQueries>
      <NodeQuery Type="ASTQueryVisitor">
        <VisitQueries>
          <VisitQuery>
            <Query Type="E_keywordCast">
```

```
            <TrueSelectors><Selector Type="NodeSelector"/></TrueSelectors>
          </Query>
        </VisitQuery>
      </VisitQueries>
    </NodeQuery>
  </NodeQueries>
 </Root>
</QueryTree>
```

Some programming languages, such as C# and Java support runtime reflection. Runtime reflection makes it easy to write generic routines for object serialization. Unfortunately this functionality is not available in ISO C++. Luckily most of our query system is automatically generated, so we could avoid writing a serialization routines by hand. We added functionality for generating serialization routines to MultiGen. This required only a few hundred lines of code, since the generator can iterate through all query fields.

We used our own flexible serialization framework. Support for different file formats can be added to the framework without having to change serialization routines. Also new attributes can often be added without breaking backward compatibility.

The biggest disadvantage of writing queries in XML is that it requires some competence in programming and extensive knowledge of the internal ASG structure used by EFES to write a complete query. We believe, however, that most users are not interested in writing new queries entirely from scratch. Most users are probably satisfied if they can simply change a few parameters of a query and run it again. Also, changing queries, or combining basic queries into more advanced queries is relatively simple and can be done by novice users.

It would be nice to have an advanced user interface that aids in writing queries. It is very useful to be able to create queries based on several template queries.

4.3.10 Aggregate queries

The ASG structure utilized by EFES has an intricate structure. Even basic language elements, such as function declarations, are represented by a large cluster of nodes in the ASG. The query tree exactly mirrors the complex structure of the ASG. Therefore, often long paths of query nodes are needed just to query a single attribute. Writing queries by meticulously reproducing the structure of the ASG not only requires profound knowledge of the ASG structure, but is also a rather tedious and unpleasant task.

We can ameliorate the situation by adding a number of shortcuts to the ASG. For example, for querying the return type of a function several query nodes are needed just to reach the return type. If a function query node has an option for immediately querying its return type attribute, then this query becomes much easier to implement.

The difficulty of this method, is which shortcuts to provide. There are hundreds of nodes in the ASG containing thousands of links to other ASG nodes. Also, providing too many shortcuts results in bloated query nodes, and having too few shortcuts will not cater to the needs of all query writers.

The complication lies in the fact that there are many different ways to look at the ASG. For example a C++ for statement, can be seen as a repetition, but also as a scope with its own set of declarations. Clearly, the query developer needs a way to create its own shortcuts.

Our query system supports this using *aggregate queries*. An aggregate query is a basically a single query node built from a multitude of sub queries. A name as well as a description for the aggregate query can be specified by the query developer. An aggregate query can define several named *ports* to which another query can be attached. Ports of sub query nodes can be linked to these ports. A query port has a type, to indicate which query nodes may be attached to it. It has also a name and a description field, where the query developer can specify an intuitive name and description of the intent of the port.

For example, the query developer may construct both a repetition query and a scope query for the for-statement. The scope query may have a "declaration " query port for a declaration query,

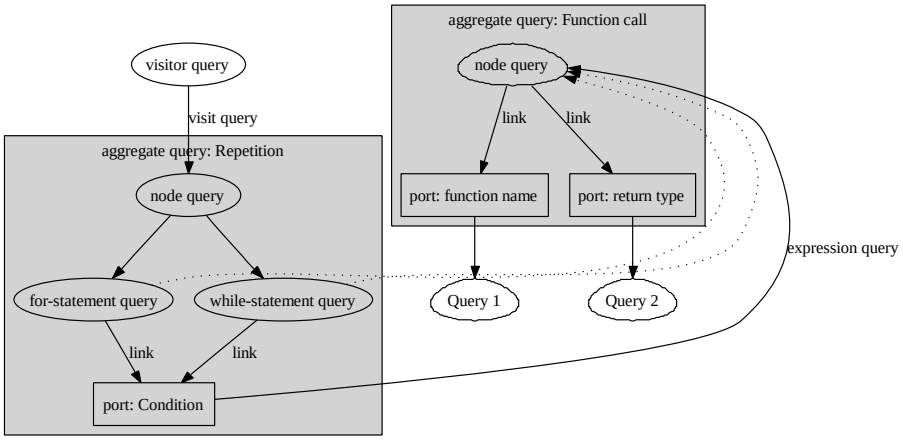


Figure 4.4: Query tree using two aggregate queries

while the repetition query has a "condition" port for an expression query.

Before a query is executed, all query sub trees attached to aggregate query ports are automatically copied to all linked sub queries. The copies are invisible to the user. Our query system does not support query node sharing, and this way the user can still change attributes of a query at a single location. The costs of copying the relatively small query trees are usually negligible.

Figure 4.4 shows an example of a query tree built from two aggregate queries. The query selects all repetitions having a function call as condition expression.

4.3.11 Properties

Creating a query from scratch is a time consuming and difficult process. But once an ingenious and really useful query is constructed, it can be reused many times over. Queries often have one or more parameters, which have to be specified before the query is executed. These values can be edited directly in XML, but that is tedious and inefficient.

We would like to expose these parameters in a graphical user interface, such that parameters for existing queries can be edited conveniently. For example, a function query may have a parameter "name" and a parameter "return type". We want to present the user a dialog where he or she is presented with a list of all query parameters, and can edit their values.

It is impossible to automatically generate intuitive names for query parameters, or list parameters automatically in a natural order. The meaning of a query parameter depends heavily on the context where it is used, what the query intents to do, and the expectations of the query user. Hence, the query developer has to provide this information manually, using a list of so-called *properties*. A property binds a name and an intuitive description to a query attribute. For each type of query attribute there is a corresponding property type. Examples of property types are *string property*, *boolean property*, and *integer property*.

There is a also a special type of boolean property, namely the *sub query property*. These properties can be used to disable part of a query tree. This eliminates the need to write a query tree for several combinations of sub queries. Instead, the query user can disable the parts of the query that he does not wish to use. An example is a function call query which selects constructor calls, member initializations, etc. besides normal function calls. Using a number of sub query properties the user can specify exactly which calls he wants to select.

The design of a query construction GUI is part of the future work of this project. An example of a property list specification in XML is given in the next section.

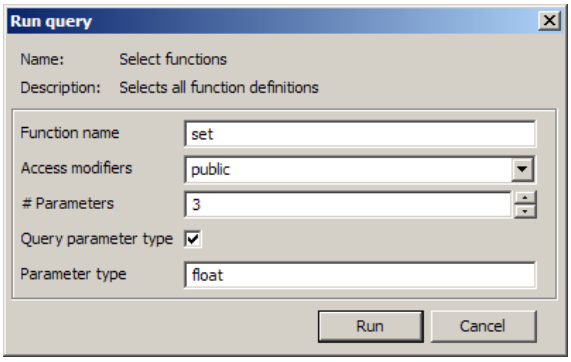


Figure 4.5: User Interface for editing query parameters

4.3.12 Editing query parameters

The property list allows for the generation of a simple graphical user interface for editing an existing query. Each property is represented in the user interface by text labels containing the name and a description of the property, and a user interface element to edit the attribute value. The type of user interface element used, depends on the property type. Interface elements are presented in the same order in which their corresponding properties are listed. This way the query developer has control over the layout of the user interface.

Ordinary string attributes, such as regular expressions or sub strings, are shown in a text edit box, where the user can specify an arbitrary string value. This value can directly be used as query attribute. Boolean attributes are shown in check boxes. For integer and character fields, ordinary text edit boxes are used. The user interface reports it to the user if invalid values are specified.

Many attributes are of an enumeration type. Attributes of enumeration type cannot be shown directly in the user interface. Enumeration constants are simply unique integer codes associated to a label with a description. The user is not always aware of all possible enumeration values. Some enumeration types have hundreds of enumeration constants, and we cannot expect the user to learn these by heart. Furthermore, the user hardly ever knows the value of a constant, but only knows what its label means. Therefore, the user interface presents enumeration attributes in a list box showing the labels of all enumeration constants. The user can select an enumeration constant from the list. The user interface uses the selected list element to create an enumeration value, which is used as query attribute. Some enumeration types are designed in such a way that their values can be used as flags. The user interface shows these attributes using exactly the same user interface elements, but for these types the user is allowed to select multiple items from the list.

Figure 4.5 shows a simple user interface for editing query parameters. The user can edit the query parameters and press 'run' to execute the query on the current selection. The following property list used for generating the user interface:

```
<Properties>
  <Property Type="String" Name="Function name" QueryId="1" />
  <Property Type="Enum" Name="Access modifiers" QueryId="2" />
  <Property Type="Int" Name="# Parameters" QueryId="3" />
  <Property Type="Bool" Name="Query parameter type" QueryId="4" />
  <Property Type="String" Name="Parameter type" QueryId="5" />
</Properties>
```

4.3.13 Query library

When we implemented our own set of example queries, it soon became obvious that we needed a convenient way to catalogue them.

We designed a *query library* for storing a collection of query trees. A query library contains a list of query trees annotated with a unique name and an elaborate description of what the query does. Query libraries can be stored in XML query library files. Query library files may either contain complete queries, or contain references to external query files. This way a single query specification can be shared between multiple libraries. The following code fragment shows the XML output for a query library named 'error queries'. It contains one query, viz. a query to select all garbage statements. The query file is stored in an external file named 'garbage.query'.

```
<QueryLibrary Name="Error queries" Description="Queries to find errors">
  <QueryItem
    Name="Select garbage"
    Description="Select garbage statements"
    QueryFile="Garbage.query"
  />
</QueryLibrary>
```

A query in the query library may be used as a building block for a new query. It is possible to attach a query tree α to an existing query tree β by using the clone operation of α. This requires only a constant amount of code. The clone operation makes an exact copy of α, and attaches this to β.

4.3.14 Query performance

Queries are often executed on very large databases. Still, we prefer to run them at nearly interactive rates.

Typical extraction units contain hundreds of thousands of selectable nodes, and a database easily contains a thousand extraction units. Queries such as 'select all if-statements' have to traverse through almost the entire ASG of an extraction unit to search for if-statements. Often the nodes that are actually selected by a query is small compared to the number of nodes traversed. Hence it is of utmost importance that the mechanism for node traversal used by a query system is efficient.

We developed an efficient visitor for the ASG that is able to traverse hundred of thousands of in-memory nodes in only a few hundreds of a second. This is much more efficient than loading an extraction unit from disk. Queries accessing nodes that are not in memory typically take longer to execute, depending on the speed of the storage device and the size of the extraction units. However, the performance is adequate for most queries and databases. The performance of the ASG visitor is achieved by storing nodes consecutively in memory, and traversing them in the same order as far as possible. We also minimized the size of ASG nodes to decrease memory requirements and increase performance. Current PC computers have very fast, but relatively small, cache memories. Another optimization that is often applicable and very effective, is to iterate only through objects of a particular type. For example, if we want to select functions matching a particular predicate, then we only have to evaluate this predicate on function nodes.

Testing the predicate of a node is extremely efficient. The query system is fully type-safe, which implies that relatively expensive string comparisons or conversions unnecessary for testing a query predicate. Moreover, a predicate is built from several very simple sub predicates. Many predicate evaluations are avoided by shortcutting predicate evaluation if the final result stays invariant.

The number of nodes that are actually selected is often relatively low compared to the tested nodes. Hence, most predicates fail after a small number of sub predicates is evaluated. We noticed that evaluating query predicates did not hamper performance.

Adding nodes to the result selection, however, does have impact on performance. Our selection data structure is implemented as a red/black tree and hence insertion takes $O(\lg n)$ time. We can probably optimize this by using a a good hash map implementation, offering nearly $O(1)$ performance.

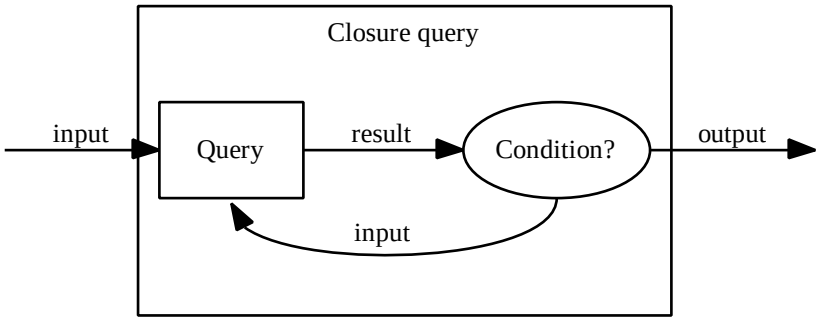


Figure 4.6: Schematic diagram of the closure query

4.3.15 Closure queries

Currently queries such as 'select all objects derived from wxObject' cannot be expressed as a single query tree. There may be n base classes to inspect on the base class chain of a class, and no query node is available for traversing base class chains.

Even if such a query would be available, traversing all base class chains of a collection of classes could be rather inefficient. Few classes may actually be derived from wxObject, but all base class chains must be traversed.

A better method is to select only the objects having wxObject as direct base class. The result of this query is a set of classes S. We can now select all objects inheriting directly from a class in S, and repeat this process until the query yields an empty selection. The result of the closure query is the union of all query results. This effectively computes the transitive closure of the base class relation in G.

Our query system offers *closure queries* for specifying such queries. The closure query repeatedly executes some query Q_n, producing S_n as output. Next S_n is repeatedly used as parameter for Q_{n+1}, until some condition C is satisfied. The condition in our example is $S_n = \emptyset$. Finally the results of the individual queries have to be combined to form the closure query result. An arbitrary set operation can be specified for combining the selections of individual queries. A schematic representation of a closure query is shown in figure 4.6.

4.4 Linker system

4.4.1 Introduction

EFES produces an extraction unit for each source file. Many symbols occur in multiple extraction units, because header files are typically included in many translation units. We wish to *link* these symbols, so that the end-user of our tool set can detect all occurrences of the same symbol in multiple extraction-units.

4.4.2 Method

We propose two kinds of linking: classical and extended. Classical linking offers roughly the same functionality as an ordinary linker, such as the one used by a C or C++ compiler after code generation. Extended linking extends this idea to other objects produced by our extractor, namely types. Object files employed by classical linkers do not have the notion of types, and are hence unable to link types.

Type linking is essential for us if we want to determine, for example, if two variables, in different translation units, have the same type. Or if we want to gather statistics about the

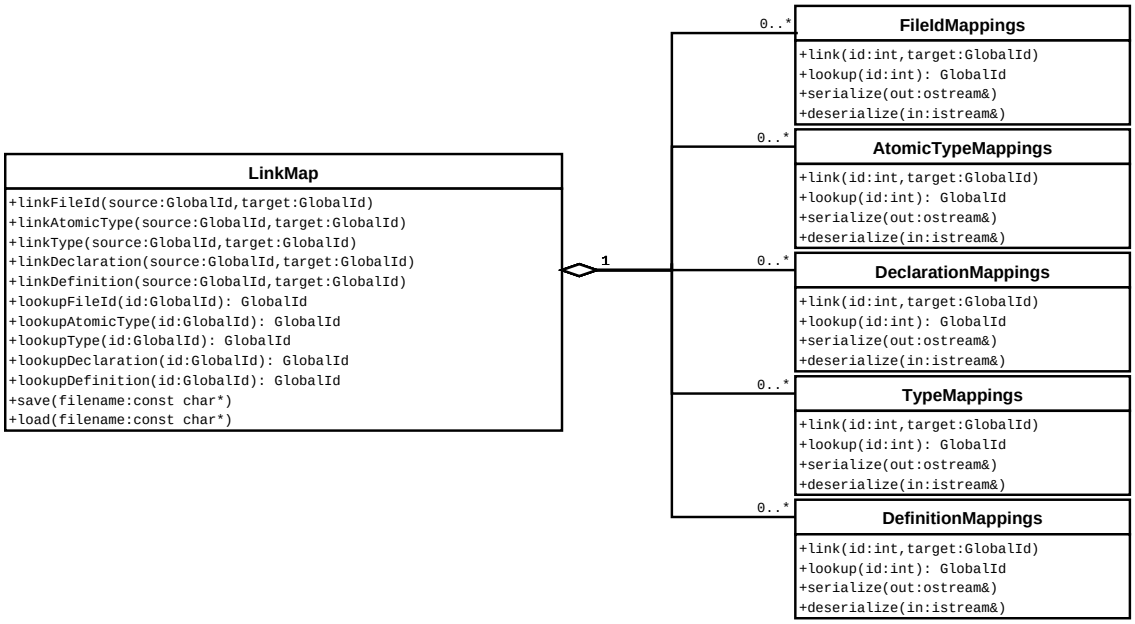


Figure 4.7: UML diagram of the link map data structure

number of unique types defined in a project.

We distinguish two kinds of types, namely compound-types and atomic-types. Compound-types are built entirely from other types, which may be either compound or atomic.

Our linker supports atomic-type linking, and optional compound-type linking. Typical translation units define much more compound-types than atomic types, which results in increased link times and a much larger *link map*. The link map is a data structure for storing link relations, and is discussed in the next section.

Another reason why we made compound-type linking optional, is because it is fairly efficient to determine if two compound-types are equal based on the equivalence of their subtypes.

4.4.3 Implementation

Linking is implemented as a postprocessing step in our tool chain. The postprocessing step generates a so-called *link map*, which maps global identifiers of source objects to global identifier of target objects. Two symbols represent the same object if and only if the link map maps them to an object with the same identifier. Figure 4.7 shows an UML diagram of the link map data structure.

We wrote a linker program, a command line utility which takes a file list as input and produces a link map as output.

The EFES user interface allows the end-user to configure the type of linking to use, or disable linking entirely. If linking is enabled, EFES automatically invokes the linker when all project files are parsed.

The end-user can also explicitly link or relink projects that are not completely processed. EFES uses the log file, which contains all output files generated during a particular run of the extractor, to generate a list of output files. This list serves as input to the command line utility.

The linker program keeps one extraction unit in memory at a time. Many projects generate multiple gigabytes of extraction units so this is essential for a practical tool. Extraction units of industrial projects are usually smaller than a few megabytes, and easily fit in main memory of a typical PC.

The linker only reads interface information from the ASG stored in the extraction units, information in local scopes is largely ignored. This roughly doubles the performance of the linker, and

decreases the memory requirements by almost a factor five on typical translation units. This does not effect the results, because classic linking ignores all symbols in local scopes. The linker does read all type information from the extraction unit, if type linking is enabled. Obviously, types are always linked, even if they are defined in local scopes.

We define type objects to be equivalent if and only if they have the same attributes, the same name, and are declared in the same scope. The linker maintains a dictionary mapping names to type identifiers. These identifiers correspond to types stored in an extraction unit. These identifiers are *persistent*, i.e. an object has the same identifier every time the file is read from disk. The EFES parser internally maintains a data structure mapping identifiers to pointers. This data structure is used by the linker for constructing a data structure to efficiently map object pointers to persistent identifiers. We denote the operation of obtaining the persistent identifier for an object x by $id(x)$.

Our type linking algorithm works as follows. The linker iterates through all types defined in a translation unit. For each type object τ the linker generates a name σ which uniquely encodes all scope information and type attributes of τ. This approach allowed us to implement the linker with only a few dictionary lookups. The costs of string concatenation are insignificant compared to the parsing of the extraction unit. The C programming language also allows *anonymous types*, i.e. types without an explicit name. We adapted the parser to generate a project-wide unique name for anonymous types based on their location.

Then we perform a dictionary lookup for name σ. If the dictionary already stores an identifier of a type object ν with key σ, it represents the same type as τ. In this case, we store the mapping from $id(\tau)$ to $id(\nu)$ in the link map. If no item is stored for key σ, we add $id(\tau)$ to the dictionary with key σ, and store the mapping from $id(\tau)$ to $id(\tau)$ in the link map. This way, all types are stored in the link map, and hence map to a type.

Classic linking works similar, but is complicated somewhat, because we prefer to link to a particular occurrence of the symbol. In case of functions, we like declarations to map to a function definition. In case of variable, all variables declared *extern* should map to the place where they are defined. This is convenient because it allows us to quickly find definitions, which is convenient for navigation in our visualization tool. Functions or variables declared in *anonymous namespaces* or declared *static* in the global scope are not linked by a normal linker. We implemented the same behavior in our linker, by ignoring *static* symbols in global scopes and by ignoring symbols in anonymous namespaces.

We maintain two dictionaries, one mapping names to function definitions, and one mapping names to sets of function declarations. For each extraction unit, we add all function definitions and declarations to the corresponding dictionaries.

When all files are processed, we link the declarations by iterating through all items in the declaration map. For each key κ, we perform a lookup in the definition dictionary. If the dictionary contains a definition, we link all declarations with key κ the this definition. If it does not contain a definition, we link all declarations with key κ to some declaration with key κ.

When all files are processed, our tool sorts the list of mappings by key, and writes it to disk. This makes lookups efficient, without requiring that EFESAPI must sort the link map every time it is loaded.

Finally, the linker writes a list of statistics to the console. EFES collects this information and stores it in the log file. The EFES API provides a link map interface that can load a link map file from disk. The link map interface provides a lookup functionality for object identifiers.

The link map interface can also report all occurrences of a symbol π. This is implemented by employing two binary searches. A copy of the link map is stored, which is sorted by target type. The EFESAPI link map can report all duplicated symbols by first searching the target symbol ρ for π, and then use a binary search to report the range where ρ occurs as target type.

Each extraction unit stores a file table. The file table is a list of all unique source files that make up the translation unit. Source locations indicate source files as indices into the file table. Comparing two source locations in different translation units is a common operation. Unfortunately comparing two local file identifiers by indexing the file table and performing a string comparison on the file names is not very efficient. Hence, we would like to map extraction unit identifiers

to project-wide identifiers. The linker generates a global file table and maps extraction unit file identifiers to global file identifiers. The file table and mappings are stored in the link map.

4.4.4 Discussion

In this section we discuss how our linker performs in practice. For all three kinds of linking we ran the linker on our suite of benchmark projects. We measured the time it takes the link all extraction units and the size of the generated link map. The results, which are shown in table 4.1, should give insight in the performance differences between the different configurations on typical projects.

	Classic		**Basic types**		**Extended types**	
Project	Time	Output (Mb)	Time	Output (Mb)	Time	Output (Mb)
Q3A	15.8s	0.520	16.3s	0.236	20.7s	2.8
Python	31.0s	0.116	32.0s	0.140	37.2s	1.91
EFES	78.1s	2.26	78.7s	7.86	259.1	20.9
Blender						
Coin3D	3m27s	4.53	3.2s	5.76		
VTK	3m41s	5.42	3m50s	5.89	6m01s	48.5
wxWidgets	270.8	7.33	283.3	8.24	606.4	82.8
Mozilla	565.4	4.62	365.6	7.01		

Table 4.1: Performance of EFESLink

It follows from the figures in the table that, even for very large projects such as VTK or Mozilla, the link map sizes are small enough to be stored in main memory of a typical PC. We also think the link times are reasonable, especially relative to parse time. Our current linker is about an order of magnitude faster than the fact extractor on typical projects.

With classic linking, the linker still needs type information. This is used for example for generating a name encoding a function symbol. This explains why classic linking is not much faster than classic linking with type linking enabled.

On all of the projects in our benchmark, basic type linking resulted in only marginally larger files and link times. Clearly, parsing the extraction unit dominates in the link timings. Hence, type linking is interesting as a preprocessing step. We think the user will link most projects with type linking enabled.

Extended type linking, on the other hand, sometimes results in significantly larger link times and link maps. For most projects the figures are still reasonable, so it can be an interesting option for the user. We think, however, that in most cases it is better to link extended types at runtime instead of as a preprocessing step. Smaller link map sizes and better link times can compensate for the extra time it takes to link extended types.

4.5 Clustering

The result of a query, a selection, is an unstructured set of ASG nodes. We want to cluster this set according to criteria provided by the user. The clustering can be utilized by the visualizations to give the user a more suitable view at the information.

For example, functions can be clustered by their number of parameters and by their return type. The source code visualization can then present functions in order of their number of parameters, and functions with the same return type can be rendered in similar colors. In this view the user can immediately spot patterns in the code, such as the amount of functions having many parameters, or the number of different return types.

Selections are clustered using a number of *cluster operations*. A cluster operation finds groups of strongly related ASG nodes, i.e. ASG nodes with similar attributes. There are two issues that

must we must address here. First, we must provide a similarity measure for ASG nodes. Second, we must provide a method for clustering similar ASG nodes. The user can provide the similarity measure using a query and a *distance functor*.

The query selects a set of attributes of an ASG node, producing a selection. The distance functor D is a function that takes two result selections S_1 and S_2 as input and returns their distance as output.

$$D(S_1, S_2) \rightarrow \mathbb{R}$$

The query system has a variety of predefined distance functors. For example, one returning the difference in cardinality of the two selections, and one counting the number of different nodes in the selections.

For clustering we use a bottom-up agglomerative clustering algorithm. We start with the individual ASG nodes, and recursively cluster the most similar ones in a cluster, until we obtain a single cluster, which is the root of the cluster tree we thus created.

Figure 4.5 shows a class diagram of the data structure we use for clustering. Cluster nodes implement the selection interface. The *contains* function of this interfaces can be used to query if a cluster tree contains a global identifier. The cluster class offers functionality to iterate through the leaf nodes of a cluster tree.

The user often applies clustering to query results, which he or she might want to store on disk for later access. EFES uses the XML format for storing hierarchical cluster trees. Selections can either be embedded into a such a cluster file, or refer to an external selection file. If a selection is embedded, then all global identifiers in the selection are listed in an XML text representation. External selections are always binary files, which may be slightly smaller or faster to write.

```
<ClusterNode>
  <ClusterLeaf SelectionFile="calls.selection"/>
  <ClusterNode>
    <ClusterLeaf>
        <Selection>
                <ExtractionUnit Id="1">
                        <![3 5 6 7 4 ]]>
                </ExtractionUnit>
                <ExtractionUnit Id="2">
                        <![ 2 ]]>
                </ExtractionUnit>
        </Selection>
    </ClusterLeaf>
  </ClusterNode>
<ClusterNode>
```

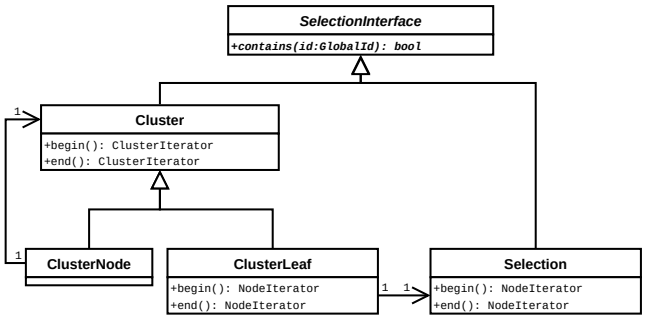


Figure 4.8: Class diagram of the cluster nodes

4.6 Query examples

In this section we will provide ten examples of queries that both commonly used and applicable in many different contexts. By giving a number of examples we hope to convince the reader that querying the ASG offers many benefits to the user compared to normal text searches and demonstrate the power and flexibility of our query system.

The queries are ordered by the complexity of their implementation. Some of the more complex queries can be implemented using more basic queries in the list.

Query 1: Select all ASG nodes
Given a selection of root nodes, it is often interesting to select all their child nodes. The user can apply this query for example to find out how many ASG nodes an extraction unit, or even an entire fact database, contains. This gives an estimation of the project size.

Query 2: Select all ASG nodes with type 'x'
Sometimes the user want know how often a C++ construct occurs in a code fragment. This query can be used to find all uses of the infamous goto statement, or all exception handlers.

Query 3: Select all ASG nodes having a name matching regular expression 'x'
Most programmers frequently search their code for named constructs. The advantage of this query, compared to normal text searches, is that only named items are searched. Ordinary text searches may clutter the result by matching text in comments, keywords or other items.

Query 4: Select all ASG nodes having 'x' and matching regular expression 'y'
The power of query 2 and query 3 is combined to form a more generic query. Many interesting selections such as 'all classes starting with wx' and 'all variables named i' can be made with this query.

Query 5: Select all functions named 'x' with more than 'y' parameters of type 'z'
This query is useful when the user wants to find functions applied to objects of some type. The query can also be used to find functions with a very large number of parameters of a certain type.

Query 6: Select all function calls
This query serves as a basic 'building block' for a myriad of useful queries. The query itself can be used for constructing call graphs and finding recursive functions.

Query 7: Select all direct subclasses of class 'x'
Not all C++ classes are designed to serve as base classes, for example a virtual destructor may be missing. Using this query the user can find where classes inherit from classes unsuitable as base. This query is often used as part of a closure query, for example to form a possible implementation for Query 8.

Query 8: Select all classes derived from class 'x'
The inheritance relation is a very strong relation. If the user wants to change the interface of an abstract class, then this query can give insight in the number of classes that require change. This query can also be used building diagrams showing the inheritance graph.

Query 9: Select all reachable functions
Useful for finding the parts of a program that are never reached, so-called 'dead code'. The query can also become the basis of a query searching for recursions or indirect calls to 'dangerous' functions.

Query 10: Select all recursive functions
Complex recursive functions are often difficult to understand for a programmer, and hence these functions often causes bugs in programs. Using this query the user can quickly identify places in the code where bugs are likely to occur.

4.6.1 Designing queries

In this section we demonstrate how each of the example queries can be constructed using our query system. We want to give the reader insight in how a query can be constructed. Constructing queries requires some knowledge of the ASG structure of EFES. We will not describe the whole structure of the ASG here. Instead we primarily focus on the thought process necessary to implement queries. Where necessary, we will give minimal information about the structure of the part of the ASG that we want to query. Hopefully this will inspire the reader to write entirely new, useful queries.

First we should elaborate on the input of a query. We noted the input is a selection of nodes. For most queries we assume that the input selection S is a forest of ASG subtrees. This is often convenient because it enables the user to simply select a scope, for example a fact database, a translation unit, or a namespace, and query the nodes in that context.

Given a selection S, we can create a selection S' containing all nodes in forest S, but we rather avoid this step for efficiency reasons. Instead, the query can traverse the nodes in the forest directly.

Query 1: Select all ASG nodes

Our query system contains a special *query visitor* for precisely this purpose. This query can serve as the root for our query tree. The query visitor can have a *visit query* parameter that is executed on each visited node. We can implement query 1 by adding an *ASG node query* as *visit query*. This query tests if the visited node is an ASG node, which is precisely what the query should select. We can finish the query by adding a *node selector* object to the visit query.

Query 2: Select all ASG nodes with type 'x'

This query is very similar to the first query, but differs by additionally querying the type of an ASG node. This can be achieved with a *type query* object. By default, queries use the "and" accumulator and hence perform a logical and between the results of their sub queries. We can transform query 1 into query 2 by adding a *type query* to the list of *node queries* of the *ASG node query*. Now the node query invokes the selector if and only if the queried node is an ASG node and has type 'x'.

Query 3: Select all ASG nodes having a name matching regular expression 'x'

Like query 2 this query is based on the first query, but additionally it queries the name attribute of an ASG node. We can query the name attribute by adding a query to the list of *name queries* of the ASG node query. In this case we want the name to match a regular expression, hence we add a *regular expression* name query.

Query 4: Select all ASG nodes having 'x' and matching regular expression 'y'

This query can be constructed by combining the second query with the thirth query. This can be achieved by adding the name query of query 3 to the visit query of query 2. The default and-accumulator makes sure ASG nodes are only selected if both conditions are satisfied. The final query tree for query 4 is shown in figure 4.9.

Query 5: Select all functions named 'x' with more than 'y' parameters of type 'z'

This query selects functions satisfying the condition that their name matches 'x' and that they have more than 'y' parameters of type 'z'. We can select all functions by creating a visitor query with a function query as visit query. Naturally, the function query must have a selector, because we are interested in selecting the functions.

The name of the function, and its parameters are not stored in the function node. Instead a function node has a declarator attribute *name and parameters*. The declarator has an attribute

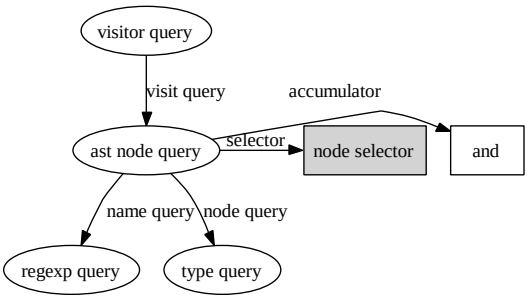


Figure 4.9: The query tree constructed for query 4

declaration representing the declaration of the function. We can construct a path to the function declaration by adding a *declarator query* with a *function declaration query* to the list of name and parameters queries of the function query. The function declaration references a variable containing the name of a function. We can add a *variable query* with a *regular expression query* as name query the function.

Thus far we constructed a query for selecting all functions with a name matching regular expression 'x'. We must strengthen the selection criterion with 'with at least 'y' parameters of type 'z' '. A function declaration contains a list of parameters, which we can query using a *list query*. Next we want to find out if more than 'y' parameters satisfy some condition. We can achieve this by assigning an accumulator with a counter and a *less then* comparison function to the list query.

For each parameter we have to query if its type is 'z'. The parameter list stores a type identifier. The type identifier references a declarator node, which in turn references a type node. In our query we can construct a path to the type node. The type node can be queried using a *type query*. The user can specify 'z' in this node. Figure 4.10 graphically depicts the query tree we constructed.

Query 6: Select all function calls

This is a relatively complicated query to implement. Finding all function calls is nontrivial because function call expressions do not directly refer to functions. Instead these nodes form the root of an arbitrarily large expression tree containing a variable expression leaf node, which in turn refers to the function. Directly searching for variable expressions yields an incorrect result, because variable expressions also occur in different contexts such as variable assignments. Another complexity in finding function calls is that many C++ constructs, such as new expressions and constructor calls, possibly result in a function call. We want a query that reports all function calls, disregarding how the call is performed.

We can find all function call expressions by using a *function call expression query* assigned to a *visitor query*. The variable expression in the expression subtree can be found by adding a *visitor query* with a *variable expression query*.

From a variable expression we can arrive, via the variable node, at the called function. We can select this function by adding selection path consisting of a variable expression selector, a variable selector, and a function selector.

Next we will extend our query such that it selects all called functions. We can do this by adding a *new expression query* and likewise queries for all other types of function calls, to the list of visit queries of the root of our query tree. Those nodes all refer directly to the function variable, so we can simply add a selector path for selecting the called function.

Query 7: Select all direct subclasses of class 'x'

We can select all classes by adding a *class query* with a *node selector* to the list of visit queries of a *visitor query*. The bases of a class are stored in a list of base classes, which we can query

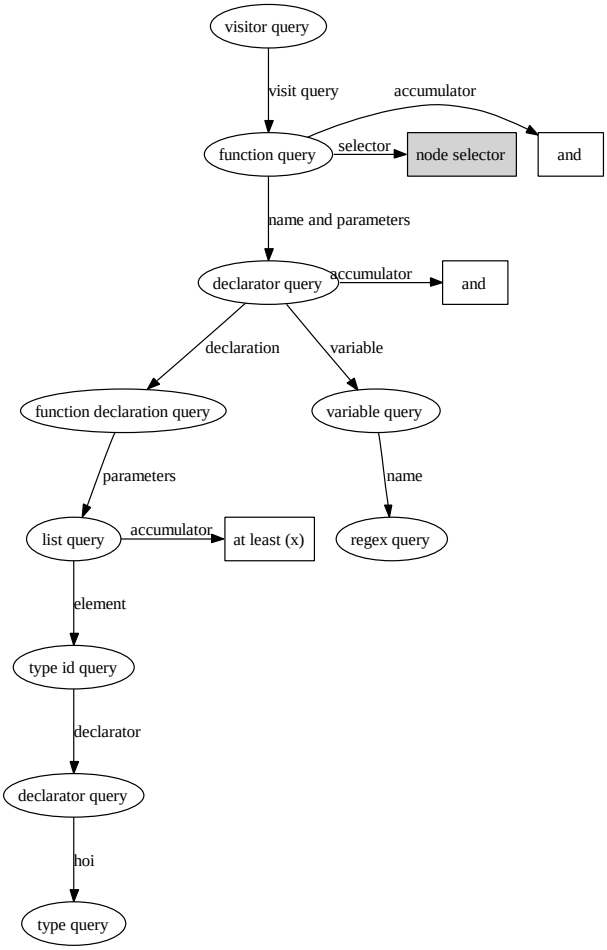


Figure 4.10: The query tree constructed for query 5

using a *list query*. If at least one of the elements in the list occurs in selection x, the class should be selected. Hence, we attach an *or accumulator* to the *list query*. We use a *selection query* as element query, for testing if the base class occurs in x.

Query 8: Select all classes derived from class 'x'

This query not only selects all classes directly derived from a class in 'x', but also all classes inheriting 'x' via a sequence of base classes. This corresponds to the transitive closure of the base class relation in the ASG. We can implement query 8 as a *closure query* of query 7. The stop condition for the closure query is that the empty set is found.

Query 9: Select all reachable functions

This query selects all functions reachable from a selection of given functions. It is clearly an undecidable problem to find all functions that are actually called using static analysis, but we are able produce a superset by assuming that all calls are actually executed. We can find all functions called by a given function by applying query 6 on the function body. By applying this step repeatedly using a *closure query* we can find all reachable functions. The stop condition for the closure query is that the function call query produces no new results, and hence leaves the result selection unchanged.

Query 10: Select all recursive functions

A recursive function calls itself either directly or indirectly via one or more other function calls. We can find a superset of the recursive functions by running query 7 on the function bodies of all functions. If the original function occurs in the list of reachable functions, then we found a potentially recursive function. Of course, this is extremely inefficient and certainly not suitable for projects with millions of functions.

If we change the query slightly to 'select all functions that may recursively call themselves in at most n steps', then we can limit the number of iterations for the closure query to n. This query can be executed efficiently, taking less than a second on projects with more than a hundred thousand function calls. In practice it still finds almost all recursive functions, even for small values of n.

The recursive function query algorithm looks like this:

```
procedure FindRecursiveFunctions(Selection s, int depth)
begin
  Selection r;
  for each f in s
  begin
    Selection calledFunctions;
    calledFunctions := findCalls(f, depth);
    if calledFunctions.contains(f)
      r.insert(f);
  end
  return r;
end
```

We can quickly implement this query creating a *function query* with a *closure query node*. We assign the query 7 as the closure query of the closure query node.

Chapter 5

Fact visualization

5.1 Introduction

In the pipeline view of our complete framework for fact extraction, querying and visualization of C++ code, this section describes the last element: fact visualization. The fact extractor and query system produce a huge amount of information. The end user can absorb the information in various ways. This section will discuss how the data is visualized.

There are many types of visualizations that we can construct from our extracted facts. Here, we focus only on a limited subset of visualizations we have actually implemented, given the limited amount of space we have. First we present our visualizations of call graphs and system structure. The second part of this chapter is devoted to the visualization of source code text in its original form enriched with extracted facts. To enable the source code visualization we have created a new visualization framework that we briefly discuss. We then go on to describe how the data is transformed into a structure for the visualization framework, and we show how we have applied the visualization framework.

5.2 Call graph and system structure visualization

A common question of software engineers is for the function call relations in a set of functions. The query system can answer this question. We have written three exporters that save the answer in the file format required by each one of three (call) graph visualizers. The class inheritance in a system is also represented in a graph. We have generated a class inheritance graph in the proper file format and we have visualized this as well. In this section we present the resulting visualizations.

5.2.1 Call-i-grapher

The complete call graph of an entire system, or even a sub system of a large program, can easily contain tens of thousands of call relations. Still, the user wants to see this call graph in a structured way. We need a way to present many thousands of call relations in a single graph in such a way that it exposes the structure of a large system.

Source code is usually organized in different modules, namespaces, files, and classes. We can use this organization for creating hierarchical call graphs. We structured several call graphs ranging in size from medium to large using two different approaches.

First, we tried to use names of global scopes, viz. namespaces and classes, to create hierarchical call graphs. However, this method proved to be ineffective for most projects. Many large projects avoid the use of namespaces entirely, and certainly not all projects are object oriented. Even projects that do use namespaces, tend to define all their symbols within a small number of

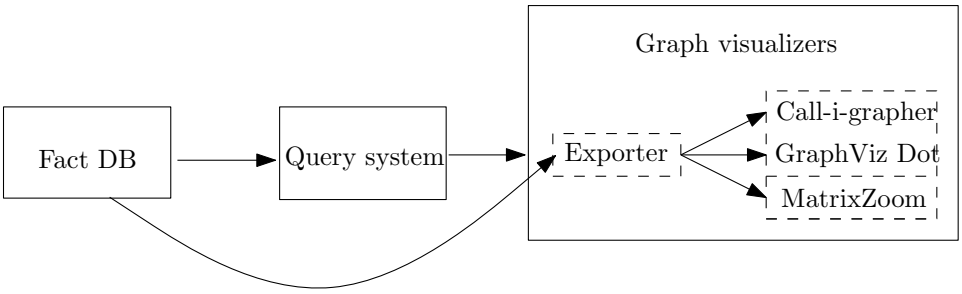


Figure 5.1: Schematic overview of the exporter pipeline

namespaces. Structuring by namespace often results merely in a sub division between application code and library code.

For most projects, the directory structure and file name proved a much better basis for creating a hierarchical call graphs. In most large projects, files belonging to a single module are stored in the same directory. Similarly, classes and functions with related functionality are frequently grouped in the same file.

For most medium sized object oriented projects, we used a hierarchy consisting of three levels, viz. module, file, and class. For larger projects, we could often add a sub module level, as module directories contained sub directories. The levels in the hierarchy can be specified by the user as arguments to our exporter tool, because it differs greatly per project which levels result in the best possible hierarchy.

CALL-I-GRAPHER is a visualization tool in development by Holten *et al.* [33] which can visualize adjacency relations in hierarchical data. It uses the hierarchy of the graph to bundle edges based on sub systems. If we structure this data well, we can insightfully visualize very large call graphs this way. Even if individual functions are invisible, the bundling ensures that we can at least see the coupling of subcomponents in the system.

We created an exporter for our EFES to write CALL-I-GRAPHER graphs. Figure 5.1 shows the complete pipeline through which source code flows before we can finally produce a call graph. The exporter uses the query system to create the call graph. The exporter can also also query and export attributes of call relations. We used a test version of CALL-I-GRAPHER that can visualize these attributes. Figure 5.2d shows virtual functions in a different color.

Graph format

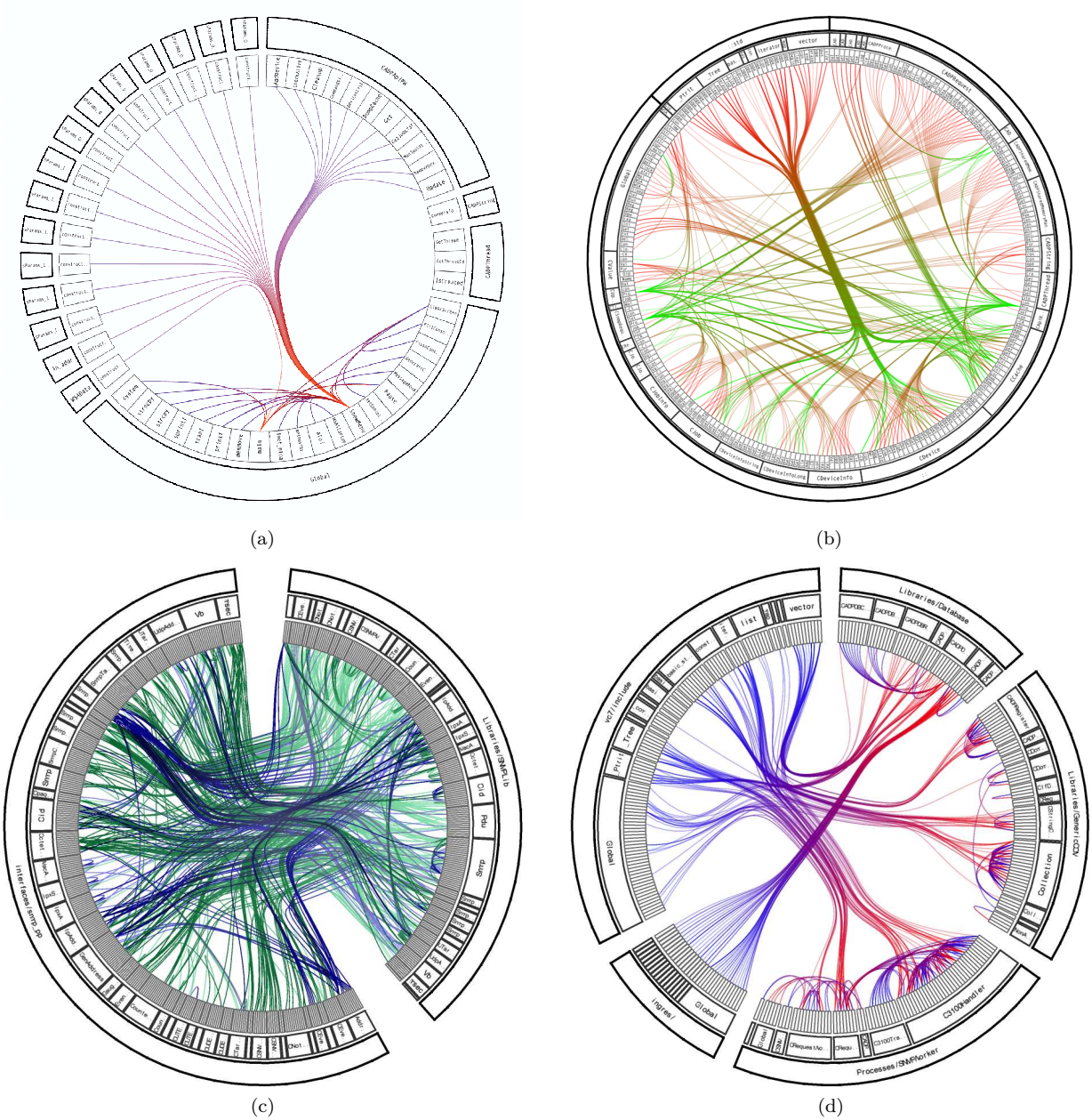
The call graph is then written to a CALL-I-GRAPHER graph. The file format used by CALL-I-GRAPHER, spreads a graph over multiple files. Figure 5.3 shows a hierarchical call graph consisting of three levels, viz. layers, units, and modules. The corresponding CALL-I-GRAPHER files are:

"modules.txt"

```
A
```

"classes.txt"

```
A.A
A.B
A.C
```



(a) (b) (c) (d)

Figure 5.2: Call-i-grapher visualizations

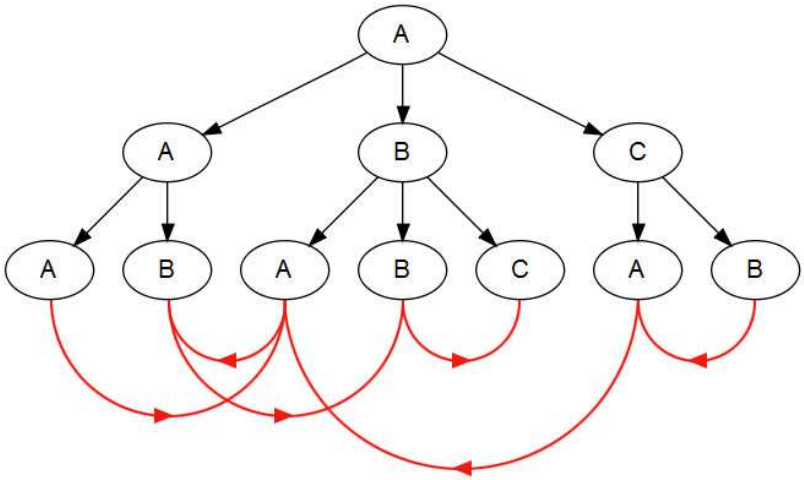


Figure 5.3: Hierarchical graph with 3 levels

"functions.txt"

```
A.A.A
A.A.B
A.B.A
A.B.B
A.B.C
A.C.A
```

"class_partof_module.txt"

```
A.A A
A.B A
A.C A
```

"function_partof_class.txt"

```
A.A.A A.A
A.A.B A.A
A.B.A A.B
A.B.B A.B
A.B.C A.B
A.C.A A.C
A.C.B A.C
```

"function_calls_function.txt"

```
A.A.A A.B.A
A.B.A A.A.B
A.A.B A.B.B
A.C.A A.B.A
A.B.B A.B.C
A.C.B A.C.A
```

The first three files declare the levels in the graph, respectively layers, units, and modules. The next two files declare to which level a sub level belongs. Finally, the last file specifies the call relations, which are shown as red arrows in figure 5.3.

Generating names

For each function in call graph the exporter queries, depending on the levels to use for the hierarchy, function name, class name, namespace name, file name, sub module name, and module name. These names are then stored in a dictionary having the function identifier as key. Multiple functions can have the same name, because C++ supports function overloading. Therefore, our exporter can generate a mangled function name by encoding the types of the arguments in the function name.

For each level in the hierarchy, the exporter maintains a set of strings. Each name in the set corresponds to a node in the graph. Furthermore, the exporter maintains a set of 'part-of' edges. 'Part-of' edges specify contain two node names, and specify to which level a sub level belongs.

Linking

Normal functions are usually declared in source files, because multiple definitions of the same function result in linker errors. If a function resides in a translation unit different from where the caller is placed, then a forward declaration must be placed in the translation unit of the caller. Frequently, this is done by including a header with function declarations. Multiple declarations of the same function are not allowed in C++. Of course, since EFES can deal with erroneous code, we have no guarantee that this is really always the case.

This implies that the file and module name of a function declaration is unsuited for generating call graphs. This way, a single function may be represented in the graph by multiple graph nodes. The function declaration is a better candidate for generating names, because it is often unique. The trouble is that the definition of the called function may be located in a different translation unit. Queries are always executed in the context of a single extraction unit, and a single extraction unit stores but the information about a single translation unit. The query can use the *link map* for resolving function definitions. The link map always links all declarations of the same function to a unique declaration. Hence, we have a deterministic way to find function declarations.

Output files

The exporter generates the name sets by iterating through all functions in the call graph. It uses the name dictionary to find the name information of a function. This information is then used to generate the complete function name, as well as all complete names for all parent levels in the graph.... The generated names are then inserted into the name sets. Furthermore, the name of a sub level is combined with the name of its level to create 'part-of' edges. These edges are then inserted into the corresponding edge sets. The exporter can then write the files declaring the hierarchy of the graph, by iterating through the generated sets and writing each name.

Finally, the output file containing all call relations is generated by iterating through all function calls, and writing the complete names of all callers and their callees.

Examples

Figure 5.2 shows several hierarchical call graphs visualized using CALL-I-GRAPHER. Figure 5.2a shows a hierarchical call graph with two levels, class name and function name. Rings closer to the center represent levels lower in the hierarchy. In this figure, the outer ring shows classes, and the inner ring shows functions. The red color indicates the caller, and the purple color indicates the callee. The figure shows that one of the functions calls the constructors of many different classes. In figure 5.2b a three-level hierarchy is shown, using namespace, class and function levels. Calls go from green to red. Because there are only two different namespaces, bundling is less effective. However, the bundling immediately reveals which classes heavily use the STL. Figure 5.2c shows another three-level hierarchy, but this call graph uses module name instead of namespace name for the top level in the hierarchy. This figure shows attributes of a call relation. In this picture, virtual calls are visualized with blue edges and normal calls have green edges. Figure 5.2d shows another call graph using the same three-level hierarchy as in figure 5.2c. Blue indicates the caller,

and purple indicates the callee. Five different components are shown in the outer ring of the graph. The edge bundling quickly reveals the coupling between the different subcomponents.

5.2.2 MatrixZoom

When looking at a high level view of a hierarchical graph, conspicuous parts often immediately stand out. We need a way to select the part of the graph that we want to view in more detail. The tool should than *zoom* into these parts of the graph. A limitation of the test version of Call-i-grapher we used, was that it had no such zooming facility. This limited the size of the graph that could be visualized without cluttering the view.

Tools such as MatrixZoom [75] can show hierarchical graphs in adjacency matrix representation. These tools are suitable for visualizing large call graphs. The tool can, for example, show callers along the vertical axis, and callees along the horizontal axis of the matrix. Function calls can then by indicated by highlighted matrix cells. Such tools can also show the call graph at a higher level. A highlighted cell can then indicate that there is at least one call between two modules.

The drawback of this method of visualization, however, is that only a single level in the hierarchy is shown at once. Of course, the number of calls at a lower level in the hierarchy can be visualized, but only in a limited way. It is easy to see whether components are coupled, but it is not immediately visible how strongly they are coupled.

Output files

We created an exporter for EFES to write MatrixZoom graphs. Figure 5.1 shows the complete pipeline for generating these call graphs. It is the almost the same pipeline as used for generating Call-i-grapher graphs. The exporter of MatrixZoom uses the same algorithm as the exporter for Call-i-grapher, because the tools use roughly the same format.

Examples

Figure 5.4 shows the call graph of a software system in adjacency matrix representation. The matrix shows call relations between modules. A red cell indicates that the module along the vertical axis contains at least one call to a function in the module along the horizontal axis. Not surprisingly, almost the entire diagonal is colored red. This indicates that most modules call themselves. A striking feature, however, is that module Base, a module that should consist merely of platform independent code, calls a function in the module Win32, which is reserved for platform dependent code. The only information that the tool can show at this level, is that there is exactly one call between the modules. Only by zooming into the code, it becomes clear that this is a factory method.

Another call graph visualization is shown in 5.5. This call graph was generated without using a link map. The module name of a called function corresponds either to the directory where its declaration occurs, or the directory where its definition is given. Which location is used, depends on whether or not the caller and the callee occur in the same translation unit. In this particular software system all header files are placed in a single directory "interfaces". The visualization clearly highlights the problem, because almost all modules call the "interface" module.

5.2.3 Graphviz

AT&T Graphviz [9] is open source graph visualization software. It has several main graph layout programs. Graphviz works by writing a graph visualization to a file, which be in a variety of standard bitmap and vector file formats. The software lacks an interactive facility for browsing an zooming call graphs, and is therefore not very suitable for visualizing large call graph.

We tried to create useful visualizations using several graph layout programs of Graphviz. Using neato we were able to get acceptable results for very small call graphs. But, we noticed that it

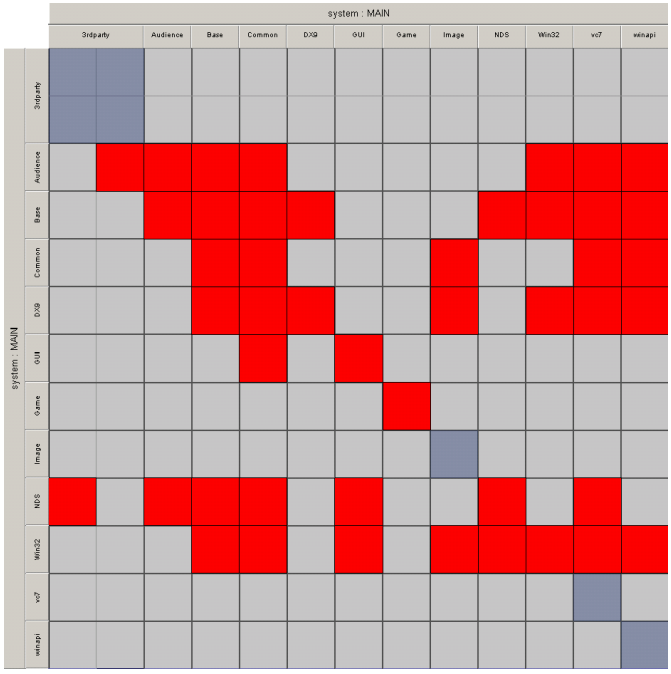


Figure 5.4: MatrixZoom visualization of a software system

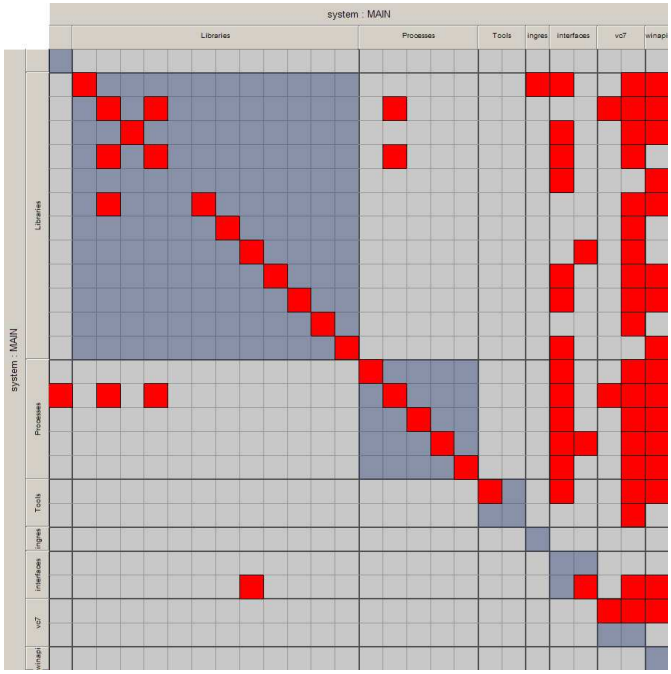
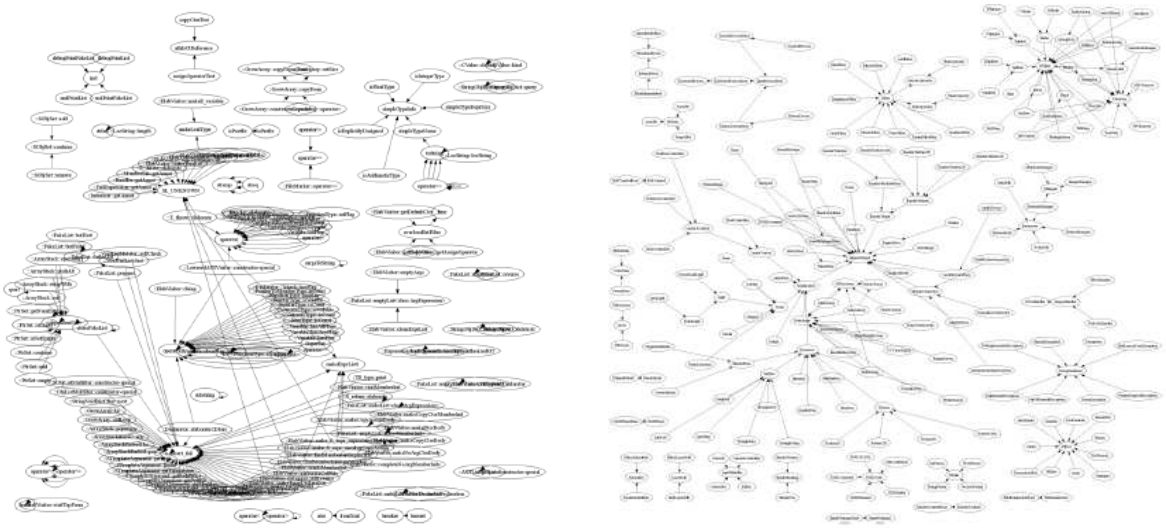


Figure 5.5: MatrixZoom visualization without linking



(a) Call graph visualization

(b) Inheritance graph visualization

Figure 5.6: Call graph visualizations using AT&T GraphViz

was nearly impossible to generate insightful visualizations for even moderately sized call graphs with only a few hundred calls.

Figure 5.6a shows a relatively small call graph visualized using Neato. At the bottom of the picture, a large cluster of functions is shown. The function at the center of the cluster is called by all functions surrounding it. Neato clearly has problems to arrange the nodes calling the function in the center, because many nodes are overlapping. It is a common situation that a single function is called by many other functions, so a good call graph visualization tool should be able to handle this situation gracefully. There is an option in Neato for reducing overlap, but we noticed that the visibility of clusters was strongly diminished if we enabled the option. Another problem of GraphViz is that the names of nodes are shown inside their shape. Functions frequently have long names indicating their meaning, which reduces the quality of the resulting graph visualization by increasing the number of overlapping nodes.

Figure 5.3 is a small hierarchical call graph visualized using Dot. The red arcs indicate calls, and the black arrows indicate hierarchy. The graph shows only a few calls, and already the resulting visualization becomes hard to interpret. There are many edge crossings, and the graph size grows rapidly when the number of calls and functions increase. Obviously this method for visualization does not scale to a graph consisting of thousands of nodes.

Figure 5.6b shows a class diagram visualized using TwoPi. The arrows indicate the inheritance relation. Arrow heads point at base classes. We used an option to reduce overlapping which decreases the visibility of clusters.

5.3 Code-based fact visualization

The source code visualization shows the ASG in a format that is almost identical to the view that a software engineer has when working with the source code, the origin of the ASG. The view will be enhanced with various graphical elements (e.g. texture, shading, colors) to reflect additional information extracted from the fact database using the query system. It is important that the enhancements keep the layout invariantly similar with a text editor's layout.

When we want to visualize the extracted ASG as source code in the original layout of a text editor, the largest problem to overcome is that the ASG does not reflect exactly the structure of the source code *text*. For many programming languages, the corresponding ASG will have a one

on one correspondence with the textual source code structure. This is however not the case for the C++ program representation in the ASG of EFES. The preprocessor is the culprit of all the difficulties involved.

If a C++ source code input does not contain any preprocessor symbols then a pre-order traversal of the ASG will visit all syntactical elements in the order of appearance in the input. A C++ source without preprocessor symbols will always consist of a single input file. When the preprocessor is involved in the compilation process, or fact extraction process, then multiple source files can create a single ASG. The preprocessor has two means of interrupting the pre-order ASG traversal property. Include directives will logically insert any preprocessed text from other files on the spot of the directive. A macro call can eliminate or duplicate its arguments and add extra code originating from other parts of varying files. The preprocessor can selectively include source code or leave it out of the ongoing compilation process by means of conditional compilation. Conditional compilation does not break the pre-order ASG traversal property.

Since we have complete information on the input we can reconstruct all the source code that the fact extractor can parse. We can not reconstruct source code that the fact extractor can not parse because the source code is included in the ASG only as a marker of non-parsable symbols. Also source code will not be represented in the ASG if conditional compilation makes it so. We can visualize *where* source code is excluded from the ASG so we can still fulfill the layout constraints.

Source code can be visualized with less layout similarities with the source files. An example of such a visualization is a list of class definitions. The class definitions presented will not necessarily be in that order in the source files and perhaps not even in the same extraction unit. The layout within the class definition would still match the source file of origin. Furthermore we can choose to visualize the ASG as it is after full macro expansion meaning that we do not visualize macro calls. We actually have chosen to visualize the source code in the exact same way as it is seen in the original input. The other two strategies require only minor adjustments to the algorithm presented.

The source code visualization can have several *selection* inputs. One selection will determine what gets visualized. All other selections are used for enriching the visualization. As a simple example, consider a visualization of selection $S1$, describes a code fragment. We create a selection $S2$ by querying on $S1$. $S2$ contains all declarations that are contained in $S1$ or that are children of nodes contained in $S1$. Similarly we create a selection $S3$ containing name qualifiers naming "i". $S2$ and $S3$ can be, but are not necessarily, a subset of $S1$. Our visualization framework lets us easily construct a visualization which shows all code in $S1$ equal to a text editor's view, all elements in $S2$ rendered in a red font, and all elements in $S3$ rendered with a blue background. Figure 5.7 illustrates the resulting visualization.

The type checker has a primary function of typing the ASG and resolving ambiguities. The type checker in our fact extractor also *lowers* the ASG. Lowering the ASG means transforming the ASG into an operationally equivalent ASG which is simplified. All those simplifications aim to make the ASG look more like a plain C ASG. This is of course not completely feasible since, for example, virtual calls do not have a simple C equivalent. We explain some of the lowering operations here. Overloaded operator applications are resolved and replaced by their respective function call. Implicit conversion operators are made explicit. Implicit references to the this pointer are made explicit. Parenthesis around expressions are always discarded; the order of expression evaluation can be determined from ASG layout. Constructor calls of simple types (int, float, etc.) are replaced with cast expressions as simple types do not have constructors. There are many more examples but this selection should demonstrate that the lowering of the ASG obscures the process of restoring the original source code from the ASG. We record certain lowering operations in the ASG, enabling a proper restoring of the ASG. For example we keep track of the number of parenthesis removed around each expression.

5.3.1 Visualization framework

Our visualization framework, schematically shown in figure 5.8, enables the visualizing of various kinds of data in a flexible manner. There are several components that interact with light weight

```
void TestPNGImage(void)
{
  ImageManager &imgMan(ImageManager::instance());

  ImagePtr img(new Image(256, 256, IMAGEFORMAT_RGBA));

  // write a simple pattern to the image buffer
  uint32 *target = reinterpret_cast<uint32*>(&(*img)[0]);
  int index = 0;
  for (int i=0; i<256; ++i)
    for (int j=0; j<256; ++j)
      target[index++] = (j + (i << 8)) << 8;

  string testFile("../TestData/pattern.png");
  imgMan.writeImageToFile(*img, testFile);
}
```

Figure 5.7: A visualization based on three selections

interfaces offering a lot of flexibility. The basis of every visualization are selections with nodes. Nodes refer to the actual data and selections represent a subset of the total available data. A node can represent a syntactical structure like a function or a compound statement, but we also use nodes to represent tokens, the smallest textual units the parser of EFES can recognize. Every node in the visualization is represented by an *Item*. An item refers to a node in a one on one relation, and augments that node with visual attributes, e.g. a shape or a texture.

Items are the primary data structure in the visualization framework. Items can have other items as children, thus forming an *item tree*. The nodes referred in the item tree can follow the same tree structure but this is not mandated in any way. As we will show later, the source code visualization builds up an item tree containing a structure similar to the syntactical structure contained in the nodes that make up the ASG. However the leave nodes being token representations do not originate from the ASG and only have that parent/child relation in the item tree.

In our visualization framework, shapes are variants. A shape can be “void”, rectangularor “code block like”. Shapes are not convex per se but the shape of an item shall always enclose the children’s shapes. More trivial visualizations would suffice with the constraint that shapes are always rectangles. We need to be more flexible because logical constructs in source code form a code block shape, as shown in figure 5.9 where the shapes of two if statements are outlined. The shape interface encapsulates the variant behavior by providing functions that hide the complexity. Only the actual drawing routines needs the ability to query shapes for exact details. When for any reason a node should not be represented visually, it can have the “void” shape. This shape does not need to be assigned any parameters nor will it ever interfere with the constraint that it needs to be enclosed by its parent item’s shape.

The View class

The view class (figure 5.8) of our visualization framework, is central to the visualization. The view class manages a single item tree with the help of other components. A selection forms the interface between data providers and visualizers. A separate class, the ItemTreeBuilder, is responsible for building an item tree from a selection. In general, different kinds of input data need different kinds of item tree builders. This is because the range of input data is very broad. The ItemTreeBuilder only determines item tree structure, the weigh and layout components will only change item attributes.

A view has only one weigher, one layout and one renderer associated with the item tree. These components all operate on the whole item tree. It would be more flexible if multiple renderers

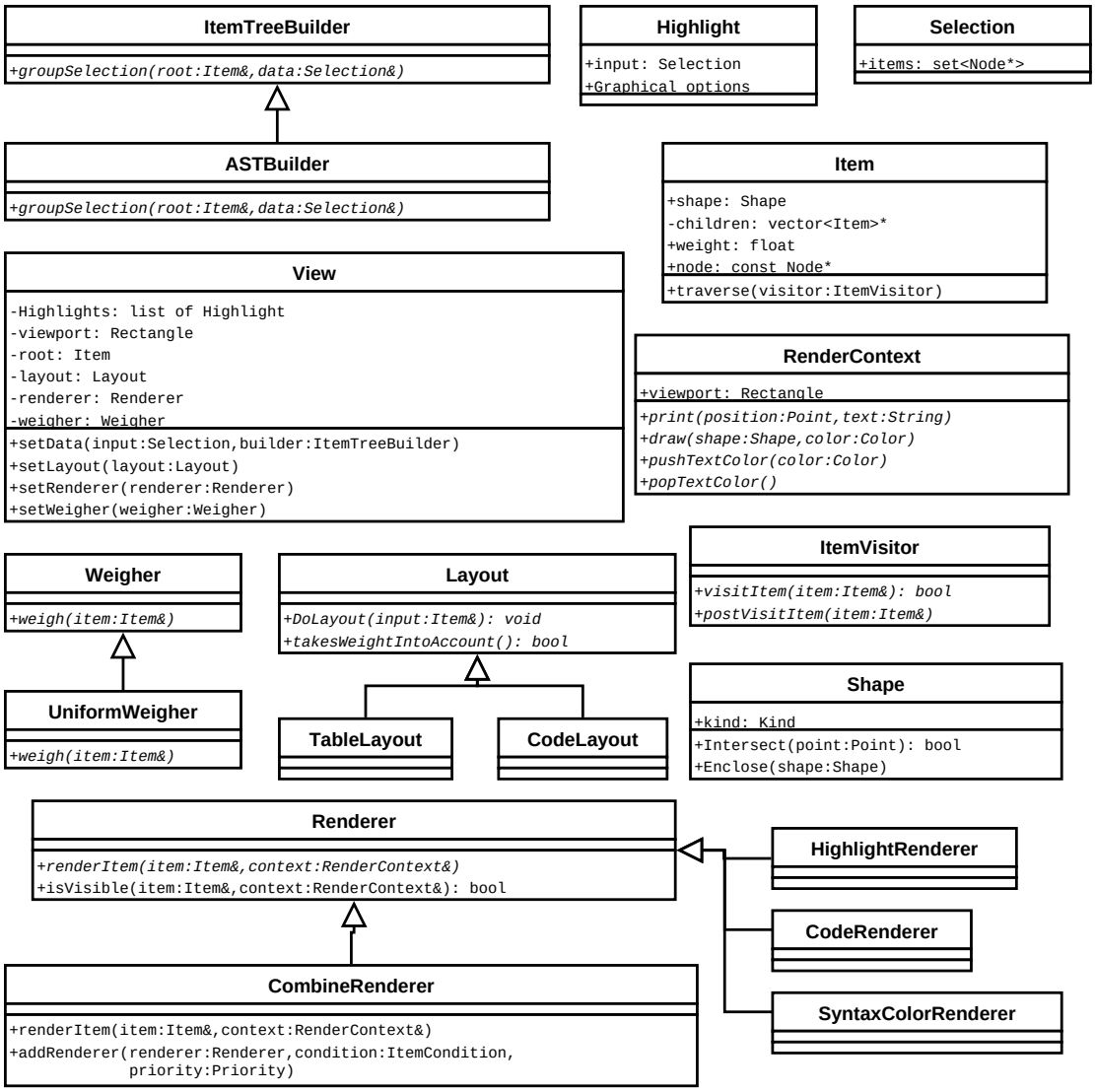


Figure 5.8: Visualization framework class diagram

```
if (lineshift == 1)
  tokenNode->row = row + 1;
else if (lineshift > 1)
  tokenNode->row = row + 2;
else
  tokenNode->row = row;
```

Figure 5.9: Generic shape of source code

could integrate so as to produce a single combined result. The view does not directly support this but a special renderer does. This approach breaks the complexity into two separate components and divides concerns properly. The same tactic can be applied to weighing and lay out but this has not been implemented as of yet. The CombineRenderer facilitates the combining of multiple renderers. Renderers are associated with a priority, within the same priority class renderers are applied in the order in which they got added to the combining renderer. The rendering order of items is a combined pre and post order traversal of the item tree. In the pre order traversal renderers can set state on the render context which can influence subsequent render operations on child items. The post order traversal can reverse the state change so as to restore render state for use by following render operations on sibling item subtrees. To facilitate these state changes, the render context provides push and pop operations on attributes like current text color.

The design and operation of the renderer class described here is a common strategy in many visualization toolkits. For example, the Open Inventor^{TM} [86] and Java 3D^{TM} [87] graphics libraries and the TableVision [88] tabular data visualization framework use very similar designs to provide custom rendering via a tree traversal and custom renderer classes encapsulating graphical state changes.

In some scenarios we associate a real-valued metric with certain nodes. Examples of such a metric are the "bugginess" of a code fragment or the byte size of a variable. We do not store such metrics directly in the ASG. In the visualization though we do want to visualize those metrics in one way or another. Items, representing nodes, have a weight property for this purpose. A Weigher class can be written that calculates the weight for each item according to the desired metric. The weighing step is performed only on explicit demand, and is also a necessary step when new data has been introduced in the item tree, but the weighing step is independent of layout and rendering. The layout algorithm(s) performed after weighing can take item weights into account when allocating space or for determining the shape of items. Rendering algorithms can color visual elements based on item weights using some application-specific color map. For more advanced applications where two or more metrics need to be displayed in one visualization, at potentially the same level in the item tree, layout and rendering subclasses can be written that support the calculation of metrics direct on the node data for use in layout and rendering algorithms.

If a weigher is associated with the view and settings for weighing change then the item tree's weights need recalculation. The layout can depend on the item tree structure, the item weights, certain render context dimensions (such as font metrics) and the target area dimensions. If one of these properties change then the layout needs to be recalculated. In all other cases the actual visualization only involves rendering of the prepared item tree.

5.3.2 Visualizing source code

If we want to visualize the ASG as source code, we need to put the ASG in the form of an item tree. ASG nodes are formed by grammar rules in nontrivial ways and may each represent varying syntax. To simplify layout and rendering steps we add so-called *token nodes* to the item tree in such a way that a pre order traversal of the item tree visits the token nodes in reading order and so that token node items are direct child of the correct ASG node items. Figure 5.10 demonstrates a generic item tree. A couple of things are worth mentioning. One header file can be included in multiple extraction units. The header file needs to be displayed multiple times because it might have a different shape in each case caused by conditional compilation and possible different scope context. Everything in an extraction unit is grouped together as it makes for one compilation unit. ASG nodes in the ASG that do not contain tokens for the file of current context are not represented in the item tree at that point; The existence of tokens from foreign files can be indicated graphically by an include directive. Tokens that originate from macro expansion, not as a macro call parameter, are not included in the item tree, not even on the place of origin in a macro definition. We have no choice because a single macro definition can bring forth many tokens and we can not visualize multiple tokens in one location. Also those tokens could have multiple different meanings while being in the same location.

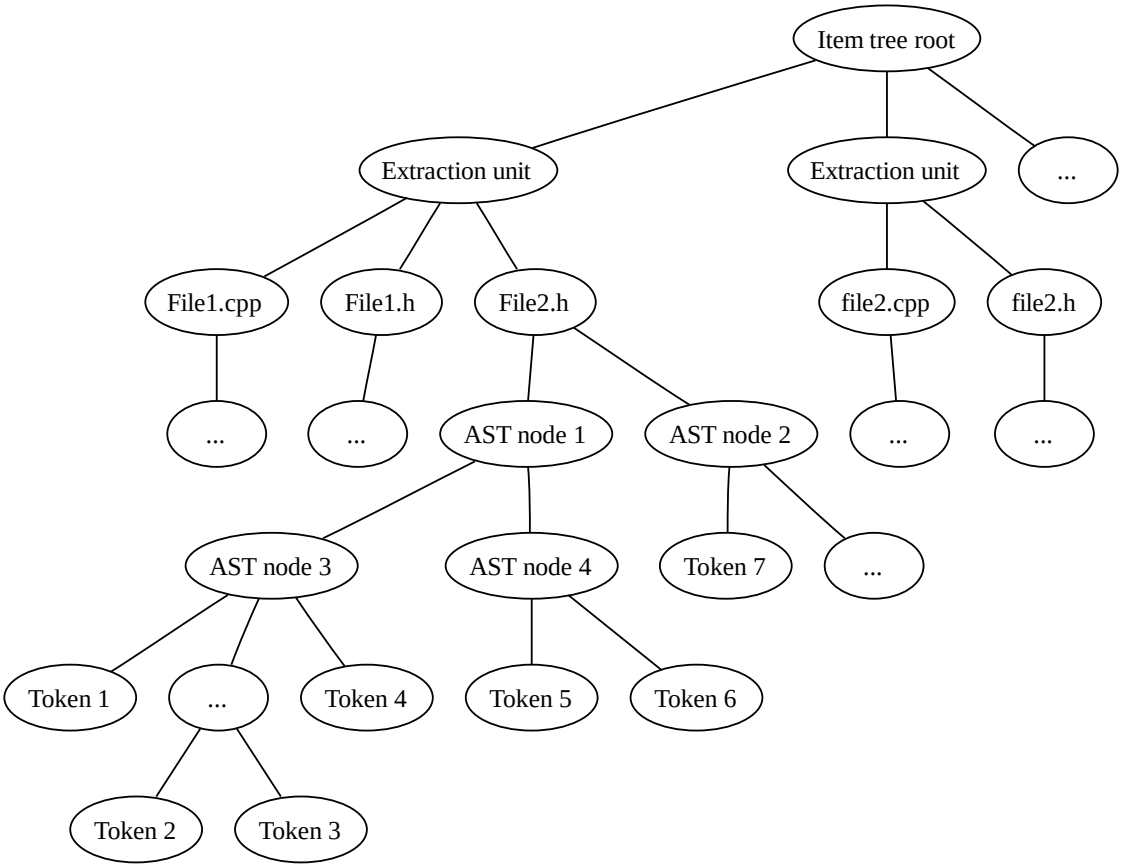


Figure 5.10: Source code item tree structure

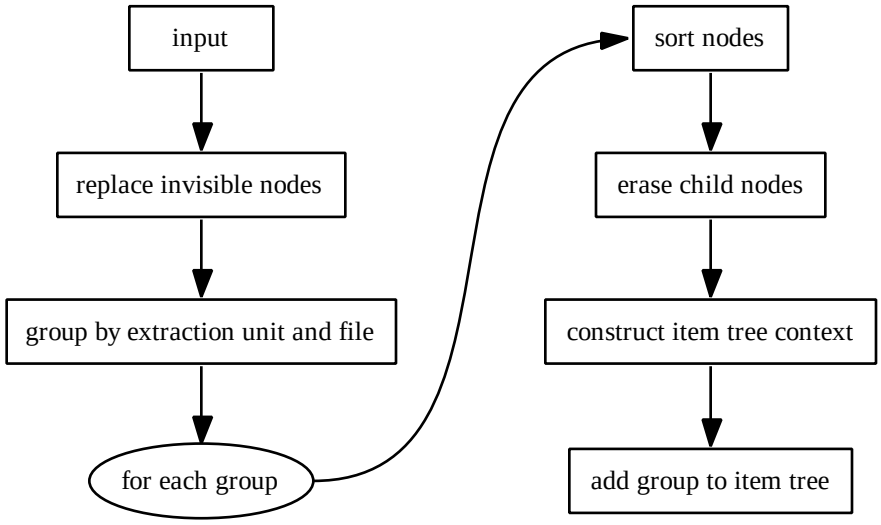


Figure 5.11: ASG item tree builder pipeline

The input of our visualization is formed by a selection. A selection, as described in , is a set of GlobalIds. A GlobalId can refer to various kinds of data but we can only visualize ASG and preprocessor nodes thus all other elements in the selection get ignored. The input, being a set, has no defined order, but we force an order on them being the order of appearance in a file. Also nodes should be grouped by file and extraction unit origin. From each smaller group of the input we can build a part of the item tree and the final result will be an item tree in the form shown in figure 5.10. The input can contain ASG nodes that are not existent in the original source. When such ASG nodes are encountered it means the user wants to visualize an invisible ASG node. There are three options on how to deal with this situation. We can either discard the ASG node, we can keep the ASG node in the selection so that any visible child elements of the ASG node will be visualized or we can search for the nearest visible parent of the ASG node so that the closest visible context of the invisible ASG node can be seen. Following steps in the visualization could be to highlight certain nodes from the input selection in some way or another. The choice on how to deal with invisible ASG nodes in the input selection influences the effect of highlighting the invisible node. We can not determine a best default choice so we leave this as an option for the visualization.

A second degenerate case is the occurrence of two ASG nodes in an input selection that have a father-child relation. ASG nodes that are visualized will have all their children visualized, on condition that they are in the same visibility context (for example both nodes origin from the same file in circumstances where only that file is visualized). Every node in the input that is a child of another node in the input and that has the same visibility context can be safely ignored. If a child ASG node is located in another file than its parent ASG node and ASG nodes are visualized per file, then the child ASG node should be grouped separately from the parent ASG node and will not be eliminated.

5.3.3 Reconstructing source code

Figure 5.11 shows “Add group to item tree” as the last step of item tree building. This step is the most complicated task of visualizing source code. What needs to be done is an inverse of the operation of the fact extractor’s parser. The parser has a number of grammar rules. A grammar rule has a certain number of terminals and nonterminals associated with it. Terminals in a grammar rule always correspond to a single token in the input. Nonterminals can correspond to zero or more tokens in the input and most of the times nonterminals translate to an abstract

syntax subtree. Each grammar rule has an action rule. The action rule typically builds up part of the ASG but that is not always the case. To support the visualization process the ASG must be rich enough to lend itself to this reverse engineering.

The tokens that form the fact extractor's input are tokens coming from the preprocessor. In most cases, the preprocessor's output does not trivially match the source file's tokens and token order. The preprocessor can manipulate the token stream on a per-token level. It can insert tokens, from nowhere (for example stringization of macro parameters) or macro definitions, it can duplicate tokens via macro and include usage and it can drop tokens (for example with conditional compilation). It even has the power to make one token out of multiple tokens with the token concatenation operator (##). This is why we need to record token locations for every single token outputted by the preprocessor. Every token is represented in the token location table and thus implicitly numbered.

ASG nodes are built up from tokens by the parser and we record what the first and last token is that makes up the text construct that resulted in the ASG node. ASG nodes that are not represented by any tokens or by imaginary tokens (for example generated by ASG lowering, sec. 5.3) are assigned a special token index so we can recognize in the visualization phase where we can suppress token output for these nodes.

When reconstructing the token stream we have two sources of text, being the ASG nodes and the preprocessor nodes. Preprocessor nodes result in texts that are not included in the preprocessor's output token stream. Also, while ASG nodes are always nested in a tree, preprocessor nodes have no such structure. When building the item tree we treat the input nodes in order of token appearance until all nodes are processed. For every ASG node we traverse its entire subtree, preprocessor nodes are handled in a parallel fashion. We have named this process *weaving*. Listing 5.1 shows the weaving algorithm in pseudo code in abbreviated form.

The algorithm described here is not complete. Every different ASG node and preprocessor node has to be treated separately and not all are as straightforward as the two kinds demonstrated. Declarations can have a number of flags. For example a variable can be declared "extern unsigned char". Such flags are stored as flags in the declaration ASG node. The C++ grammar allows the flags to appear in any order. The weaver can not always know in which order the tokens should stand because sometimes multiple orders are possible. The weaver does check for availability of space by checking the token location list and comes up with one of the possible orders that fits.

Macro calls and defines can contain tokens that have the potential to become part of the ASG. There is also a chance that this happens multiple times and with different meanings. Because of the nature of these tokens, we have no choice but to treat these tokens as plain text without any meaning, other than that they are context of a macro call or definition. Hence we suppress tokens always when they originate from macro calls. But macro calls can also be placed inside macro calls and definitions and we need to suppress those as well for the same reasons and this requires some extra checks in the `weavePreproNode` procedure.

The first condition in the addToken function highlights why we record token locations in such great detail. The condition to not introduce a token if the token was not actually there in the first place, enables us to deal with all the problems the preprocessor could cause to the weaving process. As said before, the type checker can cause problems as well by introducing ASG nodes after parsing. We record in the visibility stack if the currently treated ASG node origins from a lowering operation.

The final steps in visualizing the source code are to layout and render the item tree. First we will treat the layout algorithm. The only two layout parameters that we need is the font dimensions and the item tree itself. As demonstrated in figure 5.10 we have several layers in the item tree. There is the trunk of the item tree formed by contextual items. There are the token node items that form the leaves in the tree. There is a smaller layer of preprocessor node items that have only token node items and other preprocessor node items as possible children. Everything inbetween is taken up by the ASG node items. The entire layout is steered by the location information kept in the leave items. Their layout conditions flow upwards to the root which will have its size set to

```
global variables:
  currentPreproNode = prepro node list iterator
  itemTreeContext = stack of item pointers
  currentTokenIndex = index type
  visibilityStack = stack of booleans

procedure weave(contextItem: Item,
  astNodes: list of ast nodes,
  preproNodes: list of preprocessor nodes)
begin
  itemTreeContext.push(contextItem)
  sort(astNodes)
  sort(preproNodes)
  currentPreproNode := first(preproNodes)
  for each astNode in astNodes
    currentTokenIndex := astNode.leftSideTokenIndex
    weaveASTNode(astNode)
  while currentPreproNode < last(preproNodes)
    weavePreproNode(currentPreproNode)
    currentPreproNode := next(currentPreproNode)
end

procedure addToken(token)
begin
  if location at currentTokenIndex is real then
    while currentPreproNode < last(preproNodes)
        and currentPreproNode comes before currentTokenIndex
      weavePreproNode(currentPreproNode)
      currentPreproNode := next(currentPreproNode)
    if visibilityStack.top is true then
      create token node from token
      set token node location using currentTokenIndex
      add token node to itemTreeContext.top

  advance currentTokenIndex
end

procedure weaveASTNode(astNode: AST node)
begin
  add astNode to item tree and push on itemTreeContext
  visibilityStack.push(astNode is visible)
  switch on kind of astNode
  case ...
  case Statement_While
    addToken(TOKEN_while)
    addToken(TOKEN_LeftParenthesis)
    weaveASTNode(astNode.condition)
    addToken(TOKEN_RightParenthesis)
    weaveASTNode(astNode.body)
  case ...
  ...
  end
  visibilityStack.pop
  remove astNode item from itemTreeContext
end
```

Listing 5.1: Weaving algorithm

```
procedure weavePreproNode(preproNode)
begin
  switch on kind of preproNode
  case ...
  case define
    set defineItem = new item for preproNode
    itemTreeContext.top.addChild defineItem
    defineItem.addChild(text node with define text)
  case ...
  ...
  end
end
```

Listing 5.2: Weaving algorithm (continued)

fit the entire visualization.

Token nodes have row and column attributes. This information is set in the item tree builder for the purpose that it does not need to be (complexly) recalculated during lay out. This location information however is only used as relative position information between items. The layout can determine the order in which files appear and it can allocate extra screen space for example for headings like file name. In short, the layout determines what the rows and columns are. Each token node item is associated with a single token node and has no children. A token node, by definition, runs over a single line, thus, without any line breaks. Therefore a token node item is always assigned a rectangular shape. As demonstrated earlier with the help of figure 5.9, ASG nodes have generically a code shape and as such the ASG node items are assigned a code shape. The code shapes are always made the smallest size that fits all the children items which can have either a code shape or a rectangular shape. Preprocessor node items are assigned code shapes as well due to certain cases where a rectangle would not fit properly. An example of such a case is the C style comment. At the top level of the item tree are the contextual nodes describing files, extraction units and the visualization in its entirety. They are all assigned rectangular shapes.

At this point we have described the item tree builder and the layout algorithm. We do not use item weights so we do not introduce a weigher. We have an item tree with complete shape information available, and now we can do the actual drawing. The token node items are the only items corresponding to the original files. Visualizing these in a standard font is sufficient when one wants just a source code view. We have many more options and we will discuss some of them here.

The source code visualization makes use of the combine renderer. The basis for our visualization is a renderer that renders token node items. The text color is taken from the context stored in the render context. This token renderer is added to the combine renderer with a very late priority as the text should probably be written on top of any other visual elements. Also the token renderer is set to render only for token node items.

Since we have both token nodes as well as ASG nodes in the item tree we have the option to display lexical *and* syntactical information, both at the same time. The lexical renderer can set a state in the render context. For example it could set the font to a bold face (of course being constraint by the constant font dimensions) for certain token nodes. The lexical renderer should be given an early render priority as it needs to set state for the token renderer. Also it can be bound to render only for token nodes. The lexical highlight can be switched on and off in the visualization either by means of a flag on the lexical renderer, or by removing and adding it back to the combine renderer (of which the second option provides a more efficient rendering operation).

Syntactical highlighting can be done in various ways. One example is to draw code shape figures as a background to the token node item's visualized texts. In our program we choose to highlight syntax via text color, because we want to reserve background drawing for the so called selection renderer which will be discussed later. The syntax renderer sets the text color,

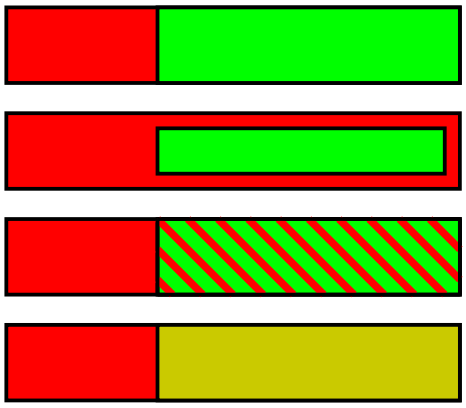


Figure 5.12: Options for the combining of highlights

an attribute of the render context. The syntax renderer can be given any priority because the renderer will always be executed before (and after in post rendering) the renderers operating on child items. It needs to be only associated with ASG node items. In a similar way, preprocessor nodes can be highlighted.

Syntactical highlighting has been achieved in the past. The Visual Code Navigator, by Lommerse *et al* [47], highlights C++ source code syntax by means of colored and shaded cushions. In comparison to their work, we generalize in the sense that we can customize the highlighting in any way we want by user-specified combinations of renderers which are associated with user-specified selections of elements in the source code. Another generalization over their work that is not directly visible, is the application of a single data structure containing both syntax and text, whereas the Visual Code Navigator employs a separate data structure for syntax and one for text. This allows our program to lay out text any way we want and to take excerpts from source code matching a syntactical context (e.g. a switch statement) and display them on their own.

In combination with the powerful query system, the selection renderer is a potent component of our source code visualization. Queries yield a selection, say s. The selection renderer is executed for a certain set of nodes $s \cap v$ where v is the set of nodes referred to in the item tree. The test is set as a condition on the renderer in the combine renderer. The priority of the selection renderer is set in such a way that it will always execute before the token renderer but after possible state changing renderers, even though there are no means for including token nodes in the typical input selection. The selection renderer draws a background, which we will call the highlight, for each node included in the selection. Each selection renderer has a color associated with it so multiple selections can be visualized at the same time. A problem occurs when a part of the ASG needs to be highlighted with two colors. An ASG node can be in one selection (for example a class definition) and a child ASG node can be in another selection (for example a class member declaration). One ASG node can also occur in two selections simultaneously. In these cases, one highlight can overlap another highlight, sometimes completely occluding the underlying highlight. Say we have drawn two highlights, highlight A and highlight B. If B overlaps A, but A is still shown before and after the occurrence of B then the full size of both A and B will be obvious to the observer, even when A and B are drawn in an ordinary way. When B starts, or ends at the same position as A then the observer can not easily see the full size of A. Even though the full size of A can be derived from text, the observer should not need to look at the text at all to observe the highlights in the correct way. In cases where B completely conceals A then obviously the observer does not have a proper view of the highlights either. Figure 5.12 demonstrates the situation where B, drawn with a green color, overlaps A, drawn with a red color.

In figure 5.12, the first picture shows the situation without a solution. The second picture shows a solution where the shape for inner highlight B is offset inwards to preserve screen space to display the surrounding highlight A. The third picture demonstrates drawing overlapping highlights with

```
m_camera->setFarPlane(50000.0f);
float distance = 300.0f + 260.0f * cosf(sinf(theta)*1.534f);
theta = 0.5f * Math::Pi;
const float yPos = 90.0f;
```

Figure 5.13: Combining highlights with alpha channel blending

patterns, that will show the two highlights' colors. The fourth picture demonstrates blending as a way to indicate two highlights being drawn on one area. Each solution has its drawbacks. All these drawbacks occur in more complex visualizations where more than two highlights overlap. The first two solutions can cause problems for readability of text, because high contrast between colors cause the text to loose contrast with its background. In the second solution it is possible to define alpha channel masks that create a pattern and reduce contrast, which can be seen as a combination of solution two and three. Solution three requires skill from the observer to see what the original colors for the blending operation have been which can be especially difficult when dealing with three or more highlights. Solution one would work better if we allow for extra spacing between lines and letters so that all highlights still enclose the full text. The result would be that the high contrast borders between highlights do not run under text leaving the text still well readable. This does, however, also result in less LOC on a still visualization. We have implemented solution two with alpha channel masks which we demonstrate in figure 5.13.

The source code visualization allows zooming out to a level where one line of text is displayed on a single line of pixels. The zooming is performed in terms of scaling the font. By scaling the font the font dimensions change. When the item tree gets laid out again the resulting dimensions will be smaller as well. At the point where lines are a single pixel in height some problems occur. When the selection renderer draws a background for a certain piece of text it might be that the text completely occludes the background. The case where this is likely to happen is when the background is drawn for one or two lines of text. For this case it is possible to have the selection renderer change the font color attribute, besides drawing the background, so the selection will strike out better. The syntax renderer can still intervene by overriding the font color in some part of the subtree rendering and therefore we provide the option to disable syntax highlighting with a single switch.

When the source code visualization is zoomed out a lot, the column of text that is the source code becomes quite narrow. This leaves a lot of screen space unused. To utilize this screen space the visualization supports column wrapping. Columns can be presented as fixed width, fitting the widest line of text present in the source code, or fixed width at a certain limited width clipping potential long lines. The columns can also be presented as variable width meaning that the width of a column on the screen is as wide as the widest line visible in that column. Figure 5.14 shows a visualization with maximal width columns and a minimal zoom level. The source code runs from top left to bottom right, each column starts where the column left to it has ended. The text in the visualization is displayed as single pixel high lines and are colored according to a certain color map.

5.3.4 Interaction

Interacting with the visualization allows the visualization to come to life. In our case all interaction is done with the mouse. Coordinates in the visualization correspond to an item in the item tree. Depending on the location of the mouse, feedback is given to the user. The visualization can be clicked on. Depending on where is clicked a pop up menu is shown informing the user of the available options. In day to day usage of the program the user knows which items need to be clicked but items do not correspond to text in a one on one relationship. The text is always represented by token nodes and they are almost never subject to interaction. Therefore the visualization needs to be improved to better support for interaction.

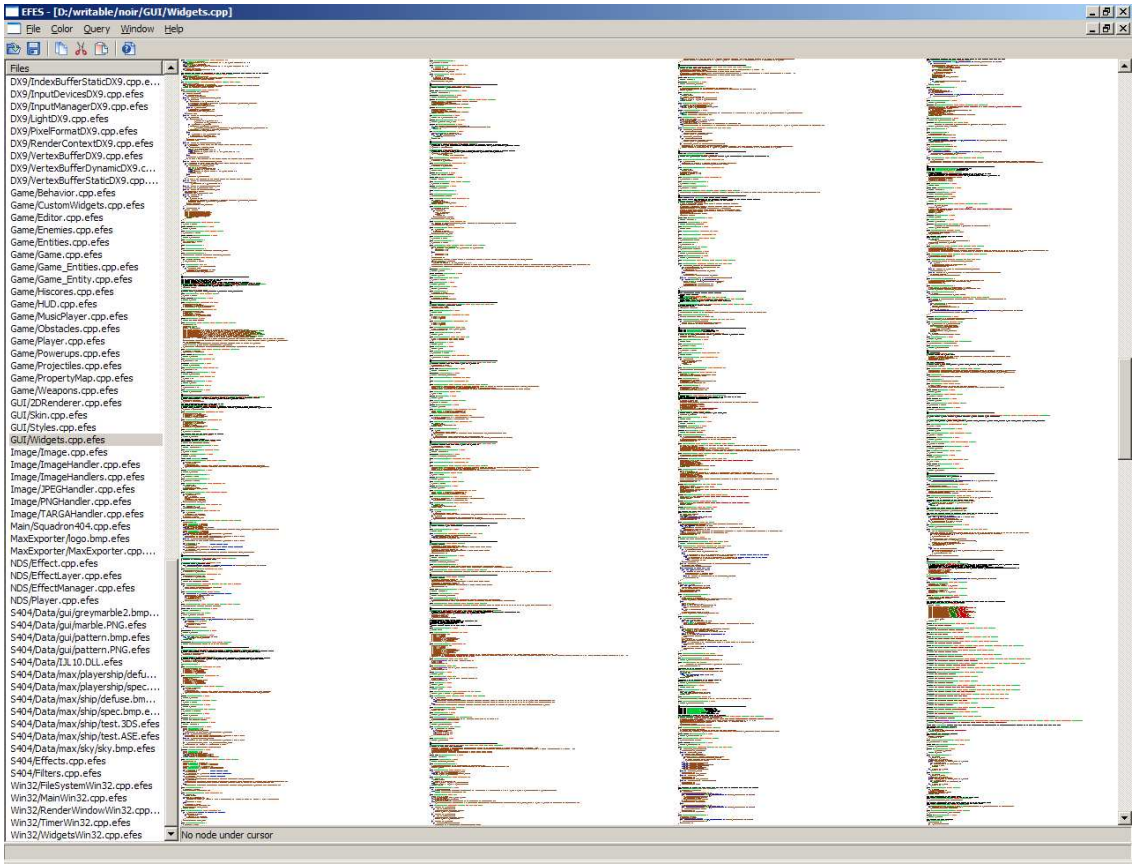


Figure 5.14: Example source code visualization

To enable the easy selection of an item we can provide the user with visual handles for items. These interactive elements will then be visualized as an overlay on top of the item tree visualization. Drawing the handles for every item produces a lot of clutter. We can reduce the number of handles by showing only those that are in the vicinity of the mouse cursor. When the user moves the mouse away from the visualization all handles will disappear yielding the original visualization. When handles are drawn they should occlude as little text as possible. Also it is better if they occlude text from other constructs that are likely not in the user's direct interest, rather than their own. Handles must never occlude other handles because that would render the whole concept of the handles useless. Since handles are visual aids, they can show the kind of construct which they represent by shape, color or animation.

One solution is to place handles at the front of items along the top side. Positioning the handles in this way can lead to handle occlusion. Handles that would occlude other handles on their natural position will be positioned at the right of the handle they would otherwise occlude. Applying this step recursively can lead to a number of handles being displayed in a row which the user can pick from without problems. Without handles, certain syntactical constructs would be completely occluded by others with the result that that construct can not be selected. Figure 5.15 demonstrates the handles' visual appearance. The handle are colored according to the color of the represented syntax as set in the global syntax coloring options (used by the syntax renderer). Because handles drawn like this have the potential to occlude text we define vicinity as follows: Given an item i being the deepest item in the item tree intersecting the mouse cursor coordinate, all items in the item tree on the path from the item tree's root to i, that have an ASG node associated with them, have their handles drawn. By selecting items in this way there is no syntactic construct that draws a handle on top of the text of the line that is intersecting the mouse cursor coordinate. Thus if a line of text is, to the user's taste, interesting and occluded too much by handles, the user can move the mouse over to that line and thereby removing the offensive handles. Advantageous to this approach is the direct visual feedback available to the user. A disadvantage is that the handles need to be sufficiently large in size so that the user can easily move the mouse cursor onto them. By being as large as in the demonstration they become easily distracting. Quite likely the shape and color can be tuned to a better style but we have not pursued that option.

Instead we have chosen an alternative less visual approach. The user gets visual feedback on the deepest located item that is the context of the mouse cursor. The feedback is given by drawing that item with a distinct background color in a fashion similar to the selection renderer. When the user clicks inside the visualization a menu will pop up showing several options for the deepest item in the item tree that is context of the mouse click. All but one option are related to that item. The other option is a gateway to the pop up menu belonging to the item's parent. This strategy can be applied recursively up to the item tree root. This method allows for interaction with syntactic elements (represented by items) that do not get any screen space, as demonstrated in figure 5.16, which is one of the goals for any method of interaction with this visualization. Advantages of this approach are: Options for items on different depth levels can be browsed fast, there is no need to calculate and redraw the screen on mouse moves and there is no obstruction of the text as long as there is no pop up menu.

The result of interaction can be delivered in exactly the same way as the results of queries. In fact, the interaction operations are implemented in terms of the query system. The results of a query can be presented in various ways. We can show results as highlights which is appropriate for queries that return a number of items. An example is a query that returns all free functions. We can also show the result by zooming and scrolling towards the result in the visualization. Condition for this feedback however, is that the result is but one node. Still, if a query results in a list of nodes, one could still want to zoom in on a particular node in that list. That is why we find it practical to present the results of a query in a list besides any other kinds of feedback. This list then contains a textual description of each node that helps the user identify the node. Interacting with this list allows the user to zoom in on a particular node in that list or to give a chosen node a particular highlight in the visualization.

The query result list can also prove useful in the following scenario. Given a certain extraction unit consisting of a single file. The file has two functions of identical type and name. In normal

```
    // apply the pointer type constructor
    if (!dt.type->isError()) {
      dt.type = env.tfac.syntaxPointerType(tokenIndex, cv, dt.type, this);
    }
  }

  // recurse
  base->tcheck(env, dt);
}

// almost identical to D_pointer ....
void D_reference::tcheck(Env &env, Declarator::Tcheck &dt)
{
  env.setLoc(tokenIndex);
  possiblyConsumeFunctionType(env, dt);

  if (dt.type->isReference()) {
    env.error("cannot create a reference to a reference");
  }
  else {
    // apply the reference type constructor
    if (!dt.type->isError()) {
      dt.type = env.tfac.syntaxReferenceType(tokenIndex, dt.type, this);
    }
  }

  // recurse
  base->tcheck(env, dt);
}
```

Figure 5.15: Handles shown for items in context

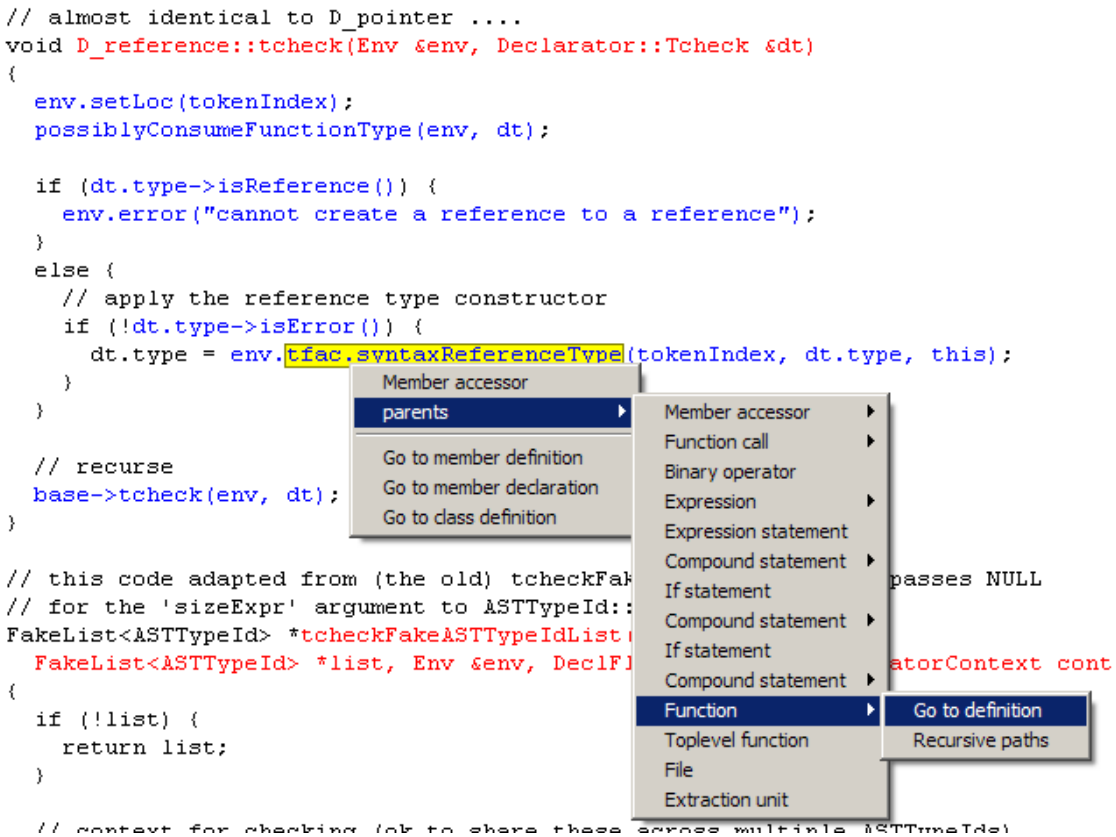


Figure 5.16: The pop up menu for a certain item

circumstances this form an ambiguity that a compiler can not accept. Our fact extractor can accept this input. Suppose a user interacts with the program in the following way, the user clicks on a function call which brings up a menu. One of the choices in the menu is to go to the called function's definition. The user makes that choice but there are multiple definitions. The result of the query is two nodes which are both displayed in the query result list and the visualization zooms in on the first node in the list. The user can still choose to zoom in on the second node allowing for complete convenient access to all the available information.

Chapter 6

Applications and evaluation

6.1 Benchmarking of fact extraction

We now discuss in detail the capabilities of our fact extractor. First, we present how EFES behaved when tested against the CppETS C++ fact extractor benchmark [63] (Sec. 6.2). Next, we present how EFES measures in terms of performance with Columbus, the major competitor we identified, and explain the performance differences (Sec. 6.3). Finally, we discuss several technical design points of EFES (Sec. 7.2).

6.2 CppETS Benchmark

CppETS was proposed by Sim *et al.* [63] as a benchmark to compare C++ fact extractors. CppETS characterizes fact extractors along the dimensions of robustness and accuracy. Robustness is roughly equivalent to our own fault tolerance (Sec. 2.2.1). Accuracy maps to a combination of our correctness, full coverage, and compliance requirements. CppETS consists of several so-called test buckets, 25 in total. A test bucket consists of a small C/C++ code fragment and a set of questions to be answered on that fragment using the fact extractor under scrutiny. CppETS was used to compare the following extractors: CPPX, Columbus, Ccia (a commercial tool from AT&T), the Rigi C++ parser (built atop of IBM's VisualAge commercial tool), Src2srcML [17] and TkSee/SN citecppets.

We applied the CppETS 1.0 benchmark (the only one publicly available for download) on EFES as follows. For every test bucket, we first constructed the required queries using the query API which was introduced in Sec. 3.7. The queries were very simple to implement using our API, taking a few tens of lines of C++ which instantiate a few of our query API classes. Next, we ran the extractor on the test bucket, applied the constructed queries on the resulting database, displayed the query output as text, and validated it manually by looking at the test bucket code. Table 6.1 details the effort it took to answer several of the CppETS queries, showing the test bucket name, as in the benchmark (column 1), the question number, as in the benchmark (column 2), the total amount of lines of C++ code we had to write to construct the query, the number of query API class instances used to construct the query (column 3), and the total time it took from reading the question to answering it (column 4).

Our experiment described above proved that EFES was able to easily and successfully pass the CppETS tests. Yet, passing these tests successfully does not guarantee a fact extractor can be used to analyze industrial projects of millions of LOC. There are several respects in which the CppETS benchmark is limited, as follows. First, no test questions target highly ambiguous C++ code, which is exactly where most C++ extractors have serious problems. Such code can parse correctly only if the extractor correctly implements the C++ lookup rules, i.e. maintains a correct symbol table. One such example are the complex C++ name lookups in presence of multiple virtual inheritance. Second, most tests target only one one language feature at a time, while

Test bucket	Question	Query LOC	Query API classes	Total time (min)
Syntax/Enum	2	7	4	5
Syntax/Enum	3	10	5	1
Syntax/Exceptions	1	8	5	5
Syntax/Inherit	3	6	4	2
Syntax/Namespace	1	2	3	1
Syntax/Namespace	2	9	5	5
Syntax/Operator	1	10	5	5
Syntax/Struct	1	35	13	7
Syntax/Union	1	7	5	5
Prepro/1	1	3	3	1
Prepro/1	2	3	3	1
Prepro/1	3	5	4	2
Prepro/Pragma	5	5	4	1
Missing	2	8	6	2
Missing	3	3	3	1
Missing	4	10	8	3

Table 6.1: Statistics of EFES on the CppETS benchmark

real-life code combines several such features in complex ways, e.g. member templates which are part of a template class, or a template class that is derived from one of the template arguments, to give just two examples. Such constructs massively complicate parsing and few extractors can handle them properly. Third, robustness in parsing C++ dialects is tested only at lexical level, e.g. whether Visual C++'s `__cdecl` or GNU g++'s `__attribute__` keywords are understood. Robustness against dialect issues above the lexical level, e.g. the peculiar `for` scoping or `static const` initializers of Visual C++ 6.0, is not tested. All in all, many extractors can successfully pass the CppETS benchmark, yet perform erroneously or not robustly on large real-life projects. In particular, CppETS does not test the scalability of an extractor. Its test buckets contain only very small code fragments, on average 50 LOC, the largest C++ and C fragments being 800 LOC and 160 LOC respectively. In total, all CppETS test buckets span under 5000 LOC. Hence, we designed a separate set of tests to test the scalability of EFES. These are described next.

6.3 Scalability Benchmarking

Besides the CppETS benchmark described previously, we created several benchmarks to assess the performance of EFES. We ran EFES on a 2.8 GHz PC computer with 1024 MB RAM. As benchmarks, we used the source code of several popular applications and libraries, focusing on very large projects of hundreds of files and millions of LOC. To compare the extraction performance of C versus C++ input, we considered both types of code bases. We discuss next a selection of our test suite.

For C fact extraction, we used the Quake 3 Arena [36], Python [59], and Blender [14] projects. Compared to C++, many C programs make less use of standard libraries. Moreover, the C programming language is relatively small and easy to parse. On the other hand, many C programs use extensively many preprocessor features.

For C++ fact extraction, we considered the VTK [39], Coin3D [68], wxWidgets [64] and EFES projects. VTK and Coin3D are two well-known, high-level libraries for interactive graphics and data visualization. Both are written in a relatively portable subset of C++ and make little use of templates, STL, exceptions and other more recent C++ features. wxWidgets is a popular cross-platform library for user interface construction. It is interesting as it is structured to benefit from compilers with support for precompiled headers. EFES, our own fact extractor, uses STL extensively and also other more advanced C++ features, e.g. exceptions. We included it as it allows us to make some observations about the costs of analyzing STL code.

Project	Files	MLOC	Size (MB)	language	PCHs	STL	Missing includes
Quake	93	0.7	5.8	C	-	✓	-
Python	294	1.4	36.8	C	-	-	-
EFES	280	2.5	46.8	C++	-	✓	-
Blender	200	6.7	14.2	C	-	-	-
Coin3D	1084	27.4	184.4	C++	-	-	-
VTK Common	168	33.5		C++	-	✓	-
VTK(no win32)	1006	5.2	83.6	C++	-	✓	✓
VTK	1006	155.2	812.7	C++	-	✓	-
wxWidgets	152	36.7	261.2	C++	✓	-	-
Mozilla	5791	205.4	987.8	C/C++	-	-	✓

Table 6.2: Projects used for benchmarking

Finally, we consider the popular browser namely Mozilla, which is a hybrid C and C++ project [51]. Mozilla is also the largest project in our test suite, spanning 2.5 MLOC without the headers and over 200 MLOC including headers, so it is an excellent test case to assess the robustness of our tool for industry-size code.

The selected code bases are a good mix of complete applications, e.g. Mozilla, Quake 3, Blender, and EFES, and libraries, e.g. Coin3D, VTK, Python, and wxWidgets. Libraries are usually written in a much more consistent style and have a relatively simple architecture. On the contrary, applications generally consist of parts having quite different code structure and style, ranging from small unit tests to entire subsystems with complicated relations. Both types of software test our fact extractor in different ways.

Table 6.2 gives an overview of the projects used for benchmarking, i.e. project size (counting included files), language used, whether precompiled headers (PCHs) were available to the Columbus fact extractor, and usage of STL. For all projects, except Mozilla, all include files were available. For Mozilla, some include files used by some of its test applications were not available. Lines of code are counted before macro expansion. The total size shown in the table reflects the size of all input, comments excluded, after preprocessing with a Visual C++ 7 profile. For VTK, we provide separate results of its Common (core) subsystem, since this is the most complex, dense C++, part of the VTK code base.

The results of the Columbus fact extractor are with filtering the contents of standard system headers. This is achieved by not including standard headers in the Columbus project file, which filters the respective information from the output. Therefore, the results of Columbus should be compared with the results of EFES shown in Table 6.4.

Table 6.3 shows the fact extraction performance, i.e. total extraction time and output file size, for the Columbus tool, except for Mozilla. When we analyzed Mozilla, the final linking step of Columbus failed with a crash and no output was produced. We are not sure why this happened, as several publications report the successful analysis of Mozilla with Columbus [23, 44]. One outlier in this benchmark is Python, which takes over three hours to parse with Columbus, although its size is roughly double that of Quake, which gets parsed in under four minutes. We have controlled this result on several machines. The slowdown is caused by some Python header files which are often included and which causes the parser to get 'off track' and generate a large amount of error messages. This proves again that exceptional situations encountered by a parser can have a huge impact on performance.

Tables 6.4 and 6.5 show the performance of EFES with the Visual C++ 7.0 profile and filtering of system header information enabled and disabled, respectively. We see that filtering roughly doubles the performance. The throughput metric in Table 6.5 shows the input processed per second by our tool. The variance in the results between different projects is caused by different code density and code complexity, and by the effectiveness of our filtering method. The line count that we perform is a bit different from a straightforward lines-in-a-file counter. A translation unit is basically a list of topforms. Each syntactic construct has location information (Sec. 3.4),

Project	Parse time	Output (MB)
Quake	3m38s	5.8
Python	3h00m10s	1.5
EFES	19m09s	46.8
Blender	8m21s	2.1
Coin3D	1h00m30s	183.4
VTK	2h00m06s	812
wxWidgets (PCH)	41m46s	6
wxWidgets	2h00m24s	261
Mozilla	7h44m00s	987

Table 6.3: Performance of the COLUMBUS fact extractor

from which we can derive the number of lines it occupies. For a file, we produce two line counts: The sum of line counts of all its topforms before, respectively after output filtering. We use this to calculate the effectiveness of our filtering method, which is shown in the table. The average number of analyzed lines per second gives a good indication of how complicated or dense the code is, a metric which can vary largely from project to project. Since we count lines before macro expansion, a relatively lower performance is reported for code which uses many macros which expand to large fragments. An example hereof is the ELSA parser, part of EFES, which has very dense code and uses many complicated macros. We can see that it takes about three times longer to parse a single line of code from the ELSA parser than from any other C++ project considered here.

Another obvious fact from Tables 6.3 and 6.4 is that COLUMBUS, although slower in parsing, produces a much more compact output than EFES. This is a clear indication that out output format can be improved to deliver more compact storage and potentially an even higher speed increase.

EFES writes its output in such a way that the original ASG can be restored, thus fully addressing the completeness requirement. This includes complete location information: begin row/column and end row/column are correctly specified for every node. We used this feature to test the correctness and completeness requirement. We use the extracted ASG to reconstruct a textual version of the input source code and compare it with the actual input via a `diff` operation. The comparison result was positive, even on complex files, which makes us trust the correctness of our EFES extractor in general.

We also write basic type information that cannot be directly deduced from the ASG. Overall, this is at least as much type information as COLUMBUS [18] reports. EFES correctly reports includes, conditionals, defines and macro applications, which explains why its output and running time are relatively larger for code bases using many macros, e.g. EFES which uses the ELSA parser (see Table 6.4).

In most projects, the preserved lines after filtering amounts to only a fraction of the total number of lines. Therefore, output reduces to a fraction of the original output size when filtering is enabled (compare Tables 6.4 and 6.5). This is not surprising, as in the considered projects by far the most lines of code come from standard headers. In particular, the STL headers are quite large, as they contain both interface and implementation of template classes. Hence, the particular STL implementation specified in the compiler profile (Sec. 3.3) is an important factor for performance. For example, we saw that switching from Microsoft's Visual C++ 8 to Visual C++ 7 headers immediately resulted in an extraction performance boost of more than 10% for the VTK Common code base.

Some projects we tested, such as VTK and wxWidgets, contain several files which include the platform-specific `windows.h` header and its include subtree. This is a big performance hit, as this subtree spans almost 200 KLOC. In this case, the filtered output can be less than a percent of the total parsed lines. For example, processing the VTK code base using a profile without the Windows headers takes less than 24 minutes, whereas including information from these headers

Project	Lines/sec	Filtered (%)	Time/file (msec)	Output (MB)	Parse time	Throughput (MB/sec)
Quake	21119	32	0.3	69	30s	0.20
Python	30344	40	0.8	452	3m42s	0.17
EFES	9033	47	1.0	455	4m05s	0.19
Blender	19747	37	0.6	180	1m10s	0.20
Coin3D	36607	10	0.7	1300	12m27s	0.24
VTK Common	73960	1	2.7	176	7m33s	0.13
VTK(no win32)	3510	63	1.4	1250	23m53s	0.06
VTK	54563	2	2.8	1300	47m23s	0.29
wxWidgets	23636	12	10.2	4000	25m52s	0.16
Mozilla	205409	3	1.3	442	2h09m00s	0.12

Table 6.4: Performance of EFES with system headers filtering and the Visual C++ 7.0 profile

in the output brings parsing time to over 47 minutes (Table 6.4), with output filtering, and over 2 hours without filtering (Table 6.5). We noticed that many files in the VTK Common code base do not even use the Windows API, so it becomes interesting to run EFES on such types of code bases without a Windows profile, if we want to maximize performance. A similar strategy of skipping several include files, also for maximizing performance, is implemented by the SNIFF+ tool [13].

Conversely, for large projects that have a set of large headers included in (almost) every source file, it becomes interesting to use *precompiled headers* to improve extraction times. A given header is processed only once, instead of once at every inclusion point, and the extracted facts are stored in a precompiled header (PCH) file, thereby improving performance. A header is suitable for precompilation only if all macros defined before its inclusion have either no effect on the precompiled header or are always defined identically, i.e. its semantics are independent of the inclusion context. Currently, EFES has no support for precompiled headers. Fortunately, our extractor is fast enough, as demonstrated by the above results, to still handle projects using large headers in very short time. Adding this support, as well as the related support of incrementally parsing files when changed, is quite straightforward, and should bring the performance figures of tables 6.4 and 6.5 closer to each other.

EFES is very efficient on C code. The number of lines processed per second is sometimes relatively low, e.g. for the Quake and Python projects (Table 6.4), since we count them before macro expansion, as described previously, and these C projects use macros extensively. Moreover, many of the C projects we tested seem to depend to a smaller extent on library headers than C++ projects. Hence, in these cases a larger percentage of lines is retained in the output after filtering. This again reduces the number of lines processed. Finally, we note that most C standard headers are relatively small in size and quick to parse, as compared to C++. For example, for the Python project about 50% of the lines were removed by filtering standard header information. In contrast, in C++ code we noticed that over 90% of the code lines usually come from header files.

Table 6.5 shows that, without output filtering, output sizes and timings are roughly doubled. However, these results show also that the performance of our fact extractor is still very good, surpassing COLUMBUS, even when all information is preserved. In some cases, e.g. the Python code base, the difference can be hugely in our favor, i.e. 3 minutes vs. 3 hours. The only case when COLUMBUS was just marginally faster than our extractor was for the wxWidgets code base, i.e. 42 minutes versus 48 minutes needed by EFES, due to an extensive use of very large precompiled headers which, as explained earlier, improve performance.

Project	Time/file (sec)	Output(MB)	Parse time
Quake	1.3	422	2m00s
Python	1.2	1000	5m48s
EFES	0.8	770	4m30s
Blender	0.8	304	1m38s
Coin3D	1.2	4290	21m19s
VTK Common	8.1	1044	22m48s
VTK	7.3	23000	2h00m06s
VTK (no win32)	1.6	1892	26m50s
wxWidgets	19.0	11000	48m10s
Mozilla	2.4	4750	3h00m53s

Table 6.5: Performance of EFES with no filtering and the Visual C++ 7.0 profile

6.4 Reverse engineering scenarios

6.4.1 Scenario A: Porting software

Goal

A programmer downloaded the source code for a C++ parser named Operator++. Operator++ is written for the GCC compiler and is only available for the UNIX platform. The programmer works on the Windows platform, and is only familiar with the Visual Studio 6 compiler. Now his manager asks him how long it would take him to port this program to Windows. The programmer cannot immediately tell this, as he has never seen the code before. He has no impression about the size of the project and the distribution of platform dependent code. The programmer decides to use EFES to get answers.

Project setup

The programmer creates a new EFES project file for Operator++ using EFESProject, a graphical user interface for project setup. He specifies the include and source directories, and chooses the VC6 profile from a list of standard profiles. He then selects that he wants to use normal linking

Estimating project size

First, the programmer wants to investigate the size of the source code. He runs the extractor on the Operator++ project, and waits while the extractor does its job. The progress window informs him that there are 300 source files in the project. After 5 minutes, the extraction process is completed, and a log file is written to disk. The programmer opens the log file and looks at the results. The statistics window of the extractor shows the total number of filtered lines processed. This indicates the number of LOC after processing, and is in this case a few million lines. Combined with the fact that the project consists of 300 files, the programmer already knows that he is dealing with a rather large project. Another metric shows that just the source files consist of 100 KLOC. The programmer now estimates the total project size around 200 KLOC.

Finding missing headers

Now the programmer has an impression about the project size, he wants to know which part of the code is Unix specific. First, he wants to figure out which files make use of platform dependent code. He selects a query to search for selecting all missing includes, by opening the query library containing queries for finding errors. Then the programmer runs the query on the entire project. Almost instantly a number of files is highlighted, which apparently have missing includes. Most of the files occur in the *Base* module, having functionality such as timers, sockets, and data

structures, but some other files are highlighted as well. The result window indicates that exactly 200 missing includes were selected over 10 different files.

Finding missing functions

Although an overview of missing headers may give an impression about the number of files with platform dependent code, it cannot display the amount of Unix specific code in these files. Headers often contain a multitude of function declarations, and may introduce many new types. Missing a single header may result in many thousand calls to unresolved functions.

The programmer opens all files with missing headers. He uses the maximum level of zooming, such that he sees the complete source code of the file on screen. Then he highlights all unresolved functions, by running a query from a query library. The selection highlight indicates that only a few functions are missing, most of which are clustered in a single construct. The programmer zooms in at this part of the code to make the function names visible. It turns out that most function calls are executed in functions of the same class, and most calls are related to timer functionality. the programmer estimated that it would take him only a few hours to implement the same functionality using the Windows API.

Next the programmer adds another highlight, in a different color, for indicating all uses of missing types. He recognizes that most missing types are clustered around unresolved functions, with only a few exceptions. He zooms into the cluster and notices that most missing types are structs used as function arguments. Figure 6.1b shows a source code fragment with several highlighted selections.

Finding code in GCC dialect

Programmers often write code against the rules enforced by a specific compiler. The programmer writes a code fragment, which may or may not be accepted by the compiler as valid code. If the compiler accepts the code, the programmer continues to his next problem. Otherwise, he refines the code and tries to compile it again. Source code accepted by a compiler, however, may still be invalid C++. Compilers for C++ are extremely complicated tools and are hardly ever completely free of bugs. Moreover, many C++ compilers, including GCC, introduce several non-standard extensions to the language.

The programmer wants to find out which part of the code is nonstandard C++. He does this by running a query to find all places in the code where code is written in the GCC dialect. The query searches both for applications of GCC extensions, as well as patterns of nonstandard C++ accepted GNU GCC. It turns out that three files are affected, but most search results are centered in a single file.

Visualizing a call graph

Finally, the programmer wants to see a call graph of the classes in the program. He knows which classes are platform dependent, and hopes the call graph reveals where and how often these classes are actually used. He uses the call graph exporter for Call-i-Grapher to create a call graph from the entire project. He creates a hierarchical call graph consisting of two levels, viz. module, and class. He opens the call graphs with the Call-i-Grapher tool and creates a new visualization containing all classes. Apparently, the graph is still too big for visualization, since the resulting visualization unintelligible. The programmer recognizes that a large part of the graph visualization is taken up by the standard library, which he settles by removing the standard library from the visualization. Instantly, Call-i-Grapher produces a much more comprehensible graph visualization. By increasing the intensity of the bundling, the programmer can easily discern which modules refer to the base library. He discovers that all calls to platform dependent classes are done by a small number of class situated in a single module. Moreover, some non-portable classes are never actually used.

Conclusion

By analyzing the combined results of the code investigation, the programmer is able to estimate how long it takes to port this project to Windows. It took him only a few hours in total, before he could communicate to this manager that he expects porting to take about two weeks of work. This is a magnificent result. Especially considering that the programmer has never seen the code before. This course could promptly last ten times longer, if it were carried out with traditional tools.

6.4.2 Scenario B: Handing over source code

Goal

Programmer X worked on a large C++ software system for two years at a medium-sized software company. The system consists of about 200 KLOC scattered over 200 files. Due to a company restructuring programmer X is assigned to new project, and his current project is transfered to a fresh programmer.We will refer to the fresh programmer as programmer Y. The company reserved two weeks for the source code transfer. During this period programmer X is still available for questions, while programmer Y must become acquainted with the source code. The company uses Visual EFES for the transfer.

Opening the fact database

The software company runs EFES every night on the source code as an automated process. Hence, the new programmer does not have to create a new project file for EFES. The fact database containing all extraction units is publicly available on the company file server. Programmer Y opens the fact database and all the files of the source code are presented to him.

Clustering code

Programmer Y, being overwhelmed by the sheer number of source files, wants to order the files by their contents. First, he clusters files by LOC. He opens the largest file, and zooms out at maximum level. The structure of the code seems to have a very regular pattern. Programmer Y sees that the file has a small comment block at the beginning. He zooms in on the comment block which states that the file is automatically generated, based on a grammar specification.

Programmer Y figures he needs a different clustering operation. One of the queries in his library selects all loops and conditionals. He expects the number of results produced by this query a good metric for the complexity of the source code in a file.

By far the most complex files according to the cluster criterion, are the files 'TypeChecker.cpp', 'TemplateInst.cpp', and 'ScopeEnv.cpp'. Programmer Y opens the files and displays them at maximum zoom level on a single screen. He orders the functions by the amount of LOC in their function body.

Coloring complex code

Programmer Y selects a color profile highlighting loops and conditionals in a flashy orange color on a white background. The resulting visualization is shown in figure 6.1a. He notices that there are many small conditional statements and loops spread over the entire file. The source code itself seems hard to comprehend, so he hopes the code is richly documented. He changes the color assigned to comments to black, and is glad to notice that about 30 percent of the code consists of comments. The programmer zooms in on a comment block of a function, and tries to understand what it does.

(a) Color profile for highlighting complex code

(b) Visualization of multiple selections

Figure 6.1: Source code visualizations

Source code navigation

The programmer notices that the code has a highly irregular structure. He zooms in on an a larger function body, and notices many function calls and object declarations. By moving the mouse cursor over one of the function calls, the status bar of the program confirms that it is indeed a function call. Programmer Y notices a call to a function with an unusual name, and is curious about its definition. He presses the left mouse button, and the tool immediately presents a visualization of the function definition. Once he grasps the intention of the function, he presses backspace to move back to the function body of the caller. Another striking feature is a large switch-statement based on an enumeration value. The programmer wants to know where these enumerations are defined. He moves his mouse over the enumeration object in the switch expression, and presses the right mouse button. A context menu appears with, among other options, an option 'goto declaration'. The tool instantly presents the declaration of the enumeration variable, located in the same function body. The programmer moves with his mouse over the type specifier of the declaration and selects 'goto definition' from the context menu. The tool loads the header file where the enumeration type resides, and displays it.

Programmer Y asks programmer X about the intention of these files, whereupon programmer X answers that all three files are for type checking the ASG, once it is fully constructed. The "TemplateInst.cpp" file contains type check functionality specific for template code. The file "ScopeEnv.cpp" is a complex data structure for maintaining the scope environment while traversing the ASG. Programmer X advices programmer Y to have a look at the error handling of the type checker. Furthermore, he should have a look at the objects allocated on the stack. Guaranteed destruction of those objects is often used to write exception safe code.

Color profiles

Programmer Y highlights all code related to exception handing by switching to a different color scheme. He applies a color scheme for highlighting error handling code in bright colors. Catch

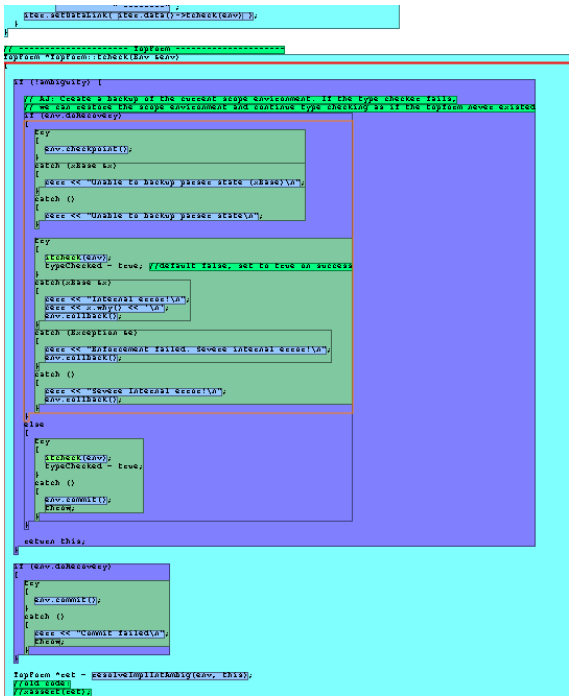


Figure 6.2: Source code visualization of the `TopForm::tcheck` function

statements are shown in a dark red color, while catch statements are in a dark purple color. The color scheme also highlights string literals, because these often contain intuitive error messages directed to the user. String literals are shown in a dark green color. All other constructs are visualized in light colors, and are barely visible against the white background.

Queries

Next, programmer Y runs a query to select all objects allocated on the stack, and highlights the selection in a conspicuous color. Many small highlights occur in function bodies. He zooms in on one of the objects, and recognizes that it is a variant of the scope guard micro pattern [4]. The destructors bring the parser's scope environment to a consistent state once the function returns, either as the result of a return-statement or because of an exception.

Reachable functions

Programmer X tells that programmer Y that he should have a look at function `TopForm::tcheck`, the central function of the type checker. From here, many other objects are constructed, and functions are called.

Programmer Y uses a query to search the fact database for the `tcheck` member function of the `TopForm` class. The programmer adds highlights for comments, if statements, exception handlers, and function calls. Figure 6.2 shows the resulting source code visualization. Visual EFES highlights the function selection, which the programmer Y uses as input for a reachable function query. He specifies that he wants to export a Dot call graph of the query result showing all functions reachable in at most 3 calls.

The resulting call graph is relatively small and unstructured, so the programmer is able to obtain an adequate visualization using Dot from Graphviz. The call graph gives the programmer insight in the most important routine of the type checker, and reveals the calls to other type check functions.

Chapter 7

Discussion

7.1 Introduction

This chapter discusses the successes and failures we encountered when we implemented our tool set. It covers each component of VISUAL EFES, viz. fact extraction, fact querying and enrichment, and fact visualization individually.

7.2 Fact extraction

Besides producing the ASG output, we added support in EFES to calculate various statistics and metrics, mainly for benchmarking and quality checking. We gather statistics on e.g. line count, parse error ratio (correctly versus incorrectly parsed code), number of type checking errors and warnings, (missing) includes, type lookup (semantic) errors, and internal parse errors (Sec. 3.2.2). Understanding the complex correlations between such metrics and various attributes of the input code, e.g. input and included code size, frequency of various language constructs, amount of incorrect or incomplete code (missing declarations, syntax errors), is quite hard. Reading the metrics as text is quite difficult, just as when reading the often cryptic warnings and errors of a compiler. We added the option to EFES to create a standard database table whose rows are the extracted project files, and whose columns are the statistics of interest, e.g. file size, included code size, number of syntax errors, number of type check errors, number of missing includes, number of extracted types, and so on. For large projects, such a table can contain hundreds of rows and tens of columns. To help understanding these data, we have used the novel 'table lens' visualization technique [69]. The table lens renders a table as a standard spreadsheet with two extra enhancements. First, it draws colored bar graphs atop of each column. This allows one to easily follow the variation of each attribute and also visually detect correlations between attributes in several columns. Second, the table lens can be zoomed out until every table row becomes one pixel row, i.e. the complete table is reduced to a set of column bar graphs. This allows visualizing huge tables compactly on a single screen to quickly detect correlations.

Figure 7.1 shows the table lens visualizing the EFES output of a small C++ project with 46 files and 35 KLOC (without system includes). The column names are indicated for a number of attributes of interest. The table rows are sorted increasingly on number of includes (rightmost column). Several interesting facts are visible here. First, only two files parsed with 1, respectively 4 errors (column 'Parse errors'). These occur as we used the Visual C++ 7 profile for extraction, while the project was compiled with Visual C++ 6, and Visual C++ 7 defines `xor` as a reserved keyword. These errors generate garbage tokens (column 'Garbage tokens'), as explained in Sec. 3.2.1, as well as several subsequent type check warnings and type check errors. Another interesting observation is that several attributes seem to be strongly correlated (columns 'Input size', 'Output size', 'Filtered lines', 'Total lines', 'Type check warnings', 'Type check errors', and 'Number of includes'). For roughly the first half of the files (upper half of Fig. 7.1), all these

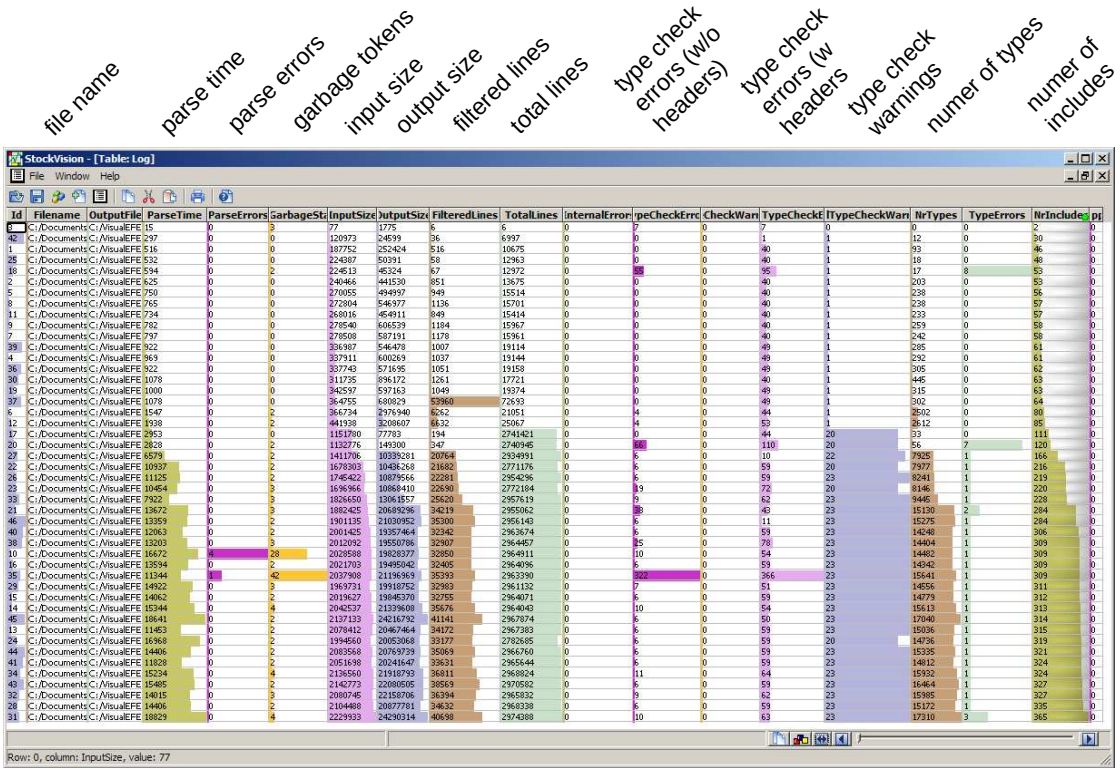


Figure 7.1: Table lens visualization of EFES extraction statistics

attributes have small values. For the other half of the files (bottom half of Fig. 7.1), their values are quite large. Somehow, our code base seems to 'naturally' split into two halves. Upon closer inspection, we saw that all files in the top half are platform and library independent, whereas the ones in the bottom half include wxWidgets headers, which includes Windows headers. Moreover, there is an undefined macro which is used in many wxWidgets headers. The above explain the statistics: Platform and library independent files parse quickly and error free. Platform and library dependent files parse slowly, as they include a lot of code from the Windows headers. Also, these generate many type warnings and errors because they all include an 'incorrect' library header which refers to missing types due to the undefined macro.

We noted that CANPP, the preprocessor of Columbus, can hang. CANPP has no facility to prevent this from happening. This is unfortunate, because it can stop the processing of an entire batch. We avoid this problem by adding a timeout-based watchdog which terminates the extractor for that file on hanging. This guarantees that the batch always continues while marking the file that has failed. EFES uses the Libcpp preprocessor, which is an extremely stable preprocessor library, also used by the GCC compiler. Given its widespread use, it is highly unlikely that internal errors occur in this preprocessor. And even if such errors ever occurred, this has no consequence for the extraction of facts from other translation units in our setup.

Internal errors in the type checker of EFES are more likely to occur, because it is not as well tested as Libcpp. We handle such errors at a finer granularity than the translation unit, namely the topform level. If a topform contains problematic code, it is treated as whitespace. The remainder of the code is processed as if the problematic topform does not exist (Sec. 3.2.1). Columbus lacks such a recovery mechanism, even though several internal errors did occur when using it in our tests. The result of an internal error is that no output is generated for the problematic translation unit. Hence, if most (or all) translation units of a project include a problematic header file, which can happen, little (or no) output is generated by Columbus.

From another perspective, there are many technical analogies between DMS and EFES. Just as ourselves, Baxter *et al.* name in all their papers about DMS the *scale* of the nowadays software as the "'principal issue" behind the success or failure of program analysis tools [11]. Because of this, there are several technical and design decision similarities between DMS and EFES. Both approaches strongly advocate that using a GLR grammar for parsing C++ massively simplifies and cleans up the parser design. Just as DMS, we cleanly separate parsing from non-context-free issues such as lexical scoping and type checking. In both DMS and EFES, the parser is automatically generated from the GLR C++ grammar. For this, we use the ASTGEN tool [61], whereas DMS uses proprietary tools. Just as DMS, EFES separates lexing and parsing, although the two can be conceptually integrated in a single parse step. In both cases, the advocated reason is efficiency. Finally, both DMS and EFES save the extracted information in a 'hypergraph' accessible via a clearly defined API and construct subsequent program analysis tools atop this API. However, there are also important differences between DMS and EFES. First and foremost, EFES focuses strictly on C/C++, whereas DMS targets an open set of languages, via a so-called 'domain specification' definition [11]. The higher layers of the DMS toolset provide program transformation functions, whereas EFES focuses *strictly* on being a good C++ fact extractor. Not surprisingly, DMS is far more complex than EFES. DMS was built on a period of about 15 years by a team containing over 5 persons a year [57]. DMS has about 1,5 MLOC C++ core code [2]. In contrast, EFES was built by essentially 3 people over a period of around one year, and has around 60 KLOC C++ core code.

Currently, EFES supports both an XML output format as well as our own binary output format. The advantage of our binary format is that is much faster to read and write than XML, as described in Sec. 3.5. It is also very compact, which is important when analyzer output can span tens of gigabytes, as e.g. for the VTK and wxWidgets code bases (Table 6.5). EFES does not support as many output formats as COLUMBUS. However, since we output all information about the source code (requirement (req. 7)), separate translators can be easily written that convert to other formats, e.g. COLUMBUS or Datrix [35, 12].

Although EFES supports most language features of C++, a very few of the more complicated features, e.g. template template parameters and template class template members, are only partially supported. The level of template support of EFES is superior to what COLUMBUS offers, but not yet on par with modern compilers, e.g. Microsoft Visual C++ 8 and GCC 4.0.1. This issue will be addressed in the future. Overall, our template support level is largely adequate to analyze most nowadays software. Most projects we analyzed made little use of such complex template features. When they did so, this was by including some template library such as the STL. In such cases, the error recovery of EFES (Sec. 3.2.1) handles parsing as if the unsupported template constructs are missing from the input. This allows us to produce almost complete output even in such extreme cases.

Finally, we constructed a so-called linker tool, similar to CANLINK in COLUMBUS. The linker is useful when examining the several output files produced by EFES in the analysis of projects containing several source files. The linker reads all output files, identifies the possible occurrences of the same topform in different files, and saves these in a per-project index file. This file can be further used e.g. by program analysis or visualization tools. A traditional object code linker identifies the occurrences of the same functions or global data objects in several files. Our linker goes beyond this, as it is capable to identify multiple occurrences of any kind of topform, e.g. type declarations.

7.3 Fact querying and enrichment

We added a generic query system to EFES. The query system matches a pattern specified by a query tree built from query nodes. Query nodes types exactly mirror ASG node types and are entirely generated ASG. The query system is very easy to maintain, because our query nodes are automatically generated. Whenever the ASG changes, our query system is automatically updates as well.

The biggest problem we encountered during the implementation of our query system, was that several ASG node types were directly specified in C++. This meant that we had to write a MultiGen description for these node types. Unfortunately, these types used multiple inheritance regularly, whereas the tool was designed specifically for specifying single inheritance trees. We were able to work around this problem with a slight adaptation of MultiGen and by augmenting the generated code with a few lines of handwritten C++ code. This is done in such a way that is has little impact on the maintainability of the system.

A disadvantage of matching patterns in the ASG, is that the parts of the ASG structure must be reflected in the query tree. This requires thorough knowledge of the highly intricate ASG structure of EFES. We could improve the situation by adding several shortcuts to the query nodes, such that not only ASG nodes directly referenced can be queried, but also a number of ASG nodes further away from the queried node. For example, it is nice to directly query the return type of a function, instead of having to create a query for the function declarator object first. However, these shortcuts would decrease maintainability and result in bloated code. Instead, we gave the user the ability to define his or her own query shortcuts using aggregate queries, which is much more flexible.

We discovered that it was very laborious to write queries using the C++ query API. Every time we had to change a query, we had to recompile the program and reload the fact database. We worked on a large fact database, consisting of more than a hundred extraction units, and reloading the entire database took a few minutes time. Obviously, having to reload the database after each query modification is unacceptable.

Clearly, we needed a way to add or change queries while the Visual EFES tool was running. We considered several options for achieving this, viz. binding the API functionality to a scripting language, using query DLL's through a plug-in API, and XML serialization. In the end, we settled on XML serialization, because XML is a standardized declarative language for which various supporting tools are available. Another benefit of using XML is that it is less error-prone. Queries are often written using trial-and-error methods, and it is undesirable that an accidental mistake of the query developer crashes Visual EFES. An erroneous query specified in XML will not cause the query system to crash. On the contrary, usually a descriptive error message indicating the query error is provided to the user. In contrast, a query specified in a scripting language, or through a DLL, has access to the unsafe C++ API, and can easily pass invalid objects to the API which may result in a crash. The XML serialization and de-serialization code is automatically generated using MultiGen.

We are extremely pleased with the performance of the query system. Currently, the query execution time is dominated by the time it takes to read extraction units from the storage device into memory, and inserting identifiers into the selection data structure. We were aware of the importance of parser performance, and aggressively optimized the file format and the parser at an early stage of the development process. The selection data structure is already asymptotically optimal, but can be replaced by more performant implementations at any time. We applied several optimization strategies to make sure the actual ASG traversal takes but a fraction of the total query time.

7.4 Fact visualization

In chapter 5 we have presented our visualization framework and the source code visualization which is an application of this framework. The complex part of source code visualization is the creation of a data structure to visualize. In this section we will discuss in more depth the problems we found and the first approach we took.

To visualize an ASG as source code in the *original* layout (i.e. as seen in a text editor) we need to store layout information in the ASG. Our first approach was to include location information of the beginning and the end of each syntactical structure. Location information means, the row, column and file of the located character and if the character (thus the complete token) origins from macro expansion or not. When we read the ASG containing this information, for visualization, we

found there to be cases where we need more layout information then we had. Certain syntactical constructs can contain up to six separate tokens, in which case four of them would have to be laid out by heuristics. A qualifier for example has five possible tokens associated with it. This proved to be too complicated and error prone in the sense that the layout would deviate too far from the original. Another great issue with this approach was that we did not know of each token if they originated from macro expansion or not. Macro expansion and include operations can introduce single tokens that might not match the beginning or end of a single syntactical construct. Hence that token would be considered as an original originating from the text, without involvement of the preprocessor, which would break the layout if we tried to fit it in anyway. Many of the fact database formats that we have seen record location data at one or two points of syntactical constructs only, and hence can not be used for precise recovery of the original layout.

We build up an item tree, the data structure for visualization, that exactly reflects the syntax structure. It also contains all the visible tokens as leaves. We claim that all tokens are placed in the item tree in the context of their matching syntax. There is one inconsistency though, which we will demonstrate with an example. Consider the following code:

```
long inline long getDateTime();
```

The function has return type `long long` (a GCC extension) and is specified to be an inline function. The syntax tree normally does not include tokens and there arise no problems constructing it. However, we construct a data structure containing both syntax and tokens. Both `long` tokens need to be included in the *type specifier* subtree but the `inline` token does not belong there. This is a result of C++ syntax which allows for declaration directives and type specifier keywords to be mixed. If the syntax would require the type specifier to be a consecutive uninterrupted set of tokens then no problems would arise. A solution to this problem would be to complicate the data structure by splitting up one item into two items but we choose to enclose the `inline` keyword in the *type specifier* subtree.

Chapter 8

Conclusions

8.1 Introduction

Our main goal was to design and implement a source code visualization tool that gives both programmers and managers insight in large C++ code bases. This has been accomplished with our visualization tool. The tool gives insight in code bases at multiple levels of detail. At the lowest level it can present source code annotated with all kinds of visual cues, and at a higher level it can create graphs showing dependencies between modules.

An important component of the visualization tool is the fact extractor, which is responsible for extracting information from source code. We were unable to find a fact extractor that satisfied our requirements adequately, so we had to develop our own solution. Our extractor complies with all our fact extractor requirements, and proved to be the right front-end for our source code visualization tool.

8.2 Fact extraction

We presented the architecture and design of EFES, a robust fact extractor for industry-size C and C++ code bases. First, we gathered an explicit set of requirements that fact extractors should fulfill for being competitive. Next, we surveyed how a number of popular fact extractors address these requirements. We narrowed our analysis to the two most performant ones, Elsa and Columbus. We described how we extended Elsa with two types of error recovery, enhanced preprocessing, output filtering, and a project concept, to deliver a performant fact extractor solution. EFES has a very fast GLR parser. On most projects in our benchmark, EFES performed three up to ten times faster than Columbus. The only limitation EFES has as compared to Columbus is the lack of precompiled header support. However, the speed of the original Elsa parser and our own error recovery architecture largely compensates for this.

Although more robust than all other parsers we examined, EFES can not yet parse the latest extensions to the C++ language perfectly. We limit the impact of such parse errors to the bounded extent of the enclosing topform, report them in the ASG cleanly, and continue parsing. Given the Elsa parser design, based on a GLR grammar and modular parser and type checker separation, we can (and will) easily remove these minor limitations.

C++ parsers come in many flavors, ranging from the classical solutions using LALR(1) grammars up to modern ones using GLR grammars. GLR grammars provide major parser design advantages: A GLR C++ grammar is compact and simple to understand and adapt, and disambiguation using type checking is done modularly after parsing, simplifying the design. However, using a GLR grammar only facilitates, but does *not* guarantee success in building a C++ parser. Our experience is that the overwhelming part of the design, development, debugging, and optimizing effort goes into the very complex symbol table and scoping environment that drives the type checker during disambiguation. In this sense, we are relatively less optimistic than Van den Brand

Extractor tool	1 fault-tolerance	2 completeness	3 compliance	4 cross-references	5 preprocessing	6 full coverage	7 output completeness	8 scalability	9 portability	10 availability
BisonXML	−−	++	+	−−	++	++	−−	++	+	++
Src2srcML	++	+	+	−−	−−	−	+	++	++	++
Doxygen	++	−	−	∘	+	−−	+	++	++	++
Sniff+	++	−	∘	+	+	−	−−	+?	++	−−*
DMS	++	++	∘	++	++	++	++	++	++	−−*
Gccxml	−−	−?	−−?	+?	++?	++	−−	++	∘	++
Cppx	−−	+	+	++	++	+	−−	++	∘	∘
Columbus	+	++	+	+	++	++	+	+	++	−−*
Elsa	−−	++	+	++	−−	++	+	∘	+	++
EFES	++	++	+	++	++	++	++	++	++	++

Scale: −− (very limited), −, ∘, +, ++ (excellent)
? This result is an educated guess
* Closed source tools

Table 8.1: Survey summary

et al. [71, 72] that parser technology can be easily and modularily reused for heavily context-dependent languages such as C++ *and* when one requires disambiguated output. True, a GLR grammar and/or parser might be modular and compositional [72]. However, the type checker, like in the case of C++, is neither modular nor compositional.

A final word on extractor design. As in many other fields, there is no silver bullet for parsing and extracting facts from C++. Practitioners in this field should carefully weigh their specific application requirements against what extractors offer, and then choose. When this analysis suggests no extractor is good enough and/or available (e.g. due to closed source), one has to build a new extractor, typically by extending an available one. Our main contribution here has been to detail all the technical steps which are of highest importance when one ventures to accomplish this task.

As it is right now, there are virtually no tools on the market that can compare themselves with the orchestrated set of features provided by Visual EFES for parsing, querying, and visualizing information at the level of C++ source code. Given the amount of effort we put in this project (around 3 man-year), the obtained results, and the fact that relatively more restrictive and less performant and scalable systems of the same kind were developed with around 10 times these resources (30 man-year), we claim that our endeavor has been successful. Visual EFES is only at its beginning. A multitude of extensions and consolidations of the current framework are possible. We envisage exciting possibilities in adding support for visual query design, code metrics, code clone detection and pattern detection capabilities, and novel forms of fact visualization, all concurring to the provision of a truly industry-grade, professional environment for interactive visual analysis and navigation of large C++ code bases.

8.3 Fact querying and enrichment

Our objective was to design and implement a generic system for querying the structure of source code. This is accomplished with our highly flexible query system.

The query API offered by EFES can be used to obtain refined information, such as call graphs, design patterns, metrics, or code comparison operators.

Complex queries for the query API can be stored in XML and loaded while the system is running. This is a convenient and safe method for writing queries. Errors in erroneous queries

are reported to the user, and can hardly ever result in crashes. Multiple query files can be can be categorized in a query library.

More advanced users can query aggregate queries, which can function as easy to use building blocks for other query developers. Novice users can combine existing queries and specify query parameters in a straightforward way. The query system shows a graphical user interface for editing query parameters. Queries from a query library can be combined using for example the binary operators.

Another important point of the query system is maintainability. A query system that needs adaptations every time the ASG structure is changed is a constant nuisance and quickly becomes out of date. Our query system hardly needs maintenance, because it is automatically generated using MultiGen. It is updated synchronously with the ASG structure.

Given the very promising results at this point, we aim to use EFES as the core element of a software analysis and understanding framework, similar to Columbus, DMS, or WindRiver's Workbench, by combining it with several visualization and interactive navigation techniques for source code [47, 73, 70], more sophisticated code metrics [16, 22], and other program analysis techniques.

8.4 Fact visualization

We have applied two kinds of visualizations and reported about both, in this thesis. The graph visualizations are applications of existing visualizations and we will not discuss them here, but we will discuss the source code visualization.

We have succeeded in constructing a data structure containing all visual information, the tokens, as well as their full context. This allows for an advanced and highly dynamic visualization. The high quality fact extractor provides a fully annotated ASG, enabling our visualization. As we have shown, we can provide a view of the original source code that looks identical to the view provided by a source code editor. This view forms a familiar starting point for the user. The complete visual data structure has links to the fact database which allows for the most complex annotations upon this basic view.

The visual data structure can be constructed on any scale. This means that we can visualize everything in a single view ranging from an entire extraction unit up to a series of expressions. Our visualization framework allows for different layouts and looks of the same data. We can display the data as source code but, with simple modifications, we can display the data in a tree map.

The source code visualization can harness the power of the query system by displaying the result directly in the visualization, either as a new text representation or as annotation on an existing visualization. This is possible since both the query system and the visualization communicate through selections on the fact database.

The visual environment supports the persistent storage of syntax color configurations. This allows the user to make configurations for different goals. Since the user can switch between the color configurations instantly the same visualization can show different patterns in the source code. This idea is parallel to having persistent queries that can be applied on the same visualization yielding a view of patterns showing details of any degree of complexity.

A conclusion on source code visualizations in general. The more one wants to annotate source code in a visualization, the less lines of source code can be fit on the screen at the same time.

8.5 Future work

8.5.1 Fact extraction

The EFES fact extractor is working extremely well within its limits of C++ language support. There is however a fundamental lack of support for certain language features or language extensions. One example of these features is support for template template parameters. These are very advanced features that are not in widespread use, however the fact extractor has room for

improvement in this direction and will be imperfect as long as it does not support at least the complete standard C++ language.

8.5.2 Fact querying

Although our current query system is extremely powerful and flexible, its most important shortcoming is that it requires a skillful programmer to write a query from scratch. The developer must have reasonable familiarity with the ASG structure utilized by EFES. Aggregate queries certainly facilitate the process of writing queries, and enable inexperienced programmers to write compelling queries. But, implementing queries using purely prefabricated aggregate queries restricts the flexibility of the query system.

The usability of the query system can be significantly enhanced, if queries can be written in a language similar to C++, but supplemented with certain wild cards. We can add that functionality to our query system by modifying our C++ parser in such a way that it constructs the query tree in roughly the same way as it currently builds the ASG. This is the same approach as proposed by [83]. Because most users of our tool are already familiar with C++, they can almost instantaneously specify complex queries. For programs without C++ knowledge, using prefabricated queries stays the best entrance gate to writing queries.

Currently queries are stored as XML files, which the query developer can edit using his favorite XML editor. We would like to have a graphical designer user interface for queries. We prefer a node based query editor, because such an editor is flexible, easy to implement and maintainable. The central benefit of a graphical query designer, compared to an XML editor, is that it can offer the user superior feedback. The XML parser is only invoked when the query is opened, and has limited knowledge about the intentions of the query developer. In contrast, a query designer can immediately provide an intuitive message when the user tries to create an invalid query. More strongly, the designer can even enforce several constraints by simply not showing invalid options. The designer can also assist the user by showing a list of nodes that may be attached to a specific query attribute.

Furthermore, the user interface can aid in abstracting query nodes into aggregate queries. The user can select a cluster of query nodes, and press a button for grouping them into a single aggregate query. Subsequently, the user can define ports for the newly created aggregate query and link the ports to attributes.

8.5.3 Fact visualization

C++ projects can be structured in a way such that different extraction units have a different interpretation of the same file. The general example is where a header file has conditional compilation on a certain condition, or uses a macro, where this condition or macro call has been defined differently between extraction units. Because of the chance of this happening, we offer views on header files based on the extraction unit they are in. In a typical project people intend to void the case where extraction units have interpretation differences of a header file, and often they succeed in avoiding this. An improvement of our system in general is the ability to detect if one header file is always interpreted in the same way in every extraction unit. If such a header file is found, we can offer the visualization of the header file in its own right, without the need to select a specific extraction unit that contains the header file.

Appendix A

Data access API

A.1 Introduction

EFES generates very large databases containing all information about source code. EFES uses its own highly optimized file format for storing this immense amount of data. This results in files that are an order of magnitude smaller, and faster to handle than a similarly structured XML file. Like XML, the EFES file format allows the user to selectively read data. EFES comes with an open object-oriented C++ API called EFESAPI for reading the files.

A.2 Programming interface

The entire interface of EFES API resides in the EFES namespace. This way potential name clashes with your own code are easily avoided, making it easier to integrate EFESAPI easy in your own applications. In the remainder of this document we will refer to symbol names without explicitly qualifying them with the EFES namespace.

The most important class in EFESAPI is the `ExtractionUnit`. An extraction unit is an output file produced by EFES. It stores all information about a single C++ translation unit. The file name of the extraction unit is usually the name of the original source file with the ".efes" extension as suffix. An `ExtractionUnit` object can be used for loading, saving, and accessing the information of an extraction unit.

EFES produces a fact database file (*.factdb) when it finishes a project. This is an XML file storing all extraction units that were produced by EFES as well as some additional information about the project configuration. The EFESAPI contains a `FactDB` singleton class for accessing and manipulating fact databases. You can obtain a reference to the singleton instance by calling the `GetFactDB` function. The function `FactDB::load` loads a fact database file. The function throws a `FileOpenError` exception if the file cannot be opened for reading. If the file is corrupt EFESAPI throws a ParseError exception.

Loading a fact database

```
EFES::GetFactDB().load("test.factdb");
```

Each extraction unit has a unique identifier in the fact database. Identifiers are numbered consecutively starting from zero. The `FactDB` class has a member function size, returning the total number of extraction units in the list.

The function `FactDB::get` accepts an extraction unit identifier as parameter and returns a reference counted extraction unit, if the file exists. Reference counting is automated using a smart pointer class from the Boost libraries. If the reference counted object expires, all data stored in

the extraction unit, e.g. AST and type information, is automatically freed from memory. The get function throws a `FileOpenError` exception if the file cannot be opened for reading.

Obtaining extraction unit objects:

```
for (int i=0; i!=GetFactDB().size(); ++i)
  boost::shared_ptr<ExtractionUnit> file = GetFactDB().get(i);
```

Of course we do not require you to work with fact database files produced by EFES. Alternatively, you can manually compile a fact database in code. Manually created `ExtractionUnit` objects can be added to the database using the `ExtractionUnit::addUnit` function. Using `ExtractionUnit::save`, the current fact database can be written to a file.

Once an `ExtractionUnit` object is obtained, the contents of the extraction unit can be read into memory. There are two overloaded read functions for this.

Overloads of the `ExtractionUnit::read` function:

```
void read(bool readAST, bool readTypes, bool readPrepro);
void read(BinReadVisitor &visitor, bool readASTStrings,
          bool readTypes, bool readPrepro);
```

EFESAPI distinguishes three kinds of data in an extraction unit: the abstract syntax tree (AST), type information, and the preprocessor information. Both read functions allow you to specify which parts of the extraction unit you want to read. The second overload also accepts a visitor object, which allows you to selectively read the AST. We recommend using the first overload if you want to read the entire AST, because that is slightly more efficient.

It is easy to create your own visitor object by creating a new class derived from `BinReadVisitor`. By default, this visitor reads all nodes of the AST.

The `BinReadVisitor` object contains a visit method for all types of AST nodes. If a visit method returns false, all AST nodes of that type, and all their children, are skipped. You can override the visit methods to return false.

The following visitor skips compound statements and expression statements and their children, and reads all other nodes.

Creating a custom visitor by deriving from `BinReadVisitor`:

```
class LinkVisitor : public EFES::BinReadVisitor
{
   bool visitS_expr() {return false;}
   bool visitS_compound() {return false;}
};
```

Often, however, you cannot decide on a per-type basis what you want to read. For this, `BinReadVisitor` offers the `visitChildren` and `postVisit` sets of functions.

The `visitChildren` methods are called before the children of a node are read. Partial information about the node, such as its location, is passed as arguments to the function. By default, these functions return true. By returning false, all children of the node are skipped.

The `postVisit` functions, as the name suggests, are called when a node, and all its children, are stored in memory. The function allows you to decide, based on the complete information about the node and its children, whether you want to keep them in memory. If false is returned, all information about the node and its children will be efficiently discarded.

Once an extraction unit is loaded into memory, EFESAPI offers two ways to traverse the data. You can iterate through all nodes of a specific type, for example through all functions. This is the most efficient way to traverse data, making optimal use of processor caches.

Iterating through all declaration TopForms:

```
TF_declIterator declEnd = file.astIterators()->TF_declEnd();
TF_declIterator iter=file.astIterators()->TF_declBegin();
for (; iter!=tfDeclEnd; ++iter)
   defineVariable((*iter)->decl);
```

Secondly, you can traverse the in-memory ASG using a visitor. It is easy to construct your own visitor by deriving from the `ASTVisitor` interface and overriding the visitASTNode function.

Writing a custom visitor

```
class MyVisitor : public ASTVisitor
{
  Visit MyVisitor::visitASTNode(ASTNode &node)
  {
        return VISIT_CHILDREN;
  }
};
```

The `visitASTNode` method must return a value of enumeration type Visit. Possible return codes are:

VISIT_CHILDREN_AND_POST
Visit the node, all its children, and also do a postVisit

VISIT_CHILDREN
Visit the node, and all its children

VISIT_SIBLING
Directly move to node sibling, ignoring node children

VISIT_POSTPARENT
Directly move to the sibling of the parent, ignoring node children and all node siblings

VISIT_STOP
Stop the visit process. No further nodes are visited

The class `ExtractionUnit` has a member function ast for obtaining the root of the AST. It returns a pointer to a TranslationUnit object which supports the `ASTNode` interface. The root node can be used as a starting point for an AST traversal.

Starting a traversal at the root of the AST:

```
MyVisitor visitor;
visitor.traverse(extractionUnit->ast());
```

A.3 Global Identifiers

The class `GlobalId` stores a global node identifier. A global identifier is used to uniquely identify an AST node in a list of extraction units. Global identifiers are a combination of an extraction unit identifier, and a node identifier in an extraction unit. Global identifiers are required, because node pointers are not persistent. EFESAPI offers the function `GetSelectable` for obtaining a pointer to the identified node. The function loads the extraction unit containing the AST node into memory if needed.

Given a pointer to an extraction unit, and a pointer to an AST node, it is possible to construct a `GlobalId` in constant-time using the id functions.

Constructing a Global Identifier:

```
GlobalId CreateId(ExtractionUnit *unit, ASTNode *node)
{
  return GlobalId(unit->id(), node->id());
}
```

The id functions never throw exceptions.

A.4 Error handling

EFES API uses C++ exceptions for handling exceptional conditions. All EFES API exceptions are derived from class `Exception`. The API may also throw STL exceptions, usually to indicate more critical errors. The client application is responsible for handling these errors.

The `Exception` class has a member function `what`, returning a string containing an intuitive description of the error that occurred.

EFES API defines several classes derived from the Exception base class. `FileOpenException` is used for indicating errors when trying to open a file format. The exception may be thrown by `ExtractionUnit::openFile`. `ParseError` is used to indicate parse errors when trying to read input files. This may indicate file corruption, for example due to a version conflict. This exception is potentially thrown by the `FactDB::load` and `ExtractionUnit::read`. Most EFES API functions may throw other exceptions derived from Exception, e.g. `NullPointerException`, `OutOfBoundsException`, or `GeneralException`. These exceptions should be rare, and probably indicate version conflicts.

Bibliography

[1] R. Akers, I. Baxter, M. Mehlich. *Program transformations for reengineering C++ components*, ACM SIGPLAN Notices, vol. 39, no. 10, ACM Press, 2004.

[2] R. Akers, I. Baxter, M. Mehlich, B. Ellis, K. Luecke. *Re-engineering C++ Component Models Via Automatic Program Transformation*, Proc. WCRE, IEEE Press, 2005, pp. 13-22.

[3] A. Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*, Addison-Wesley Professional, 2001.

[4] A. Alexandrescu, P. Marginean. *Generic: Change the Way You Write Exception-Safe Code Forever*, Dr.Dobb's Journal, CMP Media LLC, April 15, 2003.

[5] P. Anderson, T. Reps, T. Teitelbaum, M. Zarins. *Tool Support for Fine-Grained Software Inspection*, IEEE Software, vol. 20, no. 4, IEEE Press, 2003, pp. 42-50.

[6] R. Anisko, V. David, C. Vasseur. *Transformers: A C++ Program Transformation Framework*. Technical Report 0310, May 2004, LRDE, France
http://www.lrde.epita.fr/dload/20030521-Seminar/vasseur0503_transformers_report.pdf

[7] G. Antoniol, M. Di Penta, G. Masone, U. Villano. *Compiler Hacking for Source Code Analysis*. Journal of Software Quality, vol. 12, Kluwer, 2004, pp. 383-406.

[8] M. N. Armstrong, C. Trudeau. *Evaluating architectural extractors*. Proc. 5^{th} Working Conf. on Reverse Engineering (WCRE), IEEE Press, 1998, pp. 30-39.

[9] AT&T. *The GraphViz Package*
http://www.graphviz.org

[10] Zs. Balanyi, R. Ferenc. *Mining Design Patterns from C++ Source Code*. Proc. 19^{th} Intl. Conference on Software Maintenance (ICSM'03), 22-26 September 2003, Amsterdam, The Netherlands, IEEE Press, 2003, pp. 305-314.

[11] I. Baxter, C. Pidgeon, M. Mehlich. *DMS: Program Transformations for Practical Scalable Software Evolution*, Proc. ICSE, IEEE Press, 2004, pp. 625-634.

[12] Bell Canada. *DATRIX abstract semantic graph reference manual (version 1.4)*. Technical report, May 2000. Bell Canada.

[13] W.R. Bischofberger. *Sniff+: A Pragmatic Approach to a C++ Programming Environment*. Proc. USENIX C++ Conference, Portland, Oregon, 1992, pp. 67-81.

[14] Blender 2.37a
http://www.blender3d.org

[15] M. G. Burke, G. A. Fisher. *A practical method for LR and LL syntactic error diagnosis and recovery*. ACM Trans. on Programming Languages and Systems, vol. 9, no. 2, ACM Press, 1987, pp. 164-167.

[16] D. Coleman, D. Ash, B. Lowther, P. Oman. *Using Metrics to Evaluate Software System Maintainability*. IEEE Computer, vol. 27, no. 8, IEEE Press, 1994, pp. 44-49.

[17] M. L. Collard, H. H. Kagdi, J. I. Maletic. *An XML-Based Lightweight C++ Fact Extractor*. Proc. of the 11^{th} IEEE International Workshop on Program Comprehension (IWPC'03), IEEE Press, 2003, pp. 134-143.

[18] Columbus CAN 3.5. Frontendart
http://www.frontendart.com/products.html

[19] V. David, A. Demaille, R. Durlin, O. Gournet. *C/C++ Disambiguation Using Attribute Grammars*, Proc. 6^{th} Stratego User Days, 2-4 May 2005, Utrecht University, The Netherlands. Also as Technical Report, LRDE, France
http://www.lrde.epita.fr/dload/200505-SUD/disamb/article-200505-SUD-disamb.pdf

[20] C. Donnelly, R Stallman. *Bison: the YACC-compatible parser generator*. Free Software Foundation, 1995.

[21] EFES Home Page
http://www.virtualice.nl/efes

[22] N. E. Fenton, S.L. Pfleeger. *Software metrics: A rigorous & practical approach*. International Thomson Computer Press, London, 1996.

[23] R. Ferenc, I. Siket, T. Gyimóthy. *Extracting Facts from Open Source Software*. Proc. 20^{th} Intl. Conference on Software Maintenance (ICSM'04), Chicago, Illinois, USA. IEEE Press, 2004, pp. 60-69.

[24] R. Ferenc, Á. Beszédes, T. Gyimóthy. *Extracting Facts with Columbus from C++ Code*. Tool Demonstrations of the 8^{th} European Conference on Software Maintenance and Reengineering (CSMR'04), Tampere, Finland, IEEE Press, 2004, pp. 4-8.

[25] R. Ferenc, Á. Beszédes, M. Tarkiainen, T. Gyimóthy. *Columbus - Reverse Engineering Tool and Schema for C++*. Proc. 6^{th} International Conference on Software Maintenance (ICSM'02), 3-6 October 2002, Montreal, Canada, IEEE Press, 2002, pp. 172-181.

[26] R. Ferenc, Á. Beszédes. *Data Exchange with the Columbus Schema for C++*. Proc. 6^{th} European Conference on Software Maintenance and Reengineering (CSMR), IEEE Press, 2002, pp. 59-66.

[27] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns: Elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[28] GCCXML C++ to XML Fact Extractor
http://www.gccxml.org/

[29] Grammatech, Inc.
http://www.grammatech.com

[30] C. Hankin, I. Siveroni (editors). *Static Analysis*. Proc. 12^{th} Intl. Symposium on Static Analysis (SAS'05), Springer LNCS 3672, 2005.

[31] D. van Heesch. *User Manual for Doxygen 1.4.5*
http://www.doxygen.org/

[32] R. C. Holt, A. Winter, A. Schurr. *GXL: Toward a Standard Exchange Format*. Proc. 7^{th} Working Conference on Reverse Engineering (WCRE'00) p. 162.

[33] D. Holten Hierarchical Edge Bundles: Visualization of Adjacency Relations in Hierarchical Data. To appear in Proc. InfoVis'06, IEEE Press, October 2006.

[34] R. C. Holt. *TA: The Tuple Attribute Language*. Technical Report, Department of Computer Science, University of Waterloo, 27 February 1997

[35] R. C. Holt, A. E. Hassan, B. Laguë, S. Lapierre, C. Leduc. *E/R Schema for the Datrix C/C++/Java Exchange Format*. Proc. Working Conference on Reverse Engineering (WCRE'00), IEEE Press, pp. 284-286.

[36] Id Software. *Quake 3 Arena source code 1.31b*
ftp://ftp.idsoftware.com/idstuff/source/

[37] Information Technology Industry Council. *ISO/IEC 14882 C++ Language Standard*, 1^{st} edition. American National Standards Institute, 1998.

[38] B. Karlsson. *Beyond the C++ Standard Library - An Introduction to Boost*. Addison-Wesley Professional, 2005.

[39] Kitware, Inc. *The Visualization Toolkit, version 4.2*
http://www.kitware.com

[40] M. Klaus. *Wind River Workbench Product Documentation*
http://www.takefive.com

[41] P. Klint. *A meta-environment for generating programming environments*. ACM Trans. Software Engineering and Methodology, vol. 2, no. 2, ACM Press, 1993, pp. 176-201.

[42] G. Knapen, B. Laguë, M. Dagenais, E. Merlo. *Parsing C++ despite missing declarations*, Proc. Intl. Workshop on Program Comprehension (IWPC'99), IEEE Press, 1999, pp. 114-122.

[43] R. Koppler. *A Systematic Approach to Fuzzy Parsing*, Software - Practice & Experience, vol. 27, no. 6, 1996, pp. 637-649.

[44] A. Günes Koru, J. Tian. *Comparing high change modules and modules with the highest measurement values in two large-scale open-source products*, IEEE Trans. Soft. Eng., vol. 31, no. 8, IEEE Press, 2005, pp. 625-642.

[45] I. Lemke, G.Sander. *VCG: Visualization of Compiler Graphs. Design Report and Documentation*. May 1994, Universität des Saarlandes, Saarbrücken, Germany.

[46] Y. Lin, R. C. Holt, A. J. Malton. *Completeness of a Fact Extractor*. Proc. 10^{th} Working Conference on Reverse Engineering (WCRE'03), IEEE Press, 2003, pp. 196-204.

[47] G. Lommerse, F. Nossin, L. Voinea, A. Telea. *The Visual Code Navigator: An Interactive Toolset for Source Code Investigation*. Proc. IEEE InfoVis'05, IEEE Press, 2005, pp. 24-31.

[48] J.I. Maletic, M. Collard, A. Marcus. *Source Code Files as Structured Documents*. Proc. of the the 10^{th} IEEE Intl. Workshop on Program Comprehension (IWPC'02), IEEE Press, pp. 289-292.

[49] J.I. Maletic, M.L. Collard, H. Kagdi. *Leveraging XML Technologies in Developing Program Analysis Tools*. Proc. 4^{th} Intl. Workshop on Adoption-Centric Software Engineering (ACSE'04) Edinburgh, Scotland, 2004, pp. 80-85.

[50] L. Moonen. *Generating Robust Parsers using Island Grammars*. Proc. of the 8^{th} Working Conference on Reverse Engineering (WCRE'01). IEEE Press, 2001, pp. 13-22.

[51] Mozilla Corporation. *Mozilla Firefox 1.5*
ftp://ftp.mozilla.org/pub/mozilla.org/firefox/releases/1.5/source/

[52] T.J. Parr, R.W. Quong. *ANTLR: A Predicated-LL(k) Parser Generator*. Software - Practice and Experience, vol. 25, no. 7, 1995, Elsevier, pp. 789-810.

[53] V. Paxson. *Flex, version 2.5: A fast scanner generator*. March 1995. The Regents of the University of California.

[54] S. McPeak. *Elkhound: A fast, practical GLR parser generator*. Technical Report UCB/CSD-2-1214, December 2002. Computer Science Division, Univ. of California, Berkeley, California, USA.

[55] S. McPeak *The Elsa C++ parser*.
Available online at: http://www.cs.berkeley.edu/~smcpeak/elkhound/sources/elsa/

[56] S. McPeak. *AST Manual*.
Available online at: http://www.cs.berkeley.edu/ smcpeak/elkhound/sources/ast/manual.html

[57] M. Mehlich, I/ Baxter. *Mechanical Tool Support for High Integrity Software Development*, Proc. HIS'97, IEEE Press, 1997.

[58] J.F. Power, B.A. Malloy. *Program annotation in XML: a parse-tree based approach*. Proc. 9^{th} Working Conference on Reverse Engineering (WCRE), IEEE Press, 2002, pp. 190-198.

[59] Python 2.4.2
http://www.python.org

[60] Qt Graphics User Interface Library
http://www.trolltech.com

[61] J. Rekers. *Parser Generation for Interactive Environments*. PhD thesis, Univ. of Amsterdam, Amsterdam, The Netherlands, 1992.

[62] Semantic Designs, Inc. *Comparison of several C++ front ends*. Available online at
http://www.semdesigns.com/Products/DMS/DMSComparison.html

[63] S. E. Sim, R. C. Holt, S. Easterbrook. *On Using a Becnhmark to Evaluate C++ Fact Extractors*, Proc. IWPC, IEEE Press, 2002, pp. 114-122.

[64] J. Smart, K. Hock, S. Csomor. *Cross-Platform GUI Programming with wxWidgets*. Prentice Hall, 2005.

[65] R.M. Stallman, Z. Weinberg. *The C Preprocessor - for GCC version 4.2.0*. The Free Software Foundation Inc.

[66] M. A. Storey, K. Wong, H. A. Muller. *How do program understanding tools affect how programmers understand programs?*. Science of Computer Programming, vol. 36, no. 2-3, Elsevier, 2000, pp. 183-207.

[67] B. Stroustrup. *The C++ Programming Language*. $3^{r}d$ edition, Addison-Wesley Professional, 2000.

[68] Systems in Motion. *Coin3D 2.4.4*
http://www.coin3d.org/lib/coin/releases/2.4.4

[69] A. Telea. *Combining Extended Table Lens and Treemap Techniques for Visualizing Tabular Data*, Proc. EuroVis'06, IEEE Press, 2006, pp. 51-58.

[70] A. Telea, L. Voinea. *Interactive Visual Mechanisms for Exploring Source Code Evolution*. Proc. VISSOFT'05, 25 September 2005, Budapest, Hungary. IEEE Press, 2005, pp. 52-57.

[71] M. van den Brand, P. Klint, C. Verhoef. *Reengineering needs generic programming language technology*. ACM SIGPLAN Notices, vol. 32, no. 2, 1997, ACM Press, pp. 54-61.

[72] M. van den Brand, A. Selling, C. Verhoef. *Current Parsing Technologies in Software Renovation Considered Harmful*. Proc. Intl. Workshop on Program Comprehension (IWPC'99), IEEE Press, 1999, pp. 108-117.

[73] L. Voinea, A. Telea, J.J. van Wijk. *CVSscan: Visualization of Code Evolution*, Proc. ACM Symposium on Software Visualization (SoftVis'05), ACM Press, 2005, pp. 47-56.

[74] L. Vidács, Á. Beszédes, R. Ferenc. *Columbus Schema for C/C++ Preprocessing*. Proc. 8^{th} European Conference on Software Maintenance and Reengineering (CSMR'04), Tampere, Finland, IEEE Press, 2005, pp. 75-84.

[75] J Abello, F van Ham. *Matrix Zoom: A Visual Interface to Semi-External Graphs* Proceedings of the 10th IEEE Symposium on Information. doi.ieeecomputersociety.org, 2004

[76] G. Sander. *Graph Layout Through the VCG Tool* Graph Drawing, Princeton, 1994, Springer

[77] S.G. Eick, J.L. Steffen, E.E. Sumner Jr. *Seesoft-A Tool for Visualizing Line Oriented Software Statistics*. Software Engineering, IEEE Transactions, Vol.18, No 11, 1992, pp. 957-968. IEEE Press.

[78] *aiSee Graph Visualization*, AbsInt GmbH
Available online at www.absint.com/aisee.

[79] Vadim Engelson. *Call Graph Drawing Interface* Available online at http://www.ida.liu.se/~vaden/cgdi/.

[80] SL Graham, PB Kessler, MK Mckusick. *Gprof: A Call Graph Execution Profiler*. Proceedings of the 1982 SIGPLAN symposium on Compiler Construction, SIGPLAN Notices, Vol. 17, No 6, pp. 120-126, June 1982.

[81] A. V. Aho, B. W. Kemighan and P. J. Weinberger. *The AWK Programming Language*. Reading, MA: Addison-Wesley, 1988.

[82] B. W. Kernighan and R. Pike. *The UNIX Programming Environment*. Englewood Cliffs, NJ: Prentice-Hall, 1984.

[83] S. Paul, and A. Prakash, *A framework for source code search using program patterns* Software Engineering, IEEE Transactions, Vol. 20, No 6, pp. 463-475, 1994. IEEE Press.

[84] K. Wong, *Rigi User's Manual* June 30, 1998

[85] ToolsFactory GmbH. *ClassVis 2.0RC1*
Available online at http://www.toolsfactory.com/classviz.html

[86] J. Wernecke, *The Inventor Mentor: Programming Object-Oriented 3d Graphics with Open Inventor*, Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1993.

[87] K. Sowizral, K. Rushforth and H. Sowizral, *The Java 3D API Specification*, Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1997.

[88] A. Telea, *Combining Extended Table Lens and Treemap Techniques for Visualizing Tabular Data*, Proc. IEEE EuroVis06, IEEE Press, 2006